An accurate analysis for guaranteed performance of multiprocessor streaming applications

Citation for published version (APA):
Poplavko, P. (2008). *An accurate analysis for guaranteed performance of multiprocessor streaming applications*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Electrical Engineering]. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR637966

DOI:
10.6100/IR637966

Document status and date:
Published: 01/01/2008

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

An Accurate Analysis for Guaranteed Performance of Multiprocessor Streaming Applications

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van
de Rector Magnificus, prof.dr.ir. C.J. van Duijn,
voor een commissie aangewezen door het College
voor Promoties in het openbaar te verdedigen op
maandag 3 november 2008 om 16.00 uur

door

Petro Poplavko

geboren te Kiëv, Oekraïne

Dit proefschrift is goedgekeurd door de promotor:

prof.dr.ir. R.H.J.M. Otten

Copromotor:
dr.ir. T. Basten

This research work was carried out at and with the support of:
Philips/NXP Research Laboratories, Eindhoven.

Printed by: Universiteitsdrukkerij Technische Universiteit Eindhoven
Cover design by: Oranje Vormgevers, Eindhoven
ISBN: 978-90-386-1418-2

'If you would be a real seeker after truth, it is necessary that at least once in your life you doubt, as far as possible, all things.'
(René Descartes, a famous French mathematitian of the XVII-th century)

To the memory of my dear mother
Larisa Pereverzeva (1942-2005),
who is my model of being persistent in work,
and who was the best mother in the world and a talented scientist.

Abstract

An Accurate Analysis for Guaranteed Performance of Multiprocessor Streaming Applications

Already for more than a decade, consumer electronic devices have been available for entertainment, educational, or telecommunication tasks based on *multimedia streaming applications*, i.e., applications that process streams of audio and video samples in digital form. Multimedia capabilities are expected to become more and more commonplace in portable devices. This leads to challenges with respect to cost efficiency and quality. This thesis contributes models and analysis techniques for improving the cost efficiency, and therefore also the quality, of multimedia devices.

Portable consumer electronic devices should feature flexible functionality on the one hand and low power consumption on the other hand. Those two requirements are conflicting. Therefore, we focus on a class of hardware that represents a good trade-off between those two requirements, namely on domain-specific *multiprocessor systems-on-chip (MP-SoC)*. Our research work contributes to *dynamic (i.e., run-time) optimization* of MP-SoC system metrics. The central question in this area is how to ensure that real-time constraints are satisfied and the metric of interest such as perceived multimedia quality or power consumption is optimized. In these cases, we speak of quality-of-service (QoS) and power management, respectively.

In this thesis, we pursue real-time constraint satisfaction that is *guaranteed* by the system by construction and proven mainly based on analytical reasoning. That approach is often taken in real-time systems to ensure reliable performance. Therefore the performance analysis has to be *conservative*, i.e. it has to use pessimistic assumptions on the unknown conditions that can negatively influence the system performance. We adopt this hypothesis as the foundation of this work. Therefore, the subject of this thesis is the *analysis of guaranteed performance* for multimedia applications running on multiprocessors.

It is very important to note that our conservative approach is essentially different from considering only the worst-case state of the system. Unlike the worst-case approach, our approach is *dynamic*, i.e. it makes use of run-time characteristics of the input data and the environment of the application.

The main purpose of our performance analysis method is to guide the run-time optimization. Typically, a resource or quality manager predicts the *execution time*, i.e., the time it takes the system to process a certain number of input data samples. When the execution times get smaller, due to dependency of the execution time on the input data, the manager can switch the control parameter for the metric of interest such that the metric improves but the system gets slower. For power optimization, that means switching to a low-power mode. If execution times grow, the manager can set parameters so that the system gets faster. For QoS management, for example, the application can be switched to a different quality mode with some degradation in perceived quality. The real-time constraints are then never violated and the metrics of interest are kept as good as possible.

Unfortunately, maintaining system metrics such as power and quality at the optimal level contradicts with our main requirement, i.e., providing performance guarantees, because for this one has to give up some quality or power consumption. Therefore, the performance analysis approach developed in this thesis is not only conservative, but also *accurate*, so that the optimization of the metric of interest does not suffer too much from conservativity. This is not trivial to realize when two factors are combined: parallel execution on multiple processors and dynamic variation of the data-dependent execution delays. We achieve the goal of conservative and accurate performance estimation for an important class of multiprocessor platforms and multimedia applications. Our performance analysis technique is realizable in practice in QoS or power management setups.

We consider a generic MP-SoC platform that runs a dynamic set of applications, each application possibly using multiple processors. We assume that the applications are independent, although it is possible to relax this requirement in the future. To support real-time constraints, we

require that the platform can provide guaranteed computation, communication and memory budgets for applications. Following important trends in system-on-chip communication, we support both global buses and *networks-on-chip*.

We represent every application as a homogeneous synchronous dataflow (HSDF) graph, where the application tasks are modeled as graph nodes, called actors. We allow dynamic data-dependent actor execution delays, which makes HSDF graphs very useful to express modern streaming applications. Our reason to consider HSDF graphs is that they provide a good basic foundation for analytical performance estimation.

In this setup, this thesis provides three major contributions:

1. Given an application mapped to an MP-SoC platform, given the performance guarantees for the individual computation units (the processors) and the communication unit (the network-on-chip), and given constant actor execution delays, we derive the throughput and the execution time of the system as a whole.
2. Given a mapped application and platform performance guarantees as in the previous item, we extend our approach for constant actor execution delays to dynamic data-dependent actor delays.
3. We propose a global implementation trajectory that starts from the application specification and goes through design-time and run-time phases. It uses an extension of the HSDF model of computation to reflect the design decisions made along the trajectory. We present our model and trajectory not only to put the first two contributions into the right context, but also to present our vision on different parts of the trajectory, to make a complete and consistent story.

Our first contribution uses the idea of so-called *IPC* (inter-processor communication) *graphs* known from the literature, whereby a single model of computation (i.e., HSDF graphs) are used to model not only the computation units, but also the communication unit (the global bus or the network-on-chip) and the FIFO (first-in-first-out) buffers that form a 'glue' between the computation and communication units. We were the first to propose HSDF graph structures for modeling bounded FIFO buffers and guaranteed throughput network connections for the network-on-chip communication in MP-SoCs. As a result, our HSDF models enable the formalization of the on-chip FIFO buffer capacity minimization problem under a throughput constraint as a graph-theoretic problem. Using HSDF graphs to formalize that problem helps to find the performance bottlenecks in a given solution to this problem and to improve this solution. To demonstrate this, we use the JPEG decoder application case study. Also, we show that, assuming constant – worst-case for the given JPEG image – actor delays, we can predict execution times of JPEG decoding on two processors with an accuracy of 21%.

Our second contribution is based on an extension of the *scenario approach*. This approach is based on the observation that the dynamic behavior of an application is typically composed of a limited number of sub-behaviors, i.e., scenarios, that have similar resource requirements, i.e., similar actor execution delays in the context of this thesis. The previous work on scenarios treats only single-processor applications or multiprocessor applications that do not exploit all the flexibility of the HSDF model of computation. We develop new scenario-based techniques in the context of HSDF graphs, to derive the timing overlap between different scenarios, which is very important to achieve good accuracy for general HSDF graphs executing on multiprocessors. We exploit this idea in an application case study – the MPEG-4 arbitrarily-shaped video decoder, and demonstrate execution time prediction with an average accuracy of 11%. To the best of our knowledge, for the given setup, no other existing performance technique can provide a comparable accuracy and at the same time performance guarantees.

CONTENTS

1

1 Introduction

This thesis concerns with the design of digital systems embedded in consumer electronics products, e.g. mobile phones, pocket computers, etc. It focuses on *multimedia embedded systems*, i.e., the tiny computer systems that are built into those devices and that perform various video and audio processing tasks.

The design objective is to create an embedded system that has low cost and low power consumption. The increasing hardware design effort in the deep-submicron VLSI (very large scale integration) technologies as well as the costs of masks dictate the requirement that the existing designs be reused as much as possible. This can be achieved using a platform, i.e., an available hardware design that can be programmed for the required functionality. Then the system is implemented just by programming the platform.

One important issue here is the fact that the programming should ensure that the embedded system meets its *real-time constraints* specifying the timing properties expected by the user. When the programming is done in a traditional timing-unaware way and the platform is chosen using intuitive rule-of-thumb methods, most likely the design will not satisfy the constraints at all or it will be characterized by unreliable timing behavior or it will be too power-hungry. Taking the timing behavior into account as an afterthought and trying to adjust it for real-time constraint satisfaction may result in multiple design iterations, involving laborious re-design of the software. Timing is currently one of the most limiting factors in the software code generation for embedded systems, as pointed out by Edward Lee in [51].

We consider it important to make the programming easier by using an implementation trajectory that is oriented towards real-time constraint satisfaction and automates the necessary

steps to reach that goal. In the ideal case, the system should become timing-correct by construction. In reality, one can expect a significant reduction in the number of design iterations. The performance analysis formalism proposed in this thesis provides unambiguous guidelines for creating an automated timing-aware implementation trajectory, which contributes to easier programming.

Another important issue, especially for portable devices, is to ensure low power consumption. Therefore, the platforms should include sub-circuits tuned for a specific class of computations that are characteristic to a limited but still reasonably large subset of applications, called an *application domain.* We choose for such domain-specific platforms and reduce our scope to the *multimedia streaming* application domain, covering various video and audio processing applications.

What kind of platform to choose? We target our studies to the *multiprocessor systems-on-chip – MP-SoC*, i.e., platforms having multiple processor cores on a single chip. We motivate this choice later in this chapter.

Unfortunately it is not easy to exploit the multiprocessor parallelism, especially when imposing the real-time constraints. This difficulty has to be addressed by a design methodology, having three main ingredients, namely, application domain analysis, appropriate platform architecture design and the mapping of the applications to the platform.

However, in addition to parallelism, another factor that complicates the embedded multimedia system design is the dynamic data-dependent execution delays of the application tasks, which can be treated efficiently by *adaptation*, i.e., the dynamic adjustment of the controllable system settings to the current computation workload. One contribution of this thesis is an analytical framework for on-the-fly performance evaluation of the running system. A tough problem that we address is predicting the throughput of an application that is mapped to several processors under the conditions of variable computation workload. In addition to that, our work also contributes to the design methodology, in terms of support for on-chip communication channels implemented using networks-on-chip (NoC), which is an important new trend for MP-SoCs.

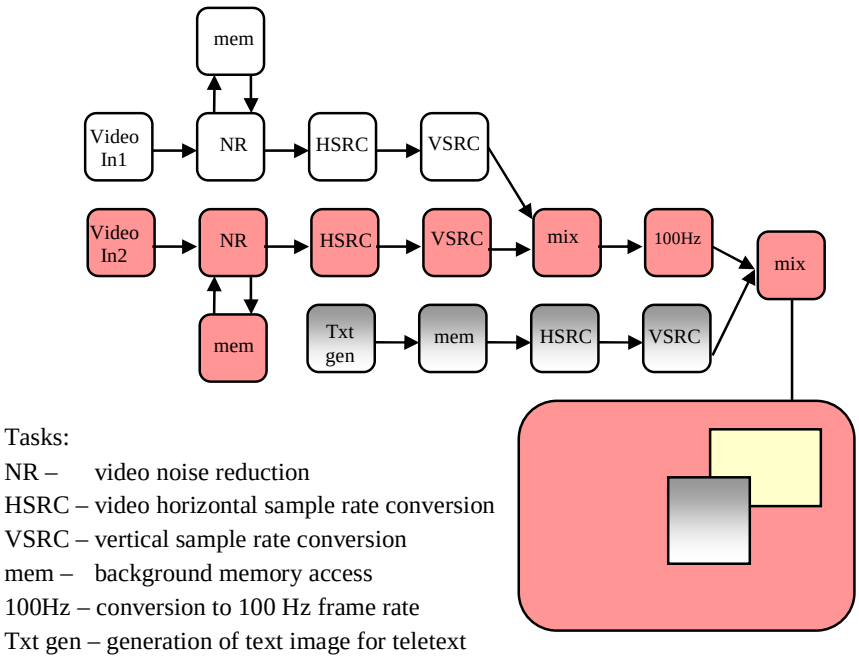
Because we bind our performance analysis approach to a certain design methodology, we describe this methodology in the first part of this chapter. Sections 1.1, 1.2, and 1.3 study the three major ingredients of a design methodology one-by-one – namely, the application-domain analysis, the multiprocessor platform architecture and the mapping of the applications to the multiprocessor.

In the rest of the chapter, we zoom into the core problems addressed by this thesis – i.e. dealing with the dynamic timing behavior of streaming applications running in MP-SoCs. Also, we analyze related work and summarize our contributions and the structure of the thesis.

1.1 Application Domain Analysis

1.1.1 Run-time Combinations of Applications

An important trend in modern consumer electronic media systems is that they are becoming more interactive, providing user interfaces with the possibility to open, rearrange and close different video/audio presentations, telecommunication sessions, etc. Interactive systems are characterized by multiple possible combinations of such activities, also known as *use cases*.



Tasks:
NR – video noise reduction
HSRC – video horizontal sample rate conversion
VSRC – vertical sample rate conversion
mem – background memory access
100Hz – conversion to 100 Hz frame rate
Txt gen – generation of text image for teletext

Figure 1.1 A use case in a multi-window TV system

For example, Figure 1.1 describes an interactive television system where the user can open multiple windows with video or teletext. The particular use case shown in the figure combines two video windows and one teletext window. The diagram of the use case can be split into three parts corresponding to each window, and we say that three corresponding *applications* are currently active in the system. Each video application continually executes a chain of tasks processing video data streams. The teletext application executes another chain of tasks. Due to the user actions like opening and closing the windows or due to the environment, the number of applications, their structure as shown in the diagram, and the basic settings (e.g. resolution or color depth) may change at run time. This corresponds to switching between different use cases. For more use-case examples, see e.g. [67].

In general, we associate an *application* with an activity that is started and stopped at run time by events originating from user actions or the environment. A media streaming application can be split into a few tasks and represented by a *task graph*, modeling the communication between the tasks. Task graphs of different applications are combined together to form one use case. In Figure 1.1, the graphs of different applications are highlighted using different color combinations.

Different applications may belong to different types, e.g., the video sample rate conversion application and teletext application, and for each interactive system one can make a list of different types of applications that can be involved in the system. Also some applications can be in different modes that can be switched due to the user actions, e.g., a video window may have color depth settings, and when it is in front of other windows it may be switched to high-quality mode.

Which particular combinations of applications of different types and in different modes will be activated at run time is not predictable at design time, and the number of possible

combinations can be exponential in the number of types and modes. In practice, the number of use cases of interactive systems can reach a few tens and even a few thousands.

1.1.2 Synchronization, Pre-scheduling and Shared Memory

For the task graphs of multimedia streaming applications, we make one important assumption on the way the tasks communicate with each other. We assume that the data are exchanged using a set of point-to-point channels where the data is communicated in one direction, *first-in-first-out* (FIFO). The channel examples are shown as arrows in Figure 1.1.

Restricting ourselves to FIFO communication is an important choice. Let us make a short overview, to position this choice in a more general context. The most general way to represent inter-task communication is a *shared memory* model. Unlike FIFO, it allows any order of writes and reads of the communicated data. Two major methods to ensure correct order of reads and writes are synchronization and pre-scheduling. Every design methodology uses a certain combination of those two methods.

Synchronization means that, prior to read/write, a task checks for certain conditions set by other tasks. To ensure real-time constraint satisfaction, this method requires performance analysis, e.g. the one proposed in this thesis. *Pre-scheduling* means putting restrictions on the order in which different concurrent tasks are executed, which can go as far as creating a detailed schedule with prescribed starting times for every task execution. The pre-scheduling makes it easier to analyze the timing properties of the system and thus reduces the need for performance analysis.

For the multimedia streaming applications implemented on programmable processors, it is crucial to use synchronization, for efficiency reasons. A major reason why we chose FIFO communication is that it is an efficient way for task synchronization. Another major reason is that FIFO is a wide-spread communication method for the multimedia streaming application domain. For the topic of this thesis, it is important to note that the assumption that the communicated data is organized in queues (i.e. FIFO memories) is a typical prerequisite for applying most known performance analysis formalisms for parallel computer systems. For example, this assumption is necessary for all formalisms we discuss later on in the related work section.

In practice, FIFO communication is not the only possible communication scheme in the multimedia streaming application domain. The order in which the data is read and written can follow a different pattern, e.g. a matrix can be first written row-by-row and then read column-by column. Worse still, the order can be unpredictable, e.g. the video movie players typically use so-called motion compensation, which can read data in many various orders, depending on the direction in which video objects move in the given movie. Therefore, certain task graph models for embedded systems support such forms of communication; for an overview and generalization see e.g. the book by F. Thoen and F. Catthoor [95].

Nevertheless, for the reasons mentioned earlier, we still insist on restricting ourselves to the task graphs with FIFO communication. We assume that the other forms of communication are handled by pre-scheduling and can be avoided in the task graph without loss of generality, by abstraction. For example, if some tasks use a fixed non-FIFO communication pattern, then one construct a fixed schedule for them and encapsulate them into one task, whose delay is equal to the length of the schedule. Note that this means that our methodology may suffer from some loss

of efficiency for the applications that extensively use shared memory communication with non-FIFO communication patterns.

Encapsulation of shared memory accesses into special tasks, dedicated for that purpose, is a universal abstraction to model shared memory communication in a task graph. Such tasks would represent the tasks executed by memory controllers, accessing shared memory on behalf of other tasks. For example, the task graph in Figure 1.1 has two such tasks, denoted as 'mem'. In general, more elaborate modeling of shared memory is possible, using special task subgraphs, as proposed e.g. by Sander Stuijk in [85].

1.1.3 Real-time Constraints

Real-time constraints are imposed on most streaming applications, which can be described in terms of throughput and latency. The *throughput* is the rate of consuming the data at the input and of producing the data items at the output. The *latency* is the time interval between consumption at the input and production at the output.

One can classify applications by their real-time constraints.

Hard real-time (HRT) applications must always maintain certain throughput and/or latency. Usually, only safety-critical applications are considered as such, but in our definition this class also includes certain entertainment applications with high quality expectations – where the user would not tolerate even smallest visible or audible 'artifacts' in the multimedia contents. Examples are high-definition television and home theater.[1]

Soft real-time (SRT) applications may sometimes fail to maintain the required throughput/latency, but they try to keep the effect of their misbehavior limited. This keeps the user still satisfied with the results. An example is capturing and displaying simple videos in a digital photo camera.

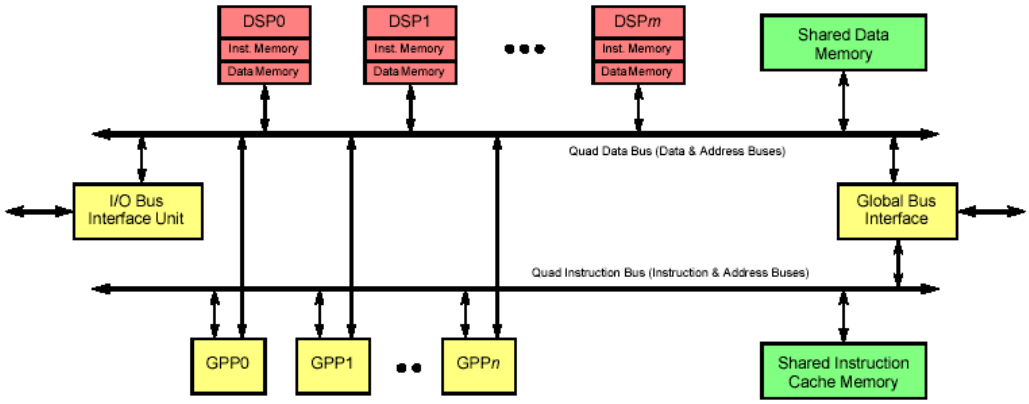
Best-effort applications do not guarantee any concrete throughput/latency, but they try to be as fast as possible, so that the user feels comfortable with them. An example is downloading a photo album from Internet. In fact, best-effort applications are not real-time.

In this thesis, under 'applications' we will usually understand soft or hard real-time applications. We also make another important assumption on the real-time constraints. We assume that the input and output data are organized in coarse-grain data chunks, usually referred to as frames, consisting of fine-grain samples, called *tokens*. An *execution run* of the application task graph should consume one input data frame and produce one output frame. We assume that the timing constraints are specified in terms of deadlines on the production of the output frames.

Under this assumption, the throughput is defined as the rate of processing the tokens and the latency is equal to the timing length of the execution run, i.e. to the total time required to consume the input frame plus some propagation delay. Thus, the latency is approximately equal to the frame size divided by throughput. Therefore, under our assumptions, the latency is directly related to throughput. Throughout this thesis, we use term *execution time* for latency, and see the problem of execution time calculation as equivalent to the throughput calculation.

We must admit that our assumption can be harmful for the latency-critical applications that do not cluster the data tokens into frames, e.g. some audio applications. An important way to alleviate this assumption would be to support the input and output task graph ports characterized

[1] Sometimes, applications for which an occasional violation of timing requirements is highly undesirable but not catastrophic are referred to as firm real-time.



GPP – general-purpose processor
DSP – processor specialized for digital signal processing
I/O – chip input/output

Figure 1.2 Cradle Technologies multiprocessor DSP: 'Quad' architecture

by periodic patterns of data token consumption and production. We believe that our analysis approach can be extended to support periodic patterns at the input/output ports, and it is an important subject for future work.

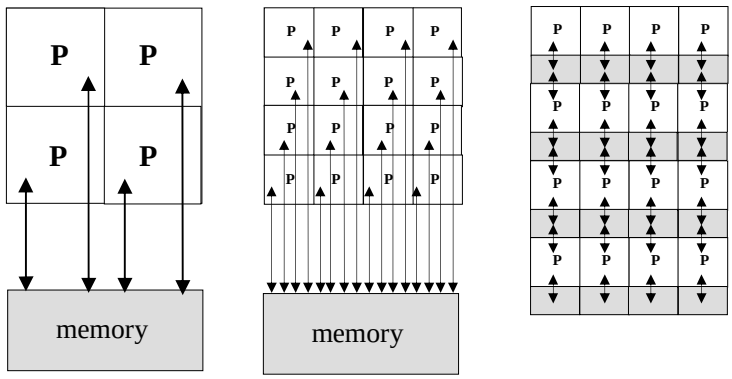
1.2 Platform Architecture

1.2.1 Platform: Domain-specific MP-SoC

The VLSI technology development is the driving force behind integrating more and more functionality in the new product generations of consumer electronic devices. The technology already allows putting multiple processor cores on a single die, organizing them as a multiprocessor system-on-chip (MP-SoC). One of the first MP-SoC architectures studied in the literature is MIT RAW [92]. Examples available on the market are platforms like Cradle Technologies Quad [17], illustrated in Figure 1.2, NXP's Nexperia™ [70], and many others. Recently, Intel demonstrated a chip containing 80 processor cores [96] in 65 nm technology.

In 130 nm technology, a MIPS R3000 processor with caches occupies around 3 mm^2, and one can estimate that, in the year 2012 (with 45 nm technology), the same processing core will shrink to less than 0.5 mm^2. With a chip area size of 100 mm^2, this will allow over 200 MIPS cores placed on a single die. However, not the area but rather the power consumption will be the limiting factor for such an integration. One can extrapolate the dynamic power consumption of the MIPS core in 2012 to be around 25 mW. With a limitation of 1W for a single chip, this would reduce the number of cores from the 200 cores mentioned above to only 40 cores. Worse still, in addition to the dynamic power, the static (leakage) power will probably limit this number even further. Note that the abovementioned 80-core Intel chip is reported to consume 98 W [96], which is a power consumption that is affordable for an experimental general-purpose computer chip but not for an embedded system.

Therefore, domain-specific MP-SoCs employ not only general-purpose processors like MIPS, but also application-domain specific processors, specialized for a limited subset of functions. As mentioned before, specialization leads to a considerable decrease in the power consumption, and



(a) 'old' technology **(b)** centralized memory **(c)** distributed memory

Feature size:	λ	$\lambda/2$	$\lambda/2$
$\frac{\text{energy}}{\text{cycle}}$	8	40	1

Figure 1.3 Spatial distribution of memory (source: [58]).

it can be very efficiently exploited in a multiprocessor environment, where one can forward each function to the processor that is specialized for it, especially if the platform is properly aligned to the application domain.

An MP-SoC platform architecture includes not only processors, but also memories and interconnection infrastructure, which consists of, for example, buses and bus bridges, as one can also see in the Quad architecture in Figure 1.2. In the rest of this section, we consider the basic memory and interconnect properties, and then we describe the platform's programming model, which characterizes the platform as a whole.

1.2.2 Memory: from Centralized to Distributed

To benefit from the increasing number of on-chip processors, the overall architecture should be decentralized. Only then the power consumption stays manageable and the performance scales up as new processors are added on the same chip.

We illustrate this in Figure 1.3(a). This example is borrowed from a presentation of Hugo De Man [58]. It depicts the topology of a platform architecture with four processors and a large memory located on chip. Assume that this picture relates to an old VLSI technology and the energy consumption per cycle constitutes 8 energy units. Assume also that when we step to the current technology the chip area allows us to increase the number of processors by a factor of 4, see Figure 1.3(b). One would expect that the energy consumption would be multiplied by a factor much less than 4, because as the VLSI technology advances the dynamic energy per processor decreases. However, we see that instead the energy has increased by a factor of 5 [58].

The reason for this is as follows. The accesses to large memories contribute considerably to the overall energy consumption. This energy cost quickly grows even further if one adds memory ports for independent accesses, which is done in this example to avoid memory conflicts between processors and thus to guarantee performance scalability.

To reduce the energy and keep the performance scalability, it is required to split the global on-chip memory into smaller *local memories* accessed by only a very limited number of processors, e.g., two adjacent elements as shown in Figure 1.3(c). The figure mentions a tremendous decrease in the energy consumption estimated as 1 energy unit per cycle [58].

In the remainder, we assume each processor has a local memory for its instructions and data, like the DSP processors in Figure 1.2.

1.2.3 Interconnect: Network-on-chip

For fast communication across the chip, systems-on-chip employ a global interconnect. Quite often this interconnect is a bus, connecting all the processors and memories together in a centralized way. However, a single bus is not scalable in the number of elements, because processors compete with each other and have to wait for their turn. When the number of processors increases, the waiting time also increases, and so does the energy consumption due to the increase of capacitive load of the bus.

Due to those problems, it is widely recognized that using only a single bus for communication is not appropriate for high-performance media platforms. Therefore, also for the global interconnect one should rather go for distributed topologies, e.g. multiple bus segments joined by bus-to-bus bridges. For example, in the Quad architecture, Figure 1.2, we see a two-bus computer architecture that can be connected to a global bus, which, in turn, can connect the given Quad to other Quads.

From this point on, we use the name *network-on-chip* – or *NoC* – referring to any interconnection network with a distributed topology.

Note that the choice we made – in Section 1.1.2 – of focusing on only FIFO form of communication and abstracting from other forms of communication offered by the shared memory model, also impacts our abstraction of the network-on-chip. Throughout this thesis, we see the on-chip network simply as a homogeneous medium being used to setup peer-to-peer FIFO channels between two processors. This makes the network topology irrelevant for us, whereas network topologies can be exploited to efficiently implement shared memory hierarchies and efficient communication between different processors in that context. Efficient organization of memory hierarchies is important for embedded system design [13]. Explicit support of memory hierarchies is a subject for future work.

1.2.4 Programming Model: Reconfigurable Streaming

A programming model describes how the programmer sees the platform. We refer to our programming model as reconfigurable streaming (RS).

The RS model has two levels. At the first level, we introduce the platform *configuration*, implementing a single use case of the system. The second level is responsible for switching from one use case to another as applications are started and stopped at run time. This level is called *reconfiguration*.

Let us first consider the configuration level, which considers a single use case, characterized by a concrete combination of running applications. The active system functionality at this level stays unchanged.

A configuration consists of

1) distribution of the use-case functionality between the processors,

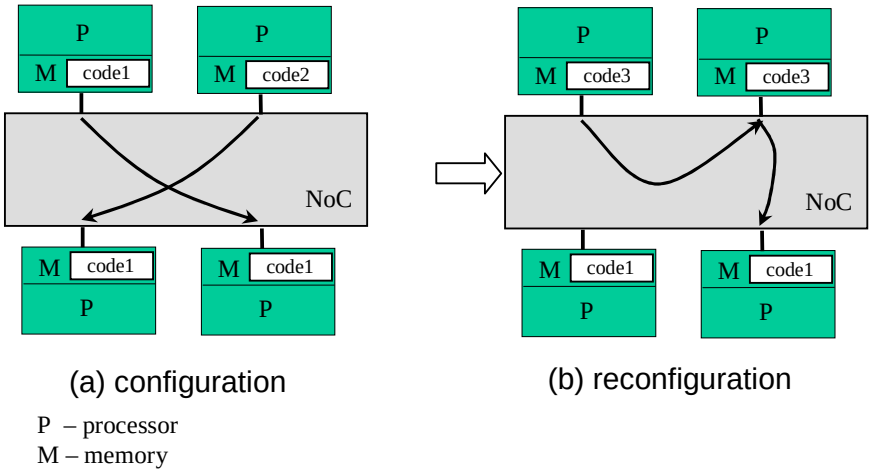


Figure 1.4 Reconfigurable streaming

2) organization of communication between the processors using a set of peer-to-peer channels going through the network-on-chip.

Figure 1.4(a) shows an example of a configuration. We see that different program codes are distributed between different processors and that the communication channels are set-up between different sources and destinations.

Reconfiguration involves setting up/tearing down the channels and reprogramming the processors. For example, Figure 1.4(b) shows a switch from the configuration in Figure 1.4(a) to another configuration, whereby some processors get programming codes that are different from the previous configuration and the communication channels are changed. In a distributed platform, a reconfiguration can take a considerable time, and then it should be done only occasionally. In our chosen application domain, it happens, first of all, when the system switches from one use case to another one, e.g., when a new application starts or when an active application adapts to the changing user requirements.

The scheme of operation of the platform can be split into three major phases repeating after some time intervals:

1. deciding upon a new configuration, or *mapping*,
2. (re-)configuring,
3. static streaming until the next special event from the user or an application.

For practical examples of reconfigurable streaming and possible implementation strategies see e.g. [67] and [35].

In this thesis, we do not model reconfiguration. We assume that the reconfiguration is relatively rare, and that it does not happen during the time intervals where critical loops of applications are active. However, because reconfiguration is a very important tool for efficient use of the hardware resources, such a topic as implementation and modeling of reconfiguration is an important subject for future work.

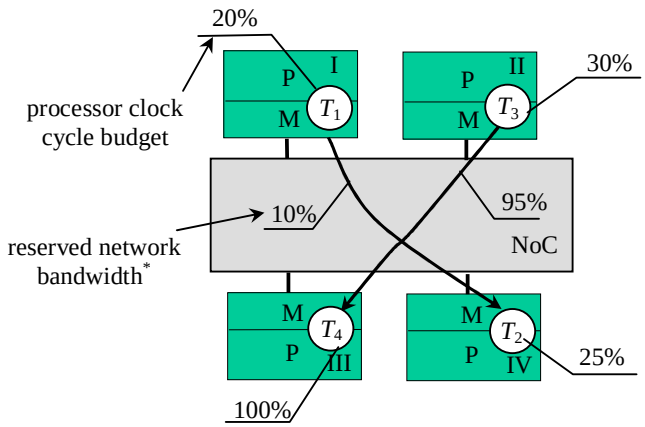


Figure 1.5 Mapping of tasks to processors

*note that the bandwidth is reserved at a certain network *path*; therefore, if two channel paths do not share any components, the sum of the reserved bandwidth can be more than 100%

1.3 Mapping and Timing Verification

1.3.1 Mapping Problem

Phase 1 in the platform operation scheme defined above – deciding upon a configuration – is a combinatorial problem. This problem is referred to as the *mapping problem.* It considers a system use case as a collection of task graphs of the active applications, e.g., such as in Figure 1.1. Given a use case and the platform, the mapping involves allocation of processors to the use case and assignment of tasks to the processors and specification of the set of channels for communication between tasks. Figure 1.5 shows an example, where four tasks are assigned to four processors and two channels serve for communication between the tasks.

The resulting configuration should satisfy the real-time constraints of HRT (hard real-time) applications, and, up to some level of certainty, of SRT (soft real-time) applications as well. Therefore, it is the ultimate goal of mapping to implement the applications such that it is possible to verify not only their functionality, but also the real-time constraint satisfaction. The latter makes the mapping problem complex.

To solve it, the design methodology should offer algorithms for timing-constrained mapping of applications to the platform. The mapping problem with real-time constraints is at least as complex as checking whether a given mapping solution satisfies those constraints. The latter is referred to as *timing verification*.

Note that, in a perfect design methodology, the timing verification should not be necessary in any foreseen situation (where for HRT applications any possible situation should be foreseen) because timing constraints in such a perfect methodology should be satisfied by construction. However, in any case, timing verification should be possible.

1.3.2 Reservation-based Approach

A major challenge for timing verification is resource sharing. The basic resources of the platform are the processors, memories and the network-on-chip. Tasks may share the same processor. The channels share the network-on-chip, like the two channels in Figure 1.4(a).

Each resource has a limited capacity. A processor can perform only a limited number of operations per second. Memories have limited size. The network components have limited bandwidth. Each application utilizes the resource capacities to a certain extent. If two applications share the capacity of a processor or a network component, the applications may delay each other, especially if the aggregated utilization of the resource in question is close to 100%. For example, in Figure 1.5, tasks T_1 and T_2 may belong to one application that shares some network resources with the application with tasks T_3 and T_4. These applications may delay each other even though those applications are functionally unrelated.

The system designer cannot avoid resource sharing and high collective utilization of the resources, because to produce a competitive product, it is required to get the highest performance out of the available hardware. Thus, if no measures are taken in the platform, non-functional timing dependencies between applications will be common. As a consequence, every running application will be dependent on all other running applications, and the combination of all running applications will have to be subject to timing verification. Timing verification under the condition of processor resource sharing is usually referred to as schedulability analysis[2] [80].

Unfortunately, it is problematic to use a schedulability analysis method in our application domain. Because of multiple functional and non-functional dependencies between the tasks, it is only feasible to perform the analysis at design time. Therefore, to support any possible use case of the designed system, one should analyze all possible run-time combinations of applications in a use case, but we have already pointed out that the number of combinations grows exponentially as the systems get upgraded with new functionality.

To avoid this difficulty, we rather choose for the *reservation-based approach*. The main idea is to reserve part of the capacity of the resources – called a *budget* – for each application at run time. Budgets are reserved in terms of capacity of the processors, network-on-chip and memories. Under these conditions, each resource behaves towards the application as an independent resource, as if there was no resource sharing. This way, one can perform the timing verification of each application independently of other applications.

Therefore, we speak of *timing composability*, meaning that relevant performance metrics of each application are invariant in any composition of the given application with other applications. The concept of timing composability is well-known in real-time systems and is explained, among other, in the work of Hermann Kopetz [47]. Timing composability drastically simplifies the complexity of timing verification, because one considers different applications separately, and not in combination with others. As we see in the next section, it also simplifies the mapping problem.

Of course, these benefits of our approach come at a certain price. Timing composability is an implementation restriction that may lead to loss of efficiency, especially for very dynamic applications [48]. An alternative to timing composability is schedulability analysis, carried out at run time. Although advanced schedulability analysis techniques for MP-SoC systems are proposed by K. Richter *et al* in [80], they are not directly suitable for being used at run time. The assessment of possible run-time schedulability analysis techniques is a subject for future work. The work of Akash Kumar *et al* [48] is an interesting example of work in that direction.

Note that a potential threat for the reservation-based approach is brought about by run-time variations in operating conditions (e.g. temperature and supply voltage). Those variations may

[2] For the single-processor case, the most famous example of schedulability analysis is rate-monotonic analysis [54]

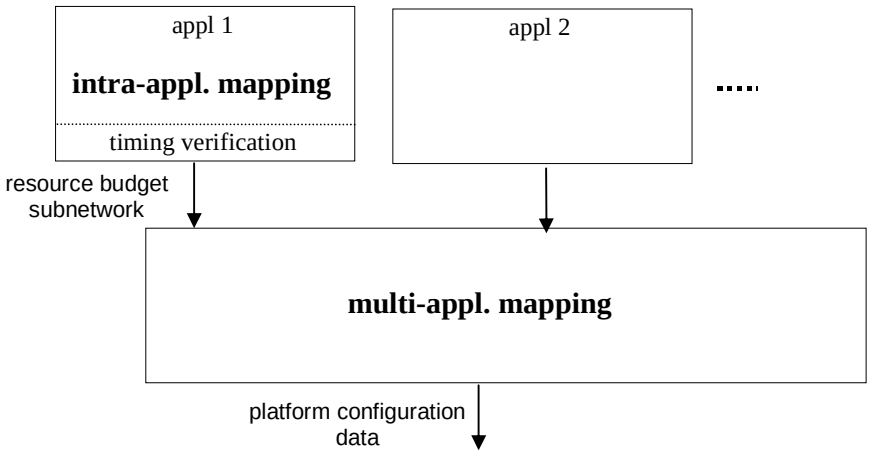


Figure 1.6 Two-stage mapping

require the operating frequency of the processors to be adjusted at run-time. Nevertheless, we ignore this problem without loss of generality. The point is that we mainly focus on the performance analysis carried out at run time, i.e. at the moment when the operating frequencies are known and can be taken into account in performance calculations immediately. Handling this problem in a broader scope – e.g. in mapping and in platform design – is outside the scope of this thesis and is a subject for future work.

1.3.3 Two-stage Mapping

As a result of the timing composability, the mapping problem can be naturally split into two stages: intra-application mapping and multi-application mapping, as illustrated in Figure 1.6.

At the *intra-application mapping* stage, for each application, budgets are reserved at different processor, memory and communication resources. For processors, this is done in terms of *processor cycle budgets* and for the network in terms of the *communication bandwidth*. For example, in the example in Figure 1.5, we reserve 20% of the clock cycles of processor I for task T_1, 25% of the clock cycles of processor IV for task T_2, and 10% of the maximum bandwidth for the channel from T_1 to T_2.

A task-graph diagram, like the one shown in Figure 1.5, consisting of tasks joined by channels, whereby each task and channel is annotated by a resource budget value, is called a *resource budget subnetwork*, meaning a logical part of the multiprocessor network on-chip that operates independently due to resource reservations. As shown in Figure 1.6, a resource budget subnetwork is generated by the intra-application mapping stage.

Note that, given all the resource reservations, a resource budget subnetwork is basically enough to reason about the application timing. Thus, the timing verification can be done already after the first mapping stage.

The second stage of mapping is *multi-application mapping*. For the applications that must run on the platform, this stage fits the resource budget subnetworks on the physical platform. The outcome of this stage is the low-level configuration data that can be loaded into the platform to set up a new configuration.

One disadvantage of the two-phase approach is that the intra-application mapping stage restricts the freedom of possible solutions that can be exploited by the multi-application mapping

stage. Another disadvantage is that it relies on the timing composability and thus can be inefficient for very dynamic applications.

At the same time, the two-stage approach has two important advantages.

First, for many applications, the system designer can perform the intra-application mapping and timing verification at design time. (The multi-application stage still has to be performed at run time.) This is possible because no run-time knowledge about the other applications running on the platform is required for that purpose.

Second, if an application of the same type is represented in the run-time combination multiple times, one can reuse the given resource budget subnetwork for every application instance.

1.4 Towards Run-time Performance Analysis

In the previous sections, we provided a general context for this thesis by sketching the general design methodology framework. Now, within this context, we are turning our attention to the main topic of this thesis, namely, the run-time performance analysis. We start the discussion of this topic by introducing the main challenge addressed using the run-time performance analysis – i.e., coping with dynamic resource utilization.

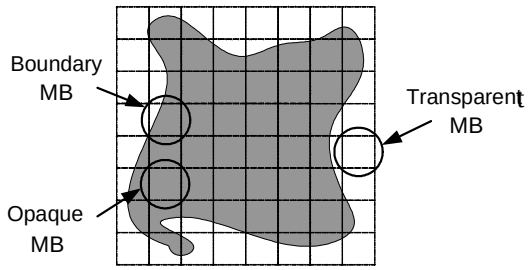
1.4.1 Sources of Dynamic Resource Utilization

In general, a streaming system can be characterized by dynamically changing levels of required utilization of the resources. The problem that arises from this fact is to ensure that the required utilization of any resource by any application does not go above the application's resource budget, because otherwise the real-time constraints will be violated.

We refer to the dynamic variation of the resource utilization as *dynamism*. One can distinguish two sources of dynamism:

1. starting new applications, stopping the active applications and adapting the active applications to the changing user requirements or environment;
2. input data dependency of the processing times.

The first source of dynamism has to do with switching between system use cases. This form of dynamism is dealt with at the multi-application level. The second source of dynamism refers to the data dependency of the processing times of tasks *inside* the applications. We see this phenomenon, first of all, in the applications that involve data compression, like MP3 audio and MPEG-4 video, because they need to process different numbers of input data bits within the time intervals of the same length. This form of dynamism is dealt with, for as much as possible, at the intra-application level, but if necessary the multi-application level is also involved.



MB – 'macroblock' – a 16x16 matrix of pixels (picture elements = dots)

Figure 1.7: An arbitrarily-shaped video object in MPEG-4 decoding

Let us give a slightly more detailed example of the second source of dynamism. The MPEG-4 standard supports arbitrarily shaped video objects on the video screen. As shown in the example in Figure 1.7, a video object is a variable-sized matrix of so-called macroblocks (MBs). Every macroblock is a fixed-size (16x16) matrix of pixels (i.e., dot elements of the picture). As it is illustrated in Figure 1.7, the macroblocks can be divided into three different types, namely opaque blocks that are fully contained in the object, transparent blocks that are fully outside the object, and boundary blocks. Because the object's shape and size, encoded in the input data stream, may change quickly, the number of processor cycles needed for processing blocks over time may change as well. However, the real-time constraints typically require the object to be refreshed at a constant rate. Thus, within regular periods of time different numbers of processing cycles needs to be spent on data processing and the resource utilization changes.

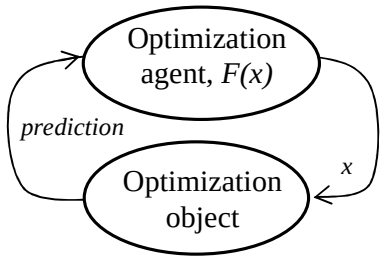
1.4.2 Three Degrees of Freedom to Cope with Dynamism

In the presence of dynamism, classical mapping, meaning an optimized binding of fixed functionality to fixed resources, is not sufficient. To ensure meeting the real-time constraints under the conditions of dynamic workload, one can exploit several degrees of freedom. Three most important of them correspond to the three ingredients of the design methodology:

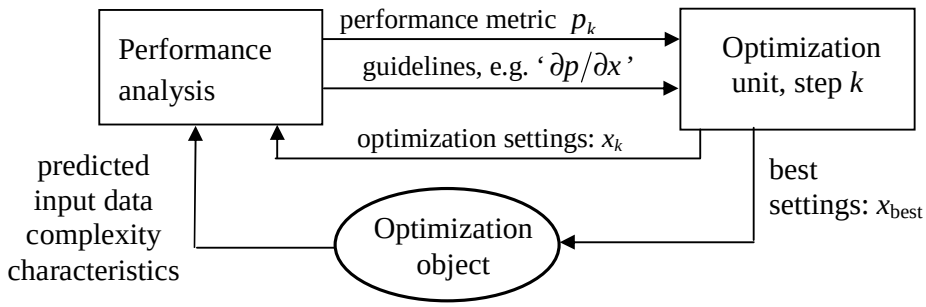
1. For the applications, the freedom is to scale the visual/audio quality up and down, referred to as quality-of-service (QoS) management.
2. For the platform, it is to scale the speed (and therefore the energy usage) of the resources up and down, called dynamic voltage/clock frequency scaling.
3. For the mapping, it is the redistribution of the resource budgets between different applications, often referred to as renegotiation.

The choice of the degrees of freedom[3] to be used depends on the possibilities offered by the application algorithm and/or the platform.

[3] An example of the other degrees of freedom is switching between different algorithms implementing the same functionality, allowing to trade off speed for memory



(a) To optimize property *F(x)* of a useful object under the predicted circumstances, an optimization agent is introduced that sets the vector of control parameters, *x*



(b) Optimization agent, expanded for the case when the predicted circumstances are input data complexity characteristics

Figure 1.8 Optimization of *F(x)* with constraints on data-dependent performance

1.4.3 Adaptation Framework to Handle Data Dependency

The second source of dynamism, the data dependency of processing times, is potentially responsible for much more frequent changes in the resource utilization than the first source of dynamism, concerning the starting and stopping of applications. Therefore, when possible, it should be handled at the intra-application level to avoid frequent reconfiguration of the system.

The applications, the mapping and/or the platform need to be enhanced with the ability to adjust themselves to the input data characteristics representing the resource requirements of processing. We refer to the run-time activity that adjusts the application/implementation parameters to the workload variations as *adaptation.*

The adaptation can be seen as solving an optimization problem with constraints on performance. Figure 1.8 shows a typical example of a framework that implements such optimization. The figure introduces an optimization agent that can exploit the available degrees of freedom – i.e. quality, speed/energy and resource budgets – to adjust the settings of the optimization object – typically, an application. This should be done such that the real-time constraints, i.e., constraints on performance, are met and the optimization objective is reached – e.g., high quality, low energy and low requested resource budget. In Figure 1.8(a), the objective is denoted as *F(x)*, where *x* is a vector of control parameters of the optimization object. As Figure 1.8(a) shows, the optimization agent requires a *prediction* of the circumstances under which the optimization object is going to operate. Only then it can take a proper decision and set

x in the best way. Figure 1.8(b) provides details on how the decision is taken by expanding the internal contents of the optimization agent.

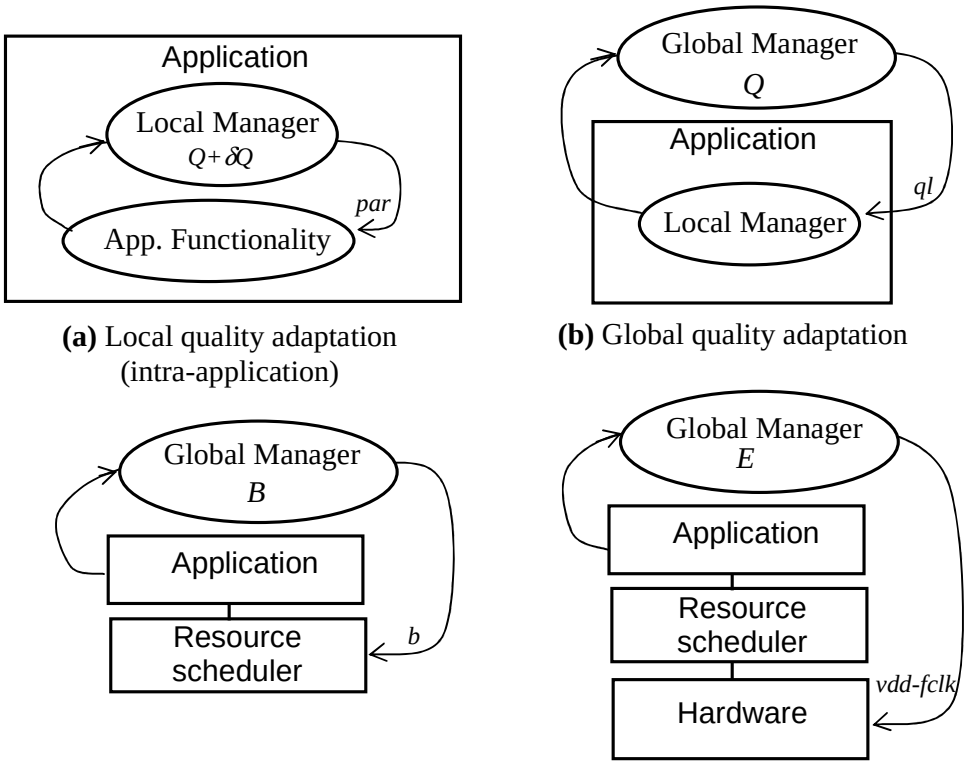
An optimization agent consists of an *optimization unit* that generates candidate solutions and a *performance analysis unit*, responsible for evaluation of those solutions. To illustrate the role of the performance analysis, we use an analogy with airplane control. One can estimate the future position of an airplane after time t given such input characteristics as current location r, speed v and acceleration a. The future position can be approximated by applying integration on v and a, and we get $r + vt + at^2/2$ for the position at time t from now. This is a relevant metric that can be used to adjust the airplane control settings such that airspace congestion is avoided.

In a similar way, the performance analysis predicts the performance metrics p relevant for the real-time constraints given the data complexity characteristics. For streaming applications, we mentioned that the relevant metric is throughput. Normally, some short-term variations of throughput can be tolerated, especially if the output is buffered in the memory buffer and particularly in case of soft real-time constraints. It is more important that the long-term average throughput, calculated using integration of fine-grain time intervals, stays within the constraint.

The performance analysis is more than timing verification. It not only derives the relevant performance metrics, but also gives *guidelines* for the adaptation. If the current mapping choice does not satisfy the real-time constraints, the guidelines show which part of the implementation is a bottleneck and should be modified, and it also shows the direction for the necessary modification. By analogy to non-linear programming, where the objective function derivative may be used as an optimization guideline, in Figure 1.8(b), we denote the performance analysis guidelines as $\partial p/\partial x$, although in reality our performance analysis approach may also give guidelines for discrete control parameters, such as a FIFO buffer memory capacity. In case the constraints are satisfied, the guidelines can help to estimate the extent to which the current control parameter settings can be relaxed, e.g., to improve the visual quality or to save power, without a risk to violate the constraints. Based on the received performance metrics and guidelines, the optimization unit may generate a new candidate solution to be analyzed, or it may decide to adapt the settings of the optimization object. Hereby, one needs to ensure that the algorithm does not run in an endless loop or at a local optimum.

Note that in the diagram in Figure 1.8(b) the generation of candidate solutions, x_k, is done from scratch, separately from the performance analysis. This approach is typical and quite universal. However, in some cases it is possible to improve the efficiency of this approach by integrating the solution generation and the performance analysis.

The performance analysis is the main topic of this thesis. At the same time, the optimization algorithm issues such as candidate solution generation and stopping criterion are beyond our scope. Although the performance analysis can also be used for the design-time optimization, we mostly focus on using it to handle the dynamism due to data-dependencies, which is done through run-time optimization, or adaptation.



(a) Local quality adaptation (intra-application)

(b) Global quality adaptation

(c) budget adaptation

(d) dynamic voltage scaling

objective/cost: '$F(x)$'	optimized variables: 'x'
δQ – fine quality level;	*par* – quality tuning parameter
Q – coarse quality level;	*ql* – quality level setting;
B – resource budgets;	*b* – budget setting;
E – energy consumption;	*vdd-fclk* – supply voltage / clock frequency;

Figure 1.9 Adaptation (run-time optimization) examples

Figure 1.9 shows four examples of adaptation considered in practice. We present them here because they are possible contexts in which our performance analysis techniques can be applied. Each of them exploits one of the three degrees of freedom introduced in the previous subsection.

Figures 1.9(a) and 1.9(b) show the adaptation based on QoS management, which we call *quality adaptation*. In related work, hierarchical control is proposed where two levels are distinguished [71]: local management (intra-application management) and global management (multi-application). At the intra-application level, one can introduce an optimization agent called a local manager (Figure 1.9(a)) which fine-tunes the quality settings of an application, whereas at the coarse level the quality is set by the global manager that oversees all the applications.

Figure 1.9(c) shows the case where the optimization object is not only the application itself, but a stack consisting of the application and the underlying scheduler responsible for resource sharing. The optimization agent assigns different resource budgets to different applications, depending on their workloads. This is only possible at the global control level, because changing the budget of one application affects the budgets of other applications. We refer to this case as *budget adaptation*.

Finally, Figure 1.9(d) presents *power consumption adaptation*, where dynamic voltage scaling is exploited with the objective to minimize the consumed energy. Because changing the frequency of a processor may affect multiple applications, this kind of adaptation is also performed at the global (multi-application) level.

In all of the presented examples, it is meaningful to consider building performance analysis of some kind into the optimization agent. In the following section, we take a closer look at the performance analysis.

1.4.4 The Required Profile for the Performance Analysis

In this subsection, we make a 'wish list' of the performance analysis properties required for our design methodology and identify major appropriate means to achieve them.

a) Guaranteed performance = conservative and, preferably, accurate

To ensure good performance, one needs to be at the pessimistic side when estimating it, because if an embedded application too often fails to meet the real-time constraints, then it can become useless. Thus, our performance analysis should provide *conservative* estimates of the performance metrics, e.g., a lower bound on the throughput. On the other side, being too pessimistic on performance can result in paying too high a price in terms of higher energy consumption and lower visual quality. Therefore, it is desirable that the estimates are sufficiently *accurate*. The required level of accuracy is determined by trade-off between the analysis overhead and the loss in the adaptation objective (such as quality or energy), $F(x)$, due to analysis error, which is often caused by analysis pessimism. To avoid a high price for pessimism, for SRT applications, we relax the conservativity requirement – by assuming the performance analysis may also give results that are pessimistic with a sufficiently high probability. In this case, we speak of *weak conservativity*, whereas a 100% guarantee is referred to a *strict conservativity*. The latter is a required for HRT applications.

In the case when the required conservativity and accuracy levels are both achieved, we speak of *guaranteed performance*. We set it as a goal, but we must admit that satisfying both requirements is not always achievable in practice; therefore, we also accept situations where only conservativity is present, but the error is beyond the limits when the estimations can be called tight.

b) Wide dynamic range and still not too far off $\Rightarrow$ run-time

We need our approach to scale up to a wide dynamic range of data-dependent processing times. To put it informally, the adaptation should be truly dynamic. However, the wider the dynamic range, the greater the uncertainty about the performance metrics of applications at design time.

Therefore, at least part of our performance analysis should be performed at run time, because then the performance analysis can make use of run-time information about the application workload expected in the near future based on run-time characteristics of input data. This reduces the uncertainty and improves the accuracy.

c) Appropriate Model: Multiprocessor parallelism and FIFO communication

Performance analysis should be based on appropriate models. There are many *models of computation* that can capture the behavior of computer applications. An important class of such models explicitly captures the parallel activities, links between them and formal rules for

interaction through the links. In that class, two characteristic and well-established models are Communicating Sequential Processes (CSP) [37] and Kahn Process Networks (KPN) [43].

The parallel activities can be characterized by the lifetime (e.g. continuous or temporary) and their nature, i.e., whether they are processor instructions, function calls or programs. We require support of multiprocessor parallelism, whereby several programs continuously run on different processors and interact with one another. Both CSP and KPN are suitable models to represent multiprocessor parallelism.

As for the links and interactions between the programs, different models may have different assumptions. In CSP, processes interact through so-called channels by defining synchronization points at which a process waits until another process also reaches a particular synchronization point. In KPN, the programs communicate streams of data to each other through the channels in a FIFO order. For this reason, KPN is very well suitable in practice for modeling streaming applications [45]. Therefore, we prefer a model that intrinsically supports FIFO channels and has a direct relation to KPN instead of CSP.

As already explained in Section 1.1.2, there are important reasons why we restrict ourselves to FIFO communication, and the other communication schemes are handled by pre-scheduling and encapsulation of the communicating sub-tasks inside the tasks of the task graph.

d) Analytical; preferably, algebraic

We prefer an analytical performance analysis approach. This means that we prefer to start from facts that one can rely upon (axioms) and to apply logical reasoning to arrive at the relevant results, the throughput estimation, in our case. In this case, one can rely upon the results and, in the case of errors, one can quickly trace them back to the wrong original assumptions.

We want even more. Because, as we have seen before, the end result should be computed at run time, we prefer that it can be expressed algebraically, i.e., as an application of a well-defined sequence of limited-complexity operations to a well-defined combination of arguments. An example of an algebraic expression is the mentioned expression for prediction of the future position of an airplane, $x + vt + at^2/2$. In the next section, we see examples for the context of SoC design.

In case of streaming and multiprocessor platforms, the axioms would specify the timing properties of 'microscopic' low-level fine-grain operations carried out on small elements of a stream and primitive network-on-chip transactions. As the end result, we should obtain a coarse-grain 'macroscopic' property, namely, the throughput of the application. This brings us to the final point.

e) Covering long execution runs

The streaming applications are typically characterized by long loops that produce a long sequence of stream data elements without interruption. Applying a brute-force approach by taking into account every stream element is not practical. We want our approach to scale up to any duration of uninterrupted execution.

1.5 Analysis of Related Work

1.5.1 From Static to Dynamic Optimization

In Figures 1.8 and 1.9, we introduced frameworks for optimization of energy consumption, resource requirements and quality. Such frameworks can be divided into two major classes: *static*, whereby the input data characteristics are constant and can be computed or estimated at

design time, and *dynamic*, whereby the end results of the performance analysis are continually updated in correspondence to the run-time changes in the input data characteristics. The static approaches can provide a foundation for the dynamic ones.

Static approaches typically obtain worst-case/best-case or statistical (e.g., average case) performance metrics either analytically or empirically, either using a specification of input data properties or a sample set of input sequences. Static optimization frameworks are applied not only to program existing multiprocessor platforms, but also in hardware/software (HW/SW) co-design.

For example, Mladen Bereković *et al* [9] propose a HW/SW co-design approach to analyze an implementation of a particular video decoding algorithm. They evaluate the resource utilization using the linear formula $\Sigma C(i) \cdot F(i)$, where $F(i)$ is the average frequency of task i and $C(i)$ is the number of clock cycles to perform the task. If the resource utilization is too high, this technique provides guidelines on which task has the highest contribution and on how much effort should be put to decrease $C(i)$ by enhancement of hardware resources.

Although in [9] the platform contains two processors communicating with each other, their analysis was focused on only one of them, performing the most computation-intensive tasks; thus they ignored various subtle effects coming from multiprocessor parallelism and communication, e.g., processor stalling when waiting for data from another processor. But in general we cannot ignore those effects. We discuss the relevant research work on static performance analysis for multiprocessors in the next subsection.

In the research on run-time power consumption adaptation, a popular approach is to extrapolate the performance metrics from those measured in the previous history. However, this work rests on the assumption that the performance metrics change rather smoothly in time. This approach does not satisfy our wish for the performance analysis to scale to wide dynamic ranges of processing time variation, and the assumption is, for example, not valid for many MPEG-4 streams with arbitrary-shape video objects.

Another popular direction in dynamic performance analysis can be seen as an extension of the static formula $\Sigma C(i) \cdot F(i)$ to the case where the task execution frequencies $F(i)$ become run-time parameters provided as the input of the performance analysis. Such an approach is used in quality-of-service adaptation of 3D-graphics applications, e.g., as described by G. Bontempi *et al* in [10]. A very similar approach for a streaming application has been proposed by A.C. Bavier *et al* in [6]. Also the detailed workload prediction model for video decoding applications proposed by Yicheng Huang *et al* in [41] can be represented algebraically with the abovementioned linear expression. An advantage of this *algebraic approach* is the intrinsic support for arbitrarily long execution runs. A major disadvantage of the work in [10], [6], and [41] is the lack of support for multiprocessor parallelism and communication.

All the examples given so far that allow for dynamism assume sequential execution. In the approaches surveyed in the next subsection the opposite is assumed: no dynamism and parallel execution. Note that allowing dynamism and parallelism in combination is a tough problem even for a standalone performance analysis, considered apart from the rest of the optimization framework.

One approach for the dynamic adaptation of energy consumption in multiprocessors is proposed by Peng Yang *et al* in [99]. We explain their key idea for dealing with the dynamism combined with parallelism below, but first explain the limitations of their work for the problems addressed in this thesis. [99] is somewhat biased to control applications, because it assumes that

the set of possible execution paths is limited, whereas in arbitrarily long execution runs of streaming applications the set of alternative execution paths can be arbitrarily large. In a later continuation of [99], Zhe Ma *et al* [55] adapt the approach to long execution runs by representing an execution run of a streaming application as a concatenation of limited-size segments, i.e. pre-calculated schedules. Unfortunately, this approach assumes that the segments are executed sequentially, one after another. This assumption will always lead to severe processor underutilization if the segments are to be pipelined. Not supporting pipelining is a serious limitation for streaming applications. To prevent underutilization due to sequential execution of different schedules, in [56] they propose timing-interleaving between the schedules. However, the method of [56] cannot be directly applied to the execution-run segments of [55], as they do not support dependencies between the segments. In [57], they consider a run-time scheduling approach that supports dependencies. For the streaming applications, their work would result in a fine-grain level of control, i.e. making a control decision for every segment, instead of handling a long execution run as one unit. Therefore, for long execution runs, they would potentially find more optimal run-time adaptation decisions at the cost of potentially larger run-time overhead. Up-to-date, we are not aware of any scheduling method that could take the inter-segment dependencies into account without analyzing every segment in the whole execution run.

Nevertheless, [99] and [55] have introduced an important way for dealing with dynamism combined with parallelism. In that work, the set of alternative application execution paths is split into subsets with similar resource requirements. Those subsets are called *scenarios*. Each scenario is considered separately using static optimization at design time. At run time, different combinations of the static scheduling solutions are activated, using the run-time knowledge about which particular alternative execution path is taken at every segment of the execution run.

An important idea of this method is that one can represent a dynamic system behavior by a discrete set of alternative static sub-behaviors. This makes the combination of dynamism and parallelism tractable for performance analysis due to separation between those two issues.

Suppose that we have indeed separated dynamism and parallelism. Then we still have to deal with parallel applications, now being static. In the next subsection, we consider a class of candidate static performance analysis methods that can be applied for this case.

1.5.2 Steady-state Performance Analysis

The related work knows a few mathematical formalisms that can be used to analyze long runs of applications having static characteristics at the microscopic level of granularity – which is, in our design methodology, the level of primitive tasks and network transactions. What all those formalisms have in common is that their long-run or macroscopic performance metrics converge to a certain state of equilibrium, the *steady state*, and become static (stationary). Those formalisms are of big interest for us because they support long execution runs in multiprocessors, taking the parallelism and interactions into account.

It is important to mention that the steady-state execution phase is preceded by a temporary *transient* phase. To extend a steady-state model to a dynamic model with multiple steady states, it is important that the transient phase can be analyzed as well.

To analyze and optimize the programming of audio streaming applications on multiprocessors, *IPC (interprocessor communication) graphs* have been proposed by N. Bambha, S. Bhattacharyya and others in [5]. These graphs are instances of the *homogeneous*

dataflow graph (HSDF) model of computation, which can be seen as a special restrictive case of KPN. These graphs assume constant task processing times and a specific way of multiprocessor communication through the system bus [83]. By choosing HSDF model of computation, these graphs severely restrict the allowed communication properties of tasks, thus sacrificing the expressive power, but getting the ability to evaluate the performance analytically in return. The analysis formalism for this kind of computational model is max-plus algebra [4], which can be used to prove that IPC graphs have static average throughput in the steady state. This throughput metric can be derived analytically from the HSDF graph. Also the transient behavior of HSDF graphs is formally defined. We come back to these models in the next subsection.

In [80], K. Richter, M. Jersak, and R. Ernst propose *schedulability analysis* for timing verification of combinations of multiprocessor applications. This approach requires the knowledge of the static worst-case and best-case performance metrics of the application tasks. That work analyzes the worst-case timing behavior of the system in the steady-state. As for the transient behavior, no formal reasoning is provided. The approach does not use resource reservation which can potentially lead to a better processor usage than approaches using resource reservations per application as we propose, but, as discussed before, the complexity of taking all interactions into account would make run-time performance analysis problematic.

The formal modeling language POOSL can be seen as a model of computation with dynamic processing times of the system components, see e.g. Bart Theelen's PhD Thesis [93]. The mathematical formalism applied for reasoning about the timing properties is Markov chains; for dataflow graphs it has been applied e.g. by Bart Theelen *et al* in [94]. Also this formalism requires the components to have some static properties. Here each processing time should have well-defined statistical moments (like the mathematical expectation and the standard deviation). However, in case of data-dependent streaming applications, where input data characteristics can change in an unpredictable way, it is not straightforward to find the conditions when an application possesses such a property, even in approximation. To avoid this issue, we leave evaluation of this approach to future work. Similar remarks can be made about other stochastic formalisms, like stochastic event graphs [4, §7, 8].

1.5.3 Conclusions on Related Work and Goal Formulation

From the analysis of the related work, we draw the following conclusions. To support arbitrarily long execution runs on multiprocessor systems, we can use steady-state performance analysis approaches. To cope with dynamic behavior, we should be able to model it as a combination of several steady-state behaviors. Because this involves transitions from one steady state to another, the transient phases should be taken into account.

HSDF/IPC graphs have a formalism for both transient and steady-state behavior and have a previous history of being used in the context of streaming applications. Moreover, they form a special case of KPN and inherit from them the support of FIFO communication (see Section 1.4.4). For these reasons, we choose to build our approach on HSDF graphs.

This implies that we accept the HSDF-graph restrictions on the communication properties of tasks. Nevertheless, in the next chapter, we explain these restrictions and argue that reasonable workarounds are often possible. We also argue that, despite these restrictions, our work is still applicable for a large class of streaming applications.

Although our HSDF models reflect dynamic behavior (i.e. data-dependent processing times), we do not model reconfigurations, i.e. changes from one configuration to another one. Our models assume that the mapping is *static*, see e.g. Figure 1.5. This is a valid assumption when reconfigurations are rare, i.e. when they do not occur when the applications are actively busy with computations. Nevertheless, because our analysis approach is *dynamic*, we believe that in future we can extend this approach to support frequent reconfigurations as well.

1.6 Contributions and Organization of this Thesis

In this thesis, we consider a generic distributed-memory multiprocessor platform and streaming applications. These applications consist of tasks communicating through FIFO (first-in-first-out) channels. To run an application, one has to perform mapping of the application onto the multiprocessor, i.e., distribute the tasks between the processors and organize the communication between them.

Multiple applications can start and stop at run time, being executed in parallel to one another. Each application usually has real-time constraints requiring it to maintain a certain throughput. To make it possible for the applications to continually meet those constraints independently of other applications, we assume resource reservations per application. This means that each application gets capacity budgets on different resources of the platform.

Within the context sketched above, this thesis contributes to the solution of the performance analysis problem. The primary task of the analysis is timing verification, i.e., checking whether an application can meet its real-time constraints. A positive or negative answer depends, in general, on the mapping and, which is very important, on the data content being processed by the application.

Our main contribution to the multiprocessor performance analysis is twofold. First, we develop new timing models for applications mapped onto network-on-chip platforms. Second, for these models, we develop conservative analysis techniques to calculate important performance metrics related to application throughput. These techniques show promising results in terms of accuracy for a highly dynamic application we use as a case study. In the remainder of this section, we explain the main ideas of our contribution and the way we present them in the structure of this thesis.

Our performance analysis techniques apply for long uninterrupted execution runs of highly dynamic applications on a multiprocessor system. To cover the long execution runs, we use a model of computation that supports multiprocessor parallelism and can be characterized by a steady-state behavior. The chosen model of computation is HSDF/IPC graphs. To support highly dynamic applications, we propose to characterize them by multiple transitions between different steady-state behaviors of IPC graphs. In Chapter 2, we introduce the basics of IPC graphs, and their main properties. We discuss the performance analysis for those graphs and extend that model such that different steady-state behaviors can be characterized. We also explain the practical use of the extended model in the context of quality adaptation.

In the past, IPC graphs have been used only with constant processing times of graph nodes and only for multiprocessors with bus communication, whereas our distributed-memory platform assumes interconnection networks with distributed topologies. Therefore, we dedicate Chapter 3 to the details of IPC graph construction, whereby we explain a generic method to model variable

processing times of the graph nodes and introduce novel methods to represent the network-on-chip communication in IPC graphs.

Having constructed an IPC graph, one can use it not only to calculate the application's throughput, but also to improve the mapping solution. An IPC graph exposes subtle dependencies of the throughput on the implementation details. In Chapter 3, we show that the performance analysis can be used to select the size of the FIFO buffers such that the required throughput can be met.

Chapter 4 first studies the theory behind the steady-state behavior of HSDF graphs with constant processing times, including some aspects that did not get much attention in the literature. Then, in Chapter 5, we turn our attention to the case with dynamic task processing times. There, we develop our novel concept of transitions between steady-state behaviors of IPC graphs in detail. This gives us a method to accurately estimate the throughput of a dynamic application and to provide guidelines for optimization also in the context of dynamic applications. As a result, we describe a throughput-related characteristic in the form $\Sigma C(i) \cdot F(i)$, traditionally used for dynamic performance analysis (see previous section). However, in our case, to cope with parallelism, the meaning of $C(i)$ and $F(i)$ in this formula are different from their meaning in the context of sequential execution.

Chapter 6 revisits the MPEG-4 shape-decoding example illustrated in Figure 1.7. We consider a reasonable mapping of this application onto a multiprocessor platform and apply our performance analysis techniques to predict its varying performance under the conditions of a varying video object shape. We demonstrate the use of our techniques in a working quality adaptation manager that satisfies the description shown in Figure 1.9(a).

Chapter 7 summarizes the thesis and discusses directions for future work.

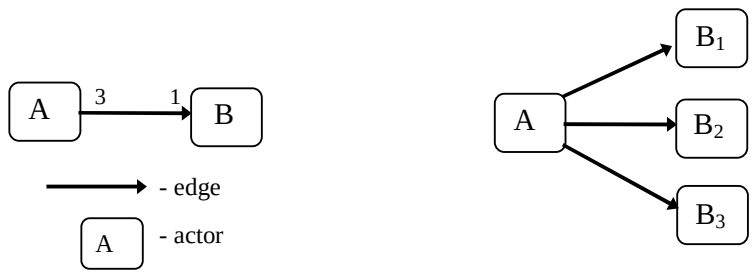
2

2 A Strategy for Implementation and Analysis

In Chapter 1, we sketched a system implementation approach that we argued to be relevant and important in advanced multimedia system design for consumer electronics products. That approach consists of two-stage mapping and run-time adaptation endowed with performance analysis. The latter is meant, in particular, to serve as an instrument to deal with high data-dependent workload variations. In this chapter, we share our point of view on how the ideas raised in Chapter 1 can be realized, with an intention to create a background for presenting the modeling and performance analysis techniques of this thesis. The multimedia system design with on-chip multiprocessors is to a large extent an open research area due to their novelty, physical design challenges and extra degrees of freedom introduced to overcome those challenges. Therefore, we often have to refer to the work on multi-chip multiprocessors, which have a lot in common with on-chip multiprocessors.

We start this chapter by introducing the behavior and implementation of a generic streaming application, in order to answer the question: what do we analyze? Afterwards, Section 2.2 gives an introduction into the existing performance analysis techniques that we use as a foundation to build our approach. We also explain how it contributes to the previous work. In Section 2.3, we discuss the adaptation techniques where the performance analysis of this thesis can be applied.

This chapter raises several relevant implementation and analysis issues that are to a certain extent answered in the rest of this thesis.



(a) An SDF graph **(b)** The equivalent HSDF graph

Figure 2.1 Examples of dataflow graphs modeling the loop of interest

2.1 The Object for Performance Analysis: Behavior and Implementation of a Streaming Application

2.1.1 The Scope of Modeling: the Loop of Interest and HSDF graphs

Design methods for embedded systems commonly assume a certain model of computation; for a detailed study of the models that are relevant for multiprocessor mapping see e.g. the book of F. Thoen and F. Catthoor [95]. The models reflect the properties of the application that are of interest for the system designer.

The streaming applications are most of the time involved in a repetitive execution of a finite set of functions applied on the data items coming from the input data streams. Therefore, it is often implicitly assumed that the application contains a loop that is mainly responsible for the processing of the application. The design effort, directed at meeting the real-time constraints at optimal cost, is focused on that loop. In this thesis, we refer to that loop as the *loop of interest*.

The model of computation that we choose for that loop is the model of *synchronous dataflow* (SDF) graphs [52], also known as the multi-rate data flow (MRDF) model of computation. It is widely used in the context of multiprocessors, e.g. in [83], [49], and [90]. Figure 2.1 gives an example of an SDF graph. The nodes of the SDF graph are called *actors*; they represent the tasks of the application. Each actor executes repetitively for an indefinite number of times. Every actor execution takes certain time, which we call the *actor delay*. Before and after the execution, each actor receives and sends portions of information via incoming and outgoing graph edges from and to other actors. The elementary portions of information exchanged via edges are called *tokens*.

The main property that distinguishes the SDF model from other dataflow models is that each actor consumes and produces a fixed number of tokens per execution per input and per output, called *consumption rate* and *production rate* respectively. For example, as annotated in Figure 2.1, actor A has a production rate of three tokens per execution at the edge going to actor B, and actor B has a consumption rate of one token per execution at that edge. As a result, each actor is expected to execute at a fixed rate with respect to one another, e.g., actor B should execute three times more often than actor A. For that reason, SDF graphs are used to represent the streaming applications characterized by a set of fixed execution rates, e.g., in video image processing.

Nevertheless, we do not restrict ourselves to such applications; in this thesis, we also support, to a large extent, applications with variable rates, such as MPEG-4 arbitrarily shaped video. In order to do that we have to overcome the limitations of the SDF model that normally lead to their usage only in the context of fixed-rate applications.

The main limitation of the conventional SDF model is that it does not allow data-dependent conditional execution of actors and conditional communication between them. It also does not allow conditional changes in the structure of the graph itself. The key idea we use to deal with the applications with data-dependent conditional behavior is to bring the conditional behavior into a different level of data granularity than the level represented by the SDF graph.

Firstly, we allow the whole graph to execute a variable number of times per given time interval. Hereby, we support applications like the MPEG-4 application introduced in Chapter 1. Recall that, in that application, the number of blocks per video frame can change dynamically from one frame to another. In our implementation, one execution of all SDF actors processes one block. Hereby, the SDF graph does not need to model the dependency of the frame decoding on the number of blocks per frame, because the frame has a higher data granularity level than the data processed by the SDF graph.

Secondly, we hide part of the conditional behavior inside the actors. We assume that the actor execution delays may change in a wide dynamic range. Thereby, when the specified application algorithm requires an actor to be skipped, we change the algorithm such that the actor is still executed but takes zero delay to execute. We refer to such zero-delay executions as ‘empty’ executions.

Note that this changes the original timing behavior of the algorithm essentially, because the ‘empty’ actor executions are essentially different from the conditionally skipped ones. The difference is that an ‘empty’ actor execution also reads and writes ‘empty tokens’. An ‘empty’ execution can be blocked, waiting for empty input tokens, thus slowing down the execution of the graph, whereas those tokens are not really required for the application algorithm to proceed.

Thus, hiding the conditional behavior inside the actors may change the properties of the algorithm such that the system performance becomes worse. Nevertheless, by giving up some performance, we buy the predictability of the timing behavior of the system. If the performance penalty is not too large, the ability to analyze the timing behavior is likely to outweigh this disadvantage, especially for real-time applications.

In this thesis, *general* SDF graphs serve only for the initial compact representation of the application. In our methodology, we mainly use a subclass of SDF graphs – *homogeneous SDF* (HSDF) graphs, also referred to as single-rate data flow (SRDF). An HSDF graph has an extra requirement that an actor may consume and produce only one token per input and output per execution.

Therefore, at the very beginning of our implementation trajectory, we translate the SDF of the application into an HSDF graph. Figure 2.1 shows an example of such a translation. Because in the original SDF model, actor B has to execute at triple the rate of actor A, in the resultant HSDF graph, this is represented by three actor instances: ‘B_1’, ‘B_2’ and ‘B_3’. For any practical SDF graph, one can compute the relative execution rates and translate the SDF graph into an equivalent HSDF graph. For the computation of the relative execution rates, see e.g., Thomas Parks’ PhD Thesis [73, §2.3] and for the translation algorithm see e.g. [32].

Being modeled as an HSDF graph, the loop of interest must satisfy certain assumptions. We assume that the *body of the loop* of interest consists of a fixed set of actors. The loop executes

multiple iterations whereby in each *loop iteration* each actor executes a fixed number of times. We also assume that each actor can communicate only a *fixed amount of data* per execution.

We put a restriction that the loop of interest is *flat*, not nested, which, although limiting the direct applicability of our approach, still keeps it useful as an approach solving a difficult part of a more general problem. Indeed, one could build on top of our technique a more general hierarchical approach for nested loops. That technique would apply our technique recursively bottom-up, from the lowest level of loop nesting to the higher levels, whereby a lower-level loop would be presented to a higher-level loop as an actor whose execution delay is calculated using the loop execution time calculation method that we propose in this thesis. However, exploring such an extension of our performance analysis approach is subject of future work.

Another quite essential assumption we take is that the timing of the loop of interest is *autonomous*, i.e., its actors can only be blocked waiting for other actors of the same loop, but they never block on external inputs and outputs of the loop. Of course, in practice, streaming applications do have external inputs and outputs (I/O) to exchange the data streams with each other and the environment. In fact, it should be technically possible to extend our approach at least to the case where a loop communicates with the other loops and the environrment using periodic communication patterns, but we leave a study on the impact of the external I/O on the loop's performance for future work. In this thesis, we assume that at any moment of time there is enough external input data and enough external space for output data in the external I/O ports, so that the timing of the external I/O does not have any impact on the performance of the loop execution. Under this assumption, the loop of interest can be considered autonomous and no external I/O ports need to be modeled explicitly. Therefore, we do not model them in this thesis.

We call a semantically defined set of subsequent loop iterations a *loop execution run*. The duration of the execution run is called the *loop execution time*. Because in a multiprocessor platform both data parallelism and pipelining can be applied to achieve the required performance, the loop execution time is approximately a linear function of the number of loop iterations in the execution run. Hereby, the linear factor of the linear function depends on the throughput of the HSDF graph (in terms of the average number of loop iterations per second). The constant component of such a linear function is normally referred to as 'latency'. However, due to the fact that we do not enforce any periodic schedule on the dataflow graph execution, the problem of finding the constant component in the abovementioned linear function becomes much less trivial than traditional calculation of latency of the given periodic schedule. For that reason, we give the constant component of the execution time a different name; we refer to it as 'lateness'.

For calculating the execution time for a loop with constant actor delays, finding *lateness* and *throughput* is a central performance analysis problem. For the calculation of throughput, efficient algorithms are available [19], [23]. Unfortunately, as we see later in the thesis, we cannot say the same about the lateness. For variable actor execution times, the expression characterizing the loop execution time is not as simple as just a linear function on the number of loop iterations. The basics for our characterization method will be introduced in the end of this chapter.

2.1.2 The implementation-enhanced HSDF model and the implementation trajectory

The SDF model of computation – and mainly its HSDF variant – is used throughout our implementation trajectory. We often refer to that model just as the SDF/HSDF graph. We

enhance the HSDF model with implementation information required for our methodology and call it the *implementation-enhanced HSDF model*. However, the functional and timing behavior of the model is identical to the behavior of the basic HSDF model widely used in the literature.

Definition: An implementation-enhanced HSDF model is a tuple $\mathbf{GPQ} < \mathbf{G}, \mathbf{T}, \mathbf{PQ} >$ where **G** is an HSDF graph, **T** is the set of *timing modes*, and **PQ** is the *implementation process network*. ♦

Thus, **T** and **PQ** are enhancement components of our model, introduced later in this section. They are data structures organically connected to the basic HSDF model **G**, carrying necessary information for the implementation trajectory.

At design time, the enhanced model is used simultaneously as a design database and a timing model for the given application. It undergoes transformations, following a trajectory from specification to executable. Together with the executable, the design flow issues a timing model of the executable, used at run time for adaptation. We call it the *IPC model*, or IPC[4] graph. The IPC model is central to this thesis. It gives the information needed for run-time performance analysis of the application executable.

Figure 2.2 shows our implementation trajectory, which constructs the IPC model and the executable at design time and employs them at run time. The trajectory consists of four parts, marked with Roman numbers. Parts I and II constitute the *design flow*, and parts III and IV constitute the *run-time management*.

In this subsection, we give an overview of the trajectory, at the same time giving the definitions of the basic features of the implementation-enhanced HSDF model. These definitions are used in this thesis to explain our timing modeling and performance analysis approach.

We start from the beginning of the design flow (the top part of Figure 2.2). First, the application designer provides the task graph as an SDF model. To construct a task graph, the application designer first divides the body of the loop of interest into *computation actors*, thus making the task-level parallelism explicit. The actors constitute the nodes of the task graph. Each actor has an *actor body*, which is a programming routine that implements the functionality of the actor. Having divided the loop body into actors, the application designer joins the actors by edges to specify the data dependencies between the actors. For each edge, a production and consumption rate is specified. The actors joined by edges that are annotated with production and consumption rates constitute an SDF graph.

Definition: The task graph is the SDF graph that is provided by the application designer as a specification of the application's loop of interest. ♦

As discussed in the previous section and shown in Figure 2.2, our implementation trajectory translates the task graph from SDF form into an HSDF equivalent. We call the result of the translation the *computation graph*, which is the first instance of the implementation-enhanced HSDF used in the design flow.

Definition: The computation graph is the implementation-enhanced HSDF model $\mathbf{GPQ_{comp}}$ that is obtained as the result of converting the task graph into its HSDF equivalent and enhancing the resulting HSDF graph **G** with enhancement components **T** and **PQ**. ♦

[4] Recall that 'IPC' stands for interprocessor communication and this name comes from previous work on IPC graphs.

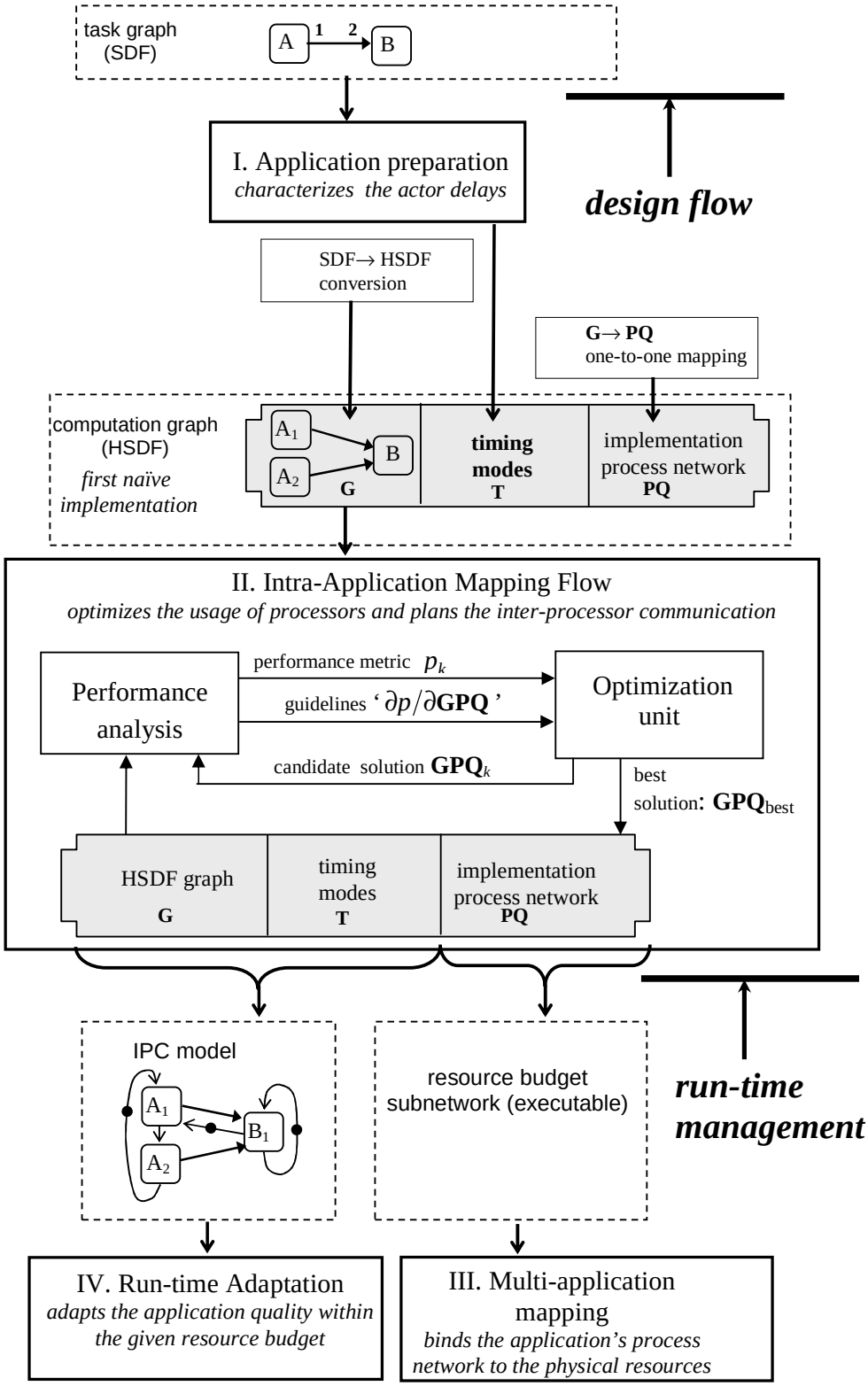


Figure 2.2 Implementation trajectory and (H)SDF Model Transformations

Below we define the basic HSDF graph and the enhancement components, motivating them and explaining how they are obtained in the beginning of the design flow.

Definition: An HSDF graph, $\mathbf{G} < \mathbf{V}, \mathbf{E}, m >$**,** is a directed graph, where **V** is the set of actors, which are the graph nodes. **V** has two subsets: $\mathbf{V} = \mathbf{V}_{\mathbf{comp}} + \mathbf{V}_{\mathbf{comm}}$, where $\mathbf{V}_{\mathbf{comp}}$ is the set of computation actors and $\mathbf{V}_{\mathbf{comm}}$ is the set of communication actors. **E** is the set of edges, and *m* – or the *marking* – gives a non-negative number of initial tokens on each edge; for example, the IPC graph shown in Figure 2.2 has three edges with one initial token shown as black dots. On the given edge, the actor at the input side of the edge is called the *producer* of the edge. For example, in the computation graph in Figure 2.2, actor A_1 is the producer for one of the edges. The actor at the output side of the edge is called the *consumer*. For example, in the computation graph in Figure 2.2, actor B is the consumer of two edges. ♦

This definition is general and holds at any point of the design flow. As for the computation graph, it contains only the computation actors, no communication actors are present in the model at the start of the design flow. Therefore, we can write:

$\mathbf{G}$ is a computation graph $\Rightarrow \mathbf{V} = \mathbf{V}_{\mathbf{comp}}$

Definition: The set of timing modes: $\mathbf{T}$. *Timing modes* are data structures defining the actor delays for all actors in the HSDF graph. Different timing modes provide different levels of accuracy of actor delay modeling, to be used for different purposes in the implementation trajectory. ♦

As shown in Figure 2.2, the timing modes are obtained from Part I of the implementation trajectory – the application preparation.

Definition. Part I: Application Preparation The *application preparation* characterizes the timing behavior of the computation actor on the processor architectures available in the target platform. It tries different implementations of different actors on different processor architectures and measures the corresponding actor delays, using profiling or worst-case execution time analysis tools. The results are stored in the design database in the form of timing modes, **T**. ♦

The application preparation and the timing modes are more rigorously defined and explained in Sections 3.1-3.3 of the next chapter.

Now the turn comes to describe the last but not the least component of our model – namely the implementation process network, **PQ**. In our approach, every HSDF graph that models the loop of interest is accompanied by the implementation information telling how the computation actors are to be bound to processors and how the data communication between the actors is to be organized.

The *implementation process network* is a graph structure that is used as the intermediate stage for binding the HSDF actors to the processors of the target platform. It groups together the computation actors that share the same processor at run time. It also groups parts of the HSDF graph that model the communication between multiple actors through the same network connection. The computation actor groups are called *processes* and the groups of communication actors and edges are called *channels*. Also, on the per-process and per-channel basis, the design flow specifies the *resource budgets*. As discussed in Chapter 1, the resource budgets realize the reservation-based approach, which makes it possible for us to handle the dynamic sets of real-time applications. Assigning the resource budgets to actor groups rather than to individual actors decreases the overhead of budgeting.

Definition: The HSDF implementation process network, **PQ**, is a tuple $<\mathbf{P},\ \mathbf{Q}, \mathbf{V}_{\mathbf{comp-body}}, \mathbf{B}>$ where **P** is the set of *processes*, which bundle computation actors in ordered subsets. **Q** is the set of channels between the processes. **P** and **Q** are nodes and edges of a directed multigraph[5], which reflects the structure of the process network. $\mathbf{V}_{\mathbf{comp-body}}$ is the set that, for each computation actor gives the *actor body*, inherited from the task graph (recall that it is the specification of the computation actor in a programming language). **B** is the *budget descriptor*, which specifies the resource budgeting for each process and channel. ♦

For the computation graph, the process network represents an initial naïve implementation, serving as the starting point for the mapping. For each actor, this implementation allocates 100% budget of a processor, and each edge gets one infinite-capacity channel. Therefore, in the computation graph, there is one-to-one correspondence between the actors and the processes, as well as between the edges and the channels. Also, the budgets assigned to the processes and channels of the computation are either maximal or infinite.

Let '$\leftrightarrow$' denote one-to-one correspondence (bijection). Then we can write:

$$\mathbf{GPQ} \text{ is a computation graph} \Rightarrow \quad \mathbf{V} \leftrightarrow \mathbf{P},\ \ \mathbf{E} \leftrightarrow \mathbf{Q},\ \ \mathbf{B} = \mathbf{B}_{\mathbf{maximal}} \quad ♦$$

Part II of the implementation trajectory, i.e., the intra-application mapping flow, applies transformations to **GPQ**. The goal of the mapping is to minimize the costs (i.e., the number of processors and channels to be used, the computation and communication budgets) while meeting the throughput constraints. Many existing multiprocessor mapping techniques fit in the iterative approach illustrated in Figure 2.2. This approach is similar to the adaptation, which we described before in Figure 1.8. There is an optimization unit that selects the best solution based on the performance estimation of different solutions and on the guidelines from the performance analysis. In contrast to the adaptation, in addition to the search for optimal *scalar* settings, like budgets, the mapping flow also brings some structural changes into the model; for example, it adds communication actors into the graph. Therefore, in the figure, we use notation '**GPQ**' instead of '*x*' for the optimized variable. Because the optimization problem is complex, it is split into a few sub-problems that are solved one after another. The mapping also differs from the adaptation problem in the sense that the mapping does not have the run-time predictions of the input data complexity characteristics, but it uses estimates of the typical values of those characteristics that are supposed to hold for the whole application run. The intra-application mapping flow is described in more detail in Sections 3.5 and 3.6.

After the mapping flow, the design is ready to be issued in the form of an executable. As shown in Figure 2.2, the final **GPQ**-model is split into two parts: the IPC model and the resource budget subnetwork.

Definition: The IPC model or IPC graph is obtained from the graph and the timing modes of the final implementation-enhanced HSDF model. The IPC graph includes two components of that model: graph **G** and set of timing modes **T**. ♦

For the completeness of illustration, Figure 2.2 shows an example of an IPC graph, which can be obtained starting from the computation graph shown in the same figure. However, we postpone a detailed discussion of IPC graphs until later in this chapter. IPC models are proposed in this thesis as timing models for run-time performance analysis and adaptation of

[5] In a directed multigraph, more than one edge can join two nodes in the same direction.

multiprocessor streaming applications. Therefore, in Figure 2.2, the IPC graph is fed to part III of our implementation trajectory.

Definition: The resource budget subnetwork is the implementation process network obtained at the end of the design flow. It represents the executable as seen by the target platform. This resource budget subnetwork has a graph structure, given by the processes joined by channels.♦

As shown in Figure 2.2, the resource budget subnetwork is used by the multi-application mapping module, Part III of the implementation trajectory. When an instance of the given application is being started on the platform, it assigns the processes and channels of the resource budget subnetwork to the physical resources. The multi-application mapper finds the physical processors for the processes and physical routes for the channels. In some sense, it can be compared to placement and routing in logic synthesis or to the application loader in operating systems. We use term 'subnetwork', to suggest that, in the context of the global on-chip network, each application operates in separate virtual subnetwork, being independent of the other subnetworks due to resource reservation.

Part IV of the implementation trajectory (see Figure 2.2) is the run-time adaptation manager. The functionality of that manager has been considered in detail in Chapter 1, see Figure 1.1. It can be responsible for run-time adaptation of application's power consumption or quality of service. Those tasks are very similar in the sense of the optimization problems being solved, and in this thesis we restrict our reasoning to the quality-of-service adaptation.

After the applications have been loaded into the platform and concrete physical resources have been allocated to concrete applications, *run-time scheduling* comes into play. It complements the functionality of the run-time management, but we do not include it as a part of the implementation trajectory. The reason is that the run-time scheduling is not involved in taking any optimization decisions; it only realizes the decisions already taken in the trajectory. In particular, the run-time scheduling manages the sharing of the platform resources such that the processes and channels definitely receive as many shares of the physical resources as specified in their resource budgets. Note that the run-time scheduling is distributed, every processor has a local scheduler and also the network components are arbitrated by local arbiters.

Given the implementation trajectory and the run-time scheduling environment discussed in this subsection, it is the purpose of this thesis to show how the implementation-enhanced HSDF model can serve for efficient performance analysis at design time and at run time. The basic *events* for our performance analysis are the starting/completion events of the actor executions. Based on those events one can completely characterize the execution of the loop of interest without going too deeply into details. For the purpose of the performance analysis, our implementation-enhanced HSDF model should adequately take into account the following features of the modeled application:

- the ordering of the events, determined by the chosen implementation paradigm,
- the delays between events, determined by the chosen run-time scheduling mechanisms.

Therefore, in the following subsections, we show the important facts about our implementation paradigm and the run-time scheduling.

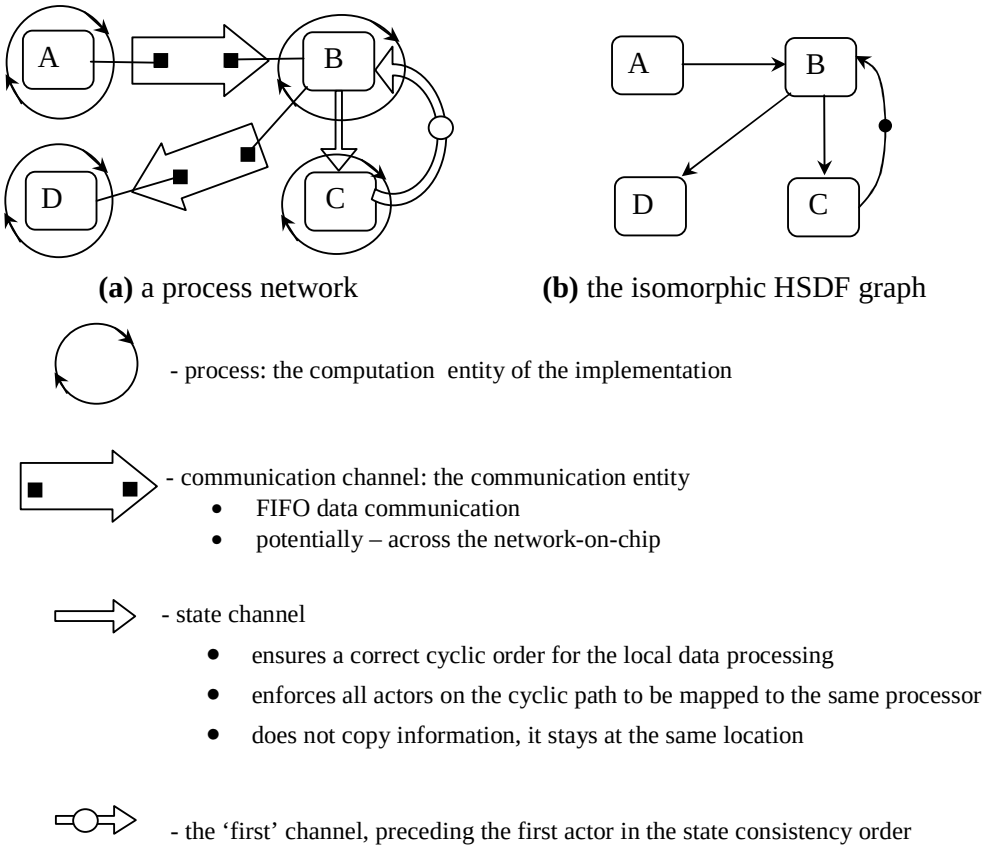


Figure 2.3 Implementation process network

2.1.3 The Implementation Process Network

The core of the implementation process network is the processes and the channels, which are entities that implement the application.

The processes are connected to the channels consistently to the computation actors contained inside them. Each actor has a body. The actor bodies are segments of software code that program the functional behavior. The computation actor bodies specify inputs and outputs through which they are connected to the incoming and outgoing channels. Hereby, the processes, in which the actors are contained, are also connected to the channels. An example of a process network is shown in Figure 2.3(a).

Recall that for the computation graph holds that each process contains just one actor and each channel contains just one edge. Given that, it is enough to have the process network to derive the rest of the implementation-enhanced HSDF model. The computation graph **G** can be easily derived from its process network, because **PQ** and **G** are isomorphic; one can find an example in Figure 2.3(b). However, during the mapping flow, the original processes are bundled together, forming more complex processes, and the same happens to the channels. For that reason, the isomorphism between **PQ** and **G** is not present anymore. Then, the process network can be seen as a coarse-grain view on the structure of the HSDF graph. To be more precise, for each channel

$q_i \in \mathbf{Q}$, a *channel macro* $\mathbf{GQ}(q_i)$ is defined, which is a substructure[6] of $\mathbf{G}$, consisting of edges and communication actors $\mathbf{V_{comm}}$, modeling the channel behavior. Similarly, for each process, $p_i \in \mathbf{P}$, a *process macro* $\mathbf{GP}(p_i)$ is defined, which is a substructure of $\mathbf{G}$ consisting of edges and computation actors $\mathbf{V_{comp}}$, modeling the behavior of the process.

To summarize the general correspondence between the process network and the HSDF graph within the implementation-enhanced HSDF model at any point of the mapping flow, we can write:

$$\begin{aligned} &\mathbf{P} \leftrightarrow \{\mathbf{GP}(p_i)\}, \\ &\mathbf{Q} \leftrightarrow \{\mathbf{GQ}(q_i)\} \\ &\bigcup_{\mathbf{P}} \mathbf{GP}(p_i) \cup \bigcup_{\mathbf{Q}} \mathbf{GQ}(q_i) = \mathbf{G} \end{aligned} \tag{2.1}$$

Let us first consider the channels. The purpose of the channels is to carry the application data structures that have a longer lifetime than a single actor execution. We classify such data as communication inputs, communication outputs and state.

The *communication inputs/outputs* contain information that can potentially be communicated between different processors. Typically, it is only worth paying the communication overhead if the sender actor and the receiver actor are capable of executing *concurrently*. For example, at a certain point in the MPEG video-decoding algorithm, a new 8x8 element DCT[7] block is extracted. It can be sent to an IDCT[8] actor and that actor can start processing it concurrently with extraction of the next DCT block, because, in order to proceed with the extraction there is no need to wait for the results of the IDCT. Therefore, it is worth specifying DCT blocks as communication input of the IDCT actor. Another reason that can justify the costs of the data communication between two actors is knowledge that there are two different processor architecture types, each implementing one of the communicating actors much more efficiently than the other.

The communication data is transferred between the actors by *communication channels* $\mathbf{Q_{comm}}$, which form a subset of $\mathbf{Q}$. For example, the process network in Figure 2.3(a) contains two communication channels. It is important to note that the communication channels are FIFO (first-in-first-out) channels, i.e., the communication outputs connected to a channel (also called channel producers) should send the information in the same order as the order in which the communication inputs (also called channel consumers) receive them.

By analogy to processes, which perform the computation by repeatedly executing the computation actors, we say that the communication channels continually execute communication transfers assigned to them. A *communication transfer* is an activity of passing one data token from a communication output of one actor to a communication input of another actor. As an activity running on-chip, a communication transfer is very similar to a computation actor, in the sense that it also operates on the data tokens, but its function is just copying the application data as it propagates through the communication network. Note that as long as a communication channel is not mapped to the network, its communication transfers are empty activities. This is

[6] Note that we call it a 'substructure' not a 'subgraph', because, a channel macro can consist, for example, of simply one edge, whereas a subgraph should be a graph by itself.
[7] Small piece of video image represented in a Fourier domain using Discrete (Fourier) Cosine Transform
[8] Inverse Discrete Cosine Transform

the case when all actors joined by a channel are mapped to the same processor, because no data copying is required in that case.

Definition: A communication channel q_{comm} is a channel that can be mapped to the network-on-chip. It is defined as a tuple $< m, \mathbf{TQ} >$, where m is the number of initial tokens – or *marking* – and $\mathbf{TQ}$ is a set of *communication transfers* that are continually repeated by the channel. The transfers are defined as triplets $\mathbf{TQ}(q_{\text{comm}}) = \{(v_{\text{prod } j}, v_{\text{cons } j}, z_j)\}$, whereby the *j*-th communication transfer is defined by the *producer actor* $v_{\text{prod } j}$, the *consumer actor* $v_{\text{cons } j}$ and *data token size* z_j. Any channel in the computation graph has only one producer and one consumer, and it continually repeats only one transfer. In the mapping flow, some channels may be bundled together, forming channels with multiple transfers. All the producer actors of a communication channel should belong to the same process. The same requirement holds for all the consumer actors. Data tokens propagate through the communication channel in a FIFO order, which means multiple communication transfers can be pending at any moment of time and they complete in the same order as the order in which they start.♦

In contrast to the communication inputs and outputs, the actor *state* refers to the data structures that, although being exchanged between the actors, are kept *locally* within the memory system of one processor. A guideline for an application designer to identify a data structure as the state is the case when the data dependencies impose a cyclic order in which the actor executions should access that data structure. Note that it can be executions of the same actor or different actors. A good example is the dependency based on parsing the input bitstream in the video/audio decoding algorithms. Each parsing operation needs to wait for the result of the previous operation to know the location where it should start further parsing.

In the case of cyclic dependency, the actors (or the actor) can only execute sequentially. Then, especially when the actors are best fit for the same processor architecture, there would be hardly any reason for spreading the actor executions between multiple processors, so the data structure as well as the actors can be kept local.

In the process network of the computation graph, the actors sharing the same state are joined by a special kind of channels – the state channels $\mathbf{Q}_{\mathbf{state}}$, which form the complementary subset of $\mathbf{Q}$, thus $\mathbf{Q} = \mathbf{Q}_{\mathbf{state}} + \mathbf{Q}_{\mathbf{comm}}$. The state channels representing one state form a cyclic path. One channel in the sequence is marked as the *first* channel. It is defined as the channel preceding the first actor in the order imposed by the state. One can see an example of state channels in Figure 2.3(a).

Definition: A state channel q_{state} is identified by a triplet $(m, v_{\text{prod}}, v_{\text{cons}})$. It enforces the mapping flow to assign both the producer v_{prod} and the consumer v_{cons} to the same processor. This channel is associated with the state data structures in the processor's local memory. Together with the other state channels, it enforces a cyclic order of execution, which we call the *state consistency order*. For the channel marking *m* of a state channel, it holds that $m \in \{0,1\}$. If $m = 1$ then the channel is the first channel in the consistency order.♦

So far we have mentioned only one guideline to identify the state data structures. The scope of the state is, however, broader than the algorithmic cyclic ordering. In general, it is also possible that some actors can execute concurrently, and their communication does not follow a FIFO pattern, which is the only pattern suitable for the communication channels as defined in our model. The FIFO pattern, for example, can be violated in the case when actors load and store

data at random positions in a table that survives one actor execution. If it can take an arbitrary number of iterations of the loop of interest before the data stored at the given table entry by one actor execution will be loaded by another one then no FIFO pattern can be established. To follow our approach, what one should do in such a case is to consider the mentioned table as the state. This is possible if there exists a cyclic ordering of the actor executions that ensures consistency of accesses to the table. In that case, the designer can model this ordering using state channels.

Having discussed in detail the elements of the implementation process network that are responsible for communication, let us turn to the elements responsible for computation. As for the processes, it is the best to introduce them in the context of run-time scheduling and we do that in the next subsection. In the rest of this subsection, we introduce the computation actor bodies, which are building blocks for the processes.

Definition: A computation actor body, for simplicity also identified by the corresponding **computation actor** $v \in \mathbf{V}_{\mathbf{comp}}$**,** is defined as a tuple $< APA, AT, \mathbf{AI}_{\mathbf{comm}}, \mathbf{AO}_{\mathbf{comm}}, \mathbf{AS}_{\mathbf{state}} >$, where APA – is the actor processing algorithm, AT – is the temporary data structures of the algorithm, $\mathbf{AI}_{\mathbf{comm}}$ and $\mathbf{AO}_{\mathbf{comm}}$ are the sets of communication inputs and communication outputs and $\mathbf{AS}_{\mathbf{state}}$ is the set of state data structures. A *communication input* / *communication output* is identified by specifying one of the communication channels and, within that channel, a communication transfer where the given actor is the consumer/producer. A state is identified by specifying a pair of state channels for which the actor is the producer and the consumer. ♦

We have used the term ‘actor body’ to stress the reference to the implementation of an actor and to suggest that the body contains the actor’s *APA*. From this point on, as we already have done before a few times, we always refer to the computation actor body just as computation actor, or just an actor.

For each actor, the run-time scheduling should be aware which inputs, outputs and states belong to that actor and which *APA* should be run for it. Prior to starting an actor execution, the run-time scheduling must ensure that:

1) there is at least one data token available at each communication input;

2) there is space for storing one data token available at the communication outputs;

3) the state consistency order is respected.

This way, the computation actor can run from the entry point until the exit point without any blocking, which could happen otherwise due to synchronization on communication inputs and outputs. Note that when saying ‘there is a data token/space’ and ‘available’ we implied ‘present in the local memory of the processor where the actor runs’.

A computation actor satisfying the abovementioned requirement is seen as *ready for firing* at the given moment of time (traditionally, the starting of actor execution is called ‘firing’). The reason we put this requirement is fundamental: we choose to exclude the delays due to other activities from the total delay of the actor. This is a way to separate different issues, by putting external timing factors outside the actor delay to deal with them separately. We only include into the computation actor the timing factors that have to do with processing carried out by the actor itself. Note that this is different from the classical real-time scheduling, where a task can be blocked when accessing a shared resource.

Doing the synchronization on the communication channels before the beginning of the actor execution is natural for HSDF models, and as we see in the next subsection, it is reflected in our

HSDF graphs based on what we call the *firing procedure* of HSDF actors. However, these assumptions might seem to be somewhat restrictive for a general parallel software routine, which may contain synchronization inside its body. Nevertheless, such a routine can be converted into data flow actors either by splitting them into different actors at every synchronization point or by moving the synchronization earlier, to the beginning of the routine.

Note also that, here, we implicitly made an assumption that all the actor instruction code and the actor data fit into the local memory of the processor and thus no caching is required. Caching could result in unpredictable actor delay. Dealing with limitations of the processor local memory size is a fundamental issue that is outside the main scope of this thesis. To justify our assumption, we take a hypothesis that, due to the limited size of the loop of interest, most of the actor code and data can be handled with priority and have a special place in the local memory, reserved before the application starts. Another hypothesis is that if there are still local data structures that pose problems in fitting them to the local memory, then they can be located in remote memories and accessed using FIFO data communication, explicitly modeled in the implementation process network. Such modeling was proposed by Sander Stuijk *et al* in [85] and further elaborated in [90].

2.1.4 Run-time Scheduling, Processes and their Budgets

Both the computation actors and the communication transfers have to be scheduled on their resources. In this section, we focus on the scheduling of the computation actors, postponing the explanation of communication scheduling until Section 3.4, where we introduce the necessary details about the architecture of the network-on-chip.

In streaming applications, the same set of computation actors is executed repetitively. For such applications, [83, §4] proposes the following classification of the multiprocessor run-time scheduling methods:

- fully static,
- static order,
- static assignment,
- fully dynamic.

In a fully static method, every processor has a fixed set of computation actors and the starting times of every execution of every actor are fixed and predefined. In static order scheduling, only the cyclic order in which the actors execute is fixed per processor, but their starting times are determined dynamically; they start as soon as their turn in the order comes and the input data is available. In static assignment scheduling, the assignment of actors to processors is fixed but the actors assigned to a given processor may be scheduled in any order and preemption may be allowed. The computation actor to be scheduled at run time is chosen by a criterion that should at least give every ready actor a fair chance to execute. Thus static assignment supports concurrency between actors assigned to the same processor, and its advantage over the static order case is due to the fact that it avoids the situation where a ready actor should wait for a non-ready actor just because the non-ready actor comes earlier in the static actor order assigned to this processor. Finally, in a fully dynamic method, the set of computation actors assigned to a processor may change at run time.

Recall that the reconfigurable streaming programming model, which we have assumed – see Figure 1.4 – does not change the assignment of computation actors to processors within one configuration. Although at the reconfiguration points the assignment can be changed, this happens relatively rarely; we leave the reconfiguration and the fully dynamic scheduling beyond the scope of this thesis. Also the fully static option is clearly of no or little use for us because we support applications with dynamic data-dependent execution delays.

Therefore, the choice that remains is between static order and static assignment. We support a combination of both. We apply static-order scheduling within processes and static-assignment scheduling between processes. Hereby, because we do not mix the actors of different applications in one process, we ensure that no order is imposed between the actors of different applications, so that the applications can run concurrently with respect to each other.

Now the time has come to give a more concrete specification of the term 'process'.

Definition. A process $p \in \mathbf{P}$ is defined by a tuple $< \mathbf{VP}, \mathbf{vp} >$, where the first element, denoted $\mathbf{VP}(p)$**,** is a subset of computation actors of the given implementation-enhanced model **GPQ** and $\mathbf{vp}(p)$ is an ordered sequence of elements in $\mathbf{VP}(p)$. That sequence is either empty (if no ordering has been enforced yet for the given process) or it orders all the elements of $\mathbf{VP}(p)$**:** $\mathbf{vp}(p) = (\{vp(p)\}_1, \{vp(p)\}_2, ..., \{vp(p)\}_{VP})$, where $\{vp(p)\}_i \in \mathbf{VP}(p)$ ♦

Note that there is a clear similarity between a process and a state consistency order: both impose a cyclic order on a subset of actors. However, the basic difference between them is that the states are *implementation constraints*, coming from the specification of the application, and the processes are *implementation entities*, i.e. they reflect choices made for the implementation. When the intra-application mapping flow forms processes consisting of multiple actors, it ensures that the process ordering is compatible with the state consistency ordering.

To schedule multiple processes at the same processor, we assume each processor has a *local scheduler* that implements the static assignment scheduling. A local scheduler can be implemented as a software real-time kernel, interleaving different tasks on a general-purpose processor, or as a hardware wrapper, interleaving different data streams passing through the same function hardwired in a domain-specific processor.

Let us compare the static-ordering and static-assignment scheduling by taking two extremes. If just one process runs on a processor, then this is pure static ordering. If there are multiple processes containing one computation actor each, then this is pure static assignment. The advantage of the pure static assignment is maximum concurrency, which is potentially better for the system performance if the context switching overhead due to different processes of (possibly) different applications that running in parallel is managed efficiently. The disadvantages of the static assignment scheduling are context switching overhead and concurrent resource sharing.

Recall from Chapter 1, that due to concurrent resource sharing, non-functional timing dependencies may arise between the tasks of different applications, posing difficulties for performance analysis. To avoid that problem, we decided to use resource budgeting. Therefore, we support only a certain class of static assignment scheduling methods, which we call *budget provision methods*. Using those methods, we can provide guaranteed performance, although the price that we pay for that is the risk that our performance estimations may sometimes be very pessimistic.

In our scheduling approach, budgets are assigned per process. Suppose that a certain subsequence of processor instructions in process *p* uses *t* processor clock cycles. Suppose that

budget B_p is the number of the processor cycles per second assigned to process *p*. In the ideal case, when the processor load is infinitely divisible, this would mean that the execution of that sequence would take time $d = t/B_p$. In practice, the schedulers can only provide some quanta of time to the processes, and therefore the value of *d* only comes in some neighborhood of the ideal value, such that:

$$t/B_p - \breve{q}_p \le d \le t/B_p + \hat{q}_p \qquad (2.2)$$

where $\breve{q}_p$ and $\hat{q}_p$ are positive constants, depending on the scheduling method and its settings. It is important to stress that B_p, $\breve{q}_p$, and $\hat{q}_p$ are fully controllable settings, independent on how much the multiprocessor system is loaded by applications at any given point of time. This enables us to reason about the performance of different applications independently, which is an important requirement set in Chapter 1. For conservative performance analysis, one can use $t/B_p + \hat{q}_p$ as estimate of d. (This is true because, as explained below, our framework is free of scheduling anomalies.)

We say that a computation actor is *enabled for firing* when that actor's turn in the process comes and it becomes ready. If an actor of process *p* uses *t* processor cycles, one can use Equality (2.2) to provide bounds on the time interval between the moment of time the actor is enabled and the moment of time the actor finishes.

There are different scheduling methods that can ensure budget provision. Clifford Mercer *et al* [59], in effect, introduce budget provisioning for multimedia applications and propose a correspondent modification of classical rate-monotone (RM) scheduling. In that work, the processes are presented to the scheduler as tasks whose computation time is proportional to the budget assigned to the processes. Orlando Moreira *et al* [64] propose a simple practical budget provisioning scheme based on round robin (RR) scheduling. As for this thesis, we assume TDMA (time division multiple access) scheduling, which e.g. is the same as the TDMA scheduling assumed by Sander Stuijk in [88] and [90]. When compared to RM, TDMA is simpler, because it offers fewer fine-tuning settings, and, compared to RR, TDMA works better than RR when actors have highly dynamic data-dependent delays. The point is that, unlike RR, it can preempt the actors and thus it avoids that the actor delay growth in one process considerably delays another process. We give more information on the TDMA scheduling in Section 3.1.3.

The multiprocessor-scheduling framework introduced in this section has an important property: it is *free from scheduling anomalies*. A scheduling anomaly (see e.g. [80]) is a phenomenon that may occur in multiprocessor scheduling. It manifests itself when faster execution of some tasks results in later completion of some other tasks. The absence of anomalies is favorable for ensuring a guaranteed system performance and it is proven in Section 2.2.4, where we show that our overall performance metrics increase monotonically when the actors execute faster.

2.2 The HSDF Graph: Timing Behavior and Performance Analysis

In this section, we focus on the HSDF graph, **G**, from an implementation-enhanced HSDF model **GPQ**. The HSDF graph models the behavior of the application's loop of interest. As suggested by Formulae (2.1), it can be partitioned into multiple parts, each modeling a process or a channel within one single model of computation.

For the HSDF graph model of computation, there exists a sound theoretical basis for performance analysis. In the following subsections, we describe the timing behavior, the structure and the relevant performance analysis aspects.

2.2.1 The Basics of HSDF Timing Behavior

Figure 2.4 shows two HSDF graph examples. Recall that the basic elements of HSDF are actors $v \in \mathbf{V}$ and edges $e \in \mathbf{E}$, where $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$.

By default, the graph edges represent the communication of data between actors, and we call them *data edges*. However, dataflow graphs also know edges introduced to restrict the actor execution order, or *sequence edges*. Note that both edge types have the same behavior, and we distinguish between them only for convenience. In contrast to the channels, the edges cannot be seen as implementation entities; they are primitive abstractions used to model the behavior of the channels and processes.

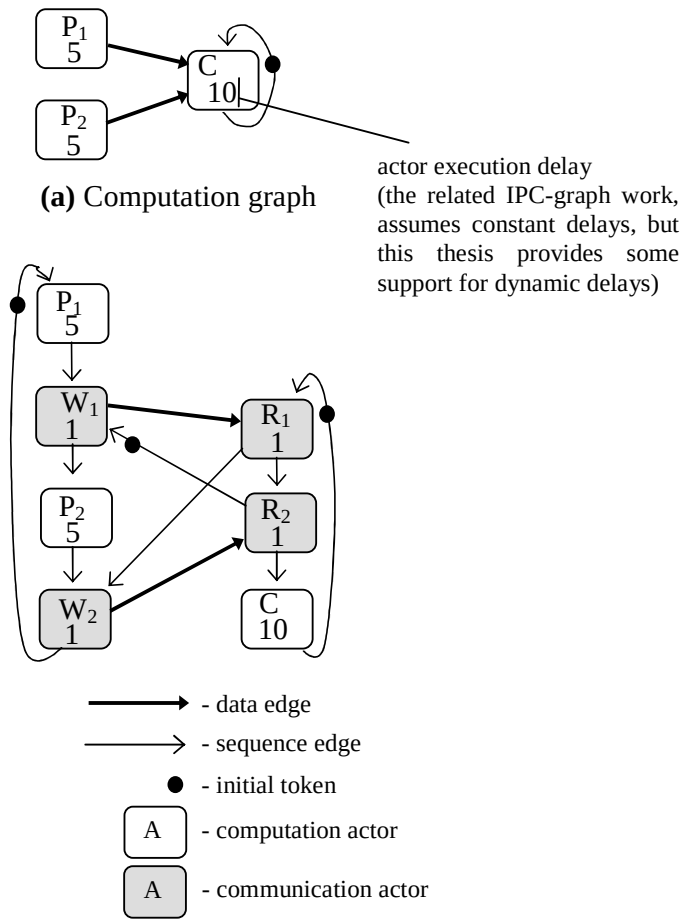
Every edge $e = (v_i, v_j)$ can potentially transport any number of tokens from actor v_i to actor v_j, and can contain *initial tokens*, which are present on the edges at start time. Recall that we call the number of initial tokens on an edge the edge's *initial marking*, usually denoted m or $m(e)$. The edges are directed, and the actor at the source of an edge is called the edge *producer*, and the actor at the sink of the edge is called the edge *consumer*. For convenience of explanation of the token order and timing, we assume that the edge producers annotate the produced tokens with *order labels* using the series of integers: 0, 1, 2, 3,… and *production times*, e.g. '1 ms', '2 ms', '3 ms' etc (the production times are not necessarily equidistant). The tokens appear on the edges at the times corresponding to their production times. If an edge contains m initial tokens, we assume that their order labels are $-1, -2, \ldots, -m$ and their production time labels are all 0.

Now we are going to explain the behavior of the HSDF actors. Hereby, it may seem that there is too little visible correspondence between the behavior of the computation actors, as explained earlier in this chapter, and their HSDF prototypes, but that correspondence is clarified in the following subsections.

To explain how actors consume/produce tokens and perform computations, we assume that each actor has its own execution counter n, originally initialized to 0. The actor behavior can be described using two procedures: the firing procedure, which initiates the actor executions, and the execution procedure.

The *firing procedure* consists of four steps:

1) at each input, wait until the production time of the token with label $n - m$, where m is the initial marking of the corresponding input edge;
2) start another execution procedure, whereby the execution gets index n;
3) annotate the input tokens obtained from step 1 as tokens *captured* at execution n;



(a) Computation graph

(b) IPC graph, modeling an implementation in a multiprocessor platform

Figure 2.4 HSDF graphs of a producer-consumer example

4) increment n by 1 and return to step 1)

Here only step 1) may take some time, steps 2), 3) and 4) are immediate. Note that step 3) only initiates an execution procedure, but does not wait until it completes. Multiple execution procedures can be initiated in parallel.

The *execution procedure* of actor v_k – also referred to simply as *actor execution* – takes index n as argument. It also consists of four steps:

1) wait for time $d(v_k, n)$, where $d(v_k, n)$ is defined as the function that models the delay of the actor execution;

2) consume the tokens captured at execution n, one token at each input;

3) produce one token at each output, annotating all the new tokens with label n and the current time;

4) terminate.

Note that, according to this definition, actors postpone the consumption of the captured input tokens until the end of the given execution, when they also produce output tokens. In the beginning of execution, the input tokens are only '*captured*'. Every token is captured only once,

to be used by only one actor execution. After being captured, the token continues to exist until the execution consumes it, in the end of execution. Although the moment of time when the tokens are consumed is not important for the timing behavior of this model of computation, our assumption that the token is consumed in the end of the execution procedure is in line with the behavior of actor implementations and is convenient when we explain the modeling of occupation of the memory buffers.

In the execution procedure, we have introduced function $d(v_k, n)$, which for every actor v_k defines a sequence of delay values in subsequent executions. We call that function the *actor execution delay*. In our implementation-enhanced HSDF models, the actor delay is determined by one of the timing modes from the set of timing modes **T**. Different timing modes are used for different purposes in our implementation trajectory. The timing modes are introduced in more detail in Section 3.1.

As for the edges, they do not have any delay, and the tokens produced on an edge become immediately visible to the consumer's firing procedure.

In HSDF models, a key notion is the notion of an *iteration*. HSDF iterations are labeled with the same index as actor executions. Iteration n is a set of actor executions that all have the same index n. We call a semantically defined set of subsequent HSDF iterations an *execution run*.

The reader can probably already see an analogy between the properties of HSDF graphs and the properties of the loop of interest. Indeed, the HSDF model also contains a fixed set of actors that are executed unconditionally, once per iteration. We did not introduce any external inputs and outputs for HSDF models, because, just like the loop of interest, they are assumed to execute autonomously. The terms 'iteration' and 'execution run' have the same meaning for the loop of interest and the HSDF model.

Here we should put an important remark that, unlike the HSDF actors, the actor bodies, which implement the computation actors, do not label the tokens with the labels representing their order. The FIFO property of the communication channels ensures the proper order of data production and consumption. We will see later in this section that in the important subclass of HSDF graphs, used for the performance analysis, the same FIFO property holds for the edges, and no ordering labels are really necessary for the tokens.

Note that from this definition of HSDF actors we see that the actors are not allowed to synchronize with each other using a general shared memory model, but all the synchronization should be organized using the edges, thus implying the FIFO order of event handling. We do not support any synchronization schemes that cannot be modeled using dataflow edges. Any non-FIFO forms of communication happening inside the loop of interest should be handled by pre-scheduling and hidden from the HSDF model by abstraction.

HSDF models are used in our work to model both the computation and communication parts of the application, both before the mapping and after the mapping, as we see in the next two subsections.

2.2.2 Computation Graph

We use the term 'computation graph' not only for the implementation-enhanced model **GPQ** prior to mapping, but also for the basic HSDF graph **G** contained in that model. In the latter meaning, the computation graph is an HSDF graph that expresses the behavior of the loop of interest prior to mapping, when there is no processor assignment, no budgets and no

communication through the network. It represents the computation actors and the data/state dependencies between them that follow from the application algorithm implemented in the process network **PQ**.

In the computation graph, the data edges represent the communication channels, the sequence edges represent the state channels and the computation actors represent the processes. The data edges get the same initial marking m as the communication channels. A sequence edge gets marking $m = 1$ if it models the first channel in a state consistency order, and otherwise it gets marking $m = 0$. A cyclic path (cycle) containing only one initial token forces the actors to execute sequentially in a cyclic order. The computation graph contains only the computation actors, and no communication actors.

For example, Figure 2.4(a) shows a simple computation graph with three actors, namely, two 'producers' ('P_1' and 'P_2') and one 'consumer' (C). In each iteration, first each producer must produce a data token and then the consumer may execute and consume the data tokens. The consumer has a sequence edge that indicates the state dependency of the current execution on the previous execution.

As already mentioned earlier in this chapter, the intra-application mapping flow can be reflected by a sequence of transformations of the HSDF graph, starting from the computation graph and finishing by the IPC graph. In Chapter 3, we consider a mapping flow that can be appropriate for our generic multiprocessor network-on-chip platform and the construction of the correspondent IPC graph. Now we introduce IPC graphs, using the original work summarized in the book by S. Sriram and S. Bhattacharyya [83] as the illustration example, indicating the concepts that we can immediately borrow from that book.

2.2.3 Modeling the Computation and Communication together: IPC Graph

Also the term 'IPC graph' has in our work a dual meaning: a model **GPQ** defined in the previous section as well as the graph **G** contained in it. In the latter meaning, an IPC graph is an HSDF graph that models the execution of the application on a multiprocessor architecture [5], [83 §7]. In the multiprocessor platform assumed in [5], [83], the communication is realized through a global bus and global memory, it supports no budgeting and assumes only one process per processor. However, in their work as well as in our work, an IPC graph can be seen as the result of a transformation of the computation graph, whereby extra edges and actors are added to the original graph. The IPC graph actors must have two essential properties:

1) the delay of an actor execution is *independent of the starting time of the execution*; and indeed, as we described, in the implementation, the delay can be bounded from above by expression $t/B + \hat{q}$, which, being invariant on the absolute starting time, could be used as a model for HSDF actor delay, although for the TDMA scheduling we use a tighter upper bound (see Chapter 3);

2) the time for waiting until the actor gets ready for execution (the blocking time) is not part of the actor execution delay; in the HSDF graph, the readiness on the inputs is ensured by the incoming data edges; the readiness on the state is ensured by the incoming sequence edges. The incoming sequence edges also ensure the readiness on the outputs, as it is shown below for the bus-oriented IPC graphs of [5], [83].

Three conditions are missing in the computation graph to ensure that an execution of a computation actor in the HSDF model **G** starts, ideally, at the same time as when it starts in the process network running in the platform. Because these conditions are rooted not in the application functionality, but in the implementation, we call them implementation conditions.

Remark. Implementation conditions for HSDF modeling:

1) the communication delays should be taken into account;

2) the actor should be ready also on the communication *outputs* (the communication resources should be ready to accept the data being written to the outputs);

3) the actor should be enabled, i.e. its turn should come to execute in its process. ♦

These requirements are rooted not in the application functionality but in the implementation, and they are taken into account in the IPC graph. Let us now take a look on how it is done in the IPC graphs in the previous work, [83], explicitly defining the elements that are also re-used in our work.

To take the first condition into account, in addition to the computation actors, coming from the computation graph, IPC graphs contain *communication actors*.

Definition. Communication actors $v \in \mathbf{V_{comm}}$ are actors modeling the delays of the communication transfers **TQ** between actors running on different processors[9]. The communication actors copy the data from one physical memory to another across the communication network. ♦

In the previous work, the communication actors are 'write' and 'read' actors – see Figure 2.4(b). A 'write' actor copies one data token from the local memory of a processor to the global memory. In the example, for each 'write' actor, there is a corresponding 'read' actor, which copies the data token from the global memory to the local memory of another processor. These actors are annotated with appropriate delays, satisfying the first of the above three implementation conditions.

To satisfy the other two conditions, in the previous work, for each processor and for the global bus, a cycle is introduced in the graph, being similar to the cycle enforcing the state consistency order. We call those cycles, the 'process cycles' and the 'bus cycle'. Recall that, to introduce such a cycle, sequence edges are added into the graph, whereby the actors are put in a specific static order, and an initial token is placed on the sequence edge at the input of the first actor in the order.

The bus cycle includes all the communication actors. By ordering them, this cycle eliminates the bus conflicts between the bus transactions. It ensures that the output data tokens are written to the global memory only when the global communication and memory resources are available. This is in line with the second implementation condition, and, in addition, this ensures that the communication delays are also independent of the time when the communication actor starts.

For example, Figure 2.4(b) shows an IPC HSDF graph for the example of Figure 2.4(a) assuming a two-processor case, where the producers are assigned to one processor and the consumer is assigned to the other one. It includes write actors (W_1 and W_2), read actors (R_1 and R_2) and two data edges, derived from the computation graph, to pass data tokens from the

[9] The communication transfers between actors mapped to the same processor do not involve any delay, so they are modeled only by edges.

producers to the consumer. The graph contains one bus cycle (W_1, R_1, W_2, R_2)*. Note that only two sequence edges are introduced to enforce the cyclic order there, namely, (R_1,W_2) and (R_2,W_1). The other two sequence edges are not necessary, due to the presence of the data edges.

In our work, we do not use the bus cycles, because we propose IPC models for on-chip networks. In Chapter 3, we model communication channels by a structure $\mathbf{GQ}(q)$ (a channel macro) that is much more complex than a cycle of sequence edges.

In the previous work, a process cycle enforces an order in which actors execute on the given processor. In this thesis, we assume that processes can share the same processor, but because the processes get separate budgets, each process can be modeled by a separate cycle in graph **G**.

Definition. Process cycle $\mathbf{GP}(p) = < \mathbf{VP}(p), \mathbf{E}(p), m(p) >$ is a subgraph of **G** that models process *p*. Recall from the definition of process in Section 2.1.4 that **VP**(*p*) is the set of actors that belongs to process *p*, ordered according to the process order, determined by ordered sequence of actors **vp**(*p*). Therefore, if the process order is not empty, the set of edges $\mathbf{E}(p)$ joins the actors in a path in the order defined by sequence $\mathbf{vp}(p)$, in addition also including an edge with marking 1 that joins the last actor to the first actor in the order. If the process order $\mathbf{vp}(p)$ is empty, then $\mathbf{E}(p)$ is also empty – that is the case for all processes in the computation graph, but it should not be the case for any process in the IPC graph.♦

In the example in Figure 2.4(b), there are two process cycles, namely, (P_1, W_1, P_2, W_2)* and (R_1, R_2, C)*. Note that if the computation graph contains channels that join actors that are finally assigned to the same process, then, at the end of the mapping flow – in the IPC graph – the processes enforce orderings that are consistent with all the data/state dependencies implied by those channels. Therefore, those channels are not needed and not present in the final process network. For example, in Figure 2.4, cycle (R_1, R_2, C)* is consistent with the cycle C* of the computation graph and therefore the channel from C to C is not present at the end of the mapping flow and is not represented in the IPC graph by any edge.

Note that Figure 2.4 assumes constant actor delays, which is in line with the previous work on throughput analysis of the IPC graphs. In this thesis, it is our goal to also support variable actor delays in performance analysis. A more detailed discussion on this subject is postponed until Section 2.2.6.

2.2.4 General IPC Graph: Restrictions and Properties

There are two basic facts about IPC graphs that make their performance analysis far from trivial. First, to mimic the behavior of the computation actors executed in a multiprocessor, IPC graphs assume so-called self-timed execution, presented in this thesis as default behavior of HSDF actors. In self-timed execution, each actor executes as soon as the actor firing procedure sees input tokens with specific order labels, without aligning the starting time to any periodic timing grid, as it is often the case in dataflow scheduling. Second, we have seen that the dependencies of actors in IPC graphs are cyclic. Fortunately, there exist theoretical studies concerning the timing properties of the self-timed execution of cyclic *static-delay* HSDF models, where $d(v_k, n)$ does not depend on *n*. Such HSDF graphs can be characterized by a steady-state throughput that can be calculated using efficient algorithms. The possibility to calculate the throughput is, after all, a major reason why IPC graphs have been employed.

In this subsection, we consider the restrictions and properties of ‘general’ IPC graphs, where $d(v_k, n)$ may depend on *n*. These restrictions and properties are needed to provide basic facts for

reasoning about IPC graphs, and they specify some features any IPC graph should possess, no matter whether it is an instance of the IPC model proposed in [83] or of the IPC model that we introduce in Chapter 3. In the following subsections, we describe the steady-state throughput analysis results that we apply and complement in this thesis. We provide either intuitive proofs or motivation of the mentioned facts, without giving formal proofs.

Postulate: An IPC graph is a strongly connected live FIFO graph:

1) a graph is *strongly-connected* if the following holds: for any two actors v_i and v_j that belong to the graph's actor set **V** there exists a path in the graph from actor v_i to v_j; this property follows from the fact that IPC graphs are built of substructures containing cycles;

2) an HSDF graph is *live* if any cycle contains at least one edge with a non-zero number of initial tokens; liveness means that any actor in the HSDF graph can eventually always fire again [4 §1]. We require this property to ensure that IPC graphs never deadlock;

3) an HSDF graph is a *first-in-first-out* (FIFO) graph if, when this graph executes, for any edge of the graph, the tokens produced earlier in time always have smaller order labels than the tokens produced later in time; this property reflects the FIFO order of processing of the stream elements by the computation actor bodies and the FIFO property of the communication channels; it is a fundamental requirement for the performance analysis of IPC graphs.♦

Note that an HSDF graph can be non-FIFO only if actors have dynamic execution delays. This is true because the firing procedure ensures by definition that each execution with index $n + 1$ starts no earlier in time than the execution with index n (the verb 'wait' in step 1 of this procedure implies going forward in time). Because each execution annotates the produced tokens with its index, a violation of the FIFO condition can only happen if, for some n, execution $n + 1$ completes earlier than execution n, but we have just seen that execution $n + 1$ could not have started earlier. Thus, to finish earlier, execution $n + 1$ has to take less time to execute: $d(v_k, n + 1) < d(v_k, n)$. Therefore, all HSDF actors with static delays abide the FIFO property.

So, for those IPC actors that have static execution delay, there is no problem, but what about actors with dynamic execution delay? Tokens can overtake each other only when multiple actor executions overlap in time. A sufficient condition that excludes overlapped actor executions is that there is a cycle in the graph that contains the given actor and has only one initial token. Obviously, this property holds for all actors in the IPC graphs discussed in the previous subsection, because each actor in those graphs belongs to a process cycle and every process cycle has only one initial token. Thus, all actors in those graphs hold a 'license' for having dynamic delays and still being FIFO graphs.

In the following lemma, we summarize the discussion on the FIFO property.

Lemma 2.1. (Sufficient condition for the FIFO property) An HSDF graph is a FIFO graph if any actor with dynamic actor execution delays is contained within at least one cycle that has only one initial token.♦

The FIFO property leads to a fundamental equality that traditionally serves as a foundation for reasoning about the timing behavior of HSDF graphs:

Lemma 2.2. (Evolution equation of a FIFO HSDF graph) Consider an HSDF graph **G** that possesses the FIFO property. Let $x_k(n)$ denote the time when actor v_k completes the execution with index n; let us define $x_k(n) = 0$ for $n < 0$. Consider actor v_i, and let $v_{j(1)}, v_{j(2)}, \ldots, v_{j(P)}$ be the list of the producers of all edges $(v_{j(p)}, v_i)$ in graph **G** that have actor v_i as the consumer,

$p = 1, \ldots, P$. Let m_p denote the initial marking of those edges. In that case, the relationship between the completion times of actor v_i and of all the producers of input tokens for that actor is as follows:

$$n \geq 0 \Rightarrow x_i(n) = \max_{p=1..P}(x_{j(p)}(n - m_p)) + d(v_i, n) \tag{2.3}$$

Equality (2.3) is called the *evolution equation* of actor v_i. ♦

Proof In a FIFO HSDF graph, the tokens being consumed by an arbitrary actor v_i at an arbitrary input p have the following property: their production times, $x_{j(p)}(n - m_p)$, are monotonically non-decreasing in n. This means that their maximum, given in Equality (2.3) is also monotonically non-decreasing. Now we observe that the three steps of the actor firing procedure imply that, in every iteration, this procedure does the following:

1) it either waits for the moment of time given by that maximum expression and starts exactly at that moment,
2) or it initiates an execution immediately if that maximum expression gives an earlier time than the current time.

However because the maximum expression is non-decreasing and because, in the first iteration, situation 1 is necessarily true, we see that situation 2 can never occur. Thus, we have proven that the execution starting time exactly equals the maximum expression from Equality (2.3), so the completion time is given in the right part of that equality. ♦

Remark (FIFO property and validity of IPC graphs) The reason we discuss the FIFO property is, first of all, the fact that the actors in general may have variable execution delays and, in general, this can lead to out-of-order production of tokens – i.e., the actors may violate the FIFO property. However, the implementation entities being modeled by the IPC graphs – i.e. the processes and channels – enforce the FIFO property of the communication transfers by construction. For example, as we will see in Section 3.4, in the point-to-point network-on-chip connections used to implement the channels in the generic platform assumed in this thesis, the data packets cannot overtake each other and always arrive in the same order as they depart. It is for this reason that we postulated above that the IPC graphs, which model the FIFO channels, should themselves possess the FIFO property. ♦

Remark (The longest path in the unfolded graph) Note that the evolution equation, Equality (2.3), has the form of the Bellman's equations [50] for the longest path lengths in an acyclic graph with weights equal to delays d. The nodes of that graph would correspond to actor executions in different iterations. We call that graph the unfolded graph and introduce it in Chapter 3. In Chapter 5, we use the graph unfolding to analyze the transitions between different steady states of the variable-delay HSDF graph. ♦

Remark (General evolution equations) For actors in more general HSDF graphs, which may or may not possess the FIFO property, we can write general evolution equations similar to Equality (2.3), but we need to introduce extra variables 'y', giving the starting time of actor execution. The equalities look as follows:

$$n \geq 0 \Rightarrow y_i(n) = \max\left(y_i(n-1), \max_{p=1..P}(x_{j(p)}(n - m_p)) \right) \tag{2.4}$$

$$n \geq 0 \Rightarrow x_i(n) = y_i(n) + d(v_i, n) \tag{2.5}$$

whereby Equality (2.4) models the firing procedure and Equality (2.5) models the execution procedure. However in practice one can expect IPC graphs that satisfy the conditions of Lemma 2.1, and thus we can avoid introducing extra variables.♦

We conclude this section with a fact that is also fundamental and that holds for any HSDF model where actors behave according to the definition given in this thesis – i.e., where they use the labels to process the incoming tokens in-order and follow the self-timed execution method.

Lemma 2.3 (Monotonicity of HSDF graphs) If one would increase the execution delay $d(n, v_i)$ of any actor v_i in any iteration n or postpone a start or completion of an actor execution, one would see that the completion times of all actors in iteration n and future iterations either stay the same or increase. Consequently, if one would decrease the delay of events in the model, the completion times can only stay the same or decrease.♦

Remark (Monotonicity and independence of delays on the starting times) Monotonicity is a consequence of independence of the actor execution delays $d(v_i, n)$ of the actor starting times 'y', which is implied by Equalities (2.4) and (2.5). One can prove the monotonicity by observing that operators 'max' and 'plus' involved in those equalities are monotonically non-decreasing functions on their arguments. This remark is important because, in practice, one can imagine that the delay of executing some task on a hardware resource can depend on the initial state of that resource. In our case, the resource budgeting ensures that each actor execution finishes within a time interval whose length is independent of the starting time. ♦

Remark (Monotonicity and absence of scheduling anomalies) The monotonicity property of IPC graphs implies that the scheduling framework modeled by an IPC graph is free of scheduling anomalies; see the remark at the end of Section 2.1.4.♦

The monotonicity property is needed in practice to prove that, in order to derive the worst-case execution time of the loop of interest, one can use the worst-case execution delays of the actors as the static execution delay annotations.

2.2.5 Static-delay IPC Graphs: Steady-state Timing Behavior

In this subsection we assume that the IPC model of the loop of interest has static (i.e. constant) actor delays. We also use the fact that any IPC graph is strongly-connected. The timing behavior of a strongly-connected graph with constant actor delays is, at least in the long run, periodic and the execution time of the loop of interest can be bounded from above by a simple analytical expression. We refer to the first property as *periodicity* and we use the second property (the analytically bounded execution time) as a very important ingredient of our performance analysis approach. The periodicity property gives us the *throughput* of the graph. Due to the monotonicity of HSDF graphs, if the static actor delays are worst-case, then the average period, the throughput and the execution time bound obtained from analysis give conservative estimates of those values.

Suppose an IPC graph is given. Let us consider a *simple cycle* in the graph, i.e., a cyclic path that does not include any actor more than once. The cycle may contain both data edges and sequence edges; in fact, the performance analysis does not distinguish between them. In the remainder of this section, we refer to simple cycles just as cycles. Let us define the *cycle length* as the sum of execution delays of the actors in the cycle. Because the execution delays are static, the cycle length is a constant value. Let us also count the number of initial tokens on the edges of the cycle and call their total number the *cycle depth*.

We define the *cycle mean* as the cycle length divided by the cycle depth. A cycle with the maximum value of a cycle mean among all cycles in the graph is called a *critical cycle*.

The *maximum cycle mean* (MCM) of the graph can be calculated in polynomial time [19]. Intuitively, it signifies the average time distance any initial token in the critical cycle has to travel, until it comes to the next starting position of another initial token (or itself). If the same would happen with every initial token in every cycle of the graph, the graph would come into the same state as where it started (a period is completed). The tokens of the critical cycle are the 'slowest' ones in this sense and, due to the fact that the graph is strongly connected, they constrain the speed of the whole graph.

Now we formulate an important corollary of the classical theorem about the periodic behavior of HSDF graphs, formulated e.g. as Theorem 3.112 in [4]. In Chapter 4, we come back to the theorem and formulate it in full detail, but here we only pick up a weaker statement, which nevertheless conveys the result that is most often used in practice.

Theorem 2.4 (Periodicity) Let **G**(**V**,**E**) be a live strongly-connected HSDF graph with static actor delays. Let λ be the MCM of graph **G**. Let $x_i(n)$ denote the completion time of the execution of actor v_i with index n. Then, we have:

$$v_i \in \mathbf{V} \Rightarrow \lim_{n\to\infty} \frac{x_i(n)}{n} = \lambda . \tag{2.6}$$

♦

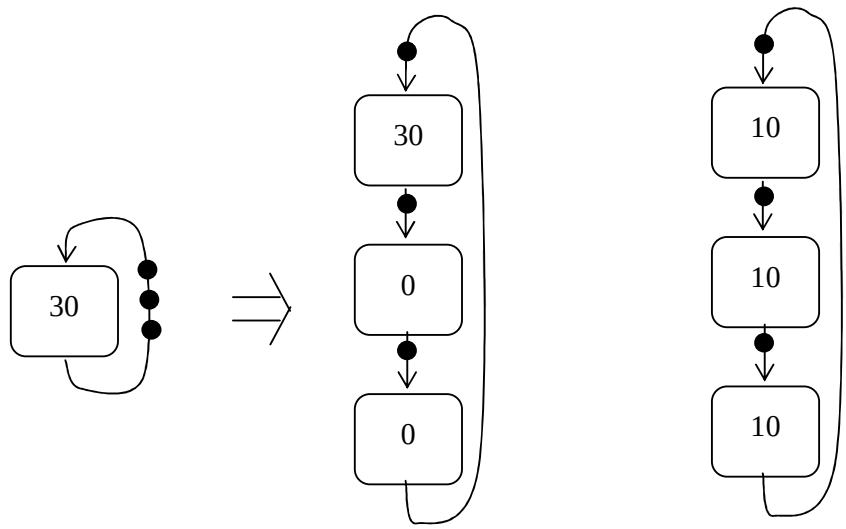
According to this theorem, the MCM is the average time interval between iterations of graph **G** in case the number of iterations on which the average is evaluated is large enough. Henceforth, we call it simply the *average iteration interval*. Note that the actual intervals between iterations may vary, despite the fact that the actor execution delays are static.

Note that for more general dataflow graphs, SDF (i.e. multirate dataflow), Theorem 2.4. does not hold in general, but it is still possible to calculate the average iteration interval by first translating the graph into an HSDF graph and then calculating the MCM. However, for such graphs, often in practice a different calculation method for the average iteration interval appears to be more runtime-efficient, i.e., state-space exploration, as demonstrated in the work of Amir Hossein Ghamarian [23], [22].

Now suppose that the amount of data in bytes produced by the loop of interest per iteration at the external outputs is constant; let us denote it $z(\mathbf{G})$ (Note that because we do not reflect the external outputs in the graph, this value cannot be derived from the graph but has to be annotated by the designer.) Let us define the *average throughput* θ of the loop of interest, in bytes per second, as the ratio between the amount of data produced by the graph at the outputs and the time interval within which they were produced, evaluated for a long enough interval of uninterrupted execution of the graph. Thus defined, average throughput is inverse proportional to the average iteration interval:

$$\theta = \frac{z(\mathbf{G})}{\lambda} \tag{2.7}$$

Definition (Execution time of *N* loop iterations) The *execution time* of an execution run of N iterations, denoted Δ_N, can be defined as the latest completion time of any actor execution within the first N iterations of the HSDF graph execution. ♦



(a) A cycle and the equivalent special cycle

$\lambda = 10,\ M = 3,\ \Delta_N = 30 \cdot \lceil N/3 \rceil$

(b) The balanced cycle with the same λ and M

$\Delta_N = 10 \cdot N$

(a lower bound to the result in (a))

Figure 2.5 A special cycle and a balanced cycle

The following lemma gives an important relationship between the execution time and the MCM value λ:

Lemma 2.5 (A lower bound on the execution time of *N* iterations) The execution time of any live static-delay HSDF graph **G** that contains at least one cycle is bounded from below by:

$$\Delta_N \geq \lambda \cdot N \tag{2.8}$$

where λ is the MCM of graph **G**. ♦

Proof Let us pick up a critical cycle in graph **G** and transform the graph by removing all actors and edges that do not belong to that cycle. This can only lead to a decrease in the value of Δ_N. Let M be the depth of the remaining cycle; then the length of that cycle is λM. One can further transform the remaining cycle such that Δ_N and λ stay invariant. Every edge with initial marking m more than 1 is split into m edges with initial marking 1 and a new actor with delay zero in between. Every chain of actors joined by edges with zero initial marking is replaced by one actor with the delay value equal to the sum of the delays in the chain. As a result, we obtain a cycle with the number of actors equal to M where M is the depth of the cycle, and where every edge has one initial token. Let us call such as cycle a *special* cycle. An example of the conversion of a cycle into the equivalent special cycle is shown in Figure 2.5(a).

One can show that for any number of iterations N the fastest cycle among all possible special cycles with depth M and length λM is the cycle whose delay is evenly distributed between the actors, i.e., where each actor has delay λ. Hereby under the 'fastest' cycle for the given number of iterations N we understand the cycle with the smallest execution time Δ_N. We call a special cycle with evenly distributed delay – a *balanced* cycle. An example of a balanced cycle is shown

in Figure 2.5(b). That balanced cycle has the same length and depth as the cycle in Figure 2.5(a), but it has an execution time that is never larger than the execution time of the non-balanced cycle. The general statement that the balanced cycle is the fastest cycle is intuitive and not difficult to prove, and we leave the proof to the reader. That statement also proves Formula (2.8), because the execution time of the balanced cycle is equal to λN.♦

Remark (The lower bound and availability of the initial tokens at time zero) The statement that the balanced cycle is the fastest cycle follows from the fact that in the definition of the HSDF timing model all initial tokens are assumed to be available at time 0. Thus, if we allowed arbitrary release times of the initial tokens in the HSDF graph, we would have to generalize this lower bound. ♦

Now we know that the execution time grows at least as fast as $\lambda \cdot N$. Can we bound that growth from above? From Equality (2.6), it follows that $\lim_{N \to \infty} \Delta_N / N = \lambda$. From this fact and Formula (2.8), we get the following result.

Lemma 2.6 (The bounds on the execution time) With the preconditions of Theorem 2.4, the execution time of N iterations of graph **G** is bounded as follows:

$$\lambda \cdot N \le \Delta_N \le (\lambda + \delta_N) \cdot N \qquad (2.9)$$

where δ_N is a sequence that converges to 0 as N goes to infinity. ♦

Lemma 2.6 is a corollary of Theorem 2.4 and Lemma 2.5. In the next subsection, we discuss this result and its relationship to the rest of this thesis.

2.2.6 Performance Analysis: Discussion, Objectives and Related Work

Due to the fact stated by Theorem 2.4 and the observation that streaming applications process long sequences of data, the research on multiprocessor mapping of DSP applications has focused on issues concerning the MCM of HSDF graphs, λ. At the same time, the bounds on the graph execution time, characterized by δ_N, have hardly received any attention, apparently because they are only significant for smaller N.

This situation, however, changes when we generalize the DSP applications that have, more or less, static actor delays to streaming applications characterized by highly dynamic variations of actor delays. Putting worst-case delay values and applying the static-delay analysis techniques can yield in general too pessimistic performance characteristics with arbitrarily large relative error. Therefore, for the dynamic case, we can refine the static analysis by splitting the long execution runs into multiple smaller ranges and make use of the approach known in calculus as integration. For each of the smaller ranges, δ_N can have a significant impact, because according to Lemma 2.6, δ_N is non-negative, and thus ignoring it is likely to lead to error accumulation. In Chapter 4, we consider the static-delay theory of HSDF graphs in more detail, whereby we also present our findings about δ_N. In Chapter 5, we study the transitions between smaller ranges and apply the integration principle. This way we generalize Formulae (2.9) and (2.7) such that, for dynamic-delay HSDF graphs, we get analytical expressions having at least the same but potentially much better accuracy than the case when we use (2.9) and (2.7) with static worst-case execution times. And, just like the worst-case estimates, our techniques also provide upper bounds on the execution time and average iteration interval.

Now, having mentioned some possibilities we are going to employ, let us step back and position them in terms of the related work on IPC graphs and in terms of the goals set in Chapter 1, Introduction; also let us be more specific about the means we use to achieve the goals.

In [83, §7.6], a survey is done on the existing performance analysis approaches applicable to dynamic-delay IPC models. The objective of the considered approaches was to find the average iteration interval, but now for the dynamic-delay case. Let us denote it Λ. Being an extension of λ for the dynamic-delay case, this performance metric has got considerable interest in research. In the survey of [83], the authors conclude that analytical approaches – all based on stochastic models – capable of computing Λ cannot be used in practice due to the non-tractable number of states in the *state spaces* employed in those approaches.

Therefore, in [83], they also considered approximations or *bounds* on Λ. Among those, the following are the basic bounds:

$$MCM(\mathbf{G_{max}}) \geq \Lambda \geq MCM(\mathbf{G_{ave}}) \geq MCM(\mathbf{G_{min}}) \qquad (2.10)$$

where $\mathbf{G_{max}}$, $\mathbf{G_{ave}}$, and $\mathbf{G_{min}}$ are static-delay graphs where worst-case, average and best-case actor delays are used instead of the real dynamic delays. All of the inequalities, except for the least trivial one, $\Lambda \geq MCM(\mathbf{G_{ave}})$, follow from the monotonicity property. Inequality $\Lambda \geq MCM(\mathbf{G_{ave}})$ deserves further attention, because it is counterintuitive and because it also says that, although a naïve 'practical' approximation of dynamic delays with average values cannot provide any assurance of analysis accuracy, one can prove that it leads to optimistic performance estimates. Note that the inequalities in (2.10) are only valid when the limit value Λ exists and when the number of iterations for which the average actor delays in $\mathbf{G_{ave}}$ are evaluated is large enough.

For conservative analysis – which is the focus of our thesis – in the survey of [83 - §7.6], the authors are particularly interested in the problem of giving an *upper* bound on Λ which would be accurate enough, because, as we already mentioned, using $\mathbf{G_{max}}$ can lead to very poor accuracy. However, in the survey the authors witness that attempts to give such a bound fail in practice, due to inaccuracy introduced when trying to approximate the huge number of possible states in the state space of the variable-delay graph by smaller computationally tractable models.

On one hand, the IPC analysis goals we set for ourselves are closely related to calculating an upper bound on Λ. Based what we said in the beginning of this chapter, the main goal of the performance analysis technique is to calculate a tight upper bound of Δ_N, i.e. the execution time of N graph iterations. Let us denote the average iteration interval of N iterations as Λ_N, where $\Lambda_N = \Delta_N / N$. Let us use notations $\hat{\Delta}_N$ and $\hat{\Lambda}_N$ for the upper bounds on Δ_N and Λ_N respectively. Thus our goal is to obtain $\hat{\Delta}_N$, which is equivalent to obtaining a value for $\hat{\Lambda}_N$, i.e. $\hat{\Lambda}_N = \hat{\Delta}_N / N$. If the latter has a limit value for $N \to \infty$ then it is an upper bound on Λ. This shows the similarity between the goal of our performance analysis and the goals of the other researchers that are surveyed in [83 - §7.6].

On the other hand, there is also an essential difference between our goals and finding bounds on Λ. Recall that our IPC models are primarily meant to be *run-time* models, using available run-time a-priori information on input data complexity characteristics to calculate run-time approximations of performance metrics – see Figure 1.8. In this way, our work is different from the related work surveyed in [83], which we see as pure *design-time* analysis trying to characterize the whole set of possible run-time situations that comply with a certain stochastic

model. Since we can use run-time information, we can avoid the problem of unmanageable state spaces.

Fortunately, as mentioned in Chapter 1, in our case, the availability of the run-time information reduces the uncertainty about the run-time behavior, and in this way we are in a better situation to predict the performance than the pure design-time methods. We also involve a state space in our model, but we can keep adjusting the state definitions at run time, based on the available run-time information. Instead of trying to oversee an arbitrary infinite execution run with a huge number of states, we oversee a particular finite one with a small state set adjusted at run time, so we are better off. In general, that is a reason to believe that our approach can reach a sufficient level of accuracy at low computational overhead. Our case study, presented in Chapter 6, witnesses sufficient accuracy results for implementing a highly dynamic streaming application.

2.3 A Mathematical Framework for Implementing Applications

In the previous sections, we explained how analysis models are constructed and used to obtain performance metrics, which is only one of the two important goals of performance analysis. In this section, we look at the basics of the second goal, namely, providing optimization guidelines. Recall that we have introduced the use of performance analysis both for run-time adaptation (Figure 1.8) and for intra-application mapping (Figure 2.2). In this section we focus on the former. In particular, we refer to QoS adaptation, because that has been subject of our studies. Nevertheless, we believe that our techniques are more widely applicable, because we have seen evidence in the literature that adaptation of budgets and of frequency/voltage (see Section 1.4.2) – have much in common with QoS adaptation.

A considerable part of this section is dedicated to the notion of ‘parameters’, which has a special meaning in this thesis and which is essential for run-time adaptation. Strongly coupled with this notion is the notion of ‘parameter coefficients’, also explained in the next section.

2.3.1 Introduction to Parameters

The run-time characteristics of the input data we referred to at the end of the previous section, are, in fact, *complexity parameters*[10], or parameters. They are reflected in our implementation-enhanced HSDF model, being involved in some timing modes in the set of modes **T**. We introduce the parameters here, because this is important for understanding the practical use of our modeling approach. We start by giving an example of a parametric performance analysis model.

Example (Parametric performance analysis). A. Bavier *et al* [6] studied the run-time prediction of MPEG-2 video frame decoding times on a single processor. Their prediction uses extrapolation of the decoding time measured for the previous frames. However, they have ascertained the fact that previously measured values alone do not carry enough information, and satisfactory predictions could only be obtained when using a priori run-time information about three different block types (I, P, and B) in the video frames. In their best prediction model, the frame decoding times Δ_{frame} are evaluated as a linear expression:

[10] These parameters are related to the theory of algorithm complexity.

$$\Delta_{\text{frame}} = \alpha_{\text{I}} \cdot I + \alpha_{\text{P}} \cdot P + \alpha_{\text{B}} \cdot B \qquad (2.11)$$

where *I*, *P*, *B* are *parameters* giving the number of blocks of type 'I', 'P' and B in a frame, provided at run time in the frame header. Character 'α' is used to denote the *parameter coefficients*, giving the processor cycle cost estimates for each block type.

In [6], the authors show that the parameters capture most of the dynamic changes in the frame decoding times. The parameters can change in an unpredictable way; therefore, the information about their values has to be provided in the frame headers. The values of parameter coefficients do not change so much, and, depending on the desired accuracy, one can consider the coefficients to be static or one can derive them based on extrapolation. ♦

The parameters are variables dependent on the application input data; they are specific for the given application. They count the number of times certain conditions in the state of the application algorithm repeat themselves when the input data is processed according to the application algorithm; the impact of the specific parameter is given by the coefficient. Thus the values of the parameters do not depend on the hardware architecture, but the values of the coefficients do. In fact, we referred to the parameters in Chapter 1 when we talked about the representation of performance metrics by a linear expression $\Sigma C(i) \cdot F(i)$. In that context, the $F(i)$ are parameters and the $C(i)$ are coefficients.

Including the parameter values into the input data headers involves certain overhead in the number of bits, being undesirable, certainly for video coding applications, striving to achieve good data compression. Nevertheless, we see that the MPEG-4 standard provides for inclusion of a certain set of complexity parameters into the video frame headers as an optional extension [42]. A question arises: what is the motivation behind paying the overhead of parameters?

2.3.2 Parameters and QoS Adaptation

Before giving an answer to this question, let us consider the basics of QoS adaptation, because it provides a motivation for applying the run-time performance analysis proposed in this thesis. Hereby, let us, first of all, bring into attention the fact that there are two basic kinds of complexity parameters. We call them *active parameters* and *passive parameters.* The active ones are those that can be adjusted to scale the quality of application output, and the passive parameter values are characteristics that cannot be changed. Applications having active parameters are called *scalable*. Among modern multimedia applications, 3D-graphics applications can be distinguished for good scalability. There, an example of an active parameter is the number of triangles used to render a 3D object and an example of a passive parameter is the percentage of the video screen occupied by the object, as shown in a paper by J. Bormans *et al* [11].

Given a scalable application, the objective of QoS adaptation is to maximize an audio/visual quality metric while meeting the real-time constraints – which means maintaining the required minimum throughput. When we have hard real-time constraints, the minimum throughput constraint is rigid. Thus, one can think of a combinatorial optimization problem, where an expression of the form $\Sigma C(i) \cdot F(i)$ – which can be used to express the throughput – is involved to express the throughput constraint. In such a problem, the active parameters are variables $F(i_{\text{active}})$ to be optimized and the passive parameters $F(i_{\text{passive}})$ are values specified in the problem instance. To complete the picture of a combinatorial optimization problem, one can imagine that the objective – a quality metric – can also be expressed in terms of parameters; examples are given in the works of J. Bormans [11] and N. Ngoc [69]. In fact, the QoS adaptation approach we have

just depicted fits the general adaptation framework we have described in Chapter 1 using Figure 1.8.

[25] and [41], in effect, present two representative variations of the adaptation framework of Figure 1.8, which avoid the need in encoding the a-priori parameter values in the input data frames. S. V. Gheorghita *et al* [25] automatically insert workload predictors into source code, which, as soon as the most influential conditional branches in the execution run are taken, send corresponding signals to the adaptation manager. Those signals imply that the parameters in the current run can take values only in a certain limited range, which is enough for the adaptation manager to estimate the workload and adapt the operating voltage and the clock frequency. Unfortunately, the usage of workload predictors is only possible to control execution runs consisting of one iteration of the loop of interest, so encoding of parameter data in the headers is still necessary to control longer execution runs. Yicheng Huang *et al* [41] present an *offline* video decoding QoS adaptation framework that runs the decoding application on a high-performance compute server such as a PC, in order to extract the application's workload parameters, use them to estimate the workload for a certain embedded system architecture and then adapt the quality such that the workload of that system does not violate a workload constraint. Because in that approach the parameters are extracted offline, they do not need to be encoded in the input stream headers; but nevertheless this approach fits into the framework shown in Figure 1.8. The approach of [41] demonstrates the usefulness of a-priori knowledge of parameter values for workload estimation/ However, because one cannot accompany every embedded multimedia device with a compute server that would perform the quality adaptation for that device, encoding the parameters in the input data frame headers is a more acceptable solution in general case.

The overhead of parameters in the input data headers can be justifiable in many practical cases. We also believe that the parameter overhead can be efficiently controlled. For this, one can reuse various techniques invented and widely applied to encode the useful multimedia content, video and audio. Advanced encoding of parameters is, however, beyond the scope of this thesis and is a subject for future work.

2.3.3 Parameters and HSDF Performance Analysis

The purpose of this subsection is to briefly show how HSDF performance analysis is extended based on the parameters defined in our parametric timing modes.

Our parametric timing models distinguish two levels of hierarchy in the loop of interest: the loop-level and the actor-level. Respectively, the parameters are also divided into two levels.

The loop-level parameters count the loop iterations having specific properties and their coefficients provide the impact of those iterations. The upper bound on the loop execution time we obtain in Chapter 5 has the form of a linear expression, which can be seen as a generalization of Equality (2.11):

$$\Delta_N \leq \alpha_{\mathrm{sc}1} \cdot I_{\mathrm{sc}1} + \alpha_{\mathrm{sc}2} \cdot I_{\mathrm{sc}2} + \ldots \qquad (2.12)$$

where $\alpha_{\mathrm{sc}\,j}$ are loop-level coefficients and $I_{\mathrm{sc}j}$ are loop-level parameters. The linear terms of this expression are contributions of different *scenarios* (or, to be more precise, of different scenarios and scenario transitions; these terms are introduced in detail later in this thesis). The idea is that the scenarios define the subsets of the loop iterations – based on certain application-specific

properties – such that the iterations that belong to the same scenario have similar contribution to the total execution time. The loop-level parameters count the number of iterations in the subset of the scenario.

In the example application for which Equality (2.11) has been constructed, the loop of interest processes one block of pixels per iteration and the specific property for grouping the iterations into different scenarios is the type of the block as defined in the MPEG-2 standard. However, in general, scenarios do not need to be bound to any classification defined in the standard from which the application is derived. We introduce scenarios in Chapter 5.

In fact, our work on scenarios can be classified as part of scenario-based approach; a broad-scope extensive overview on using that approach in embedded system design can be found in the paper of S.V. Gheorghita *et al* [29].

Note also that the method of A. Bavier [6] and also of most of the other work on timing models for adaptation, e.g. G. Bontempi's work [10], assume that the contributions of different computations to the execution time add up together, which is a valid assumption only when the application has only one thread of execution. In our case, there are multiple parallel threads of execution – the processes. Because of that, we compute the contribution of scenarios differently, using IPC graph analysis. For example, a linear expression that can already be written based on Lemma 2.6 is the following:

$$\Delta_N \leq (\lambda_{\max} + \delta_{\max}) \cdot N \qquad (2.13)$$

where $\lambda_{\max}, \delta_{\max}$ are the MCM and the maximum δ_N of IPC graph $\mathbf{G_{max}}$ (see Equality (2.9)), with worst-case execution times. In this case, there is only one scenario that includes all N iterations of the loop of interest. Instead of one scenario, Chapter 5 proposes multiple scenarios and explains the algorithms to find the loop-level coefficients. The common property between the coefficients defined there and coefficient $\lambda_{\max}$ mentioned here is that they are also calculated from the analysis of various paths through the IPC graph.

Not only at the loop level, but also at a finer-grain level – the actor level – the execution delays are data-dependent. Unlike the loop-level expressions, the actor-level expressions can be non-linear functions on the parameters, and we support this case. Nevertheless, in Chapter 3 we make an observation that under reasonable general assumption, any actor execution delay can be accuirately translated into linear form, and therefore we use linear expressions for illustration purposes. In Chapter 3, we show that one can model variable actor execution delays in the form:

$$d(v_k, n) = R\left(C_{k,0} + C_{k,1} \cdot \xi_{k,1} + C_{k,2} \cdot \xi_{k,2} + \ldots, \quad v_k\right) \qquad (2.14)$$

where $\xi_{k,\omega}$ are actor-level parameters, $C_{k,\omega}$ are constant coefficients and R is a stepwise-linear function that takes into account the limited budget assigned to the process where actor v_k belongs. Note that, as we see in Chapter 3, function R can be represented algebraically (i.e. using a simple set of arithmetic and 'ceiling' operations), and thus the whole right part of Equality (2.14) is an algebraic expression. Recall from Section 1.4.4 that the use of algebraic expressions is an important requirement for our performance analysis approach.

Whereas loop-level parameters I_k count the iterations of the top-level loop of interest, actor-level parameters $\xi_{k,\omega}$ count the iterations of the lower-level loops, hidden inside the actors.

We conclude this subsection by an important remark.

Remark. The values of active parameters should be set prior to the loop execution run. Recall that the active parameters are the parameters that can be set by the run-time QoS

adaptation algorithm. We would like to stress that in our approach it is not allowed to set them *during* the loop execution run. Therefore, the QoS adaptation cannot be part of functionality included in the computation actors of the loop of interest. The reason for that is the fact that QoS adaptation algorithms typically set the active parameter values based on the current slack (the time left until the deadline). If we included the QoS adaptation, then the execution delays of some actors would become dependent on their starting times, which would harm the validity of our analysis techniques. ♦

2.3.4 Implementation Trajectory

In this subsection, we reconsider the basic implementation trajectory given in Section 2.1.2 – see Figure 2.2 – now updating it by exposing the use of parameters. Here we still avoid most details on the contents of the intra-application and multi-application mapping stages, postponing them to the next chapter.

The purpose of this presentation is to give an overview on how the parameters are supported and applied in our design methodology. When reasoning about the use of the parameters in practice, we tried to come up with a 'recipe' that is simple but still general enough. For a concrete and detailed example we refer the reader to our application case study in Chapter 6.

First, we discuss the variant of our methodology for soft real-time (SRT) applications, and then we consider the differences for the hard real-time (HRT) case. For SRT, we follow a philosophy similar to the one often followed in the domain of QoS for consumer terminals: design for average-case resource utilization and the best-quality setting of active parameters and, at run time, whenever the resource utilization goes above average, lower the quality as much as necessary to avoid too many deadline misses. In line with that approach, when dealing with SRT applications, our implementation trajectory uses average actor delays to perform the mapping to the platform. Strictly speaking, this is not exactly the same as targeting the average resource utilization, because, as we learn from Formula (2.10), using the average actor delays is a technique that tends to lead to optimistic estimation of the average performance and consequently it also leads to underestimation of the resource utilization. However, we do not have any better generic approximation of the average resource utilization, as we said we can only approximate it better at run time, and there are no general techniques available yet to approximate it better at design time.

Below, a brief specification of the trajectory follows. Some terms used in the specification have not been introduced yet; we indicate them using Italic font. The explanation comes after the specification. Note that to be able to later extend this specification to the hard real-time case, in what follows, we use the words 'typical' and 'typically' instead of 'average-case'.

Specification of an implementation trajectory.

I. Application preparation

1. Parameter identification (application designer, design-time)
   a. identify actor-level parameters ξ_ω,
   b. identify the scenarios and loop-level parameters I_{scj},
   c. identify which parameters are passive and which are active.
2. Actor-level *characterization* (system designer, design-time)
   - calculate the actor-level coefficients, $C_{k,\omega}$.

3. Typical timing and constraints evaluation (system designer, design-time)
   a. calculate the typical actor-level parameter values $\xi_{\text{typical}\,\omega}$,
   b. calculate typical actor delays, based on $\xi_{\text{typical}\,\omega}$, $C_{k,\omega}$, and Equality (2.14),
   c. calculate the *typical required throughput* θ_{required}.

II. Intra-application mapping (system designer, design-time)

1. Intra-application mapping of static-delay computation graph
   - minimize the resource usage assuming typical actor delays and a throughput constraint of θ_{required}.
2. (optional) Preparation for loop-level characterization
   - find analytical expressions for loop-level coefficients $\alpha_{\text{sc}\,j}$.

III. Multi-application mapping (system, run-time)

IV. Run-time QoS adaptation manager (application, run-time)

1. retrieve *characteristic passive parameter values* from the frame header,
2. select the values for active parameters, based on the adaptation algorithm,
3. use the parameter values to calculate the actor delays for different scenarios, for each scenarios applying Equality (2.14) ,
4. perform loop-level characterization
   - from the obtained actor delays, calculate loop-level coefficients $\alpha_{\text{sc}\,j}$ either based on the IPC graph analysis or using the prepared analytical expressions (if available),
5. find an expression for an upper bound on Δ_N using Equality (2.12),
6. use the expression to verify that real-time constraints are met, and if the quality metric is maximized stop the adaptation algorithm[11]; if not go back to Step 2.♦

Let us consider the parameter identification first, i.e., Part I.1. It is performed once for a big class of target multiprocessor platforms that should support the given application. The identification of parameters in Steps ‘a’ and ‘b’ is essentially independent of the target platform architecture. For the parameter identification at actor-level – Step ‘a’ – one can apply an existing profiling-driven automated technique by S. V. Gheorghita *et al* [28], [24]. However, they do not provide a way to derive algebraic expressions for the dependency of actor execution delays on the parameters, their method to estimate the execution delays are based on lookup tables, which are accurate only for a limited range of parameter values. To really represent the actor-level execution delays algebraically, e.g. as in Equality (2.14), one can do static control-flow analysis of every actor, like the one we sketch in Chapter 3. [28], [24] also present an automated technique for identifying scenarios and hence also for identifying the loop-level parameters – Step ‘b’. We discuss Steps ‘a’ and ‘b’ in Sections 3.2 and 5.3 respectively. Note that along with identifying the parameters, the application designer also selects the most significant parameters to be encoded in the data headers. Distinguishing between passive and active parameters, Step ‘c’; it should be done by the application designer him- or herself, based on the knowledge of the application algorithm.

In contrast to the first subpart, Part I.2 targets a concrete multiprocessor platform, and therefore it is the job of the system designer. This subpart performs *characterization*, which we define as ‘finding the parameter coefficients’. Part I.2 calculates the coefficients at the actor-

[11] The algorithm can also be stopped if a time-out is reached.

level, i.e., for the actor execution delays. In Section 3.3, we discuss how well-known execution delay analysis techniques such as linear regression for the execution delays measured in a platform simulator, as proposed e.g. in [6], and worst-case execution times, e.g. [53 - §3], can be modified to obtain conservative coefficients, which is necessary for conservative performance analysis.

Part I.3 is the last subpart of the application preparation. It evaluates the possibilities and requirements for the application under typical conditions, which are used to allocate the platform resource budgets at design time. Step 'a' evaluates the typical values of passive parameters. This can be done experimentally, by measuring them for 'typical' sample input streams, e.g. Mladen Bereković *et al* [9] calculate the average values of parameters for the given MPEG-4 'profile', i.e., for the given set of workload conditions defined in the MPEG-4 standard. As for the active parameters, determining the typical values means selecting the parameter settings that the QoS manager will 'typically' use at run time. One way or another, the application designer is in control over those values because it is he/she who designs the QoS adaptation manager. Step 'b' combines the calculated typical parameter values and the coefficients calculated for the given target platform by actor-level characterization. Note that knowing the typical parameter values for the given application and the coefficients for the given platform, one can quickly evaluate the typical actor delays even without profiling the given application with the platform's profiling tools. Finally, step 'c' evaluates the typical throughput requirements of the application. For many streaming applications the throughput requirements are fixed, but for some of them they may change at run time. For example, in the MPEG-4 arbitrary-shape decoder, the video frame can grow or shrink, whereby the required number of blocks per second also changes. For such application, the 'typical' requirements should be evaluated at this step.

The main task of Part II is intra-application mapping. We observe that most of the work on design-time mapping is done for static delays, e.g., [36], [49], [83], [66], [88], and [90]. Therefore, during the mapping, in Part II.1, we abstract the actor delays as static delays, i.e., the *typical delays*, calculated in Part I.3. Given the typical delays, at the beginning of the mapping flow one can assume that each actor gets 100% budget on the fastest native processor architecture that can execute the given actor. This enables the designer to immediately evaluate whether the fastest possible implementation can satisfy the typical required throughput constraint, and if it is the case, to continue by relaxing the resource budget requirements in the mapping flow.

In our implementation trajectory, the objective of mapping is to minimize the number of resources (i.e., processors and channels) and the magnitude of resource budgets (i.e., processor cycle budgets and channel bandwidth budgets) under the performance constraint (the required typical throughput). An example of an intra-application mapping approach that considers similar kinds of applications and platforms as assumed in this thesis is presented by Sander Stuijk *et al* in [88], [90].

To explain Step 2 in Part II, recall that our performance analysis uses an expression in the form of Equality (2.12) to yield a conservative estimate of Δ_N at run time. That expression uses loop-level coefficients $\alpha_{\mathrm{sc}\,j}$ .Recall also from the remark in Equality (2.13) that in this thesis we define run-time algorithms to calculate the loop-level coefficients based on analysis of various paths in the graph. One of those algorithms is MCM analysis. However, Amir Hossein Ghamarian *et al* [22] propose automatic design-time derivation of analytical expressions for a graph's MCM, λ, as a function of actor delays as unknown variables. Those expressions

can be used for quick and accurate MCM evaluation at run time instead of performing run-time MCM analysis. This is, in fact, in line with Step 2 in Part II of our implementation trajectory. At Step 2, the designer derives analytical expressions for the loop-level coefficients as functions on the actor delays that are known only at run time. Note that Step 2 of Part II is optional, because one can run our analysis algorithms at run time.

Thus, [22] partially automates this step. However, in Chapter 5, we define also other coefficients that we need for run-time estimation of Δ_N. For those coefficients, either these automatic techniques have to be extended or the designer can try to derive analytical expressions manually, exercising his/her analytical skills. For the IPC graph in our case study in Chapter 6, we arrive at simple formulas for all loop-level coefficients in a 4-actor graph.

Part III, the multi-application mapping, is not involved with parameters. It fits the resource budget subnetwork (obtained from the intra-application mapping flow) into the available unoccupied physical resources. Closely related topics are studied, for example, in the work of Orlando Moreira *et al* [65] Sander Stuijk *et al* [89], Srinivasan Murali *et al* [67], and Andreas Hansson *et al* [34].

In Part IV, we see an iterative adaptation procedure, which is in line with Figure 1.8, whereby Steps 2 and 6 in Part IV represent the optimization unit and the other steps represent the performance analysis. To be able to estimate the execution time, our method needs to calculate special actor delay values that represent the actor delays in different scenarios. They are obtained from the actor parameter values that are characteristic for those scenarios and we refer to those values as *characteristic parameter values.* Those values are retrieved from the input data headers at Step 1. Having calculated the actor delays for different scenarios at Step 3, our method fills them in into the IPC graph and uses certain graph-path analysis algorithms (or the formulae derived at Step 2 in Part II) to calculate the loop-level coefficients at Step 4. The loop-level coefficients go into an expression that estimates the loop execution time, Δ_N, based on Equality (2.12) .

In the ideal case, the QoS adaptation procedure can immediately – without multiple iterations – find the appropriate values for the active parameters using the execution time expression. For example, suppose that the expression for the execution time gives:

$$\Delta_N \leq 10 \cdot I_{\mathrm{sc}1} + 5 \cdot I_{\mathrm{sc}2}$$

and suppose that the header provides value $I_{\mathrm{sc}1}$=40 and suppose that $I_{\mathrm{sc}2}$ is an active parameter, being at the same time the quality metric that has to be maximized. Suppose that the real-time constraint is $\Delta_N \leq 500$. In that case, to meet the constraint and to ensure the maximum quality, the manager can quickly solve this linear programming problem and come up with value $I_{\mathrm{sc}2}$=20. Note that although in general Δ_N might be a non-linear function (if some active parameters are actor parameters), one can be sure that it will be monotone on its arguments, which is favorable for optimization.

In case of hard real-time (HRT) applications, one can use the same trajectory, but the term ‘typical’ will mean in this case ‘worst-case’, such that one can always ensure that the deadlines are automatically met for the typical active parameter setting. It is not necessary to involve a QoS adaptation manager here, but one may choose to use one to maintain, whenever possible, quality settings that are better than worst-case. Such a QoS manager would always be safeguarded by the possibility to resort to the worst-case setting.

2.4 Summary and Notes

In this chapter, we have chosen and motivated a multiprocessor scheduling method and the basic timing models for that method – IPC graphs, enabling the application throughput analysis for the given implementation. We have argued that they can be extended to support a network-on-chip platform and data-dependent execution delays. Therefore, those models can constitute a mathematical framework for run-time adaptation, making it possible to predict the performance metrics and identify performance bottlenecks in the given implementation based on a-priori characteristics of the dynamic computational workload available at run time. The details on how the extended IPC graphs can be built are presented in the next chapter of the thesis.

Another major topic we studied is application dynamism due to data-dependent variations in the execution delays. Under those conditions, we wish to analytically derive the performance metrics that can be guaranteed by the system. We observed that a closely related problem addressed in the literature has proven to be too tough to solve analytically. Our hypothesis is that the reason for that is an attempt to *globally* cover all possible run-time situations. Nevertheless, for the purposes of run-time QoS adaptation, we have identified a possibility to exploit available run-time information on the temporarily *local* run-time situations to derive local performance metrics analytically. Later on in this thesis, in Chapter 4, we build a necessary basis for that idea and in Chapter 5 we work it out in detail. We will see there more evidence that this problem is challenging even under the current basic assumptions that the IPC graph is autonomous (no inputs and outputs) and that there is no conditional communication between the graph nodes.

In the end, we would like to mention our major sources of inspiration for the ideas explained in this chapter and mention some related work. The original idea of IPC graphs and throughput analysis comes from the book of [83] and earlier papers by the same authors. The ideas on providing bounds on the performance metrics of HSDF models and on processor scheduling come from discussions in a multiprocessor networks-on-chip project at Philips Research Labs Eindhoven (nowadays NXP Semiconductors), and one can find more on this subject in [31], [7], [8], as well as in our own work [75]. We owe such a subtlety as the FIFO property and other general statements about the HSDF model to the fundamental book of F. Baccelli *et al* [4], but Lemmas 2.5 and 2.6 are original for this text. Last but not the least, the idea of characterization of the resource requirements of streaming applications using HSDF models with parameter expressions for actor delays comes from our collaboration with the video coding architectures group at our university, see e.g. [72], [77]. A publication of Clara Otero Pérez, Liesbeth Steffens *et al* [71] and related whiteboard presentations from the authors provided us a good introduction into the domain of QoS management for streaming applications.

In this chapter, we implicitly touched upon an important topic – the possibility to embed models of multiple local schedulers of different resources (i.e., processors) into one single dataflow ‘super-model’, such that the ‘super-model’ enables the schedulability analysis that takes into account not only the behavior of separate schedulers but also the – possibly cyclic – dependencies between them. This is, in effect, done in our implementation-aware HSDF, whereby we model the run-time scheduling by using a scheduler-dependent actor delay determined by the upper bound given in Equality (2.2). Similar ideas were developed by Rob Hoes in his Master Thesis [38]. Maarten Wiggers *et al* [98] analyze the new possibilities opened by such super-modeling for the run-time scheduling theory in general and introduce more

accurate and elaborate models for a certain general class of local schedulers, whereby every application task is represented by a *pair* of actors as opposed to one actor, as in this thesis.

A work that is closely related to our actor-level parameter identification is presented by Yicheng Huang *et al* in [41]. They study the decoding workload estimation for different video coding standards. The processing times are obtained using lookup tables from data-dependent conditions to either a constant number of clock cycles or to a simple parameter function. This is equivalent to introducing a separate Boolean parameter for every condition and using as a linear combination of those parameters and the lookup values. They can achieve good accuracy – [39] claims at most 2.7% average error for sequential execution on a single processor. They do not model parallelism and communication, which would be required for parallel execution on multiple processors. We present such modeling techniques, as well as the details of actor-level processing time modeling in the next chapter.

3

3 Design-Time Trajectory: IPC Model Construction

This chapter focuses on the design flow introduced in the previous chapter. Recall that that flow consists of the application preparation and the intra-application mapping. The goal of the flow is to generate a resource budget network that typically satisfies the timing requirements and the IPC graph that can be used for performance analysis. In this chapter, we consider the design flow mainly from the point of view of IPC graph construction.

Sections 3.1-3.3 are dedicated to the application preparation – or Part I of the trajectory. Whereas, in Chapter 2, we considered the structure of computation graphs, in those sections, we fill in the delay values into that structure. Hereby, we introduce the actor-level parameters and coefficients in more detail. As for the scenarios and loop-level parameters, which are also defined during the application preparation, we postpone their detailed treatment until later – Chapter 5.

In the implementation trajectory, the application preparation is followed by the intra-application mapping – or Part II. Before considering the mapping flow in Section 3.5, we give the necessary details on the multiprocessor architecture in Section 3.4. For the last design-time step, Step II.2 – the derivation of analytical formulas for the loop-level coefficients – we hardly can provide a general methodology, but we give an example in Chapter 6.

In Section 3.6, we consider a few important miscellaneous properties of IPC models.

The methodology presented in this chapter is a combination of different ideas, coming from different sources, including some original ideas. The summary on the literature sources and our

own claims are for the most part postponed until the 'Summary and Notes' section in the end of this chapter.

3.1 Timing Modes

3.1.1 Timing Mode: Processing Times and Computation/Communication Delay Relations

In this subsection, we introduce the timing modes, which provide the actor delay values in the implementation-enhanced HSDF model.

Before introducing the timing modes, we take a quick look on how the actor delays are computed in a timing mode and how the timing modes fit in the global picture of performance analysis. Recall that a computation actor in the IPC model starts at the moment when it is enabled, i.e., when its turn comes to execute and the input tokens as well as the free places for the output tokens are available. However, at the moment of time when an actor is enabled, rather than executing the process that runs the given actor, the processor may be busy with another process, and afterwards it may keep switching between processes. Nevertheless, recall from Section 2.1 that the processor scheduler guarantees that the computation actor will complete the necessary processing in time d, such that:

$$d \leq t/B + \hat{q} \tag{3.1}$$

where t is the number of processor cycles required for the processing in the given actor execution; B is processor cycle budget assigned to the process per unit of time; $\hat{q}$ is a constant depending on the scheduling method and settings.

In Chapter 2, we did not yet detail the relation between the processing algorithm of the computation actor and delay annotation $d(v_k, n)$. We only suggested that expression $t/B + \hat{q}$ could be used as such, because it gives a conservative estimate of the delay and because it is independent of the time when the computation actor is enabled.

In the latter statement, we implicitly make a certain basic assumption that often holds by default. We assume that value t is stable under any possible starting conditions, and the real execution on a processor will not take more cycles than t. In fact, for conservative timing analysis, we do not require that t gives the exact number of processing cycles, we only need an upper bound, preferably a tight one.

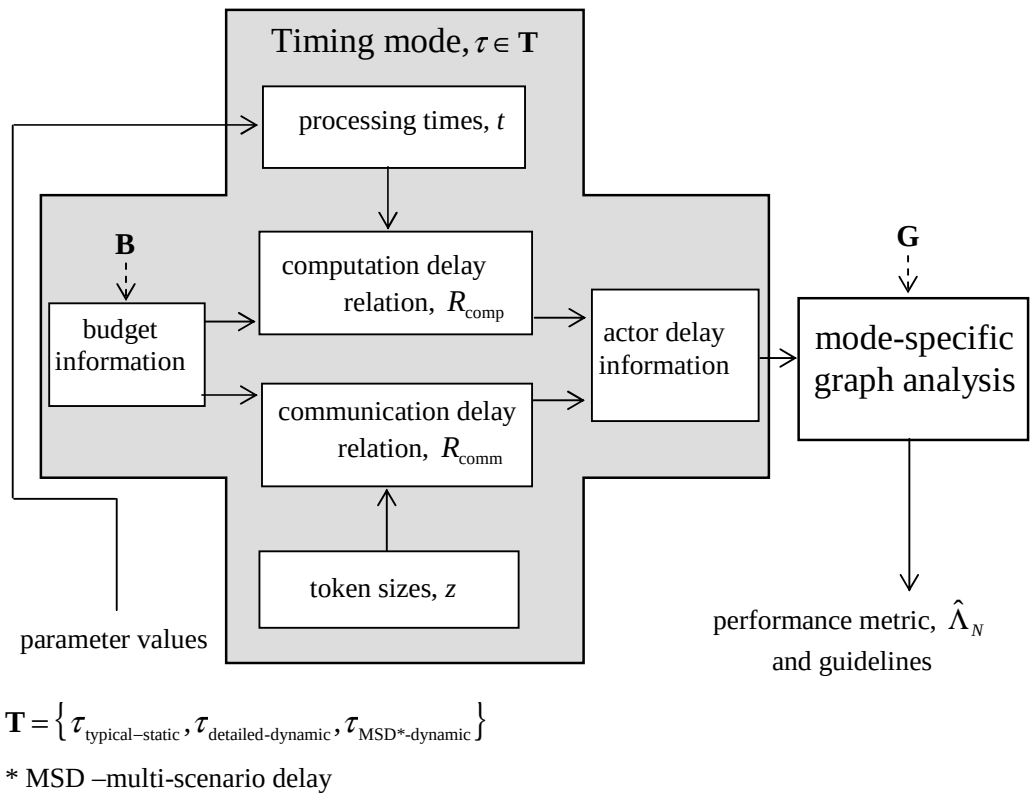
Postulate. (Actor processing time) There exists an upper bound on the number of cycles required for an actor execution, which only depends on the contents of the input data streams of the application. We call that bound *actor processing time*, denoted $t(v_k, n)$. ♦

The processing time postulate is motivated and discussed in detail in the next subsection. In an implementation-enhanced HSDF model, every timing mode defines a method to estimate the actor processing times and a relationship between the processing times t and the delays d.

Definition (Computation delay relation[12]) The delay of a computation actor is related to the processing time by equality:

$$v_k \in \mathbf{V}_{\text{comp}} \Rightarrow d(v_k, n) = R_{\text{comp}}\big(t(v_k, n), v_k\big) \tag{3.2}$$

[12] Although, in fact, R_{comp} and R_{comm}, defined later, are functions, I refer to them as 'relations', so that I can easily distinguish them from the other functions.



$\mathbf{T} = \{\tau_{\text{typical–static}}, \tau_{\text{detailed-dynamic}}, \tau_{\text{MSD*-dynamic}}\}$

* MSD –multi-scenario delay

Figure 3.1 Timing modes and performance analysis

where $\mathbf{V}_{\text{comp}}$ is a set of HSDF computation actors, and function R_{comp} is called the *computation delay relation*, and its value gives an upper bound on the computation actor delay for the given processing time and the budget to which actor v_k is assigned. We assume that R_{comp} is a monotonically increasing function on t. For any scheduling method, one can select R_{comp} in the form $R_{\text{comp}}(t, v_k) = t/B(v_k) + \hat{q}(v_k)$, where $B(v_k)$ is the budget assigned to the process where actor v_k is contained and $\hat{q}(v_k)$ is an additional factor that depends on the scheduling algorithm used by the processor where actor v_k is executing. Nevertheless, for TDMA scheduling, which we adopted in Chapter 2, there exists a tighter upper bound that cannot be expressed in that form, and we introduce it in the last subsection of this section.♦

In fact, the postulate and the definition above give us the timing mode components for the computation actors. A timing mode also has components for the communication actors. This is illustrated in Figure 3.1. As shown in that figure, similar as in case of the computation actors, the delay of communication actors is determined by the amount of work they perform and by the communication budgets.

Definition (Communication delay relation) The delay of a communication actor is related to the data token sizes by equality:

$$v_k \in \mathbf{V}_{\text{comm}} \Rightarrow d(v_k, n) = R_{\text{comm}}\big(z(v_k), v_k\big) \qquad (3.3)$$

Function R_{comm} is called the *communication delay relation*, and its value gives an upper bound on the communication actor delay for the given size $z(v_k)$ of the data token transferred by the communication actor and the budget of the channel to which actor v_k belongs. The communication budgets are introduced later in this chapter. ♦

Definition (Timing mode) A timing mode $\tau \in \mathbf{T}$ of an implementation-enhanced HSDF model $<\mathbf{G},\mathbf{T},\mathbf{PQ}>$ is a tuple $<t, R_{\text{comp}}, z, R_{\text{comm}}>$, where t is a function, $t(v_k, n)$, that defines the processing times of computation actors, R_{comp} is a function, $R_{\text{comp}}(t, v_k)$, that defines the computation delay relation, z is a function, $z(v_k)$, that defines the sizes of the data tokens transferred by the communication actors, and R_{comm} is a function, $R_{\text{comm}}(z, v_k)$, that defines the communication delay relation.♦

Recall that we assume that the sizes of the data tokens transferred by the communication channels are fixed. Because the channels get guaranteed bandwidth budget, this means that the communication delays are constant. Figure 3.1 reflects that fact by showing that only the computation delays depend on parameters.

An implementation-enhanced HSDF model has several timing modes, reflecting the actor timing with different accuracies. Different timing modes are meant for different purposes and situations. In each situation, only one timing mode is active. Which performance analysis method is applied in the given situation depends on which timing mode is active. As shown in Figure 3.1, given the actor delays and the structure of HSDF graph **G**, the mode-specific analysis method should estimate performance characteristics, such as an upper bound $\hat{\Lambda}_N$ on the loop iteration interval Λ_N.

The timing modes can be split into static and dynamic modes. Static modes assume that the processing times – and, consequently, the actor delays – are static. Recall that in the mapping flow, we use a static-delay timing mode that assumes that the actors have typical delays. In Figure 3.1, that mode is denoted as $\tau_{\text{typical-static}}$. In that mode, the loop iteration interval, Λ_N, can be efficiently approximated by its limit value for $N \to \infty$, i.e. as the MCM of the graph, λ. λ is used as a constraint for the optimization steps of the mapping flow, considered in Section 3.5.

However, under conditions of dynamic computation delays, only the dynamic timing modes, using run-time information about the parameter values, can ensure good accuracy in the general case. We introduce two dynamic timing modes: *detailed dynamic mode* and *multi-scenario-delay (MSD) dynamic mode*.

The detailed dynamic mode needs parameter values that characterize each actor in every loop iteration. In that case, every computation actor gets a sequence of accurate delay annotations, $d(v_k, n)$, for the whole loop execution run, $n = 0, 1, ..., N-1$. At design time, such information could be used to perform a simulation of the graph execution with accurate timing. From the simulation, one could obtain the loop execution time, Δ_N, and then $\hat{\Lambda}_N$ is equal to Δ_N / N. However, similar computations at run time would involve too much overhead. Nevertheless, we use the detailed mode as a foundation to define the multi-scenario-delay mode and to evaluate the accuracy of that mode experimentally. In fact, the detailed mode is the key mode for the identification of actor-level parameters, the first design task of the implementation trajectory. Therefore, this mode is central to Section 3.2.

Also in the multi-scenario-delay (MSD) mode, run-time parameter values are required that characterize all the iterations in the loop execution run, but not in full detail. The purpose of the MSD mode is to reduce the overhead of the detailed mode while still preserving good accuracy. The MSD mode and the HSDF graph analysis in that mode are defined in Chapter 5.

In the remaining two subsections of this section, we revisit the concept of processing time and the computation delay relation, which are components of any timing mode.

3.1.2 Processing Time

In this subsection, we revisit the concept of processing time in detail. In fact, the postulate formulated in the previous subsection says that one should be able, for any given set of valid input data streams, to derive beforehand a sequence of guaranteed processing times for each actor in the computation graph.

This statement implies two underlying requirements. Firstly, it presumes that, all the processing required to perform an actor execution – namely, the computation of the data outputs and the update of the internal state – constitutes a sequence of processor instructions that depends only on the data inputs being consumed in the given actor execution and on the internal state (which is, in general, predetermined by the contents of all data tokens consumed from the input data edges before the actor execution by all the actors that belong to the same process). In practice, a violation of that rule is only possible in an unlikely case when the actor contains a loop with a conditional number of repetitions that is chosen based on the current time value obtained from a timer or a random number generator. We cannot consider the values read from a timer or a random number generator to be part of actor input or state, because that could make the actor delays sensitive to the actor starting times and the behavior of the other processes running on the same processor. Fortunately, such situations are not typical in streaming applications, which easily satisfy this assumption. Therefore, we can speak of a deterministic, even though data-dependent, sequence of processor instructions required for processing within one actor execution for the given input data streams. Let us call that sequence of instructions the *actor processing duty*.

Secondly, our postulate requires that the given processing duty take a tightly bounded number of processor cycles. Violations of that rule can happen, first of all, due to ‘improper’ use of instruction and data caches. The ‘improper’ use of caches, in our terms, means the possibility of cache misses when the processor does the actor processing duty. It is ‘improper’, because it violates the basic requirement that, before the actor execution starts and during its lifetime, the actor state must be fully available in the physically local part of the memory system (see Section 2.1). To avoid ‘improper’ use of caches, one can either avoid the use of caches in the loop of interest at all (by mapping the actor state, instructions and input/output data to the local scratchpad memories) or, in case the cache has some advanced control features, by instructing the cache to pre-fetch the required data and to keep it as long as it is needed.

In addition to caches, also conditional branch predictors can contribute to cycle count variation of an actor processing duty. Whether this factor is important depends on how many conditional branch instructions are contained in the actor instruction sequence and how much their processor cycle usage can vary. If the actor is ‘sensitive’ to conditional branches, this poses a threat to our processing time analysis techniques, especially when, at the start of actor execution, previously executed actors may influence the predictor state or when it can be disrupted by context switches. To avoid this, processor architectures could quickly save and restore the status of the branch predictors, and then it is possible to keep our assumption valid even in the ‘sensitive’ situation simply by extending the definition of actor processing duty by also including the initial state of the branch predictors. Now, not only the sequence of

instructions but also the initial state of the branch predictors would predetermine the processing time. One would only have to ensure that the first part of our assumption still remains valid, namely, that also the state of branch predictors is determined only by the input data of the process where the actor is contained, but never by the other processes or by the starting time of actor execution. This can be ensured by appropriate saving and restoring of that state at the context switches. In case the save/restore facility is not available one can account for the worst-case effect of the unknown state in the processing time analysis, considered in the next two subsections. In this case, one sacrifices the tightness of actor delay estimation for the sake of conservativity.

Let us summarize the meaning of the two basic assumptions discussed in this subsection. Both of them follow the same philosophy: they permit the actor processing time to be influenced only by the internal state and input data, which are only determined by the actor's predecessors in the computation graph. To ensure that processing times are close to the real execution cycle counts, one can involve the management of caches/scratchpad memory and the management of the branch predictor units, thereby incurring certain costs. To avoid that cost, one can take conservative assumptions about the behavior of the hardware units. We can add to that that one can choose to use different strategies for different actors. The HSDF analysis techniques of the next two chapters help to determine which actors are critical for the performance. Then one can choose to pay the cost of tight processing time estimation only for the critical actors.

3.1.3 Computation Delay Relation under TDMA Scheduling

Recall from Section 2.1 that each processor has a local scheduler managing multiple processes on a single processor. In this thesis, we adopt a time-division multiple access (TDMA) scheduling for processes, where time is divided into periods, and each period is split into several time slots of possibly different size assigned to different processes.

The computation delay of an actor includes the processing time and the time the actor execution was initially postponed and subsequently preempted by the scheduler. To have a conservative model, our computation delay relation (see Equality (3.2)) assumes worst-case delay of postponement and preemption. In case of the TDMA scheduling, the computation delay relation is given by:

$$v_k \in \mathbf{V}_{\text{comp}},\quad D = \frac{t(v_k, n)}{F_{\text{clock}\,k}} \quad \Rightarrow R_{\text{comp}}\big(t(v_k, n), v_k\big) = D + \left\lceil \frac{D}{T_{\text{B}\,k}} \right\rceil \cdot (T_{\text{T}\,k} - T_{\text{B}\,k}) \tag{3.4}$$

where D is the processing time measured in absolute timing units (rather than in processor clock cycles), $F_{\text{clock}\,k}$ is the clock frequency of the processor to which the actor is mapped, $T_{\text{T}\,k}$ is the TDMA period of the processor's local scheduler, $T_{\text{B}\,k}$ is the time slot reserved for the process that contains actor v_k. The 'ceiling' part of the expression accounts for the worst-case number of time intervals when the given actor execution has to wait because the processor is busy with the timing slots that are different from the timing slot of the given actor and $(T_{\text{T}\,k} - T_{\text{B}\,k})$ gives the worst-case delay of one such interval.

Note that $B_k = F_{\text{clock}\,k} \cdot T_{\text{B}\,k} / T_{\text{T}\,k}$ gives the processor cycle budget, measured in clock cycles per unit of time. Then, by using the property that $\lceil x \rceil < x + 1$ in Equality (3.4), after some algebraic manipulations, we obtain:

$$v_k \in \mathbf{V}_{\text{comp}},\ \ t = t(v_k, n),\ \hat{q} = T_{\text{T}\,k} - T_{\text{B}\,k} \quad \Rightarrow R_{\text{comp}}\big(t(v_k, n), v_k\big) \le \frac{t}{B_k} + \hat{q} \qquad (3.5)$$

This result proves that the TDMA scheduling is a budget-provision scheduling method according to our definition given in Section 2.1. It also proves that we use a tighter upper bound on the real actor execution delay than the bound that is suggested by Equality (3.1).

3.2 The Identification of Actor-Level Parameters

In this section, we cover part of the first design task to be performed for a given application in our implementation trajectory – the application preparation. According to the specification in Section 2.3.4, this part of the design flow is responsible for expressing the dynamic data-dependent execution times of the application as functions of *parameters*, i.e. workload characteristics of the application input data streams. These functions should be expressed in *algebraically*, which is required in our performance analysis approach. In this section, we only consider actor-level part of the application preparation, which characterizes the actor processing times. This is the low-level part of the application preparation. The study of the other essential part of application preparation, working at the level of the whole HSDF graph, is postponed until Chapter 5.

A major part of actor-level application preparation is the detecting the actor-level parameters, i.e. the parameters that determine the actor processing times. Currently we are not aware of any automated parameter identification techniques that would be able to not only detect the set of parameters, but also give an algebraic expression of the processing time as a function of the parameters. On one hand, the automated parameter-identification method proposed by S. V. Gheorghita *et al* in [28], [24] partly solves this problem, because it can automatically detect a set of input data variables that determine the processing time and thus can be used as parameters. Unfortunately, on the other hand, that work calculates the processing times from the input variables by means of a performing a lookup in a lookup table, which is only suitable for the cases where the set of possible values of every parameter is limited.

Therefore, instead of using the input variables, we assume that the actor-level parameters are (implicit) functions of the input variables that count the number of executions of different source-code sub-blocks and thus always have a close-to-linear contribution. Our algebraic expression for the processing time is thus a linear combination of parameters. In this section, we propose and discuss a manual method to detect such actor-level parameters. (Note that the linear parameter functions to express execution times are often exploited in the design and performance analysis; the examples which we already discussed before are [9] and [6].)

This section is organized as follows. In Section 3.2.1, we introduce the linear actor-level parameter functions. Hereby we set and justify our goal of identifying the actor-level parameters, by arguing that they can be used universally for various streaming applications. Section 3.2.2 is the core subsection of this section, discussing how to define suitable linear contributors to the processing time of the given computation actor. In effect, that section introduces our manual

```
Input:  i(n), n = 0, 1, 2…
Output: | i(n) · i(n–1) |
    i ← i(n)
    out ← i*i_1;
    i_1 ← i;
    if (out<0) {
     out ← -out;
    }
Answer: out
```

C_0, C_1, C_{WCET0}

(a) actor source code

$$t_1(v,n) = C_{\mathrm{WCET0}}$$
$$t_2(v,n) = C_0 + C_1 \cdot \xi_1(n)$$
$$t_3(v,n) = C_0' + C_1' \cdot \xi_1(n) + C_2' \cdot \xi_2(n)$$

t_1 – WCET

t_2 – taking the 'if' operator into account

t_3 – taking a branch predictor into account

$$\xi_1(n) = \begin{cases} 1, & i(n) \cdot i(n-1) < 0 \\ 0, & \text{otherwise} \end{cases}$$

$$\xi_2(n) = \begin{cases} 1, & \xi_1(n) \neq \xi_1(n-1) \\ 0, & \text{otherwise} \end{cases}$$

(b) parameter functions and parameters

Figure 3.2 Modeling a simple actor with parameter functions

parameter identification method. Note that for simplicity we often use term 'actor parameter' instead of 'actor-level parameter'.

The actor processing time expressions do not have to be linear, as our analysis approach works also for non-linear actor-level expressions. However, since our manual parameter identification method detects linear contributors, in this thesis we use linear actor-level models, for illustation purposes.

3.2.1 Expressing the Processing Times as Actor Parameter Functions

In modeling the actor processing time, we adopt a hypothesis that it can be computed as a linear function on data-dependent arguments with data-independent coefficients. We call that function an *actor parameter function*. In this subsection, we argue that this hypothesis is general enough to be widely used in practice. However, first we need to introduce the parameter function in detail.

Suppose that the contents of the application input streams is given. Remember that the computation actors can be implemented on processors of different types. The actor parameter function is a linear function on a set of variables, and it can be expressed as follows:

$$t(v_k, n) = C_{k,0} + C_{k,1} \cdot \xi_1(n) + C_{k,2} \cdot \xi_2(n) + \ldots + C_{k,\Omega} \cdot \xi_\Omega(n) \qquad (3.6)$$

where $C_{k,\omega}$, $0 \le \omega \le \Omega$, are constant coefficients that depend on the processor type chosen for actor v_k and $\xi_\omega(n)$ are variables that depend on index n and on the input data streams. These variables are called *actor complexity parameters*. Note that, typically, multiple actors may share these parameters, which is why the parameters in Equality (3.6) are not indexed with the actor index.

A special case can be distinguished when all coefficients except $C_{k,0}$ are equal to zero. In that case the parameter function has a constant value. According to the definition of the processing

time, that value gives then an upper bound on the number of cycles required by the processing. Such an upper bound is usually referred to as *worst-case execution time* (WCET) [53].

The derivation of WCET can be done for a broad class of processor architectures and application software. It is a broad research field, and some major results are published e.g. in a book by S. Malik *et al* [53]. The parameter function is a generalization of WCET, namely, it gives a conditional WCET value given that complexity parameter values are known. Similar to WCET, it can be derived based on the analysis of the internal structure of the actor code.

Example: Processing time: an actor with an 'if' operator. Consider an actor that computes expression $out(n) = |i(n) \cdot i(n-1)|$, where $i(n)$ is the sequence of numbers encoded in the input data stream. Figure 3.2(a) shows a possible implementation of such an actor using a C-language-like pseudocode. Figure 3.2(b) gives three alternative parameter functions for this implementation. Function t_1 is a WCET, and its sole coefficient covers the total clock cycle count of the whole actor body. Function t_2 has Boolean parameter ξ_1 that takes value 1 only if the condition in the '*if*'-operator is satisfied. Its coefficient C_1 corresponds to the worst-case cycle count contribution of the operator body. Multiplied by ξ_1, it contributes to the total only when the condition is true and the operator body is executed. Function t_3 gives a more accurate expression of the processing time in case the architecture contains a branch predictor. It is taken into account by an extra term, $C_2' \cdot \xi_2$. Here C_2' stands for the cost of a wrong prediction. The definition of parameter ξ_2 assumes that the branch predictor expects the previous condition to repeat in the current execution of the operator. If the algorithm of the branch predictor is indeed as assumed in the definition of this parameter, then introducing this parameter makes it possible to select values for coefficients C_0' and C_1' that are smaller than for coefficients C_0 and C_1, because then C_0' and C_1' do not include the cost of a wrong prediction. In case the given actor is mapped to a processor having a different branch predictor or none, C_2' should be set to 0 and costs of the wrong prediction should be included in C_0' and C_1'. ♦

From the above example, we see that the application designer can anticipate a certain type of processor architecture and try to select the parameters accordingly; however, once the set of parameters are chosen, it should remain the same, no matter to which processor type the actor is mapped to. That is, in fact, what we mean when we say that the parameters are independent of the processor hardware architecture.

Let us make a few claims about the generality of actor parameter functions. First of all, they are *general enough for a broad class of processor architectures*. Indeed, one can find the worst-case contributions of different internal parts of each computation actor, such as the body of a conditional operator or a loop. These contributions can serve as coefficients in Equality (3.6) and can be obtained by applying standard WCET techniques to individual parts of each actor. Note that Equality (3.6) may seem to restrict us to non-pipelined architectures[13], because it *adds* the contributions of actor parts, whereas their execution may overlap in time. However, we have performed multiple experiments on the computation actors of the JPEG and the MPEG-4 decoding applications[14], and we have observed that, for RISC architectures, which are pipelined, using the sum of the part contributions as parameter function leads to fairly accurate processing

[13] In fact, virtually any modern processor architecture is pipelined.

[14] A few more details on these experiments are mentioned later in this section, but they are mainly reported in Chapter 6.

time estimates, which can be explained by the parts being large enough, making the timing overlap between the parts negligible compared to the timing lengths of the parts themselves. Of course, in case when the overlap is considerable compared to the sizes of the actor parts, simply adding the WCET contributions of each part may lead to quite pessimistic estimates. Nevertheless, for such cases, we envision that, to reduce the pessimism and improve the accuracy, one can perform extra analysis to compute the *minimum overlap* between a given part and any other part that may precede it during an actor execution. This minimum overlap can be subtracted from the coefficient, and this makes the parameter function less pessimistic. We have not experimented with this idea at the level of individual actors, but we do use a similar idea at the level of multiple actors, and we describe the underlying method in Chapter 5.

The second generality claim we make is that a linear parameter function is *general enough for any algorithm* that may be used in the application. To support this claim we observe that *algorithmic complexity theory* can represent the processing time of virtually any algorithm as an algebraic function on the input data characteristics. It is, for example, well known that the complexity of an efficient algorithm for sorting an array of elements is $O(N \cdot \log(N))$, where N is the size of the array. Equivalently, one can define the processing time of that algorithm as: $t = C_0 + C_1 \cdot N \cdot \log(N)$. Although this function is not linear, we can transform it into a linear one by variable replacement: $\xi_1 = N \cdot \log(N)$. Not any function one can think of can be translated into the form of Equality (3.6), but, intuitively, any complexity function can. Take for example function $\log(N + C)$, where C is an architecture-dependent constant. It is not possible to translate it into an accurate linear form of architecture-independent variables, except by an infinitely long Taylor expansion. Luckily, complexity theory does not come up with such 'strange' complexity functions, because it represents the results in form $O(f)$, where f is an expression that does not include any architecture-dependent constants.

The question about the generality of linear expressions has been raised in [91]. That article, by the way, proposes linear expressions like Equality (3.6), but for the consumed energy, rather than for the consumed clock cycles, which nevertheless boils down to the same reasoning as ours. The author's remark that it is not always 'possible' to obtain a linear expression, referring to the 'greatest common divisor' (GCD) algorithm as a counterexample. We find it worth studying that example and the meaning of 'possible' in their sense to get a further insight into the nature of actor complexity parameters.

Example: Processing time: GCD computation algorithm. Consider the following algorithm:

```
Input: integer n and k, where n ≥ k
Output: GCD(n, k)
  repeat {
    p ← n;
    n ← k;
    k ← p mod n;
  } until k=0;
Answer: n
```

The processing time of this algorithm is $t = C_0 + C_1 \cdot \xi_1$, where ξ_1 is the number of iterations of the 'repeat until' loop required to complete the computations. Obviously, ξ_1 depends on the input data, integer values n and k. In case of the sorting algorithm, we had the same parameter function, and ξ_1 could be expressed analytically, using a logarithm of the size of the input array.

Now a question arises on whether also for the GCD algorithm we can express ξ_1 analytically. We should admit that we are not aware of any simple way to do that. We can only give upper bounds on ξ_1, e.g. $0+n$, $1+k$, $2+n \bmod k$. Here each bound assumes that the loop, once it has started, executes at least a given number of iterations – the first operand of the addition – 0, 1, 2. The second operand of the addition gives an obvious upper bound on the number of remaining iterations, equal to the initial value of variable 'n' before the remaining iterations begin.♦

In [91], only informal reasoning on this subject is provided, but it can be formalized as follows. They consider it 'possible' to use a parameter function only if all the parameters can be expressed as analytical functions of the input elements. Here 'input elements' are such characteristics of the input data structure that can be computed in linear time, e.g. array *N* in case of an array sorting algorithm, and *n* and *k* in case of the GCD computation. Clearly, for the GCD example, using complexity parameters is 'impossible' according to the definition of [91].

This example actually demonstrates the fact that to compute a complexity parameter for an algorithm may require as much computational resources as the algorithm itself. For example, although one does not have to execute the sorting algorithm to compute its parameter ξ_1, to compute ξ_1 accurately for the GCD algorithm one may have to execute the algorithm itself, extending it with a so-called *counter variable* that counts the number of iterations of the 'repeat until' loop, as shown by highlighted lines in the algorithm description below:

Example: Introducing a counter into the GCD algorithm.
Input: integer *n* and *k*, where $n \geq k$
Output: **ξ_1 of the GCD computation algorithm**

```
n ← n;
k ← k;
xi1 ← 0;
repeat {
 xi1 ← xi1+1
 …
} until k=0;
```

Answer: xi1 ♦

Introducing the counters into the application algorithm and using the modified algorithm itself to compute its complexity parameters is a fallback solution that would always work. Therefore, we do not require that the complexity parameters can be computed 'easily' – in a constant time, or in a linear time, or whatsoever. That is what makes our actor parameter function general.

However, the run-time quality/energy optimization managers, which are the ones that would make use of our timing models, may not wait until the application algorithm itself would compute parameters because the managers have to estimate the application performance beforehand. Therefore, the parameters have to be computed externally, without putting any considerable load on the resources of the embedded multiprocessor system-on-chip. This can be done by the external system that generates the input data streams, whereby the streams are extended with headers containing pre-computed values of actor complexity parameters, such that the managers can easily retrieve them. The practical examples of such external systems are, of course, the encoders of video input streams. As far as we know from practice, one can say that, in order to generate the parameters for the decoding applications, the encoders do not need to run

the decoding themselves, because the same parameters typically influence both the encoding and decoding complexity. It is thus sufficient to enhance the encoders with the counter variables and to let the encoders provide their values based on those counters.

Note that we do not encode parameter values for each actor execution, but rather encode a set of parameter values that characterize the execution run as a whole, without sacrificing too much accuracy – the method is described in Chapter 5. Also, one can restrict the set of encoded parameters to include only those parameters that are essential, thus saving space in the headers (whereby for the skipped parameters one will have to make conservative assumptions).

3.2.2 Constructing an Actor Parameter Function

The processing time of any computation actor can be represented in the form of Equality (3.6). All actors share the same set of parameters, $\{\xi_\omega\}$, which we denote $\mathbf{\Omega}$. However, each actor v_k uses only a subset of set $\mathbf{\Omega}$, denoted $\mathbf{\Omega_k}$, such that only parameters in $\mathbf{\Omega_k}$ have influence on the processing time of that actor. Coefficients $C_{k,\omega}$ for the parameters ξ_ω that do not belong to set $\mathbf{\Omega_k}$ are zero for any processor architecture. We have $\mathbf{\Omega} = \bigcup_k \mathbf{\Omega_k}$, and different sets $\mathbf{\Omega_k}$ may have common elements.

Thus, to construct a parameter function for a given actor v_k, two tasks need to be accomplished:

1) a proper subset of actor complexity parameters $\mathbf{\Omega_k}$ should be detected;

2) the coefficients $C_{k,\omega}$ should be computed for all suitable processor architectures represented in the target multiprocessor platform.

Those two tasks are interrelated, but in this subsection we first focus on the first one and then on the second one.

One way to specify the parameters of $\mathbf{\Omega_k}$ is first introducing their counter variables into the source code of the actor, and then, when this has been done for every actor v_k, determining which parameters of different actors are identical, in order to save in the number of parameters. Detecting the parameters of an actor and deciding on their number is not trivial for automation, and that subject deserves further investigation, which is out of the scope of this thesis. So far we have used only a manual approach that partly relies on familiarity with the application algorithm and on empirical data.

The parameter identification method we apply in this thesis uses the so-called *profiling* approach, measuring the processor clock cycles consumed by a running executable. Therefore we require a preliminary version of the application executable to be built and executed on the platform itself or on a sufficiently timing-accurate platform simulator. Note that the timing-accuracy requirement for a multiprocessor simulator is relatively easy to meet given our assumption that the local memory system of every processor is free of cache misses and bus/memory port conflicts at each actor execution.

The profiling infrastructure should support a profiling interface for measuring the number of processor cycles spent between any pair of user-specified breakpoints in the source code of each actor. In the infrastructure, it is favorable to assign no more than one process to one processor, thus avoiding the run-time scheduling. The point is that, in the presence of scheduling, one would have to separate the cycle counts of different processes. Note that not having to

implement the local processor schedulers and other operating system services may also greatly simplify the prototyping infrastructure and favor high simulation speed.

Here we describe the parameter identification method using a practical example: the variable length decoding (VLD) actor of the JPEG video image decoding application presented by Erwin de Kock in [46]. The input of that actor is the bitstream containing the coded representation of the image in the JPEG format and the starting position in the bitstream for a 16x16 pixel matrix, called a minimum coding unit (MCU). The bitstream represents an MCU using a combination of the Huffman-tree coding and the run-length coding techniques, as specified in the JPEG standard. The VLD actor applies the Huffman and the run-length decoding to obtain a representation of the pixel matrices in the discrete cosine Fourier domain, ready for the subsequent inverse Fourier cosine transformation to be carried out by another actor. One execution of the VLD actor processes one MCU.

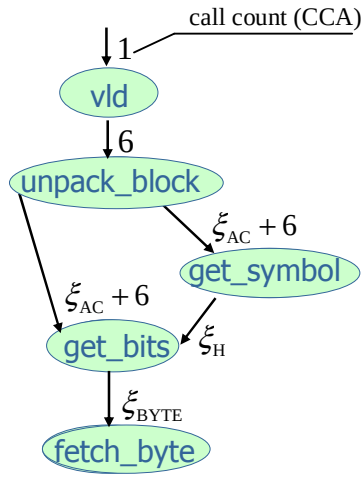
The method that we use builds a timing model of the actor internals by splitting the actor control flow into parts called subroutines and blocks. We call that model the *subroutine call graph (SCG)*; it is similar to the call graphs used in profiling. From the control flow, only the information relevant for the construction of the actor parameter function is preserved in an SCG. Thus, to a large extent, the model ignores the order in which different subroutines/blocks are executed inside the actor; it only counts the number of times they are executed. Given our earlier observation that the timing overlap between different parts of the actor can be efficiently accounted for or ignored, this is enough to construct a parameter function that achieves the desired level of accuracy.

Below we first give the summary of our method's algorithm and then explain it in more detail.

Algorithm (Informal): **Detecting the complexity parameters for a given actor**.

1. Build an initial SCG graph.
2. Determine actor parameters and *call count annotations* (CCAs) for the subroutines.
3. For each non-visited subroutine in the SCG:
    a. Build a control flow diagram, hiding superfluous details in blocks
    b. Determine the CCAs for each edge in the diagram
    c. Exclude the blocks with processing times showing considerable variations and insert them into the SCG as new subroutines
    d. Based on the CCAs of the diagram nodes, compute the parameter subfunction
4. Compute the actor parameter function as the sum total of the parameter subfunctions. ♦

To build an SCG at Step 1, one first has to identify the actor subroutines, which are modeled as graph nodes. The directed edges of the SCG graph specify the relation 'the source subroutine calls the sink subroutine'. The subroutines may be identical to procedure/function calls of the source code or may correspond to the source code segments that the designer wishes to separate from one another because they correspond to different stages in the application algorithm and it is easier for the designer to analyze them separately. The subroutines should not share any source code lines with one another, and they should cover together all the lines of the actor source code. Figure 3.3 shows the subroutine call graph for our example, the VLD actor. It contains four subroutines, the VLD actor itself and the functions that are called during the execution of that actor.



(a) the SCG

ξ_{AC} – the number of AC symbols in the MCU;

MCU – the portion of input data processed per execution of the VLD actor;

ξ_H – the total number of bits in the Huffman codes contained in the MCU;

ξ_{BYTE} – the total number of input stream byte boundaries crossed when reading the MCU bit-by-bit;

it also holds that:

$$\lfloor \xi_b/8 \rfloor \le \xi_{BYTE} \le \lceil \xi_b/8 \rceil$$

where:

ξ_b – the total number of bits in all codes of the MCU;

(b) relevant information about the VLD algorithm

Figure 3.3 VLD example: subroutine call graph (SCG) and parameter definition

The symbolic analytical annotations shown in the figure at the SCG edges are the call count annotations – CCAs, introduced at Step 2. A CCA stands for the total number of calls to the given subroutine per actor execution. If a subroutine is called from different subroutines, it has several CCAs and several incoming SCG edges that come from the calling subroutines[15]. In an actor SCG, there is always one and only one subroutine that has a CCA that is not associated with any incoming edge, and that is the subroutine corresponding to the actor itself, or the 'top-level' subroutine. By definition, that CCA is set to '1'. In Figure 3.3, the 'top-level' subroutine is VLD.

Being represented as symbolic expressions, the CCAs are different from the call counts in traditional profiling, because the profiling only provides the concrete values of data-dependent call counts measured for the given input data sample. In general, a CCA may constitute any analytical expression involving the complexity parameters and algorithmic constants. Those constants have concrete values, e.g. '1', '6' and '8' in Figure 3.3, and they are architecture-independent (as opposed to architecture-dependent constants, which we denote using character *C*, and use as the coefficients of parameter functions).

We see that specifying the CCAs is the core of this parameter identification method, because it is here where the designer directly introduces the actor parameters. A CCA is specified based on examining the source code of the calling subroutine and understanding the application algorithm.

For example, in the VLD algorithm, each MCU consists of 6 pixel blocks, and subroutine 'unpack_block' is called for each block, so it gets CCA 6 – see Figure 3.3. The coded representation of an MCU can be split into so-called 'DC' and 'AC' symbols, there being 6 DC symbols and a variable number of AC symbols per MCU. For each symbol, 'unpack_block' calls two subroutines: 'get_bits' and 'get_symbol'. Therefore, we assign to both subroutines a CCA equal to $\xi_{AC}+6$, where ξ_{AC} is the total number of AC symbols in the given MCU. The definition of all parameters in this example can be found in Figure 3.3. They should be clear for

[15] Thus, if a subroutine can call itself recursively, it gets a loop self-edge, for which it is both the source and the sink.

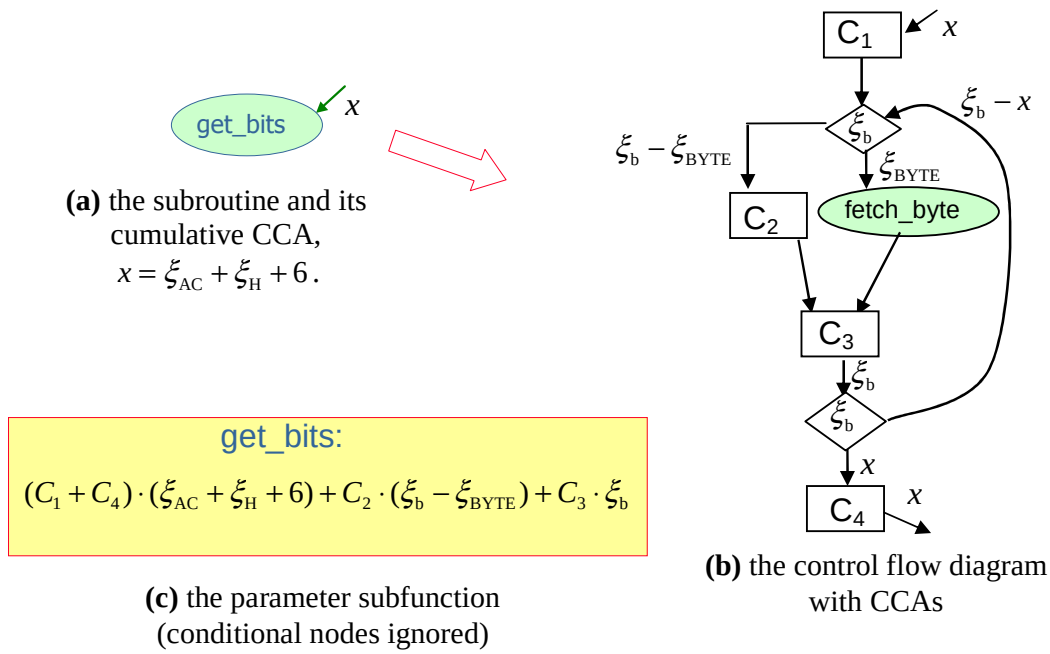


Figure 3.4 The derivation of a parameter subfunction using the control flow diagram

the readers familiar with the VLD algorithm, but it is not necessary to understand those definitions in detail to follow the explanation of this parameter identification example.

Before we turn to Step 3 of the parameter identification method, let us comment on Step 2 for the general case. The SCG example given in the figure only represents the ideal case, where the designer is able to relate all the CCAs to a small set of meaningful algorithmic variables – the parameters – that have a clear definition in terms of the application algorithm. Hereby, the purpose of the SCG graph is to provide an overview of the actor source code such that it is easier for a human designer to analyze it in a structural way and not to overlook important details.

In general, it may be the case that the actor algorithm is so complex that the designer is not capable of finding all the relationships between the CCAs and the algorithmic variables within the available time. In that case, he/she may choose to introduce a new parameter for each CCA. The price paid for simplification is possibly too many parameters. [16]

Step 3 of the identification method builds – in Step 3a – a *control flow diagram* (CFD) of the subroutine to be analyzed. Figure 3.4 shows an example of the diagram for the 'get_bits' subroutine of the VLD actor. The diagram consists of block nodes, denoted as C_p, conditional nodes and successor subroutine nodes. The block nodes, or blocks, are different parts of the subroutine that cover all the processing done by the given subroutine excluding the processing done by the successor subroutines, called from the given subroutine. A block may have any number of entry points, but it may have at most one exit point, so a given block node either serves as predecessor to another node – like block C_1 in the figure – or exits from the subroutine – like block C_4. Thus, a block node does not provide conditional branches, which is the task of the conditional nodes. A conditional node has at least two successors – the conditional branches,

[16] One can also try to use automated techniques for finding the source code variables that have the largest impact on the processing time, e.g., the methods proposed by Valetin Gheorghita *et al* in [24], [28]. One can focus on the analysis of only those parts of the source code where those variables and the directly related variables are involved and express the CCAs in terms of those variables.

and every time the control flow arrives at a conditional node, the node decides which conditional branch is taken.

Similarly to the edges of an SCG, the edges of a CFD have call count annotations (CCA), which use parameters. A CCA shows the number of times when the block at the source of the edge is followed by the block at the sink of the edge. For example, the edge from C_3 to the following conditional node has annotation ξ_b. Every CFD must have one or a few entry and exit edges. An *entry edge* of the diagram – in our example the edge that goes into node C_1 – is an edge that has no source node. Such an edge points to a node where the subroutine can be entered. Similarly, an *exit edge* – in our example the edge leaving node C_4 – is an edge through which the subroutine can be finished.

Not only the edges, but – for convenience – also the nodes may have CCA annotations – giving the total number of times the block is executed. For example, the conditional nodes in Figure 3.4 are annotated with CCA ξ_b. Note that the annotation of the nodes is superfluous if there is annotation of the edges, because – obviously – the CCA of a node is equal to the sum of CCAs of its incoming edges.

The assignment of CCAs in a control flow diagram – which is the task of Step 3b – follows the same guidelines as the assignment of CCAs to the subroutines – in Step 2. However, in addition, the designer can use the control flow diagram as extra help, because it imposes three rules on the CCAs. *Rule* 1 says that the sum of CCAs of all entry edges of the diagram is equal to the sum of the CCAs of all exit edges, which, in turn, is equal to the cumulative CCA of the subroutine being analyzed. The latter can be determined from the SCG graph as the sum total of the CCAs of the incoming SCG edges. In Figure 3.4, we denoted the CCA of the entry and exit edges as 'x', and from Figure 3.3 we see that $x = \xi_{AC} + \xi_H + 6$. *Rule* 2 says that the total CCA of the edges to a successor subroutine node is equal to the CCA of the edge between the current subroutine and the successor subroutine in the SCG graph. In Figure 3.4, we see that that this rule applies to the incoming edge of node 'fetch_byte'. *Rule* 3 says that the CCA of any node is equal to the sum of the CCAs of all incoming edges and to the sum of the CCAs of its outgoing edges. In our example, the last rule helps us to find the CCA of the edges whose CCA was not yet known after applying the first two rules. From the knowledge of the actor algorithm, we know that the loop inside the 'get_bits' subroutine executes one iteration per bit. Therefore, we annotate the conditional nodes of the loop with CCA ξ_b. Using Rule 3 and the annotation of the entry edge, we annotate the loop edge entering the top conditional node with CCA $\xi_b - x$. Because, the CCA of 'fetch_byte' is ξ_{BYTE}, we can immediately calculate the CCA of node C_2. At this point, it is obvious how to derive all the other CCAs in this example.

If the reader is familiar with WCET analysis methods, he/she probably has noticed similarity of our control flow diagram to the control flow graph being automatically built by WCET tools. It is, in any case, useful here to give some comment on the WCET approach. WCET analysis exposes so-called *basic blocks*, which are typically quite small (a few lines of the source code in a high-level language). The basic blocks are split from one another by definition at every conditional or Boolean operator exposing the latter as a conditional node in the control flow graph. Hereby, it is more or less safe to assume that a basic block typically consumes the same number of processor cycles (exceptions are processor instructions with data-dependent cycle counts). As opposed to the basic blocks in case of WCET analysis, in our case the designer may choose to hide much bigger parts of the source code inside the blocks, including conditional and

Boolean operators. Hereby we take care that the variations in the block processing time stay insignificant and come back to this issue in one of the next sub-steps, but at this sub-step it is definitely desirable to make the blocks as large as possible, to only expose the major details of the subroutine's control flow. Apart from the desire to keep the diagram manageable, we would thus avoid introducing multiple insignificant actor parameters, which might result from taking into account all the fine-grain details of the control flow. Ignoring parameters with insignificant impact on the processing time helps to reduce the performance analysis overhead.

To explain Step 3c – i.e. the exclusion of blocks with variable processing times – we first make a remark that a major hypothesis of the described method is that each block consumes a constant number of the processor clock cycles every time it is called. In such an ideal case, this method would provide an ideally accurate parameter function; the higher the variations of the block processing time, the higher the error. Therefore, in Step 3c, for each block, we check the magnitude of variations of its processing times, and the blocks with considerable variations are turned into subroutines so that those blocks can be split into smaller blocks in the later iterations of the algorithm. The corresponding block nodes of the diagram are changed into subroutine nodes, and they get into the list of non-visited subroutines. Note that the described algorithm finally converges because splitting of blocks cannot continue indefinitely; in the worst case one gets to the basic-block level of granularity, and the basic blocks have stable processing times. To decide whether the processing of a block is stable enough, one can, for example, measure its processing times for a representative sequence of input data samples and build a histogram of the measured processing times.

In the last sub-step of Step 3, we use the CCAs of the diagram nodes to find the total contribution of the given subroutine to the actor processing time excluding the contribution of successor subroutines. We call it the parameter subfunction of the subroutine. We also use the term 'exclusive' for it, because it excludes the successors. Just as the CCAs, it is computed in symbolic form, as the sum total of the contributions of all blocks and conditional nodes. For every block C_p we use symbol C_p to denote its processing time, which is assumed to be constant. The contribution of each block is its processing time C_p times its CCA. The number of the processor cycles consumed by a conditional block may depend on which of its branches is taken. The contribution of a conditional block is thus a sum total of a few terms defined as the branch's cycle cost times the CCA of the branch. Note that, if blocks are large enough, the contribution of the conditional nodes is small compared to the contribution of the blocks, and then the conditional nodes can be ignored, just as it is done in Figure 3.4.

In Step 4, when all subroutines have been visited, the designer adds the subfunctions together and obtains the final actor parameter function. Thereby, the designer has certain freedom in deciding which variables will finally act as parameters, trying to arrive at an expression using as few parameters as possible and still being accurate enough. First of all, the designer may try to group the variables that belong to the same CCA to one parameter, because those variables have the same coefficient. Also, the designer has to check that the set of all parameter variables extended by a non-zero constant forms a linearly independent set; the check can be performed empirically or based on the knowledge of the algorithm. This condition can be violated e.g. when one parameter is proportional to another one or differs from another one by a constant. Such a linear dependency is eliminated by replacing one variable by a linear combination of other variables plus a constant value. This way, the designer also ensures that all parameters are, in fact, variables, such that no parameters have constant values for any input sequence. Such

parameters should be excluded from the parameter set by replacing them by their constant values. It is also desirable to exclude approximately-linear dependent parameters, i.e. such parameters that stay in a relatively small neighborhood to a linear combination of other parameters.

Let us illustrate the reduction of the parameter set by an example. If the designer decides that all variables in the subfunction in Figure 3.4(c) are to be parameters, then we obtain a set with 4 parameters. However, in that subfunction, parameters ξ_{AC} and ξ_{H} have the same coefficient, and thus they can be grouped together into one parameter, equal to expression $\xi_{\mathrm{AC}} + \xi_{\mathrm{H}} + 6$. Also, according to information in Figure 3.3, parameters ξ_{b} and ξ_{BYTE} have an approximately linear relationship between one another, to stay conservative one can replace term '$-\xi_{\mathrm{BYTE}}$' by its upper bound $-(\xi_{\mathrm{b}} - 7)/8$, because it holds that $(\xi_{\mathrm{b}} - 7)/8 \le \lfloor \xi_{\mathrm{b}}/8 \rfloor \le \xi_{\mathrm{BYTE}}$. Thus, the number of parameters for that subfunction can be reduced to two.

Unfortunately, for the total VLD actor parameter function, saving in the number of parameters by grouping ξ_{AC} and ξ_{H} does not work, because, as we see from Figure 3.3, the call count of subroutine 'get_symbol' includes only one of those parameters. Fortunately, the expansion of the other subroutines into control flow diagrams does not yield extra parameters, and finally we obtain an actor parameter function with three parameters:

$$t(v_{\mathrm{VLD}}, n) = C_{\mathrm{VLD,0}} + C_{\mathrm{VLD,H}} \cdot \xi_{\mathrm{H}}(n) + C_{\mathrm{VLD,AC}} \cdot \xi_{\mathrm{AC}}(n) + C_{\mathrm{VLD,b}} \cdot \xi_{\mathrm{b}}(n) \qquad (3.7)$$

The designer can express each actor coefficient $C_{\mathrm{VLD},\omega}$ as a simple algorithm-specific linear function of the various C_p, i.e., the processor cycle costs of different blocks and conditional nodes. For example, from Figure 3.4(c), it is obvious that $C_{\mathrm{VLD,0}}$ contains term $(C_1 + C_4) \cdot 6$. This fact is used in one of the methods for computing the actor coefficients, presented in Section 3.3.

3.3 Calculating Actor Coefficients

Having obtained the symbolic expression, to finalize the construction of the parameter function, one has to calculate the values of the coefficients for different processor architectures represented in the target platform. Recall that, in terms of our design flow, we refer to this task as *actor-level characterization*, i.e., the characterization of actor execution delays.

In line with our requirements to the performance analysis method, we strive to obtain conservative estimates for the coefficients – upper bounds. Nevertheless, we base our method on the profiling approach, i.e., we use measurements of the processing times obtained from running an application executable with certain representative input data sequences. That approach, being to a large extent empirical, does not always yield upper bounds that are reliable for 100% of the actor executions. The advantage of this approach is, however, that it requires less sophisticated design automation tools effort as, for example, WCET analysis tools. We believe that our profiling-based methods are suitable in for our conservative performance analysis framework for three reasons:

1) Our methods allow controlling the level of confidence to any desired level below 100%.

2) The transition from the detailed actor timing mode to the multi-scenario delay mode – later on in the implementation trajectory – increases the model pessimism, thus compensating for occasional lack of pessimism in the detailed mode.

3) The chosen application domain (i.e. multimedia streaming) mostly includes soft real-time applications, which can tolerate deadline misses if their probability is low enough.

We use two alternative actor-level characterization methods:

1) direct measurement combined with control flow analysis;

2) linear regression with consideration of confidence intervals for the coefficients.

The direct measurement method is more laborious, but it can, in general, provide more reliable results than the second method. In particular, if one invests enough effort into the direct measurement method, one can obtain values of coefficients that are 100% guaranteed to be conservative; we say that in this case we have obtained a *strictly conservative parameter function*. The method is based on the measurement of the longest processing time of every block C_p and every conditional node in the control flow diagrams of the actor subroutines. The effort to ensure that the processing times measured are really the worst-case times may range from simply registering the longest time ever measured in an arbitrarily chosen long input sequence to artificially creating an input sequence where the worst-case path conditions for the given block really occur and using that sequence to directly measure the worst-case processing time[17]. Once we have measured the worst-case processing times C_p of all nodes, we can derive the actor coefficients $C_{k,\omega}$ using the relationship between C_p and $C_{k,\omega}$ described at the end of the Section 3.2.2.

We applied this method to the VLD actor using the JPEG executable presented in [46], which we ran on the ARM7TDMI™ processor architecture using the ARMulator™ simulator [3] and assuming a single-cycle access to the local memory. When measuring the costs of the blocks, we have ensured that we obtain the delays of their worst execution paths (by making sure that the worst-path conditions for every block occur in the representative input stream). The results are presented in Figure 3.5, where we see that the actor parameter function is indeed an upper bound, still being very close to the real processing time measurements.

The linear regression method with consideration of confidence intervals yields actor-level coefficient estimates that are upper bounds with a probability close to 100%. Due to the fact that the probability is still below 100%, we call such a parameter function a *weakly conservative parameter function*. The advantage of the linear regression method is that it requires less routine work to be done. For this method, one only needs to perform processing time measurements for whole actors rather than for separate code blocks contained in the actors.

Note that linear regression is most often used to obtain linear coefficients that are conservative with only a 50% probability. As explained below, one can exploit so-called confidence intervals, calculated during linear regression as a by-product, to increase this probability to any desired level below 100%.

In the rest of this section, we give a detailed explanation of the use of linear regression and the confidence intervals for a weakly conservative parameter function.

[17] In this extreme case, the direct measurement method almost ceases to be a ‘profiling-based’ method, and resembles the WCET method.

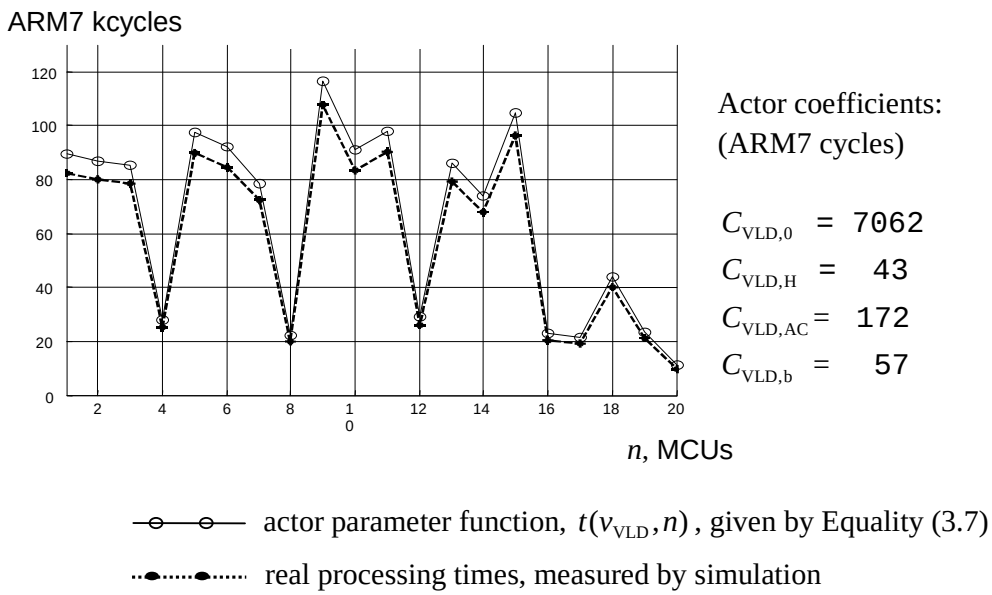


Figure 3.5 The processing times of VLD actor for 'PHILIPS' logo JPEG image

We assume that the subset of parameters for every actor v_k is known (having been derived e.g. using the method from the previous section). We denote that subset as $\mathbf{\Omega_k}$. Let $\boldsymbol{\xi}_k(n)$ denote a column-vector whose first element is '1' and the other elements are parameters in set $\mathbf{\Omega_k}$; i.e. $\xi_k(n) = [1, \xi_1(n), \xi_2(n), ,..., \xi_{\Omega k}(n)]^{\mathbf{T}}$, where $\Omega_k = |\mathbf{\Omega_k}|$ and n is the actor execution index. (Here, without loss of generality, we assume that the parameters ξ_ω contained in set $\mathbf{\Omega_k}$ have indices ω in the range $1 \ldots \Omega_k$.) Let $\mathbf{c}_k$ denote the column-vector of the corresponding coefficients $C_{k,\omega}$. Then, using matrix algebra, we can rewrite Equality (3.6) as an inner product of two vectors:

$$t(v_k, n) = \boldsymbol{\xi}_k(n)^{\mathbf{T}} \cdot \mathbf{c}_k \qquad (3.8)$$

The linear regression method for computing vector $\mathbf{c}_k$ requires a sequence of values of $\boldsymbol{\xi}_k(n)$ for $n = 0 \ldots N - 1$, where $N > \Omega_k$. Let us consider how to obtain that sequence in practice.

First of all, one needs to create a version of the application executable instrumented with parameter counters, as discussed in Section 3.2.1. Let us call that version of the executable a 'counter-instrumented' version, as opposed to the original version – or the 'normal' version. Secondly, one needs a sample of input data streams with enough data for at least N iterations of the loop of interest.

Performing a run of the counter-instrumented executable with the sample input data results in N samples of parameter vectors: $\xi_k(0),...,\xi_k(N-1)$. For the linear regression method to work, one has to ensure that the sequence is 'rich enough', i.e., it must contain $\Omega_k + 1$ linearly independent vectors. This is possible only when the linear dependency between the parameters has been excluded, as described in the previous subsection. From our experiments with the arbitrary-shaped video decoding application, described in Chapter 6, we observed that, for applications similar to that one, one is likely to have a rich enough input data sequence if it satisfies the following conditions:

1) $N >> \Omega_k$,

2) all the possible video frame types and video block types are encountered in the stream.

All the samples of the parameters together form a matrix, denoted $\mathbf{\Xi}_{\mathbf{char}\ k}$, that characterizes the given input stream:

$$\mathbf{\Xi}_{\mathbf{char}\ k} \equiv \begin{bmatrix} \{\xi_k(0)\}^{\mathbf{T}} \\ \{\xi_k(1)\}^{\mathbf{T}} \\ \dots \\ \{\xi_k(N-1)\}^{\mathbf{T}} \end{bmatrix} \tag{3.9}$$

In addition to this matrix, the linear regression method also needs a column-vector of the actor processing times, $\mathbf{t}_{\mathbf{prof}\ k}$, measured by the profiling tools of the target platform by running the 'normal' executable and feeding it with the same input stream samples.

In the ideal case, our processing time model given by Equality (3.8) would exactly represent the measured processing times. Then, whatever rich enough input sequence one would use to generate $\mathbf{\Xi}_{\mathbf{char}\ k}$ and $\mathbf{t}_{\mathbf{prof}\ k}$, one would always obtain the same vector of 'ideal' coefficients, $\mathbf{c}_{\mathbf{ideal}\ k}$, by solving the system of N linear equations with Ω_k +1 variables:

$$\mathbf{\Xi}_{\mathbf{char}\ k} \cdot \mathbf{c}_{\mathbf{ideal}\ k} = \mathbf{t}_{\mathbf{prof}\ k} \tag{3.10}$$

In that case, $\mathbf{c}_{\mathbf{ideal}\ k}$ would be equal to the unique solution of that system, which could be obtained by first reducing the number of equations in Formula (3.10) to $\Omega_k + 1$ and then by applying the standard techniques for solving a linear system with a square matrix.

In general, the ideal fit, $\mathbf{c}_{\mathbf{ideal}\ k}$, does not exist, because in addition to the actor parameters identified by the designer there can be other parameters contributing to the variability of the actor processing time. In that case, one can expect that the linear system given by Equality (3.10) does not have any exact solutions; this is likely to be the case because the number of equations N is (much) larger than the number of variables. Even if one is 'lucky' and an exact solution for the given $\mathbf{t}_{\mathbf{prof}\ k}$ still exists, for another sample sequence, there may be no exact solution. In general, placing any vector of coefficients $\mathbf{c}_{\mathbf{try}\ k}$ in the left part of Equality (3.10) would lead to some mismatch at the right-hand part:

$$\mathbf{\Xi}_{\mathbf{char}\ k} \cdot \mathbf{c}_{\mathbf{try}\ k} = \mathbf{t}_{\mathbf{prof}\ k} + \boldsymbol{\varepsilon}_{\mathbf{mismatch}\ k} \tag{3.11}$$

Therefore, in general, it is only possible to offer a conservative solution, ensuring that the mismatch is often or always positive. To arrive at such a solution using linear algebra, one can, for example, think of the following approach. One can extract from the set of measured parameter vectors a number of subsets with $\Omega_k + 1$ linearly independent vectors. Solving the system of equations given by Equality (3.10) for each subset yields an exact solution for that subset. Maximizing every coefficient from the obtained series of solutions yields coefficient values, such that the longer the series of solutions considered the higher the probability that the calculated coefficients are conservative. However, this approach has no theoretical basis that would enable the concrete estimations of that probability.

In contrast to that, the linear regression method is standard, well studied, and it can achieve the required results. Nevertheless, we still present some linear regression details in order to have a convenient way of showing how we deviate from the standard usage of that method so that conservative values of actor-level coefficients are obtained.

The 'standard' use of the linear regression method calculates solution $\mathbf{c}_{\mathbf{try-best}\,k}$ leading to the smallest sum of squares of the mismatch. The method computes it as a function:

$$\mathbf{c}_{\mathbf{try-best}\,k} = f_{\text{linear\_regression}\,\mathbf{1}}\left(\mathbf{\Xi}_{\mathbf{char}\;k}, \mathbf{t}_{\mathbf{prof}\,k}\right) \qquad (3.12)^{18}$$

'Traditionally', the linear regression method implies selecting $\mathbf{c}_{\mathbf{try-best}\,k}$ as the preferred solution, but we do not follow that approach. As already mentioned, the reason for that is that $\mathbf{c}_{\mathbf{try-best}\,k}$ is not a conservative solution; on the contrary, it ensures that the underestimation and the overestimation are balanced: $\sum_{n=1}^{N}\{\varepsilon_{\text{mismatch}\,k}\}_n = 0$. Instead, our method obtains a statistical upper bound on $\mathbf{c}_{\mathbf{try}\,k}$ such that the mismatch is more likely to be positive, but not 'too far off', to keep the error of our linear model under control.

As we already mentioned, we use the confidence intervals of the coefficients to achieve this goal. To explain the meaning of the confidence intervals, we have to consider some basic facts about the linear regression method. In fact, linear regression assumes that the measured values $\mathbf{t}_{\mathbf{prof}\,k}$ are samples of a variable – lets denote it $t_{\text{exact}\,k}$ – that is an exact linear combination of the parameters[19], but the measurements of that variable contain a random error $\varepsilon_{\text{error}\,k}$. In order for our following statements to hold, it is required that the setup must satisfy a few basic general requirements. Let us assume, for the time being, that these requirements hold and come back to them later.

According to the theoretical results on linear regression, $\mathbf{c}_{\mathbf{try-best}\,k}$ given by Equality (3.12) also appears to be the optimal estimate of the coefficients of the exact linear combination, denoted $\mathbf{c}_{\mathbf{exact}\,k}$. However the exact coefficients are unlikely to precisely coincide with the optimal estimate; they are more likely to be located in a certain neighborhood of $\mathbf{c}_{\mathbf{try-best}\,k}$. Linear regression can estimate the boundaries of such neighborhood $\mathbf{c}_{\mathbf{try-best\,min}\,k}$ and $\mathbf{c}_{\mathbf{try-best\,max}\,k}$ such that

$$1 \le p \le \Omega_k + 1 \;\Rightarrow\; \mathcal{P}r\left\{\{\mathrm{c}_{\text{try-best min}\,k}\}_p \le \{\mathrm{c}_{\text{exact}\,k}\}_p \le \{\mathrm{c}_{\text{try-best max}\,k}\}_p\right\} = p_{\text{coef}} \qquad (3.13)$$

where '$\mathcal{P}r$' stands for probability and $0 < p_{\text{coef}} < 1$ is a control setting, usual set to '0,95' in practice. As we see in Equality (3.13), p_{coef} controls the degree of confidence in the assumption that the exact coefficients are within the specified neighborhood interval from the calculated coefficients.

Interval [$\mathbf{c}_{\mathbf{try-best\,min}\,k}$, $\mathbf{c}_{\mathbf{try-best\,max}\,k}$] is called a *confidence interval*. The best solution lies in the center of the confidence interval:

$$\mathbf{c}_{\mathbf{try-best}\,k} = \left(\mathbf{c}_{\mathbf{try-best\,max}\,k} + \mathbf{c}_{\mathbf{try-best\,min}\,k}\right)/2 \qquad (3.14)$$

The bounds of the interval can be derived based on the obtained measurements:

$$\mathbf{c}_{\mathbf{try-best\,max}\,k} = f_{\text{linear\_regression}\,\mathbf{2}}\left(\mathbf{\Xi}_{\mathbf{char}\;k}, \mathbf{t}_{\mathbf{prof}\,k}, p_{\text{coef}}\right) \qquad (3.15)^{20}$$

---

[18] $f_{\text{linear\_regression}\,\mathbf{1}}(\mathbf{B},\mathbf{x}) \equiv \left(\mathbf{B}^{\mathbf{T}} \cdot \mathbf{B}\right)^{-1} \cdot \mathbf{B}^{\mathbf{T}} \cdot \mathbf{x}$

[19] Note that the literature on the linear regression often uses the term 'parameters' for what we call 'coefficients' and the term 'variables' for what we call 'parameters'.

[20] This formula just indicates that $\mathbf{c}_{\mathbf{try\text{-}best\,maxk}}$ is derived from the measured samples; the exact formula can be found in the special literature on the linear regression topic.

which also gives possibility to compute $\mathbf{c}_{\mathbf{try-best\,min}\,k}$ using Equality (3.14), but we are more interested in the upper bound.

Our method chooses $\mathbf{c}_{\mathbf{try-best\,max}\,k}$ as the preferred solution. The question arises on what can be said regarding the conservativity of this choice. As we already mentioned, it is the matter of 'probabilistic' or *weak* conservativity, as we named it. From the fact that the distribution of $\mathbf{c}_{\mathbf{exact}\,k}$ is symmetrical and from Equality (3.13), it follows that the probability of one coefficient being conservative is: $(1+p_{\text{coef}})/2 = 0{,}975$ (assuming $p_{\text{coef}} = 0{,}95$). Using the basic properties of probability values, we conclude from that that the probability that all coefficients are simultaneously conservative is between $0{,}975 - 0{,}025 \cdot \Omega_k$ and 0,975. Thus, to ensure that even in the worst case that probability is high, one can keep the number of parameters small, which can be achieved by splitting the actor body into two or more components and performing the regression for different components separately. Having this in mind, we observe that in our experiments with the video decoder that no more than nine parameters were required. Thus, in those experiments, one can be sure that the result of the actor parameter function is greater than the result of the exact linear timing model with probability at least 75%[21]. Note that one can increase the level of assurance to any required level, because $f_{\text{linear\_regression}\,\mathbf{2}}$ in Equality (3.15) supports any probability threshold $0 < p_{\text{coef}} < 1$.

As already said, these theoretical calculations hold exactly if certain assumptions about the experimental setup hold. A major assumption is that the exact linear timing model $t_{\text{exact}\,k}$ exists. That assumption is supported by an observation made in the previous section. Namely, we have observed that, by splitting the blocks into control flow nodes and introducing more parameters, the designer approaches an exact linear model. Another major assumption is that the measurements are such that the mean value of $\varepsilon_{\text{error}\,k}$ is 0. In fact, this requirement boils down to the requirement that one can define a probability distribution for $\varepsilon_{\text{error}\,k}$. This follows from the fact that in our case the error is bounded in a finite range, because the measured processing times are limited by the actor WCET and every actor parameter is also bounded. Because $\varepsilon_{\text{error}\,k}$ is bounded, it has a finite mean value. If that value is different from zero – i.e. if the error has a systematic component – then one can redefine $t_{\text{exact}\,k}$ by subtracting the systematic component from it and the new mean value of $\varepsilon_{\text{error}\,k}$ will be equal to zero. The next assumption in the linear regression method is that the error values $\varepsilon_{\text{error}\,k}$ in the N experiments are mutually independent. This requirement can be satisfied in practice as follows. Instead of selecting the N samples for the regression from the parameters/processing times measured for *subsequent* data items of the input data stream – which are likely to be dependent on one another – one can first collect a much larger set of samples, and then *randomly* select from those samples a subset with N samples.

If the abovementioned assumptions are satisfied, then there is only one requirement left that is sufficient to make the probability estimates given above to hold *strictly*. Namely, $\varepsilon_{\text{error}\,k}$ should be distributed according to the normal probability distribution law. This requirement is the strongest one and requires a special discussion.

---

[21] This observation is based on pessimistic mathematical reasoning; in practice we have not encountered any influence of splitting the actor into components on the probability of overestimation; this question has not been investigated in detail, although we have witnessed that splitting the actor into components can reduce the mismatch of regression.

Firstly, the normality of the distribution law is only a sufficient, but not a necessary condition for such characteristics of the linear regression as the confidence intervals to hold strictly. Moreover, these characteristics are known to be robust against violation of the normality of the distribution law.

Secondly, according to a well-known observation of C. F. Gauss[22], in many physical experimental setups, $\varepsilon_{\text{error}\,k}$ is typically composed of multiple error contributors that are stochastically independent of each other thus satisfying the so-called central limit theorem of probability theory, which states that the more such contributors there are the closer the distribution of $\varepsilon_{\text{error}\,k}$ to the normal distribution. In our measurements, the error contributors are blocks that have variable processing times. If one can split the blocks into multiple groups such that each group brings a contribution that is stochastically independent of the contributions of the other groups, then Gauss' observation is applicable. Recall that the contribution of a block is its CCA times its processing time. One can imagine, that if the algorithm of an actor is complex enough, its implementation can contain blocks whose CCAs and processing times are not directly related and are stochastically independent. At the same time, we must admit that, especially for smaller actors, one cannot always apply Gauss' observation. For example, in the VLD actor, the parameters (and consequently, CCAs), although being linearly independent are clearly stochastically dependent, because, typically, the greater the number of AC symbols in an MCU, the more bits are required to encode it.

Whether we can fully rely on probability and confidence interval estimates given by linear regression, needs further study and is beyond the scope of this thesis. Nevertheless, we believe that in many cases the major theoretical results can work at least as good approximations. Linear regression is very widely used in practice and one can find many literature references on the subject, e.g. [14].

We finish the discussion of the described linear regression-based method by a summary on linear regression, looking at our statements more from the practical point of view and making additional remarks.

For the linear timing models of the actors of streaming applications, given that the measured sequence of parameter vectors $\mathbf{\Xi}_{\mathbf{char}\ k}$ is 'rich enough' in the sense defined above, the method can calculate two statistical bounds $\mathbf{c}_{\mathbf{try-best\,min}\,k}$ and $\mathbf{c}_{\mathbf{try-best\,max}\,k}$. Vector $\mathbf{c}_{\mathbf{try-best\,max}\,k}$ is chosen as the preferred solution for a weakly conservative estimation of the real processing times; therefore, in case linear regression is used, the processing time model is defined as the model obtained by putting those coefficients into Equality (3.8):

$$t(v_k, n) = \boldsymbol{\xi}_k(n)^{\mathbf{T}} \cdot \mathbf{c}_{\mathbf{try-best\,max}\,k} \qquad (3.16)$$

The obtained model, $t(v_k, n)$, is weakly conservative in the following sense. Suppose that $t_{\text{exact}\,k}(n)$ are the values of $t(v_k, n)$ we would obtain if each block in the actors control flow had constant processing time equal to the long-run average of processing times they have in reality. Then,

$$\Omega_k = 0, 1, \ldots 9 \;\Rightarrow\; 75\% \le \mathcal{P}r\left\{ t_{\text{exact}\,k}(n) \le t(v_k, n) \right\} \le 97{,}5\% \qquad (3.17)$$

which means that we stay above the exact average with probability at least 75%.

[22] Carl Friedrich Gauss (1777-1855) is a famous German mathematician, whose work has made a major impact on many mathematical disciplines, but also on physics and on other sciences.

The worst-case probability of 75% can be increased to any required level below 100% if one increases p_{coef} from its default value '0,95' to a higher value. Note that although the linear regression method allows the actor-level parameter function to be non-conservative for a certain (typically small) part of actor executions, we see in practice that this is at least partially compensated by other ingredients of our performance analysis method. In particular, when we step from the detailed timing mode to multi-scenario timing mode (described in Chapter 5), this makes our timing model more conservative. Also, when we integrate the actor execution delays to calculate the loop execution time, we see that the less probable non-conservative estimations in a certain part of actor executions is compensated by more probable conservative estimations in the other part.

Before we complete the description of the linear regression method, we mention two quality metrics for the linear model obtained using linear regression. The first metric gives an estimate of the *relative error of the coefficients*, which we denote as $err(\mathbf{c})$. It is given by:

$$err(\mathbf{c}) \equiv \max_{p=1..\Omega_k+1} \frac{\{\mathrm{c}_{\text{try-best max } k}\}_p - \{\mathrm{c}_{\text{try-best min } k}\}_p}{\{\mathrm{c}_{\text{try-best } k}\}_p} \tag{3.18}$$

The second metric, the so-called *coefficient of determination*, R^2, estimates how well the terms of the obtained linear model account for the variability of $t(v_k, n)$. For example, R^2 =99% would mean that the actor parameters account for 99% of the variability of the processing time. If R^2 is small, it would indicate that one should refine the model and add more parameters to it, e.g. by doing a more detailed control flow analysis. The formula for calculating R^2 can be found in the literature, e.g. [14].

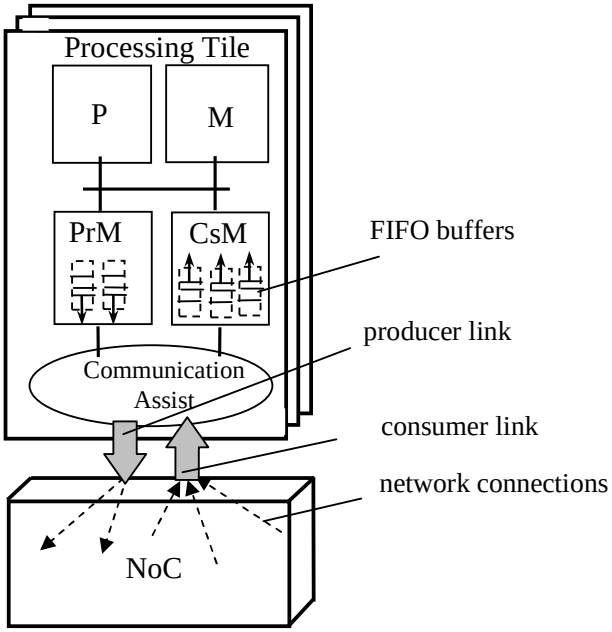
We applied the linear regression method to compute the coefficients of several actors in the MPEG-4 arbitrary-shape video decoding application running on the ARM7TDMI-based processor. We have experienced that this method requires considerably less manual effort than the direct measurement of the coefficients and gives good results, although, as one can already expect, the obtained models are not strictly conservative. We performed a few experiments, where we checked the accuracy and the conservativity (the frequency of overestimation) of the linear model obtained from a sample of one MPEG-4 sequence against a few other real MPEG-4 sequences. However, we postpone a detailed report on the results until Chapter 6.

3.4 Generic Multiprocessor NoC Architecture

3.4.1 Background

As we have said before, the HSDF models we propose in this thesis co-model computation and communication. In Section 2.2, we used related work involving bus-based communication to introduce IPC graphs. However, in this thesis, we assume architectures with *network-based* communication, which is the main topic of this section.

D. Culler *et al* [18 - §1] describe a general template for multiprocessors. It consists of multiple *processing tiles* connected with each other by an interconnection network. Each tile contains a few processor cores and local memories. If the hardware architecture and the operating conditions (clock frequency/supply voltage) of all the tiles are identical, then the multiprocessor is *homogeneous*; otherwise it is *heterogeneous*. The processing tiles contain communication buffer memories – referred to simply as *buffers*. The buffers are accessed both



P – processor
M – local general-purpose memory
PrM – local producer communication memory
CsM – local consumer communication memory

Note: M, PrM, CsM can be integrated into one physical memory but then it might become too expensive in terms of area/power to have one large double-ported memory. Alternatively, one single-ported memory can be used with appropriate arbitration between the processor and the communication assist making their memory access times independent.

Figure 3.6 Architecture template

by the local processor and the network. Each tile has a small controller, called a *communication assist*, that performs buffer accesses on behalf of the network. Many embedded MP-SoCs implemented on silicon, e.g., Daytona [1], AxPe [82], Prophid (CPA) [84], Cradle [17] and HIJDRA [8], [7], fit nicely into this general template. Cradle, Prophid and HIJDRA are heterogeneous platforms, whereas AxPe and Daytona are homogeneous.

Among these architectures, Prophid and HIJDRA are the most interesting ones for us, because they use a packet-switched communication network and provide performance guarantees for hard real-time tasks. Prophid contains application-domain-specific processors communicating through a switch, based on a time-division multiple access (TDMA) scheme, enabling guaranteed-bandwidth communication. HIJDRA uses multiple switches, referred to as *routers*, joined in a certain topology. In Prophid and HIJDRA, different tasks running on the multiprocessor communicate with each other using asynchronous message passing, meaning that processors synchronize based on the availability of data in buffers. Message passing introduces the buffer overflow issue, which is solved using a kind of end-to-end *flow control*. We reuse some ideas of the Prophid and HIJDRA architectures in our architecture template.

3.4.2 Architecture Template

The MP-NoC architecture template adopted in our work is shown in Figure 3.6. We focus on defining the issues important for the mapping flow and IPC graphs introduced in the rest of this chapter.

The interconnection network in our template is a network-on-chip (NoC). Each processing tile is 'plugged' into the network through a producer link and a consumer link. The link names illustrate that the processing tile produces data on one link and consumes data on the other link. The NoC offers unidirectional point-to-point connections. The connections must provide guaranteed bandwidth and a tightly bounded propagation delay. The connections must also preserve the ordering of the communicated data. The details of the NoC implementation (e.g. the architecture of the network router) are not important and not exposed in this template.

An example of a NoC providing these properties is ÆTHEREAL [81], [30]. It uses the TDMA scheme, for which efficient implementations of the network routers are possible. Note that several other schemes ensuring guaranteed performance in communication networks exist in the literature, but we are not aware of any of them having been implemented so far in the context of NoCs. The choice of the particular scheme does not influence the main idea and the structure of the timing models we present in this chapter.

To keep the processing tile simple, we assume only one processing core (denoted 'P' in Figure 3.6) per tile. In future work, our design trajectory can be extended to support multiple processors per tile. The local memory layout contains three blocks: the general-purpose memory ('M') for processor instructions and data, the producer communication memory (PrM) and the consumer communication memory (CsM). PrM and CsM have ports for the processor and for the communication assist.

Connections are the key logical elements for the implementation of the communication channels. Recall that the latter are the basic communication entities of applications. The communication channels that make use of the NoC are managed by the communication assists. Each channel connects two different processing tiles and transfers data in one direction. A communication channel is implemented by a *producer buffer*, a *data connection*, a *flow-control connection* and a *consumer buffer*. The buffers are located in the communication memories (PrM and CsM) at different sides of the channel. Every channel pumps data from its producer buffer to its consumer buffer through the data connection. Every communication assist can run multiple channels concurrently. The example tile in Figure 3.6 has three incoming and two outgoing channels. (Note that, in the literature, the usage of terms 'connection' and 'channel' can be different, even opposite to the usage adopted in this thesis.)

For our timing models to be valid, we require that the following memory accesses be independent:

1) the processor and the communication assist accesses to PrM and CsM;

2) the communication assist accesses to PrM and CsM;

3) the accesses initiated by different channels through the same PrM/CsM port.

In this context, with 'independence' we mean that the variation of the access time due to contention (if any) is sufficiently small. Requirement 1) is achieved by using a dual-ported communication memories and requirement 2) is achieved by making sure that the communication assist can access both those memories simultaneously. Note that we do not

necessarily demand that the memory layout be exactly the same as in the figure, as long as the independence requirements are satisfied. Requirement 3) can be satisfied if an appropriate arbitration scheme is used for the PrM/CsM ports. We believe that this can be achieved at a reasonable cost. Working out the details of such an arbitration scheme is a subject of future work. Note that recent publications of Arno Moonen *et al* [61], [62] motivate the use of a communication assist as mediator between a local processor memory system and the on-chip-network. They investigate a concrete arbitration scheme for PrM and CsM accesses and show the advantage of PrM memories for the multiprocessor performance due to the decoupling of local memory systems of on-chip processors from the network-on-chip communication medium.

The communication of a data token through a communication channel is realized as follows. First, the processor at the producer side puts a token into the producer buffer, where it waits for the tokens in front of it (in that buffer) to depart. Then, the local communication assist transfers the token into the producer link. Due to limited bandwidth of the data connection, this operation takes a certain time to execute, which we call a *transfer delay* (d_{transfer}). In the remainder, we often use the word '*transfer*' to refer to the activity of pushing the data token from the producer buffer into the network. In that sense, we may say that the transfer delay is a delay of one transfer. Note that the term 'communication transfer' we introduced in Section 2.1 is more extensive, because it also implicitly includes the arrival of the data token in the consumer buffer.

The exact value of d_{transfer} depends on the NoC implementation. In an ideal network:

$$d_{\text{transfer}}^{(\text{ideal})} = z_{\text{token}}/B_{\text{conn}}, \tag{3.19}$$

where B_{conn} is the bandwidth of the connection in bytes per second and z_{token} is the size of the token in bytes. Recall from Figure 3.1 that we call such a relationship between the communication delays and the data token sizes the *communication delay relation*.

In a real network, the transfer delay also includes the medium access delay, which depends on the scheduling mechanisms used in the network routers for multiple data packets competing for the same link. Because we deal with real-time applications, we choose to restrict ourselves to networks providing *timing guarantees*, i.e., supporting guaranteed upper bounds on the communication delays. Just as in the case of the processor scheduling, to ensure the independence of the timing behavior between different channels of different applications, we require that the data transfers in the network connections be also scheduled using budget-provision techniques. For those techniques, by analogy to Equality (3.1), we can write:

$$d_{\text{transfer}} \leq z_{\text{token}} / B_{\text{conn}} + \hat{q}_{\text{conn}} \tag{3.20}$$

where $\hat{q}_{\text{conn}}$ is a worst-case degradation of connection delay compared to ideal delay.

Equality (3.20), although being true for any budget-provision network scheduling, is not always the best way to express the transfer delay in practice, because, by analogy to Equalities (3.4) and (3.5), tighter bounds can be obtained by using knowledge about the particular scheduling method involved.

For the practical examples on the computation of the communication delays, we use better upper bounds that hold for the ÆTHEREAL NoC of [81]. That NoC uses TDMA scheduling and therefore an upper bound on the transfer delay $d_{\text{transfer}}^{(\text{Æ})}$ can be expressed in a way similar to Equality (3.4):

$$Z = \left\lceil \frac{z_{\text{token}}}{Z_{\text{min-Æ}}} \right\rceil \cdot Z_{\text{min-Æ}}; \quad D = \frac{Z}{B_{\text{LINK-Æ}}}; \qquad T_{\text{B conn}} = T_{\text{T Æ}} \frac{B_{\text{conn}}}{B_{\text{LINK-Æ}}} \quad \Rightarrow$$
$$d_{\text{transfer}}^{(\text{Æ})} \le D + \left\lceil \frac{D}{T_{\text{B conn}}} \right\rceil \cdot (T_{\text{T Æ}} - T_{\text{B conn}}) \tag{3.21}$$

where Z is the size occupied by the token on a network link, D is the transfer delay that the token would experience if it could use the whole link bandwidth without sharing; $T_{\text{B conn}}$ is the total time that is dedicated to the transfers of the given connection per one TDMA period on every link along the connection path. All three symbols with 'Æ' in the subscript are network constants, namely $Z_{\text{min-Æ}}$ is the data granularity of the network (the size of primitive packets), $T_{\text{T Æ}}$ is the TDMA period, $B_{\text{LINK-Æ}}$ is the bandwidth of a network link, which is the maximum bandwidth one processing tile can use in one direction.

The essential idea for the upper bound on the transfer delay in Formula (3.21) is to add the delay that the token would experience if it could use the whole network link – D – with the worst-case time the token has to wait for the TDMA slots. Note that the formula for Z does not include any packetization overhead (headers), because the guaranteed-throughput connections described in [81] do not need headers, because they are based on circuit-switching, where the routes followed by the packets are pre-programmed in the network routers instead of being derived from the packet headers.

For our examples, we assume that the router runs at 400MHz (20% slower than the 500-MHz clock reported in [81]), using the link bit-width 16 bits (two times less than reported in [81]) and the primitive packet size 6 bytes (also two times less). We also assume that one TDMA period takes 768 router clock cycles (the same as reported in [81]). The reason we scale down the router performance is to achieve comparable computation and communication delays in our examples, where we use the ARM7 processor architecture. All the important network constants that we use in practice are summarized in Table 3.1.

The transfer delay is not the only delay experienced by a data token in the network. Between the departure of a token from the producer buffer and its arrival at the consumer buffer there is another activity taking place: the 'tail' of the data token should propagate through the network. The propagation delay is called the *latency* of the network connection (d_{latency}). In the ÆTHEREAL network, every connection follows a fixed path through the network and the latency is a constant proportional to the number of routers along the connection path, l_{routers}:

$$d_{\text{latency}}^{(\text{Æ})} = l_{\text{routers}} \cdot d_{\text{ROUTER-Æ}} \tag{3.22}$$

where $d_{\text{ROUTER-Æ}}$ is the latency of one router, another ÆTHEREAL network constant, equal to the duration of three clock cycles of the network clock. (For readers familiar with the ÆTHEREAL network, it is appropriate to mention that we account for the network interfaces as routers because they are equivalent to the routers from the latency point of view.)

Summed with d_{transfer}, the latency gives the total delay of a network connection. The reason to separate the transfer delay from the latency is the fact that the subsequent transfers on the given connection can only execute sequentially one after another, whereas propagation of subsequent data tokens through the network is pipelined and can take place concurrently. Later in this chapter, we take this difference into account in the HSDF subgraphs that model the network connections.

Table 3.1 Network constants used in this chapter
(2x smaller link and 20% slower clock w.r.t. [81])

Notation	Meaning	Value
$T_{\text{T Æ}}$	TDMA period of the network routers	1.92 μs
$B_{\text{LINK-Æ}}$	Link bandwidth in one direction	$800 \cdot 10^6$ byte/s
$d_{\text{ROUTER-Æ}}$	Latency per router	7.5 ns
$Z_{\text{min-Æ}}$	Network data size granularity	6 bytes
$B_{\text{min-Æ}}$	Network bandwidth granularity	$3.125 \cdot 10^6$ byte/s

The other part of the network channel behavior that we have to consider here is the *flow control mechanism*, which helps to avoid consumer-buffer overflow. To implement the flow control, the communication assist at the producer side keeps a (pessimistic) counter of the number of free places (*credits*) in the buffer at the consumer side. The data token transfers are blocked when that counter reaches zero. The counter is decremented whenever a new token departs into the network. Every time the processor at the receiving side frees one or more places in the consumer buffer, it triggers a credit packet to be sent back to the sender through the flow-control connection of the channel, and the credit counter is incremented accordingly.

Because the network connections provide a guaranteed upper bound on the communication delay, we can bound the time interval it takes between the moment when the consumer frees up some space in the consumer buffer and the time when the credit counter is updated. We call the upper bound on that time interval the *credit delay*, and denote it d_{credit}. It is important to note that the credit information propagates through the credit connections that can be of the same type as the data connections. However, to transfer the credits, it is more efficient to follow different rules from those used for the data tokens. If at a certain moment of time, in a certain connection, there is a credit waiting to depart into the network and another credit is produced, the new credit does not have to wait in a queue until the earlier token departs, it can just be added to the earlier credit. Every network packet of the flow control connection carries a value equal to the number of tokens that were added to the credit during the time that the packet was waiting for departure. Thus, subsequent credit transfers occur concurrently. An important conclusion from this is that there is no need to split the credit delay into transfer delay and the latency.

In the ÆTHEREAL network, it would take at most $T_{\text{T Æ}}$ time for a credit to depart and exactly $d_{\text{latency}}^{(\text{Æ})}$ for a credit to propagate through the network back from the consumer of the channel to the producer. Therefore,

$$d_{\text{credit}}^{(\text{Æ})} \le T_{\text{T Æ}} + l_{\text{routers}} \cdot d_{\text{ROUTER-Æ}} \tag{3.23}$$

At the end of this subsection, we would like to introduce one more network constant, $B_{\text{min-Æ}}$ - the bandwidth granularity or the indivisible unit of bandwidth in the ÆTHEREAL network. The finite bandwidth granularity can be explained as follows. The TDMA period is divided into 256

slots [81] and the time dedicated to a given connection is allocated in terms of integer number of slots. Therefore, $B_{\text{min-Æ}} = B_{\text{LINK-Æ}} / 256$.

3.5 Intra-application Mapping: Computation Phase

3.5.1 The Goals of Studying the Mapping Flow in this Thesis

In the previous sections, first steps were made towards the final goal of this chapter, the construction of the IPC graphs, used for our run-time performance analysis methodology, in which the main contributions of this thesis reside. In Sections 3.1-3.3, we have characterized the computation actor delays using the actor parameter functions, and this characterization is an important part of our methodology, because we use this characterization to model the actor delays at design time and at run time. Finally, in Section 3.4, we described the platform architecture that is supported by our IPC graphs.

To explain how the IPC graphs are constructed, in this section and in the next section we study the intra-application mapping flow. This flow strives to minimize the resource usage under a minimum throughput constraint. It is important to stress that, in this thesis, it is not our purpose to propose or compare any mapping algorithms. The main purpose of this study is to show how the initial HSDF model of the application – i.e. the computation graph – is being gradually transformed by the mapping flow, through intermediate models, into the final model – i.e. the IPC graph. Although we consider the mapping flow step-by-step, we mainly focus on *what* every step has to do and how that goal is reflected in the intermediate HSDF model. The question *how* an optimal or efficient solution can be achieved at every step is beyond the scope of this thesis[23]. Although it is not our goal to give answers to this question, we also show that our HSDF models are not only product of the mapping flow, but also a performance analysis tool that the flow can use to direct the mapping decisions towards a better quality of results. The novel component here is the usefulness of the HSDF modeling techniques for the *buffer capacity minimization* for NoC-based platforms.

The paradigm of updating and analyzing the intermediate graph-theoretic model of a real design object to support the design decisions is a well-recognized paradigm in the field of electronic design automation (EDA). The tools for logic synthesis and physical synthesis exploit so-called timing graphs, which provide an intermediate timing model of the digital logic design, being updated in conjunction to the modifications made in the design by the design flow and being used to guide the decisions made in the flow. The idea to use some sort of a 'timing graph' for the multiprocessor mapping of streaming applications is less widely known, although it is advocated and thoroughly researched, originally in [83] and [5]. In that original work, the focus has been limited to bus-based multiprocessors. Later work extends the idea of using graph-theoretic mapping analysis models to the network-based multiprocessors. Examples are our publication [75], the work of Sander Stuijk, e.g. [90], and Orlando Moreira, e.g. [65]. As already mentioned, our contribution mainly lies in the buffer capacity minimization for NoC platforms and the HSDF-based modeling techniques that enable graph-theoretic formalization of that

[23] Note that the applicability of the performance method proposed in this thesis does not depend on the answer to the question how to achieve optimal results at every mapping step, because the analysis can be applied equally well for efficient and inefficient solutions.

problem. This formalization results in the ability to find bottlenecks in the given mapping solution. This ability can be used to develop iterative optimization algorithms.

The study of the intra-application mapping flow spans two sections of this chapter, namely, this section (Section 3.5) and the next one (Section 3.6). By the end of the next section, it should become clear how our IPC graphs are constructed, and, after that, in the next chapters of this thesis, we can proceed with the performance analysis of the given IPC graph. A more detailed analysis of related work is postponed until the end of this chapter (Section 3.8).

3.5.2 The Structure of the Mapping Flow

We illustrate the trajectory taken by the application's HSDF model in the course of the intra-application mapping flow by means of a hypothetical example flow, which we call the *preferred mapping flow*. We base that flow on a mapping flow proposed by Rudy Lauwereins *et al* in [49], because it was devised for a similar application domain and for a similar type of multiprocessor platforms. We refer to the latter flow as the *reference flow*.

Figure 3.7 shows an overview of both the reference flow and the preferred flow, the latter being shown in the context of our overall implementation trajectory that we introduced in Sections 2.1.2 and 2.3.4.

As we see from the figure, we split the preferred flow into two sub-phases – the computation phase (studied later in this section) and the communication phase (studied in Section 3.6). The *computation phase* focuses on the computation part of our implementation-enhanced HSDF model. It means that it primarily makes decisions about the computation actors and processes, striving to minimize the *processing* resource usage under the throughput constraint. Hereby, the minimization of the communication resource usage is seen as a secondary goal. The *communication phase* focuses purely on minimization of the *communication* resource usage in the context given by the computation phase.

Recall that – as also shown in Figure 3.7 – at a level higher than the splitting into sub-phases, our implementation trajectory divides the mapping process into two stages, namely, the intra-application mapping stage and multi-application mapping stage. This division constitutes a considerable difference of the preferred mapping flow from the reference flow. As explained in Chapter 1, the division into stages is also done in some related work and it is necessary to support dynamic starting and stopping of different applications in different combinations at run time. In the context of the intra-application mapping stage, the division into stages means postponing some optimization decisions that the reference mapping flow would consider at once. As explained below, some optimization decisions are postponed until the multi-application mapping stage, which is performed at run time. Although, in general, dividing the optimization solvers into several stages may lead to suboptimal results due to phase coupling between the stages, in our case, it is a necessary price for supporting dynamic run-time combinations of different applications.

As shown in Figure 3.7, the reference flow starts with a preparatory step, the 'resource estimation', which corresponds to Application Preparation in our implementation trajectory, because its main goal is the same, namely, the calculation of typical actor delays. Therefore, the figure reflects the correspondence between those two steps in the different flows.

In the preferred mapping flow, after the Application Preparation, follows the first step of the intra-application mapping – the *processing assignment*. The goal of the processing assignment is

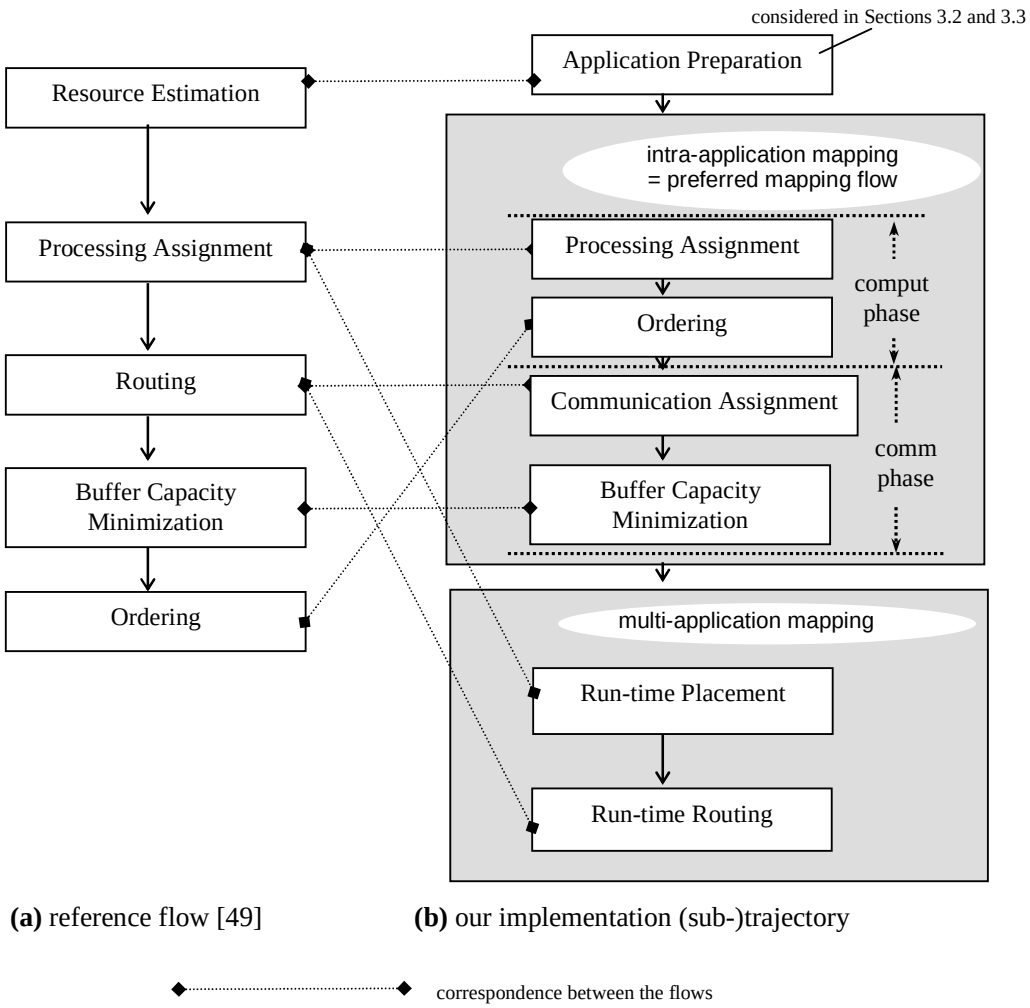


Figure 3.7 Overview of the mapping flows

to group the actors that go to the same processing tile of the NoC-based architecture, and then to subdivide every such group into sub-groups of actors that go to the same process. We study the processing assignment in Section 3.5.5.

The reference flow also has the processing assignment step; and it also groups the actors that go to the same process together. However, unlike the preferred mapping flow, the reference flow also *places* the groups, i.e. assigns them to the physical processors. In our trajectory, the placement is postponed until the multi-application mapping stage, and we refer to it as the *run-time placement* – see Figure 3.7. The term 'placement' in this context means selecting a free processing tile in the sub-array of the processing tiles that have an equivalent type. Note that if the multiprocessor platform is fully heterogeneous – i.e., if any two processing tiles have different types – then the placement is not needed.

In our preferred flow, the second step is the *ordering*, which orders the computation actors in every process. The reason why we put this step immediately after the processing assignment is to ensure that the ordering comes before the communication phase of the flow. This is a prerequisite for us to be able to demonstrate the usefulness of our HSDF modeling techniques proposed in Section 3.6 for the communication phase and in particular for the buffer capacity minimization, as promised above.

In general, whether the ordering should go before the communication phase or the other way around is an important decision when one develops a mapping flow, see e.g. discussion on this subject in [13]. In the reference flow, the ordering step is done after the buffer capacity minimization – see Figure 3.7 – and in our preferred flow it is the other way around. Both approaches are worth considering when one develops a mapping flow, but it is beyond the scope of this thesis to analyze and compare different mapping approaches. No matter which approach is chosen, one can obtain in the end the IPC graph that models the results of the mapping and that can be used for the run-time performance analysis, and we use the preferred flow as an example of how that can be done.

As we see in Figure 3.7, both flows perform some steps for optimizing the communication between the processors. Under the 'routing' in the reference flow, one understands the assignment of the computation graph edges to the network channels, calculation of the required network bandwidth, finding the physical routes through the communication network to realize the communication channels and allocating communication resources on every network router along the route to actualize the calculated physical routes (in the case of TDMA scheduling, this means allocating the TDMA slots for the packets that follow the routes). In our implementation trajectory, the calculation and actualization of the physical routes are postponed until the *run-time routing* step in the multi-application mapping stage. As for the assignment of the graph edges to the network channels and the required network bandwidth calculation, this is done in the 'communication assignment' step, which is part of the communication phase of the preferred mapping flow, studied in Section 3.6.

This completes our introduction into the structure of the mapping flow. In the remainder, we first present some preliminaries for the mapping flow and then visit the mapping steps one-by-one in more detail.

3.5.3 Preliminaries: Virtual Tiles, Budget Descriptor, Local Channels and Network Channels

Before describing the intra-application mapping flow, we have to introduce the data structure used by the flow to assign budgets to processes and channels – namely, the budget descriptor $\mathbf{B}$. That abstraction is based on a concept that is important for our two-stage mapping approach – the *virtual tile*. Let us introduce the virtual tiles in this subsection.

At the intra-application mapping stage, the processes are not yet assigned to *physical* processing tiles, but they are rather grouped together into groups, called virtual tiles. Processes belonging to the same virtual tile have to be mapped at run time to the same physical processing tile. For example, in Figure 3.8, four processes are shown communicating via three channels. Processes p_1 and p_2 have to be mapped to the same physical processor, because they are assigned to the same virtual tile. The same holds also for processes p_3 and p_4.

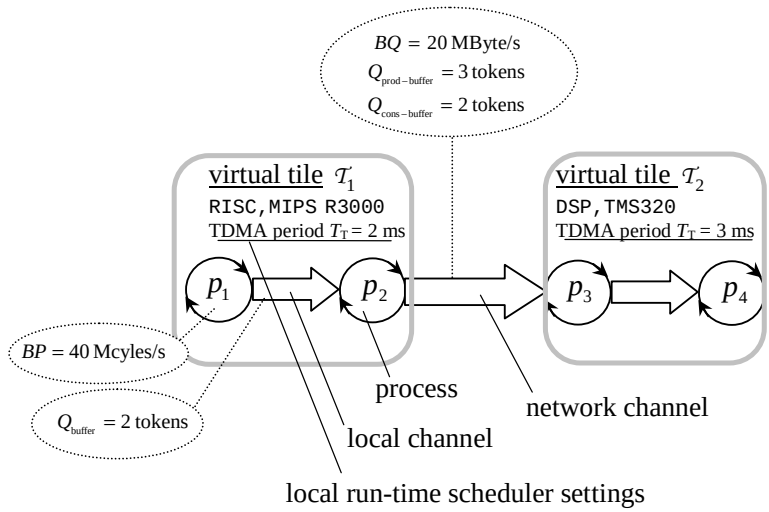


Figure 3.8 Virtual tiles, processes, channels and resource budgets

The reason to introduce virtual tiles is as follows. In the highly dynamic run-time environments where the set of active applications changes at run time, it is not efficient to bind the processes of applications to the physical resources at design time; it is much more efficient to assign a physical processor at run time, based on the processor availability at the moment when the application starts. However, to assign budgets to the processes and channels, some information about the physical processing tile has to be known at design time. This information is encapsulated in the virtual tile. At run time, the multi-application mapping module decides which physical tiles implement the virtual tiles.

A virtual tile tells, first of all, the *processor type*, which should be one of the types available in the target platform. The type of a virtual tile may be for example an 'application-specific processor (ASIP) optimized for performing the discrete Fourier transformations', or a 'RISC processor of a certain architecture such as the MIPS R3000™ or ARM7TDMI™ series' (like tile τ_1 in Figure 3.8), or 'a DSP processor of a certain architecture, e.g. a representative of the TMS320™ series' (like tile τ_2 in Figure 3.8), etc. If the target platform contains only one processor of the type specified in the virtual tile, then assigning a process to that virtual tile would mean an indirect assignment to the physical processor. However, if the target platform contains multiple processors of the same type, then assigning a process to a virtual tile of that type would still leave the choice for a specific physical processor to the run-time multi-application mapping module. In the extreme case, if all physical processors have the same type (in case of a homogeneous multiprocessor) then all virtual tiles must necessarily have that type as well and the type information does not play any role in process-to-processor assignment.

Secondly, the virtual tile gives the numerical values of the *local run-time scheduler settings* that are expected from the physical processing tile. If TDMA scheduling is used, then the settings include only the TDMA period T_{T}, measured in seconds. For example, the run-time scheduler settings for the tiles in Figure 3.8 are $T_{\mathrm{T}} = 2\,\mathrm{ms}$ for tile τ_1 and $T_{\mathrm{T}} = 3\,\mathrm{ms}$ for tile τ_2.

During the intra-application mapping flow, the virtual tiles are treated as if they themselves were the physical processing tiles. The processes get processing cycle budgets in terms of the processor clock cycles of the virtual tiles. Recall from Figure 3.1 that the process budget directly influences the delays of the actors contained in the process. From the discussion in Section 3.1.3,

it follows that in the case of TDMA scheduling, the budget and the TDMA period T_{T} imply the size of the timing slot T_{B} and the latter can be used to calculate the actor delay using Equality (3.5). Thus it is the information about the processing clock cycle budgets on the virtual tiles that we referred to in Figure 3.1 as 'budget information' and that is used in the computation delay relation for the performance analysis. From now on, we use notation '*BP(p)*' for the budget of process *p*. In the example of Figure 3.8, $BP(p_1) = 40$ Mcycles/s.

The process assignment to virtual tiles is important not only for the computation budgets, but also for the communication budgets. The processes assigned to the same tile share the same local memory system, so they do not need the on-chip communication network to communicate with one another. A communication channel that joins two processes belonging to the same tile is called a *local communication channel*. Such a channel does not involve the communication assist and the producer/consumer buffers – the architectural elements we have seen in Figure 3.6. It uses one FIFO buffer directly accessible by both the producer and the consumer. The only budgeting required for such channels is the buffer capacity, denoted Q_{buffer}, measured in data tokens. For example, in Figure 3.8 the channel between processes p_1 and p_2 is a local channel because both processes are assigned to the same tile, $\mathcal{T}_1$. That channel has a buffer capacity of two tokens.

The communication channels whose producer and consumer are assigned to different tiles use the network-based communication infrastructure discussed in the previous section. We refer to those channels as *network communication channels*. Their budgets include:

1) the producer buffer capacity, denoted $Q_{\text{prod-buffer}}$ and measured in data tokens,

2) the consumer buffer capacity, denoted $Q_{\text{cons-buffer}}$ and also measured in data tokens,

3) and the network bandwidth for data communication, *BQ*, in bytes per second.

To summarize what we have considered so far in this subsection, we can say that the resource assignment and resource budgeting information is characterized by the set of the virtual tiles, the processing cycle budgets and by the communication budgets, the latter including the communication bandwidth and the buffer capacities. All this information is encapsulated in budget descriptor $\mathbf{B}$, as defined below.

Definition. A budget descriptor $\mathbf{B}$ of a given implementation-enhanced HSDF model **GPQ** is a tuple $<\boldsymbol{\mathcal{T}}, \mathcal{T}_{\text{type}}, \mathcal{T}_{\text{sched}}, P_{\mathcal{T}}, BP, Q_{\text{prod-buffer}}, Q_{\text{cons-buffer}}, Q_{\text{buffer}}, BQ>$, where $\boldsymbol{\mathcal{T}}$ is a set of virtual tiles, $\mathcal{T}_{\text{type}}$ is a function that for virtual tile $\mathcal{T}$ specifies its processor type; $\mathcal{T}_{\text{sched}}$ is a function that for each $\mathcal{T}$ specifies the run-time scheduling settings for the processor contained in $\mathcal{T}$; $P_{\mathcal{T}}$ is a function that for each process *p* specifies the virtual tile that runs p; BP is a function that for each process *p* specifies its budget, in processor clock cycles per second; $Q_{\text{prod-buffer}}$, $Q_{\text{cons-buffer}}$, Q_{buffer} and BQ are functions of the communication channel $q \in \mathbf{Q_{comm}}$ that specify the corresponding characteristics of the channel. For a local channel, the producer/consumer buffer capacities are defined as zero, because no such buffers are used in those channels, and their bandwidth is defined as infinity, because the data gets from the producer to the consumer instantly:

$$q \text{ is a local channel} \Rightarrow \quad Q_{\text{prod-buffer}}(q) = Q_{\text{cons-buffer}}(q) = 0\,,\ 0 < Q_{\text{buffer}}(q) \le +\infty\,, \quad BQ(q) = +\infty\,.$$

Here, Q_{buffer} can also be set to infinity to model the situation when the buffer capacities are large enough to ensure that the computation actors never block due to full output channels. As for a network channel, it does not have a buffer that directly connects the producers and consumers, so the local buffer capacity is set to 0. The sizes of the intermediate buffers and bandwidth are either assigned a finite positive value or infinity. In the latter case, the communication cost through the network is being ignored. Therefore, we can write:

q is a network channel $\Rightarrow$

$$0 < Q_{\text{prod-buffer}}(q) \le +\infty,\ \ 0 < Q_{\text{cons-buffer}}(q) \le +\infty,\ \ 0 < BQ(q) \le +\infty,\ \text{and}\ Q_{\text{buffer}}(q) = 0\ \blacklozenge$$

Recall from Section 2.1.2 that the budget descriptor is an element of a tuple defining the implementation process network, $\mathbf{PQ} = <\mathbf{P},\ \mathbf{Q}, \mathbf{V}_{\mathbf{comp-body}}, \mathbf{B}>$. As it follows from the definition given above, the role of the budget descriptor is to group the processes in $\mathbf{P}$ into the virtual tiles and to set the budgets for the processes in $\mathbf{P}$ and for the communication channels in $\mathbf{Q}_{\mathbf{comm}} \subseteq \mathbf{Q}$.

3.5.4 The Computation Graph and the Initial Performance Estimates

The mapping flow starts from the computation graph. Recall that in the computation graph there are only computation actors, and there is a one-to-one correspondence between each process and an actor, each sequence edge and a state channel, and each data edge and a communication channel.

Figure 3.9 shows an example of the computation graph of the JPEG image decoding application, the same application from which we took actor 'VLD' as an example for determining the actor parameters in Section 3.2.2. We use that application as the main example for the rest of this section. The purpose of this example is to demonstrate already in this section that our methodology can be used for 'real-life' problems.

One iteration of the loop of interest of the application decodes one minimum coded unit (MCU, 16x16 pixels). In any computation graph, the communication channels transfer one token per iteration. In our example, all the communication channels assume the same data-token type, an 8×8 pixel *block*. One MCU consists of six blocks corresponding to different positions within the MCU in different color planes.

For clarity, the actors in Figure 3.9 are organized in columns and rows. The columns correspond to processing stages and the rows correspond to the six blocks of an MCU. The blocks undergo three processing stages: variable length decoding (VLD), inverse discrete cosine transform (IDCT), and scaling (SCALE), before they are fed into a color conversion stage (COLOR). The VLD actor has a self-loop state channel that is introduced there because each new iteration of the VLD actor needs the previous iteration to complete in order to know the position in the input bitstream where it can start with the decoding.

In the computation graph, it is assumed that every actor/process gets 100% processor budget, therefore each actor/process is assigned a separate virtual tile. The processor architecture type of the tile is chosen such that the processor can typically execute the functionality of the given actor with the least delay. Because 100% budgets are given to the actors, the typical actor delay is computed as the ratio between the typical actor processing time on the given processor architecture and the processor clock frequency. In our application example, we assume that the ARM7TDMI running at 133MHz is the only processor architecture type available in the target platform (a homogeneous multiprocessor). We assume that the application's real time constraints are not hard but soft so that the 'typical' delays can be interpreted as 'average' delays

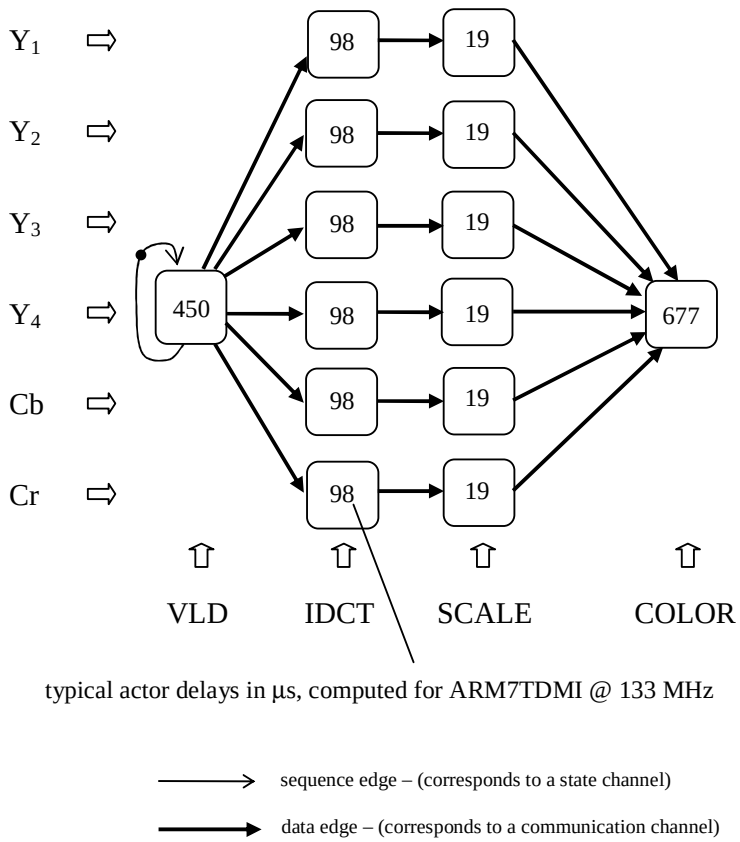


Figure 3.9 Computation graph of the JPEG decoding application

(see Section 2.3). Therefore, for the VLD actor, we had to obtain average values of each of the three parameters of that actor and then use them in Equality (3.7). As a result, we obtained the typical processing time of the VLD actor of 60 kilocycles (visually, in Figure 3.5, this value is also close to the average value). For the clock frequency of 133 MHz, this yields a delay of approximately 450 μs, which we use as the delay value for that actor, as shown in Figure 3.9. As for the other actors in this computation graph, all of them have zero parameters, so their typical processing times are equal to the corresponding actor coefficients $C_{k,0}$, calculated using the direct measurement approach described in Section 3.3. Note that for the IDCT actor, as for all the other actors in Figure 3.9, we report the results for the ARM architecture, although some embedded processor architectures can implement the IDCT operation much faster than the ARM processor. If we had assumed in this example that such an architecture type would have been available in the target platform, we would have used the IDCT delay of that architecture type in the computation graph in Figure 3.9.

As for the communication costs, they are ignored at this point of the mapping flow. Until the final processing assignment of actors to the processes and processes to the virtual tiles is done, the communication resource requirements cannot be estimated accurately anyway. The preferred intra-application mapping approach first focuses on ensuring that the computation part of the implementation meets the performance requirements and then it can accurately estimate the computation resource requirements that should be met by the communication infrastructure. This

separation of computation and communication concerns is seen as important for the future NoC-centered multiprocessor on-chip architectures [31].

This enables the initial abstraction of the communication channels by HSDF edges – the data edges of the computation graph. In the computation graph, all the communication channels are assumed to have unlimited bandwidth and unlimited buffer capacity. Because the edges in HSDF graphs have no intrinsic delay and can carry an unlimited number of tokens, it is valid to represent each communication channel just by one data edge until the mapping flow starts the communication assignment, whereby the communication bandwidth and the buffer capacities are determined.

The computation graph allows to obtain the best performance estimate, $\theta_{\max}$, or the typical throughput of any implementation of the given application on the given target platform. Indeed, in the computation graph, the computation actors are assigned the best possible processor resources that the target platform can offer, with the assumption that the platform can provide a separate physical tile for every computation actor, and communication costs are ignored. It only makes sense to start the mapping flow if the best throughput the platform can offer, $\theta_{\max}$, is not less than the required throughput: $\theta_{\text{required}} \le \theta_{\max}$. The difference $(\theta_{\max} - \theta_{\text{required}})$ can be seen as a slack that can be exploited by the intra-application flow to relax the high resource demands that are assumed initially for the actors of the computation graph. If the slack is negative, then either the application QoS/performance requirements have to be downgraded or the target platform has to be upgraded.

We derive $\theta_{\max}$ from a throughput estimate that is useful for the whole computation phase of the mapping flow. The estimate is based on the critical cycle of the HSDF graph (i.e. the cycle with the maximum cycle mean) and the critical process, i.e. the process with the maximum total actor execution delay).

Definition. Throughput estimate (for the computation phase of the mapping flow):

$$\theta(\mathbf{G}) \equiv \frac{z(\mathbf{G})}{\max\left(MCM(\mathbf{G}), \max\limits_{p \in \mathbf{P}} \sum\limits_{v \in \mathbf{VP}(p)} d(v) \right)} \tag{3.24}$$

where $MCM(\mathbf{G})$ is the maximum cycle mean of graph $\mathbf{G}$, $d(v)$ is the static actor delay assumed in the typical delay timing mode, depending on the processor type; $\mathbf{VP}(p)$ is the set of actors assigned to process p and $z(\mathbf{G})$ is the (constant) amount of information the loop of interest communicates to the implicit external memory buffers. (Because we do not explicitly model external communication, $z(\mathbf{G})$ is not present in our model, but it can be provided as an extra annotation.) ♦

In the computation graph, every process contains just one actor, so the maximum total process delay can be replaced by the maximum computation actor delay, and we get:

$$\mathbf{G} \text{ is a computation graph} \Rightarrow \quad \theta_{\max} = \theta(\mathbf{G}) = \frac{z(\mathbf{G})}{\max\left(MCM(\mathbf{G}), \max\limits_{v \in \mathbf{V}} d(v) \right)} \tag{3.25}$$

Equality (3.25) can be proven to be an optimistic – i.e. maximal – throughput estimate as follows. In the denominator, we have a lower bound on the typical iteration interval. The MCM of graph $\mathbf{G}$ – where actors have typical (i.e. average) delay – is a lower bound on the typical iteration interval, according to Formula (2.10). Also any actor delay is a lower bound on the

typical iteration interval, because, in the final implementation, the subsequent executions of every computation actor occur sequentially. Thus, the denominator of the expression in Equality (3.24) is the maximum of the lower bounds on the iteration interval, thus being itself a lower bound. In the final implementation, the graph cannot execute faster, because the transformations applied to the graph by the mapping flow respect the actor dependencies defined in the computation graph and because, the actors cannot get processor budgets that are larger than the budgets assumed in the computation graph.

Note that often, when reasoning about the throughput constraint, instead of directly using $\theta_{required}$ we use $\Lambda_{allowed}$, i.e., the maximum allowed average iteration interval implied from the throughput constraint: $\Lambda_{allowed} = z(\mathbf{G})/\ \theta_{required}$.

In the example in Figure 3.9, we see that θ_{max} is one MCU per 677 μs, or 1477 MCUs per second. Now, let us assume that $\theta_{required}$ is 1000 MCUs per second, or one MCU per 1000 μs (i.e. $\Lambda_{allowed}$ = 1000 μs). A single ARM processor cannot meet this requirement, because the sum of the typical delays is 1829 μs; therefore a multiprocessor is required. The optimization steps of the mapping flow considered in the next subsections are needed for efficient use of a multiprocessor platform. The final goal of the mapping is to issue an implementation process network of efficient structure and with efficient budget descriptor. This implies that the purpose is to generate a structure of the same type as the one shown in Figure 3.8 taking the computation graph as the starting point.

3.5.5 Processing Assignment and Ordering

In this section we consider both steps of the computation phase of the intra-application mapping flow – the processing assignment and the ordering. In terms of our implementation-enhanced HSDF model, those two steps finalize all the elements related to the processes and processing tiles, leaving everything related to the communication channels to be finalized later in the mapping flow. At the start of the mapping flow, our model is configured for the maximum usage of processors, but the steps described in this subsection strive to ensure that the processor resource usage is reduced as much as possible, while the throughput constraint – evaluated using Equality (3.24) – is still satisfied.

For convenience, we present all the tasks to be done in the processing assignment and ordering steps as a sequence of five sub-steps executed in a certain order, although in a real flow these sub-steps can be done in a different order or in parallel. For us, it is only important to mention all the optimization problems to be solved by the flow and to show their relationship to our HSDF-based model. We do not intend to describe a full mapping solution or to point to any in the literature; we only justify our flow discussions by the fact that similar mapping problems are discussed for the reference flow in [49] and, in the related work on mapping of the applications to the multiprocessor systems on chip, such as [90] and [66]. The only place where we do claim new insights in the mapping flow is the buffer capacity minimization; however, this exception refers to the communication phase of the mapping flow, being described, in the next section. In this section, we do not claim any novel results, as our reasoning is to a large extent comparable to the reasoning in [83]. An example of a mapping approach that proposes solutions to some relevant optimization problems in the computation phase of the mapping flow is presented by Sander Stuijk *et al* in [88], [90].

The sub-steps of the computation phase of our preferred mapping flow are as follows:

1) (*partitioning into tiles*) Determine $\mathcal{T}$, i.e., the final set of virtual tiles and partition the set of actors between the tiles. Because at this point the actors still correspond one-to-one to the processes, this step, in fact, modifies the mapping of the processes to the tiles, denoted in our model as function $P_{\mathcal{T}}(p)$ (in the budget descriptor). Assign a processor architecture type and run-time scheduling settings to every virtual tile in set $\mathcal{T}$, i.e. define final $\mathcal{T}_{\text{type}}(\mathcal{T})$ and $\mathcal{T}_{\text{sched}}(\mathcal{T})$ (in the budget descriptor).

2) (*partitioning into processes*) In every tile, lump the small one-actor processes into larger processes, thus effectively partitioning the sets of actors assigned to the same virtual tile into subsets – **VP**(*p*) (in the definition of the process) – containing the actors assigned to the same process *p* in the final implementation.

3) (*the budget assignment*) Assign a final budget to every process, i.e., define function *BP*(*p*) (which is included in the budget descriptor).

4) (*computation actor ordering*) For the processes that contain more than one actor, decide on the ordering of the actors within the process, i.e., define function **vp**(*p*) (which is included in our definition of a process).

The first sub-step – the partitioning of the actors between the virtual tiles – has the largest impact on the end result of the mapping. Therefore, this sub-step gets the most of our attention. It decides how many processing tiles will be used by the application, which tiles execute which actors, which processor architecture type every tile has (for heterogeneous target platforms) and how the local scheduling is organized in every tile. This step also indirectly decides which tiles communicate with each other via the network, because if there is a data edge between two actors assigned to different tiles, then a network channel is needed to implement that data edge.

The main rationale of this sub-step is to allocate as few virtual processing tiles as possible and to distribute the actors between the allocated processing tiles such that no tile gets a load that is larger than $\Lambda_{\text{allowed}} \cdot (1 - \delta_{\text{margin}}(\mathcal{T}))$. Here $\delta_{\text{margin}}(\mathcal{T})$ is the – set by the designer – tile-specific fraction of the processor clock cycles that is reserved to compensate the scheduling overhead, for the processor budget of the other applications and for the idle processor cycles due to possible inability of the later steps of the mapping flow to find such a scheduling that every processor resource is kept busy for 100% of the allocated processor budget. The *load* of a virtual processing tile is defined as the sum of the execution delays of all actors assigned to the tile, whereby for the load calculation we use the delays that assume no run-time scheduling, i.e. the delay is a ratio between the actor processing time and the processor clock frequency. Note that in a heterogeneous multiprocessor the contribution to the load of a given actor depends on the processor type assigned to the given virtual tile, also at this sub-step.

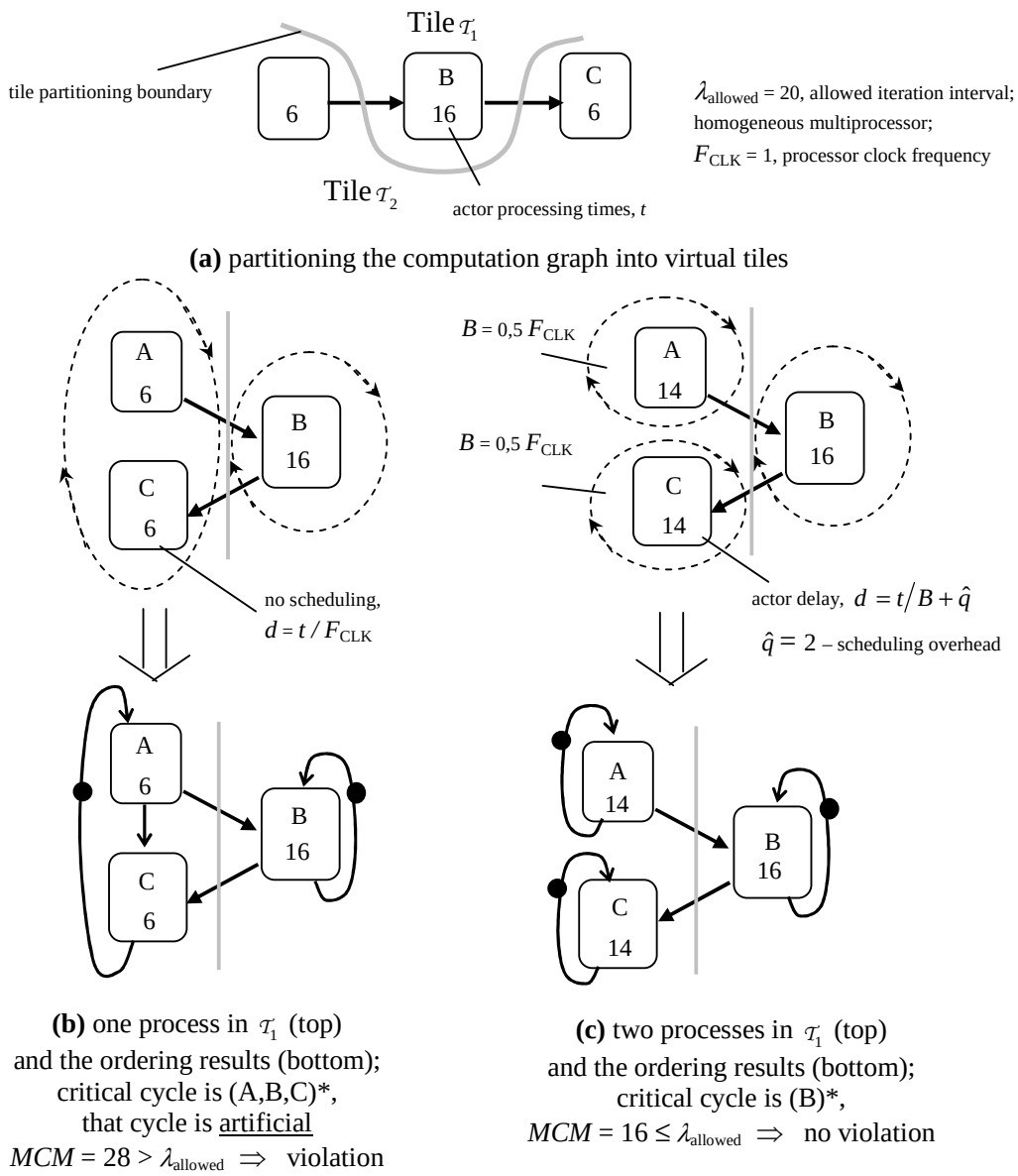
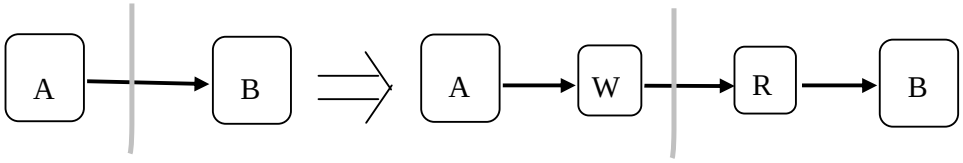


Figure 3.10 An example of partitioning into tiles and processes

In our examples, we try to allocate no more tiles than necessary to bring the maximum load below the maximum allowed iteration interval and for simplicity we assume that $\delta_{\text{margin}}(\mathcal{T})$ is zero. We also try to spread the load as evenly as possible to maximize the load 'slack' of every tile, i.e., maximize the difference between Λ_{allowed} and the tile load. Figure 3.10(a) illustrates an example partitioning of a simple three-actor computation graph into two tiles. The maximum allowed iteration interval in this example is 20 time units. In Figure 3.10(a), we assume a homogeneous multiprocessor and a clock frequency of 1 frequency unit, so that the load has the same numeric value as the processing times. Obviously, in this example, at least two tiles are needed to bring the maximum load below 20. We partition the example graph into two tiles with the load of 16 and 12.

Keeping the load below the maximum allowed iteration interval is necessary for the throughput constraint satisfaction. The reason for that is the observation that, if we assume that



if in the target architecture PrM/CsM memory modules are physically separate from the general-purpose data memory (M) – see Figure 3.6 – then the processing assignment should insert 'W' and 'R' actors to implement the data copying;

Figure 3.11 Insertion of 'write' (W) and read (R) actors at the tile boundaries

the tiles correspond one-to-one to the processes, then the load is equal to the second term in the 'max' expression in the denominator of the throughput calculation formula – Equality (3.24). One can show that the load is still a good estimate of the second term even for multiple processes per tile.

If the computation graph contains no multi-actor cyclic paths, then the second term in the denominator – and hence the load – is the only effective term in the denominator of Equality (3.24). In the case where there are multi-actor cyclic paths, then the first thing to observe is that some of those cycles may be so-called state consistency cycles, i.e., those that include only state consistency channels and no communication channels. The partitioning of actors into tiles should see such cycles as indivisible elements, as if they were actors whose load is equal to the total load of the actors in the cycle; the reason for this is explained in Section 2.1.3. If all the cycles are state consistency cycles then the reasoning in terms of load is enough to ensure the throughput constraint satisfaction. However, if there are cycles that include communication channels, then the reasoning in terms of load is not enough, because these cycles may influence the application throughput via the '*MCM*' term in the denominator in Equality (3.24). However we skip the discussion of this case to stay focused on the main topic of this section.

In our discussion of the partitioning into tiles, we have to mention two special considerations that should be taken into account in this sub-step of the flow. Both of these considerations refer to the graph edges that cross the partition borders (see for example the edges in Figure 3.10(a)). The first consideration is the communication bandwidth limitation. Based on the token sizes of the edges that come in and out of a processing tile, one can estimate minimum bandwidth required from the network link that joins the given tile with the rest of the on-chip network in both directions. The partitioning should ensure that for each tile holds that this bandwidth does not exceed the maximum bandwidth a network link can offer physically.

The second consideration to be taken into account is the insertion of extra computation actors that is necessary for certain kinds of hardware architecture of the processing tile, in particular for the case where the local general-purpose memory and the communication memories, PrM and CsM, are implemented in different physical memory modules (see Figure 3.6). In this case, the data tokens that are sent through the network channels have to be copied from the local memory to PrM at the producer side and from CsM to the local memory at the consumer side. The extra actors that should be introduced in the HSDF graph implement the data copying. Those actors are inserted at the edges that cross the tile partition boundaries, as illustrated in Figure 3.11. We call those actors *write actors* and *read actors*. When executing a write actor, the processor copies

a data token from the general-purpose memory to the PrM memory. When executing a read actor, the data goes from CsM memory to the general-purpose memory.

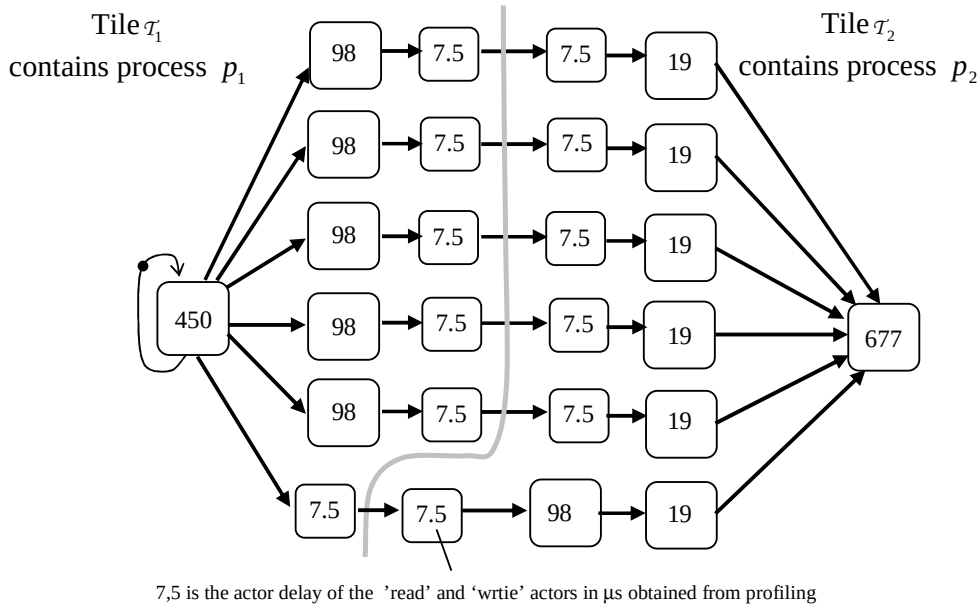
For implementing the JPEG decoding application introduced in the previous section, we assumed an architecture that requires data copying. In Figure 3.12(a), we see back all the computation actors from Figure 3.9 plus the data copying actors. As shown in Figure 3.12(a), the computation graph of the JPEG application is partitioned into two tiles, which is just enough to meet the throughput constraint (recall from above that one tile would not be enough). Because there are no multi-actor cycles in the computation graph and because we assume a homogenous multiprocessor architecture, reasoning in terms of load is enough for the throughput constraint satisfaction. The load is distributed roughly evenly between the tiles: tile τ_1 has load 985 μs and tile τ_2 has load 934 μs (calculated as the sum of actor delays in the partition). Note that we still stay below the allowed maximum iteration interval of 1000 μs, so both tiles have some performance slack.

Note that we did not mention anything on the methodology to assign the processor types $\mathcal{T}_{\text{type}}$ and the scheduling settings $\mathcal{T}_{\text{sched}}$ to the ties. This is only necessary for heterogeneous multiprocessors and for the case where the designer intentionally sets different clock frequencies or different scheduling settings – e.g. different TDMA periods – at different processors of the target platform. In fact, we do not have any suggestions on how our implementation-enhanced HSDF model can help in making these decisions.

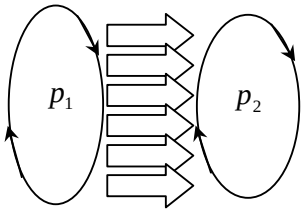
A simple solution for the second sub-step in our list – the partitioning of tiles into processes – would be to always have one process per tile. An advantage of that solution is that all actors use the maximum budget, equal to the processor clock frequency. When we introduce multiple processes per tile, the processor clock cycle budget gets split between the processes and each process enjoys only a portion of the budget. Consequently, due to such budget splitting, the actor delays get larger, and this can lead to throughput constraint violation. On the other hand, the processes are concurrent and therefore the actors of one process cannot delay the actors of another process when waiting for the input tokens. Due to this, in some cases, splitting into processes can help to satisfy the throughput constraint. This is demonstrated in the example below.

Consider again the example in Figure 3.10(a), where the computation graph is partitioned into two virtual tiles. To partition tile τ_1 into processes, there are two options, namely, either to use one process, as illustrated in Figure 3.10(b), or two processes, as shown in Figure 3.10(c).

To analyze the results of those two alternative solutions, we need to look ahead in the flow, i.e., to do the budget assignment and the ordering. In the one-process case in Figure 3.10(b), the process containing actors A and C can use the 100% of the clock cycles in tile τ_1. Therefore, no budget assignment is needed in tile τ_1, and the actor delays are equal to the processing times divided by the clock frequency. As for the ordering sub-step of the flow, the only feasible ordering in this case is ‘first actor A and then actor C’. Recall from Section 2.2.3 that the actor ordering in a process is modeled by extra edges, forming a cycle; see the bottom part of Figure 3.10(b). Due to extra graph paths introduced by the process cycle, we see in Figure 3.10(b) that the MCM of the graph is now 28 time units, which means a throughput constraint violation.



(a) the new HSDF graph and tile partitioning



(b) the implementation process network
(only the channels that join different processes are shown;
the internal channels of the processes are skipped)

Figure 3.12 Processing assignment results for the JPEG decoding application

In the two-process case, illustrated in Figure 3.10(c), each process gets only one half of the budget, i.e., 0,5 frequency units. Consequently, actors A and C have higher delay values than in the one-process case. Nevertheless, the throughput constraint is not violated, because the graph MCM in this case is 16.

This example illustrates the purpose of the partitioning of the virtual tiles into processes. The partitioning into processes helps to avoid the introduction of *artificial cycles* during the ordering sub-steps later on in the flow. We call a cyclic path artificial, if is not present in the HSDF graph before a certain sub-step of the mapping flow and gets introduced there by the mentioned sub-step (in this case – the ordering sub-step). In Figure 3.10(b), cycle (A,B,C)* is such a cycle.

On the other hand, our example also illustrates a disadvantage of multiple processes per tile – the actor delays get larger. Another disadvantage, not illustrated in the figure, is the context switch overhead. Note that introducing multiple processes per tile is not always necessary, because it is not always necessary to avoid the artificial cycles. An artificial cycle is only

harmful when its cycle mean (see Section 2.2.5 for the definition of cycle mean) is close enough to Λ_{allowed}, and sometimes smart decisions in the ordering sub-step can avoid the violation of the throughput constraint due to artificial cycles.

This finishes our discussion on the partitioning of tiles into processes. The flow sub-steps left to be discussed in this subsection are the budget assignment and the ordering. In fact, we have already mentioned the impact of those two sub-steps on our implementation-aware HSDF model, but we provide some more remarks to finish this topic.

In every multi-process tile, the budget assignment splits the processor clock frequency into budgets of different processes assigned to the given tile and the budget left free for the other applications. For the formulation of the optimization problem solved at this sub-step of the flow, two contradictory objectives can be included. On the one hand, it is desirable to minimize the resource usage, which means maximization of the budgets left for the other applications. On the other hand, it is desirable to maximize the performance slack of every process and every cycle in the graph, i.e. to maximize the difference between Λ_{allowed} on one side and the total execution delay of every process and the cycle mean of every graph cycle on the other side. The purpose of the latter objective is to increase the optimization freedom for the later steps of the mapping flow. Which of the two objectives to choose depends on how scarce the processor cycle budget resources are at the given platform for the given application domain (i.e. on how great the need to share as much processor resources between different applications as possible) and on how hard the optimization problems are for the later steps of the flow (and thus on how much slack they need). The constraints for this optimization problem should specify that the total budget at every virtual tile should not exceed the processor clock frequency and that the slack of every process and every graph cycle should be positive.

The ordering sub-step introduces process cycles $\mathbf{GP}(p_i)$ into the HSDF graph (see definition of the process cycles in Section 2.2.3). The optimization problem to be solved at this sub-step is to find an ordering of actors for every process such that when the process cycles are introduced to enforce the chosen orderings, no artificial cycle gets a cycle mean that exceeds Λ_{allowed} minus a certain margin to be used by the later steps of the mapping flow (just as it is the case for the previous sub-step, one may consider to include this margin into the optimization objectives). The requirement to keep the cycle means of all cycles sufficiently small also means that if actors v_a and v_b are assigned to the same process – lets call it process p_{ab} – and if there is an initial-token-free path from actor v_a to actor v_b in the computation graph, then v_a should be earlier than v_b in the ordering $\mathbf{vp}(p_{ab})$. The reason is that if actor v_b could be earlier than v_a then process cycle $\mathbf{GP}(p_{ab})$ would introduce an initial-token-free path from v_b to v_a and, because there is already a path from v_a to v_b, we would see an artificial cycle with zero initial tokens and thus with an infinite cycle mean. For example, in Figure 3.10(b), if we chose ordering 'first C and then A', then we would create an initial-token-free cycle – (C, A, B)*.

An objective that the ordering sub-step should pursue in a flow like our preferred flow is minimizing of the estimated number of channels in the final implementation. We will explain this objective later on in this section, when we consider the JPEG decoding application example.

The last thing that we mention about the ordering sub-step in general is that this is the first sub-step in our preferred mapping flow that modifies the structure of the HSDF graph (except that the partitioning into tiles may add 'read' and 'write' actors in some cases). After creating the process cycles, $\mathbf{GP}(p_i)$, the ordering sub-step removes any previously existing intra-process edges, i.e., the edges that join different actors in the same process (whereby, in the

implementation process network, it also removes the intra-process channels, i.e., the channels that correspond to the intra-process edges). The reason for that is that in the presence of the process cycles the intra-process edges are superfluous, as we will see in the JPEG example below.

Coming back to our JPEG decoding application example, we observe that we still need to apply the three last sub-steps of the computation mapping phase there. Recall from Figure 3.12(a) that we have partitioned the computation graph into two tiles. In the partitioning of tiles into processes, we choose to have one process per tile, and, as we will see later, this does not lead to artificial cycles in this example. Figure 3.12(b) shows the impact of the partitioning into processes on the implementation process network. We see two processes, p_1 and p_2, joined by six channels that cross the partition boundary in Figure 3.12(a). As already mentioned, at this point of the flow, the process network also contains intra-process channels, but they are removed later by the ordering sub-step, and we do not show them in Figure 3.12(b).

For the budget assignment step, in this example, we choose to follow the objective to maximize the performance slack. Therefore, we assign 100% processor clock frequency to every process (because they are assigned to different tiles) thus not leaving any free processor budget for the other applications at the two processing tiles assigned to this application.

Now let us consider the last sub-step, i.e., the ordering. Look at Figure 3.12(a), where a line partitions the graph into two partitions. We have to decide upon the actor ordering in the left partition and the right partition. Hereby we keep two requirements in mind. The first requirement is that in every process, the ordering should respect the internal edges of the process that are free from initial tokens, which means that the producers of such edges should be located earlier in the ordering than the consumers. This requirement is implied from the mentioned above requirement about initial-token-free paths in the computation graph. The second requirement is that the 'write' actors in process p_1 should be ordered in the same order as the corresponding 'read' actors in process p_2. The purpose of this requirement is to ensure that the estimated number of the network channels in the final implementation is minimal, i.e., equal to one. The point is that if the producers of the inter-process channels are executed in the same order as the corresponding consumers then the communication assignment step later on in the flow can assign those inter-process channels to the same channel.

These two requirements still leave multiple ordering choices open. We choose the ordering for process p_1 as follows. Ignore for the time being all the actors that have delay 7,5, i.e., the 'write' actors. Inspect the remaining actors column-by-column from left to right, as they are placed in the figure. In each column, consider them from top to bottom. Use this ordering, and for each actor insert its corresponding 'write' actor immediately behind it. One can see the resulting ordering for p_1 in Figure 3.13 in process cycle **GP**(p_1). (Ignore, for the time being, the communication actors shown in Figure 3.13 in the context of channel macros **GQ**(q_j).)

In partition p_2, we first consider only the 'read' actors column-by-column from left to right and in each column from top to bottom. Note that every 'read' actor has a corresponding non-'read' actor that directly consumes the token received by the 'read' actor. Having ordered the 'read' actors in this way, for each 'read' actor we insert its corresponding non-'read' immediately behind it. All the non-'read' actors that do not have a corresponding 'read' actor are placed in the end or the ordering, represented in Figure 3.13 in process cycle **GP**(p_2).

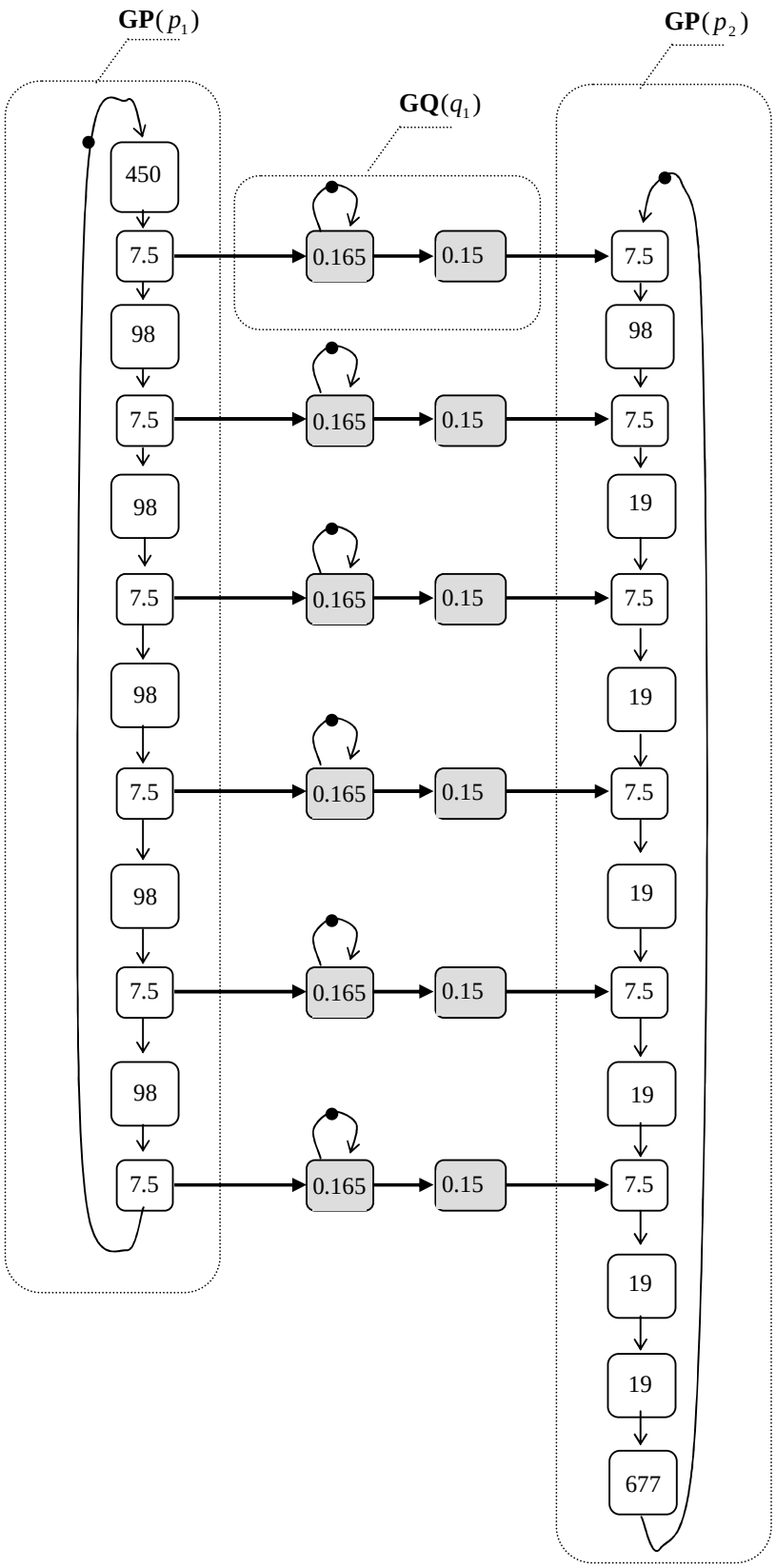


Figure 3.13 The JPEG decoder HSDF graph after ordering and and communication actor insertion

As we see in Figure 3.13, in the HSDF graph, the ordering sub-step has replaced the intra-process edges by the sequence edges of the process cycles. For example, in Figure 3.12(a), all the data edges that are outgoing from actor VLD (the actor with delay 450) are intra-process edges. In Figure 3.13, we do not see any of those edges, but all the actors that are consumers of those data edges come later than the actor VLD in the process cycle. Therefore, keeping those data edges would be unnecessary, as they would not have any impact on the performance analysis anyway.

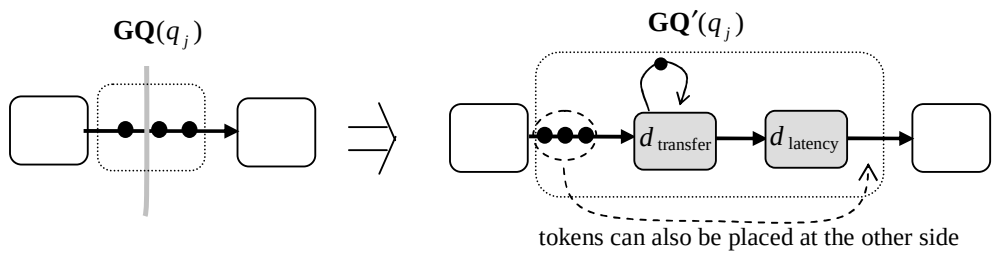
In the end of our discussion of the processing assignment and ordering, we would like to stress again that the splitting of the computation part of the mapping flow into sub-steps is not the only possible splitting and there is phase coupling between these sub-steps, and thus the order in which they are executed matters for the quality of results. Thus, it might be favorable to integrate these substeps into one optimization problem, so that the problem solver can see the whole design space of this part of the flow at a time.

This completes our discussion of the processing assignment and the ordering mapping steps. In the next section, we consider the mapping steps that follow later in the preferred mapping flow.

3.6 The Communication Mapping Phase

3.6.1 Communication Assignment and Communication Actor Insertion

If we take a look back at the computation phase of the mapping flow, described in the previous section, and ask ourselves what impact that mapping phase has on the final HSDF graph – the IPC graph – then the answer would be that that phase defines how certain subgraphs of the IPC graph will look like when the flow is completed. Those subgraphs are the process cycles, $\mathbf{GP}(p_i)$. Recall from Section 2.1.3 that the HSDF graph of our implementation-enhanced model can be decomposed into parts called *macros*. Every macro corresponds either to a process or to a channel of the implementation process network. The process cycles, $\mathbf{GP}(p_i)$ are, in fact, the process macros. The mapping phase considered in this section keeps the process macros intact and only transforms the channel macros, $\mathbf{GQ}(q_j)$, until they also reach the final form. Compared to the final structure of the process macros, consisting of just a single cycle, the final structure of the channel macros is more complex; it depends on whether the channel is a local channel, i.e., contained in a single tile, or a network channel, joining two tiles together. In this section, we introduce and explain the channel macros step-by-step. For the JPEG application example, whose graph has almost attained the structure depicted in Figure 3.13 (except that the communication actors already shown in that figure are yet to be inserted), we can now note that the process macros we see in that figure, $\mathbf{GP}(p_1)$ and $\mathbf{GP}(p_2)$, will be imported without changes into the IPC graph of that application, whereas the edges in between, now representing the six channels of the current process network, are going to undergo certain transformations. Communication actors and new edges are going to be introduced into graph **G**, so that the model adequately captures the mapping decisions taken at the communication phase. The new channel macros, $\mathbf{GQ}(q_j)$, are going to be built from the communication actors and the edges joining either a computation actor to a communication actor, or two communication actors, or two computation actors belonging to different processes.



(a) original channel macro = an edge **(b)** new macro = a graph substructure

Figure 3.14 Communication actor insertion

As shown in Figure 3.7, the first step of the communication mapping phase is the *communication assignment*, which is the main topic of this subsection. This step has the following tasks:

1) communication actor insertion (for the network channels),

2) channel number minimization (for all channels),

3) bandwidth minimization (for the network channels).

In the rest of this subsection we consider Task 1). It is the simplest task, and it is illustrated in Figure 3.14. For every channel crossing the tile boundary, we change the channel macro such that the data edge originally contained there is replaced by a graph structure consisting of two actors and four edges, as shown in Figure 3.14(b). The new channel macro contains a graph path that joins the channel producer and the channel consumer; in Figure 3.14(b), that path goes from left to right. The two new actors model the two components of the network connection delay – recall them from Section 3.4. The first actor on the path models the transfer delay, d_{transfer}, and the second actor models the network latency, d_{latency}. Therefore, the first actor is called a *transfer actor* and the second one is called a *latency actor*.

As for the initial tokens of the original data edge, one can choose one of the two possible implementations of communication channel, whereby in the channel macro the initial tokens are placed either in front or at the back of the path. This choice depends on the application algorithm, i.e. on whether, at the start of the loop execution, the content of the initial tokens can be implied by the consumer actor (e.g. all zeros) or whether it has to be pre-generated by the producer actor before the start.

Originally, the transfer actor has a sequence edge that joins the actor with itself. Recall that, unlike the network propagation activity, modeled by the latency actors, the subsequent transfer activities of the given network connection can only execute sequentially, and that is exactly what that edge is modeling.

We classify both newly introduced actors as *communication* actors because, unlike the computation actors, they are not executed by any processor; instead, they model the behavior of the on-chip network. The results of the actor insertion for the JPEG decoding example are shown in Figure 3.13. Each of the six channels in that example has attained a new channel macro similar to the one in Figure 3.14(b).

When new actors are inserted into the HSDF graph of our implementation-enhanced HSDF model, their delays have to be defined based on the budgets (recall Figure 3.1). Recall from

Equalities (3.19), (3.20) and (3.21) that the transfer delay depends on the token size. Because, in this thesis, we restricted ourselves to static token sizes, not depending on the input data content, the transfer delay does not depend on the actor parameters, thus keeping the same constant value in every timing mode. The same holds for the latency delay, assigned to the latency actor.

In the preferred mapping flow, during actor insertion we assume that each network channel gets the maximum network link bandwidth, independently from the other channels. Due to the monotonicity of HSDF graphs, this means that we assume the best performance the model can have with the given communication network architecture. Thus, similarly to the computation phase, we start the communication phase with an initial solution having the biggest chance to satisfy the application throughput constraint – if that does not happen then either the computation phase results have to be reconsidered or there is no feasible solution at all. The initial solution may use an unrealistic amount of the network bandwidth, but later on, at task 3), the bandwidth assignment should be 'relaxed' such that the bandwidth requirements get into the realistic scope while the throughput constraint is still satisfied.

As for the latency, in general, it depends on the distance between the physical processors to which the virtual tiles are assigned at run time. Our preferred mapping flow assumes that the latency values are much smaller than the computation actor delays, and therefore it assigns each latency actor a constant delay that is computed based on the maximum possible distance between the tiles in the given network-on-chip; after all, being conservative here does not imply being too pessimistic, because as long as our assumption holds the network latency does not have any considerable impact on performance.

For the JPEG decoding example, we use the instance of the ÆTHEREAL NoC that is described in Section 3.4. The token size of every data edge in this example is 128 bytes (or 64 pixels, every pixel being a two-byte word). Assuming the highest connection bandwidth, $B_{\text{conn}} = B_{\text{LINK-Æ}}$ and applying Equality (3.21) and taking the constants of Table 3.1, we get $d_{\text{transfer}} = 165$ ns. As for the latency, it is reasonable to assume that it is always possible to route a connection between any two processing tiles on chip using up to 20 routers; thus, by applying Equality (3.22) and Table 3.1, we obtain $d_{\text{latency}} = 150$ ns. The obtained numbers for the transfer delay and the latency are assigned as actor delays to the graph shown in Figure 3.13.

Task 1), considered so far, does not introduce any artificial cycles into the HSDF graph, except for the cycles due to the edges around the transfer actors. The cycle means of those cycles must be smaller than Λ_{allowed}, otherwise the mapping problem is doomed to fail in finding a good solution due to high communication delay of the platform. The communication actors may also increase cycle means of the cycles that correspond to the cycles present in the original graph, but the corresponding increase in the cycle mean must also be non-dominating, for the same reason.

3.6.2 Minimization of the Number of Channels and Bandwidth

The communication assignment task that follows after the insertion of the transfer and latency actors is the minimization of the number of channels. In the previous subsection, we denoted that task as Task 2). That task comprises a combinatorial optimization problem, and the same holds for Task 3), i.e., the minimization of the required channel bandwidth. Similarly as for the previous mapping steps, since it is not our purpose to propose new solutions to the communication assignment problems, we only give a brief description for them and point out the relationship between those problems and our implementation-enhanced HSDF model.

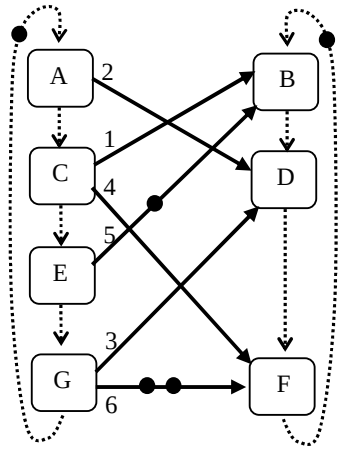
In the channel number minimization problem, the *state* channels and the *communication* channels are treated similarly, but separately. For the end result, it does not matter which channel type is considered first, so let us start by considering the communication channels. Recall, that among those channels, we distinguish between the *local channels*, staying within one tile, and the *network channels*, joining two tiles. For the latter, also the bandwidth should be minimized – Task 3). Having considered both minimization tasks for the communication channels, we explain a similar but simpler optimization problem for the state channels in the end of this subsection.

Recall that every channel at this point of the flow has exactly one producer actor and one consumer actor. We refer to such channels as *simple channels*. The end result of the channel number minimization is the merging of the simple channels into *complex channels*, which have, in terms of the definition given in Section 2.1.3, a transfer set **TQ** with cardinality more than one. The purpose of channel merging is the sharing of the FIFO buffers and the network connections between multiple transfers. Looking ahead, for the JPEG decoding example, we can mention that the channel number minimization results in the merging of all six simple channels shown in Figure 3.12(b) into one complex channel.

Note that channel merging cannot introduce deadlock if it follows the compatibility rules defined later in this subsection. Moreover, the merging of local and state channels does not have an impact on HSDF graph throughput, as it does not change the graph structure and delays. As we will see later in this subsection, merging of the network channels can decrease the throughput and thus it should be done with the throughput constraint in mind.

The preferred mapping flow splits the optimization problem of Task 2) into basic subproblems working with different comparable parts of the HSDF model. The preferred mapping flow considers different parts of the problem separately, in an arbitrary order. Due to the possible coupling between the subproblems, the preferred approach may be suboptimal, but finding exact solutions for mapping problems is beyond the scope of this thesis. We only present reasonable indications in order to demonstrate the usefulness of our modeling techniques to formulate the mapping problems and to guide the optimization algorithms.

We define the basic channel number minimization subproblem by selecting a distinct ordered pair of processes (p_a, p_b) having the property that there is more than one communication channel going from process p_a to process p_b. Those channels are candidates for merging. Figure 3.15 shows an example of the problem instance, where there are six candidate channels. The best would be to merge all those channels into one complex channel, but as mentioned in the figure, for this problem instance, the minimum number of channels is three. Soon below, we explain how a feasible solution should look like, and it will become clear why in this example the chosen solution is feasible and why it is infeasible to further merge channels.



a sequence edge path without initial tokens

a sequence edge path containing one initial token

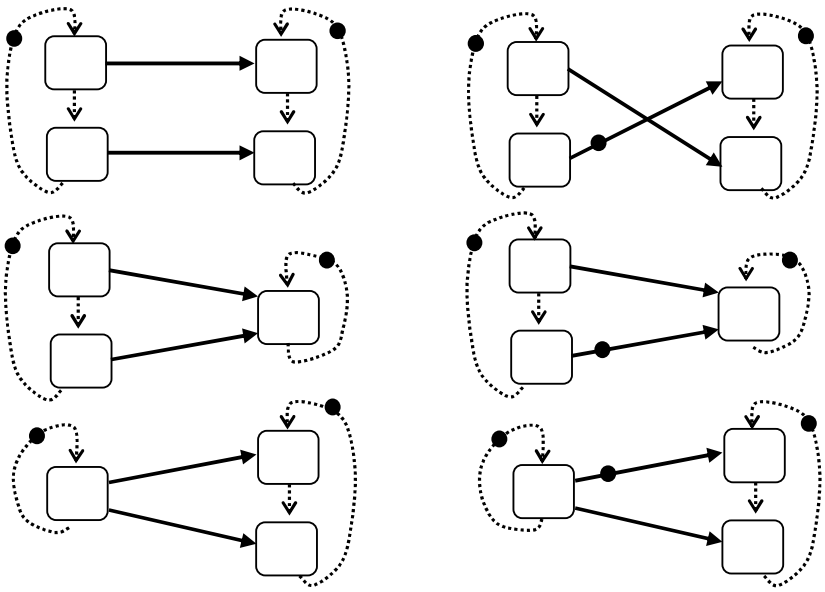
an optimal solution: three channels:
channel I: merged 1 and 3
channel II: merged 2, 4 and 5
channel III: 6

Figure 3.15 An instance of the channel number minimization problem

Note that in this example the channels are local; therefore, each channel is represented just by an edge and does not have communication actors. Note also that there is an important difference in the formulation of channel number minimization problem between the case where the channels are local and the case where they are network channels, because, merging the network channels together creates new cycles in the HSDF graph and thus can have impact on performance. This peculiarity of the network channel merging is discussed later in this subsection, and for now we focus on the compatibility of different channels for merging, where the rules for the local and network channels are similar.

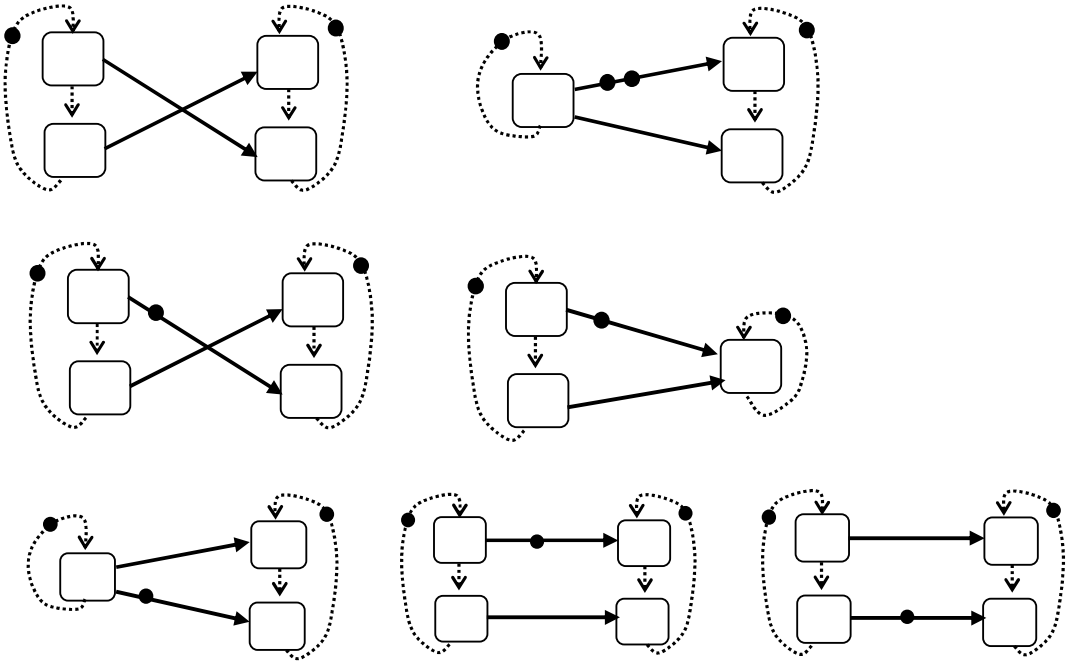
The preferred flow distinguishes only those actors in process p_a that are producers of the channels going to process p_b. Similarly, only those actors in process p_b are considered that are consumers of those channels. In general, p_a and p_b may also contain other actors, but they are not in the scope of the problem instance, and any such actors are omitted from Figure 3.15.

As already mentioned before, when forming complex channels with multiple producers and consumers, one should take the actor execution order into account to ensure the proper FIFO ordering of the communication transfers; namely, the producers should produce the tokens in the same order as the corresponding consumers consume them. For example, in Figure 3.15, simple channels ‘1’ and ‘2’ cannot be merged together because their producers execute in the opposite order to their consumers. When, like in this case, two simple channels cannot be merged together such that FIFO ordering of communication transfers can be ensured, we say that they are *incompatible*. If they can be merged together such that FIFO ordering can be satisfied then we call them *compatible*; for example, so are channels ‘2’ and ‘4’ in Figure 3.15.



(a) the patterns of compatible channels

more compatible channel patterns can be obtained by adding the same number of initial tokens to both channels in one of the patterns



(b) the patterns of incompatible channels

Note! The positions of initial tokens are essential for each pattern!

Figure 3.16 Visual detection of channel (in)compatibility

Figure 3.16(a) illustrates a few examples of compatible channel pairs. The channels are represented by corresponding graph edges in the HSDF graph as it was before the insertion of the communication actors in Task 1) – to keep the illustration simpler for the network channels. One can check all these patterns one-by-one for compatibility by checking whether the first two tokens produced can be consumed in the same order as they can be produced; with a reasonable assumption that if an actor has multiple inputs/outputs then one can adjust the order in which the actor consumes/produces tokens at different inputs/outputs to the order in which the tokens are produced/consumed at the other side of the channel. For example, the producer of the bottom-left pattern in Figure 3.16(a) can first produce a token to the top edge and then to the bottom edge, and thus the corresponding channels are compatible. Note that because the production and consumption order of the patterns repeats cyclically, the FIFO-compliance check for the first two tokens produced in the complex channel is a necessary and sufficient condition of the FIFO-compliance of the whole HSDF execution run.

If a channel pair matches the pattern shown in one of the examples of Figure 3.16(a), then the corresponding two channels are compatible. 'Matching' a pattern means that a pair of channels has the same number of initial tokens as in the pattern and the same ordering of the producers and the consumers. For example, channel pair {4, 5} from Figure 3.15 matches the top pattern in the right column of Figure 3.16(a). Every pair of channels in Figure 3.13 – if one removes the communication actors – matches the top pattern in the left column. Note that more patterns of compatible channel pairs can be obtained from any pattern in Figure 3.16(a) by adding the same number of initial tokens to both channels.

Figure 3.16(b) shows incompatible channel patterns. One can check them one-by-one for violation of FIFO ordering. For example, it is easy to see that the first two tokens produced in the top-left pattern will be consumed in the opposite order. In our example in Figure 3.15, channel pair {3, 4} matches this pattern and therefore those two channels cannot be merged together. Note that the positions of the initial tokens are essential in all patterns; for example, if one removed initial tokens from the top-right pattern in Figure 3.16(a) and the top-left pattern in Figure 3.16(b), then those two patterns would become identical.

Definition. (Compatible channels/data edges) Two simple channels or two data edges that represent simple channels in the HSDF model are called compatible if they match one of the patterns in Figure 3.16(a) either directly or after adding the same number of initial tokens to every data edge. ♦

One can show that our original criterion to distinguish compatible channels – i.e. the compliance to FIFO ordering – and this explicit definition are equivalent. This can be proven as follows. First of all, all the patterns where the difference in the number of initial tokens is two or more are obviously incompatible (e.g. the top-right side Figure 3.16(b)). The reason for that is that at start of the execution, the new tokens produced on the edge with the larger number of tokens will have to wait until the new tokens at the other edge are consumed at least two times, which obviously violates the FIFO order of productions and consumptions. All the patterns where the difference is zero or one can be reduced – by removing the same number of initial tokens at every data edge – either to one of the patterns in Figure 3.16(a) or to one of the patterns in Figure 3.16(b) (except the top-right pattern). As mentioned before, one can check every pattern one-by-one to verify whether the FIFO-ordering criterion is satisfied.

To solve the channel number minimization subproblem for a given pair of processes, one should split the set of simple channels into as few as possible subsets such that in every subset

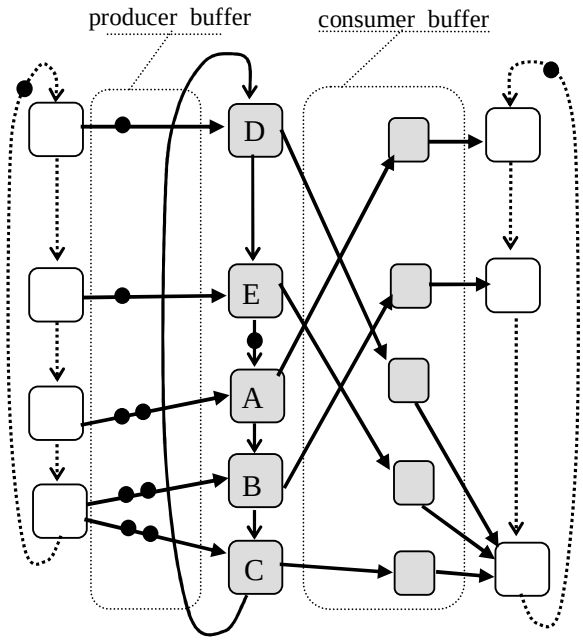


Figure 3.17 A complex transfer cycle construction example

any two channels are compatible. For the problem instance in Figure 3.15 three such subsets have been found; they are mentioned in the figure. For the JPEG decoding example in Figure 3.13, all the channels are compatible, and thus we can merge them into one complex channel.

Recall that it is not our goal to present any algorithm for mapping flow, so we do not discuss algorithms for solving the channel number minimization problem. Instead, we continue the discussion on the relationship between that problem and the HSDF model that represents the current mapping decision.

Unlike the minimization of the number of local channels, the minimization of the number of network channels leads to transformations of the HSDF graph **G** and thus it can have impact on the throughput. The point is that when one decides to merge a few simple network channels into one complex channel one has to join the transfer actors of those simple channels into one cycle, similar to the process cycle. This cycle is called a *transfer cycle*. The construction of a transfer cycle is explained in the following definition.

Definition. (The transfer cycle of a complex network channel: construction rules) Consider the set of all transfer actors of the simple network channels merged into one complex network channel. Because the transfer actors are in one-to-one correspondence with the communication transfers, introduced in Section 2.1.3, we use the same notation for that set as for the set of transfers, **TQ**. In the example given in Figure 3.17, **TQ** = { A, B, C, D, E }. Let $m_{\text{s-min}}$ be the minimum number of initial tokens per simple channel. Note that a set of simple channels can be mutually compatible only if the maximum difference in the number of initial tokens in the set is one. Therefore, only simple channels with either $m_{\text{s-min}}$ or ($m_{\text{s-min}}$ + 1) can be present in the set (in Figure 3.17, $m_{\text{s-min}} = 1$). Therefore, in general, set **TQ** can be split into two subsets: $\mathbf{TQ} = \mathbf{TQ^m} \cup \mathbf{TQ^{m+1}}$, where $\mathbf{TQ^m}$ is the set of transfer actors of the simple channels with $m_{\text{s-min}}$ initial tokens and $\mathbf{TQ^{m+1}}$ is such a set for the simple channels with ($m_{\text{s-min}}$ + 1) initial tokens.

Note that in general, the latter subset can be empty. In our example $\mathbf{TQ^m}$ = { D, E } and $\mathbf{TQ^{m+1}}$ = { A, B, C }. To define the ordering of the transfers in the transfer cycle, first we have to order the elements in both subsets, thereby forming two ordered sequences, $\mathbf{tq^m}$ and $\mathbf{tq^{m+1}}$. Sequence $\mathbf{tq^m}$ orders the transfer actors from $\mathbf{TQ^m}$ in the order that is consistent with the ordering of the corresponding producers. If any two transfers have the same producer, then one has to check the ordering of the corresponding consumers to decide upon the ordering. As for sequence $\mathbf{tq^{m+1}}$, it uses the same rules, but it orders the transfer actors from $\mathbf{TQ^{m+1}}$. For example, in set $\mathbf{TQ^{m+1}}$ from Figure 3.17, actors B and C have the same producer, but actor B has the earlier consumer; therefore actor B precedes actor C. The resulting sequences for our example are: $\mathbf{tq^m}$ = (D, E) and $\mathbf{tq^{m+1}}$ = (A, B, C). Finally, the transfer cycle order is equal to the concatenation of the two sequences, with $\mathbf{tq^{m+1}}$ coming in front. Let '∘' denote the concatenation operation, then:

$$\mathbf{tq} = \mathbf{tq^{m+1}} \circ \mathbf{tq^m} \qquad (3.26)$$

where **tq** is the ordered sequence defining the transfer cycle order. In our example, we have: **tq** = (A, B, C, D, E).

Given the ordering, the transfer cycle is constructed in the same manner as a process cycle: between every two subsequent actors a sequence edge is introduced, without initial tokens, and there is a sequence edge with one initial token going from the last actor in the sequence to the first one. The transfer cycle is introduced into the HSDF model to reflect the fact that the data tokens coming from the producer buffer of the same network channel can only enter the network connection sequentially, one after another.♦

It is worthwhile to draw attention to the 'unusual' position of the initial token in the transfer cycle in Figure 3.17 – in the 'middle' of the transfer cycle. Technically, this is the result of the transfer cycle construction rule that we have just introduced. The rule places the initial token in front of the first actor of sequence $\mathbf{tq^{m+1}}$. Only if that sequence is empty (when $\mathbf{TQ = TQ^m}$), is the initial token in the transfer cycle 'aligned' with the first producer.

As illustrated Figure 3.17, the HSDF model of a complex network channel can be split into three parts: the producer buffer model, the transfer cycle and the consumer buffer model (whereby the latter also includes the model of network latency). The producer buffer model consists of the data edges joining the producer process cycle to the transfer cycle. The consumer buffer model consists of the data edges and latency actors joining the transfer cycle to the consumer process cycle.

Note that, it is not difficult to intuitively see the correctness of the complex network channel construction in Figure 3.17 in terms of token transfer ordering. This can be done using the following remark. First, in the consumer buffer model, let us, for explanation purposes, assume that we remove the latency actors and merge the incoming and outgoing edge of each such actor into one edge. Then, for both buffer models, it holds that one can enforce a series of actor executions in such a way that all the data edges in the buffer model get the same number of tokens, and afterwards it is possible to rearrange the actors (preserving the graph structure) in such a way that no data edges cross each other and the position of the initial token in the cycle that produces the data tokens corresponds to the initial token of the cycle that consumes the data tokens. For example, in Figure 3.17, for the producer buffer model, if we enforce execution of actors A, B, and C, then all the data edges will have one initial token and the initial token of the transfer cycle will move to edge (C, D), which corresponds to the initial position of the process cycle producing the data tokens to the channel. In the same figure we see that we can shift actors

A, B, C to the top of the figure, and D, E to the bottom, such that no data edges in the consumer buffer model cross each other, whereas the initial token position of the transfer cycle corresponds to the initial token position the process cycle of the consumer process.

It is obvious that the transfer cycle can have an impact on the performance. In theory, it can become a critical cycle, limiting the throughput that the application can provide. This can be the case when the platform's communication network is not tuned well for the given application domain and too slow or when the application has unusually high communication requirements (manifested in relatively large token sizes, causing large transfer delays). In that case, the optimal solution for the channel number minimization problem may not be the best choice. However, in practice, we do not expect that a transfer cycle can become critical at this point of the flow, because, as the reader may recall, so far we assume that each channel uses the maximum bandwidth a network can provide for one single channel, resulting in very small transfer actor delays. What indeed can happen is the phenomenon that the introduction of a long transfer cycle can lead to an extension of an existing critical cycle, whereby it changes its route and uses some new edges introduced by the transfer cycle.

Note that the transfer cycle together with its incoming and outgoing data edges and also with the adjacent latency actors and their outgoing data edges – all those primitives together – now constitute a new single channel macro, which is the *macro of a complex network channel*. For example, in Figure 3.17, all the actors and edges in between the producer process cycle and the consumer process cycle constitute the macro of a channel performing five communication transfers per loop iteration and containing eight initial data tokens in total. As for the local channels, we can make a similar statement: all the edges that belonged to the simple channels merged into a complex channel now constitute a new channel macro, which is the *macro of a complex local channel*. For example, in Figure 3.15, after the channel merging, edges 2, 4 and 5 together form a channel macro of a complex local channel.

In the implementation process network, **PQ**, the new channels – complex channels – come to replace the simple channels that have been merged. The simple channels that have not been merged stay untouched; they can be seen as 'complex' channels containing just one transfer per loop iteration.

All the communication channel macros that are present in the HSDF graph at this point of the flow come as an input to the buffer capacity minimization step, which follows as the next step in the mapping flow (see Figure 3.7) and which we consider in the next two subsections.

Let us step back and recapitulate where we are in the description of the channel and bandwidth minimization tasks. So far, we have considered the basic subproblem of channel number minimization for the communication channels. Now there are a few 'smaller' topics to consider before we finish the description of the communication assignment and close this subsection. Those topics are:

1) bandwidth minimization in the network channels,

2) state channel number minimization.

We consider these topics one-by-one in the remainder of this subsection, and we start with the bandwidth minimization. As already mentioned, in an ideal situation, at this point of the flow, the transfer cycles should have no or very little impact on the performance, because so far we assume the highest network bandwidth values allocated for the channels. To ensure that the requested bandwidth can be provided in reality and to decrease the communication resource

usage of the application, one should 'relax' the bandwidth values assigned to the channels. This is done by the bandwidth minimization task, the last task in the communication assignment step. The bandwidth budgets for all channels can be minimized using a cost function like the average or the maximum bandwidth per channel. The bandwidth minimization leads to a delay increase in the transfer cycles and other cycles that include the transfer actors. Therefore, the 'relaxation' of the bandwidth values should be constrained such that none of those cycles gets a cycle mean higher than Λ_{allowed}. Also a set of constraints should be imposed that specify the maximum bandwidth the network can provide per tile per producer link and consumer link (see Figure 3.6).

The last (sub-)task of the communication assignment we have not discussed so far is the minimization of the state channels. That task is formulated and solved the same way as the local channel number minimization. The state channels are, in fact, very similar to the local channels, the major difference being that the state channels do not carry any data tokens; they only enforce some actor executions to wait for the completion of certain other actor executions. For the interested reader with a background in concurrent systems, we mention that the inter-process state channels can be implemented using semaphores – special variables that can be 'acquired' and 'released' multiple times; those variables are implemented as counters whose increments and decrements are atomic operations; trying to do more 'acquires' than 'releases' leads to blocking until the semaphore is released by another process. In the semaphore-based implementation, the channel producers would release the semaphore once per production and the channel consumers would acquire it once per consumption.

In the JPEG example, the communication assignment step joins all the transfer actors in Figure 3.13 into one cycle (to see this transfer cycle, the reader can look ahead in this chapter and consider an IPC graph for this application in Figure 3.23). The bandwidth minimization step reduces the network bandwidth assigned to the channel from the maximum to the minimum possible value in the ÆTHEREAL network, $B_{\text{min-Æ}}$ = 3.125 Mbyte/s. Due to this change, all the transfer actors (see Figure 3.13) acquire a delay value of 42 μs, which does not affect the throughput of the application in an adverse way, because the total delay of the transfer cycle is 6 times 42μs – i.e. 252 μs – whereas Λ_{allowed} is 1000 μs.

3.6.3 Modeling the FIFO Buffers of the Communication Channels

After the communication assignment, the intra-application mapping is almost complete. The implementation process network has almost reached the final form: the process network structure has been finalized, the contents of the processes and channels have been defined in terms of actors/transfers and their ordering, the budgets have been assigned to the processes and channels in terms of the processor cycles per unit of time and the communication bandwidth. The only part of the implementation-enhanced HSDF model that still has to be filled in by the intra-application mapping flow is the capacities of the FIFO buffers in the channels. Recall from Section 3.5.2 that the network channels are characterized by the capacities of the producer and the consumer buffer, $Q_{\text{prod-buffer}}, Q_{\text{cons-buffer}}$, and the local channels are characterized by the local buffer capacity, Q_{buffer}.

Before we can describe the final step of the preferred mapping flow, the buffer capacity minimization, we define how we model the FIFO buffers having *finite buffer capacities* and multiple producers and/or consumers. That modeling technique is, in fact, the main topic of this

subsection. We spend extra attention on this topic because it comprises an important part of our contribution. The modeling of finite-capacity FIFO buffers enables performance analysis of streaming applications whose channels are possibly shared by multiple producer and consumer actors and are mapped to network-on-chip. In the next subsection we use this modeling technique to guide buffer-capacity minimization.

The simplest FIFO buffer models are HSDF graph edges. They can be used to model the infinite-capacity FIFO buffers. The FIFO buffer of a simple local channel with an infinite capacity is represented by a single data edge. It is essential that the producer of that edge be enclosed in a cycle with one initial token, because that ensures that the subsequent productions do not overlap and do not overtake one another. The consumer should also be enclosed in such a cycle. For example, in Figure 3.15, we see six data edges, modeling the buffers of six simple local channels. As for the simple network channels, consider their channel macro in Figure 3.14(b). The data edge that enters the transfer cycle models the producer buffer. The two data edges separated by the latency actor model the consumer buffer, whereby the latency actor represents the latency of the token arrival to the consumer buffer.

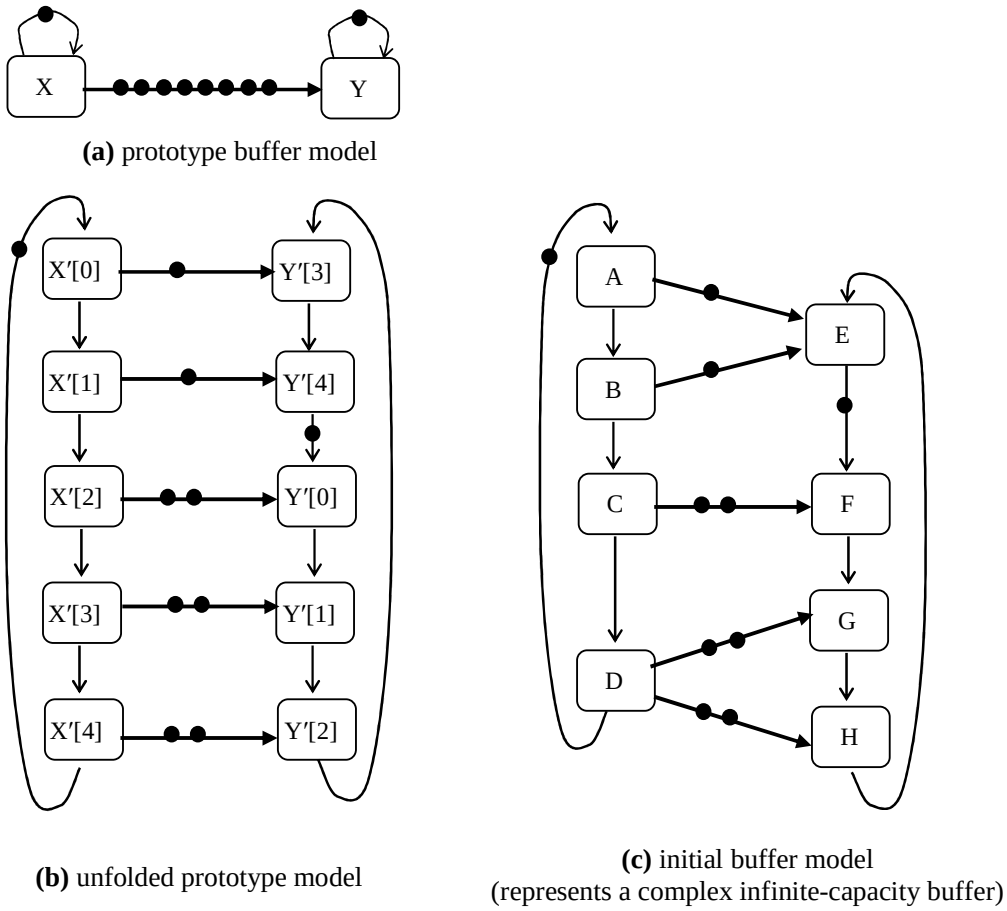
After the channel merging, the FIFO buffers of complex channels are modeled by the *collection of edges* that, before the merging, were used to model the buffers of the simple channels. For example, in Figure 3.17, the data edges ingoing into actors A, B, C, D, E (entering those actors from the left side in the figure) model the producer buffer of the underlying complex channel. The latency actors and their adjacent edges model the consumer buffer and the latency of data arrival into the consumer buffer. By analogy to the channels, we give the name '*complex buffers*' to the buffers having multiple consumers and/or multiple producers.

Buffer models that contain only edges going from the producers to the consumers represent *infinite capacity* buffers. In this subsection, we assume that a finite buffer capacity is provided and explain how one can model the buffers with the given finite capacity. As we will see later, for that, we introduce special edges going in the reverse direction: from the consumers to the producers.

In our explanation, we assume that we have an HSDF model for an infinite-capacity complex buffer, like the one shown in Figure 3.18(c). We assume that now we would like to limit the capacity of that buffer to a given finite capacity value. Therefore, we study how to reflect the limited capacity in the HSDF model. Note that for convenience, in this subsection, we often refer to the HSDF models of a buffer simply as 'buffer'. In other words, we use the terms '(graph-theoretic) buffer model' and 'buffer' interchangeably. Thus, we refer to the infinite-capacity complex buffer model as *initial buffer* or initial buffer model. We refer to the finite-capacity buffer model that is obtained from the initial buffer as *final buffer* or final buffer model.

Let us take a step back and reconsider the infinite-capacity complex channels introduced in the previous subsection. To be more precise, we reconsider their buffers. We are going to show that every initial buffer can be represented by a simple meta-model that expresses the essential properties of the buffer in a simple and transparent manner. We call that meta-model the *prototype buffer*. As opposed to the initial buffer, which is, in general, a complex buffer, the prototype buffer is always a simple buffer, i.e. it has only one producer and consumer. For example, an initial buffer and its prototype are shown in Figures 3.18(c) and 3.18(a) respectively. Soon we will explain and study that example in detail.

A prototype buffer has a simple structure and there exists a clear relationship between any valid initial buffer and its prototype. Note that this relationship preserves the buffer capacity. The



(a) prototype buffer model

(b) unfolded prototype model

(c) initial buffer model
(represents a complex infinite-capacity buffer)

Figure 3.18 The graphs modeling a complex buffer

reason to introduce the prototype buffer is that one can easily create a variant of the prototype buffer that has a finite buffer capacity. Then, using the relationship between the prototype and the complex buffer one can translate the finite-capacity variant into the complex-buffer form and thus obtain the model of a complex buffer with a finite buffer capacity.

First let us study the infinite-capacity buffers. Let m be the number of initial tokens in the initial buffer and H be the number of simple channels merged into the initial buffer (or, equivalently, the number of transfers in the complex channel per loop iteration). For illustration purposes, we use an example with the initial buffer having $m = 8$ and $H = 5$. The graph model of the initial buffer is illustrated in Figure 3.18(c). Visually, it may seem that the edges of that buffer are incompatible (some edge pairs seem to not match the compatibility patterns in Figure 3.16), but one should beware the ‘unusual’ position of the initial token of the consumer that explains the counterintuitive visual effect. To verify visually that in reality all edge pairs are compatible, one can pull the top consumer to the bottom of the figure and then use the patterns of Figure 3.16 again. However, we intentionally have placed all the consumers with the smallest number of the initial tokens at the top of the figure and, as a by-product, we see that no data edges visually cross each other. It can be shown, that one can place the consumers of any complex buffer this way.

For clarity and simplicity, we are now making an important assumption, namely, we assume that all the transfers of the complex channel that contains the given complex buffer have the

same data token size, *z*. Later on, we will show how to waive that assumption and keep the model accurate.

Here we need to introduce the notion of the initial buffer model more thoroughly.

Definition (Initial buffer model: equal token size case) The initial buffer model can be seen as an extraction from the application's HSDF graph that models only the particular complex buffer. It is a graph whose set of vertices is the set of producers and consumers and the set of edges includes:

1) one data edge per one communication transfer (i.e., per original simple channel that was merged into the channel);
2) one sequence edge per subpath of the process cycle (or transfer cycle) joining two subsequent buffer producers/consumers, whereby the sequence edge should have the same number of initial tokens (either zero or one) as the corresponding subpath.

For example, the sequence edge between actors A and B in Figure 3.18(c) may correspond to a token-free chain of actors that come in the process ordering in between actors A and B and that do not access the initial buffer and – for that reason – do not appear in the buffer model. ♦

Now let us consider the prototype buffer model. The prototype buffer model built for the initial buffer example in Figure 3.18(c) is shown in Figure 3.18(a). A prototype buffer always has a single producer, single consumer and just one data edge. If all transfers of the initial buffer have equal token sizes then the prototype buffer carries the same number of initial tokens as the original buffer. In Figure 3.18(a), we see that all the initial tokens, which were distributed between different edges in the original graph, are neatly collected on the same edge.

Let us now explain the relationship between the buffer models. The producer (or consumer) of the prototype buffer represents all the producers (consumers). One execution of the prototype producer (consumer) corresponds to one production (consumption) to (from) the initial buffer. In the example of Figure 3.18, executions 0 and 1 of prototype consumer Y corresponds to executions 0 of consumers F and G. Executions 0 through $(H-1)$ of Y (recall that $H = 5$) cover the executions of consumers F, G, H, and E in loop iteration 0. Note that E consumes two tokens, and thus corresponds to two executions of Y. Executions H through $(2H-1)$ cover their executions in loop iteration 1, and so forth.

In fact, this relationship is better explained by means of *unfolding* the prototype graph.

Definition (Graph unfolding and folding with factor H) HSDF graph $\mathbf{G}'$ is called an *unfolded representation* of HSDF graph $\mathbf{G}$ with unfolding factor H if every actor v_k in graph $\mathbf{G}$ is in one-to-one relation with a distinct set of H actors: $v'_k[0]$, $v'_k[1]$, …, $v'_k[H-1]$ in the unfolded graph $\mathbf{G}'$. Hereby the graph structure of an unfolded representation should imply a certain relationship between its behavior and the behavior of graph $\mathbf{G}$. To define that relationship, we first extend the relation between set $v'_k[j]$ and actor v_k by defining also a *correspondence* between the *executions* of $v'_k[j]$ and v_k. We define that any execution n' of actor $v'_k[j]$ in $\mathbf{G}'$ *corresponds* to the execution $n' \cdot H + j$ of actor v_k in $\mathbf{G}$. Then the relationship between the behaviors of $\mathbf{G}'$ and $\mathbf{G}$ is defined by two requirements given below:

1) Every token produced and consumed in $\mathbf{G}$ should be related to a unique token produced and consumed in $\mathbf{G}'$. Let a be the producer and b be the consumer of a token in $\mathbf{G}$. Then the producer in $\mathbf{G}'$ should be related to a, i.e. it should belong to set $a[0]$, $a[1]$, …,

$a[H-1]$. The consumer in $\mathbf{G}'$ should be related to b, i.e. it should belong to set $b[0]$, $b[1]$, …, $b[H-1]$.

2) The reverse statement to 1) should also hold, and thus the relation between tokens in **G** and in $\mathbf{G}'$ should be a one-to-one.

3) The delays of actor executions in $\mathbf{G}'$ are equal to the delays of the corresponding executions in **G**: $d(v'_k[j], n') = d(v_k,\ n' \cdot H + j)$

We say that graph **G** is the *main* graph with respect to an unfolded representation. The main graph is obtained from its unfolded representation by *folding it back* with the folding factor H. ♦

For example, Figure 3.18(b) shows an unfolded representation of the prototype buffer model in Figure 3.18(c) with unfolding factor 5. Actor X is related to actors X′ [0]…X′[4] and actor Y is related to actors Y′ [0]…Y′ [4].

As follows from Lemma 3.3, given below, the graph unfolding not only postulates a one-to-one relation between an actor in **G** and a set of H actors in $\mathbf{G}'$, but also implies a one-to-one relation between an edge in **G** and a set of H edges in $\mathbf{G}'$. Note also that Lemma 3.3 also implies that all unfolded representations of a given graph **G** are isomorphic (i.e., have identical structure); therefore, in the remainder, we speak of 'the' unfolded representation instead of 'an' unfolded representation. Note also that H, the (un)folding factor, must be a positive integer, and if $H = 1$ then $\mathbf{G}'$ is isomorphic to **G**.

Remark (Similarity of the relationship between the main and unfolded graphs and the relationship between the prototype and the initial buffers) The definition given above can be interpreted as follows: the executions with index n of actors $v'_k[0]$, $v'_k[1]$, …, $v'_k[H-1]$ in the unfolded graph $\mathbf{G}'$ represent the executions with index $n \cdot H$ to $(n+1) \cdot H - 1$ of actor v_k in graph $\mathbf{G}$. Thus, hereby we see a similar relationship as the relationship we have introduced between the initial and the prototype graph.

For example, in Figure 3.18(b) we see that the unfolded prototype graph looks similar to the initial graph, shown in Figure 3.18(c), although there are differences, on which we will elaborate later. ♦

In fact, due to property 3) in the definition, graph (un)folding keeps the timing behavior of the HSDF graph essentially intact; the main graph and its unfolded representation have closely related evolution equations, whereby one can be obtained from the other by variable replacement:

$$j = 0..(H-1) \quad \Rightarrow \quad x'(v'_k[j], n') = x(v_k,\ n' \cdot H + j) \tag{3.27}$$

where $x'(...)$ are the starting time variables in graph $\mathbf{G}'$ and $x(\ldots)$ are the variables of graph **G**.

We have explained the correspondence between the actors of the prototype graph and its unfolded representation. Now let us explain the correspondence between the edges of those graphs.

Definition (Positions of the actors in the unfolded graph) From the previous definition, it follows that any actor in $\mathbf{G}'$ can be identified as actor $v'_k[j]$ for some valid k and j. We refer to index j as the *position* of actor $v'_k[j]$.♦

Lemma 3.3 (The characterization of the unfolded graph edges) Let $\mathbf{G}'$ be an unfolded representation of graph $\mathbf{G}$ with unfolding factor H. Then every edge $e = (v_a, v_b)$ in $\mathbf{G}$ is in a

one-to-one correspondence with a distinct set of edges $\{e'_j\}$, $j = 0..(H-1)$ in $\mathbf{G}'$. Those edges are defined by:

$$j = 0..(H-1),\ m = m(e) \quad \Rightarrow \quad e'_j = \left(v'_a[j],\ v'_b[(j+m) \bmod H]\right), \tag{3.28.1}$$

$$j = 0..(H-1),\ m = m(e) \quad \Rightarrow \quad m(e'_j) = \begin{cases} \left\lfloor \frac{m}{H} \right\rfloor, & \text{if } (j + m \bmod H) < H \\ \left\lfloor \frac{m}{H} \right\rfloor + 1, & \text{if } (j + m \bmod H) \geq H \end{cases} \tag{3.28.2}$$

where $m(e)$ denotes the number of initial tokens on edge e.

This lemma can be summarized as follows. Firstly, it says that every edge in $\mathbf{G}$ corresponds to a set of H edges in $\mathbf{G}'$. For example, the edge from the producer to the consumer in Figure 3.18(a) has resulted in H edges in Figure 3.18(b), going from the producers to the consumers. Secondly, this lemma says that the abovementioned set of H edges can be split into two subsets, the set with m_s initial tokens per edge and the set with $(m_s + 1)$ initial tokens per edge, where $m_s = \left\lfloor \frac{m}{H} \right\rfloor$. The first set has producers with smaller positions: $j = 0...(H - m \bmod H - 1)$. The second set takes the rest of the range of possible producer positions and may be empty. To give an example of those two sets, in Figure 3.18(b), the set consists of two edges with one initial token and three edge with two initial tokens.♦

We skip a rigorous proof of this lemma, because the statement of this lemma becomes almost straightforward when we make two remarks. The first remark is that execution n of an actor in $\mathbf{G}$ corresponds to the execution n' of the corresponding actor at position j in $\mathbf{G}'$ if $n' = \lfloor n / H \rfloor$ and $j = n \bmod H$. The second remark is that in graph $\mathbf{G}$ (respectively, also in $\mathbf{G}'$) any token produced by execution n (resp. n') at any edge e (resp. e') is consumed by execution $n + m(e)$ (resp. $n' + m(e')$) of the consumer actor. Based on these two remarks and the requirements 1) and 2) from the graph unfolding definition, after some straightforward technical derivation one can obtain Equalities (3.28).

Let us now make three important observations.

Firstly, based on the Lemma 3.3, one can show that the data edges of the unfolded prototype buffer model are mutually compatible – for example, we already discussed that property for Figure 3.18(b). So, in other words, by *unfolding a simple buffer one obtains a complex buffer*.

Secondly, one can also show that the reverse statement is also true for a quite general case. Suppose that the buffer has only *simple producers/consumers*; i.e. produces/consumers that have only one outgoing/incoming data edge in the given initial buffer – e.g. in Figure 3.18(c), actor C is a simple producer and D is not, we call such actors *complex producers/consumers*. Let H be the number of transfers in the initial buffer and let all transfers have equal data token sizes. Under these assumptions, the structure of initial buffer can be shown to be an unfolded representation of the prototype buffer, with an unfolding factor H. In fact, this property follows from the requirement, introduced in the previous subsection, which states that the simple channels included into a given complex channel are *compatible*. In fact, this requirement 'enforces' that the edges of those simple channels follow the pattern defined by Lemma 3.3. Thus, *if one folds back a complex buffer with simple producers and consumers, one obtains a simple buffer*.

Thirdly, recall that in case all token sizes in the initial buffer are the equal, then the prototype buffer contains the same number of initial tokens *m* as the initial buffer. Therefore, from the first two observations, it follows that, for any positive integer *H*, the prototype buffer has a unique initial buffer with simple producers/consumers. That buffer can be obtained by unfolding the prototype buffer with factor *H*.

Now we have all the important ingredients to give a general description of the relation between the initial buffer and the prototype buffer for the case where the transfers have equal token sizes. We need to formulate that relation because later on we use it to step from infinite-capacity buffers models to the finite-capacity buffer models.

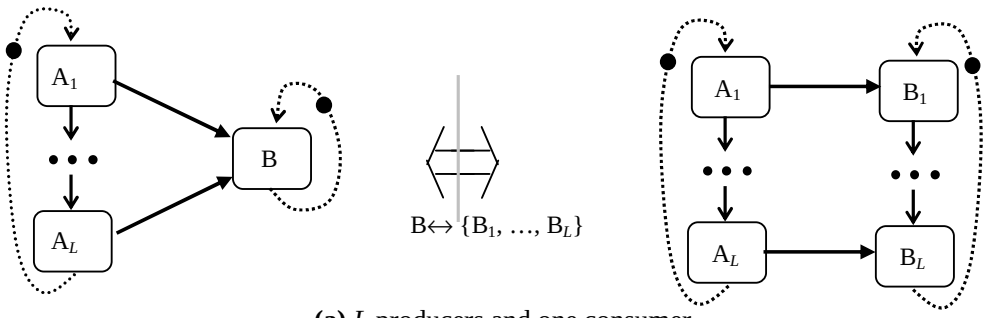
Definition. (Relationship between the initial buffer and its prototype buffer: equal token-size case). Consider an initial buffer with the number of transfers *H* and the total amount of the initial data tokens *m*. Suppose that all the transfers have an equal data token size.

(From the initial buffer to the prototype buffer) First, the initial buffer is transformed to the *unfolded prototype buffer*, for example the buffer in Figure 3.18(c) is transformed into the buffer in Figure 3.18(b). The purpose is to split the complex producers and consumers into a few simple ones, e.g. in Figure 3.18(c) producer D is split to obtain actors X′ [3] and X′ [4]. When splitting the complex producers/consumers, the set of so-called *groupings* of producers/consumers is recorded, where a grouping is the relation between an actor in the initial buffer and a set of corresponding actors in the unfolded prototype graph. The rules for splitting are introduced later in this definition (for a preview, see Figure 3.19). The obtained set of groupings can be used for the reverse transformation. At the second step, one transforms the unfolded prototype buffer model into the prototype buffer model by folding it back with the folding factor *H*.

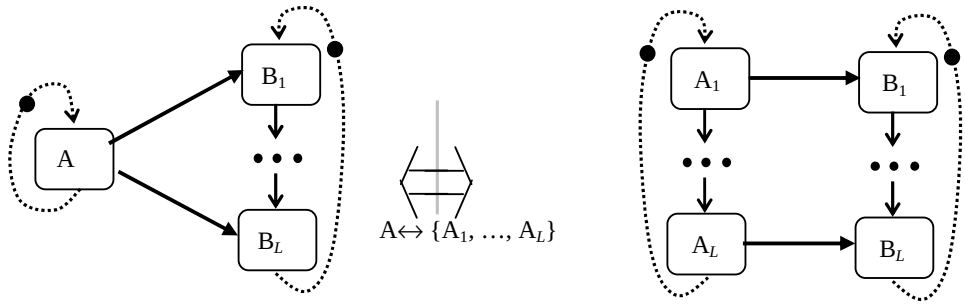
(From the prototype buffer to the initial buffer) First, the prototype buffer is unfolded with the unfolding factor *H*, hereby obtaining the unfolded prototype buffer. Then, one uses the set of groupings to transform the unfolded prototype buffer into the initial buffer.

The transformations from the initial buffer to the unfolded prototype buffer and back are illustrated by four patterns in Figure 3.19. Let us first consider the first two patterns, which consider the case where the data edges attached have equal number of initial tokens (zero in the figure, but one can add any number of tokens to every data edge).

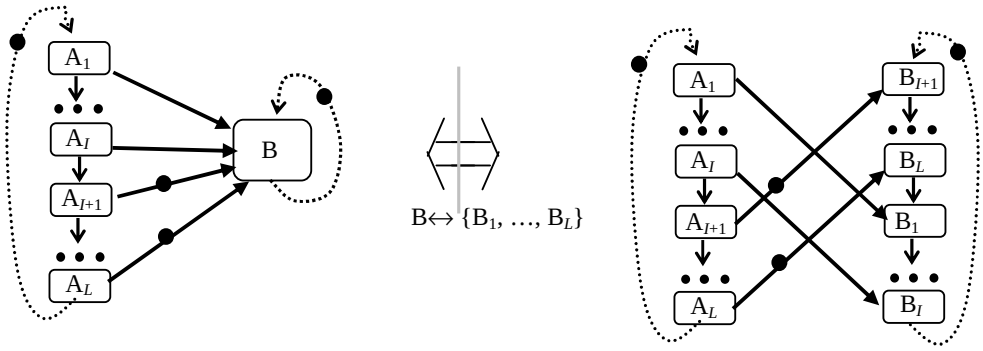
In the first pattern, in Figure 3.19(a), we see a situation where the initial buffer contains a complex consumer B (respectively, complex producer A in the second pattern in Figure 3.19(b)) with multiple incoming edges coming from a set of *L* producers { A_1, A_2, …, A_L } (resp. multiple outgoing edges to consumers in Figure 3.19(b)). In that case, in the unfolded prototype graph, consumer B (resp., producer A) is split into an ordered set of consumers { B_1, B_2, …, B_L } (resp., producers { A_1, A_2, …, A_L }), each consumer (resp., producer) being joined by an edge with a distinct producer (resp., consumer). The ordering of the new set of actors should be in line with the ordering at the other side of the buffer. Every such local transformation is recorded as an element in the set of groupings, where a *grouping* defines an actor and a set of corresponding split actors. For example, the grouping created by the transformation in Figure 3.19(a) is (B, { B_1, B_2, …, B_L }).



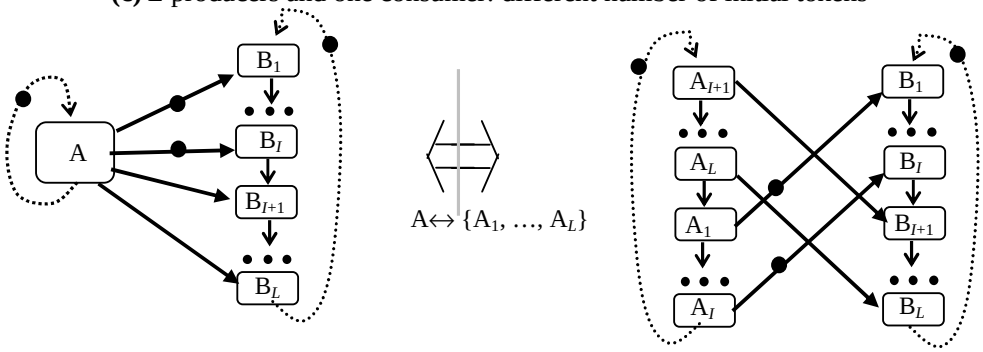
(a) *L* producers and one consumer



(b) *L* consumers and one producer



(c) *L* producers and one consumer: different number of initial tokens



(d) *L* consumers and one producer: different number of initial tokens

Figure 3.19 The relationship between initial and unfolded prototype buffer

Figures 3.19(c) and 3.19(d) are generalization of Figures 3.19(a) and 3.19(b), taking into account the fact that part of compatible data edges may have one extra initial token. In that case the splitting of complex producers and consumers should make sure that the data edges with different token sizes are compatible to each other.

If all groupings are available, the reverse transformation (from the split graph to the original graph) is possible, and done by replacement of each set of actors by one actor. For example, in Figure 3.18(b), two groupings are involved: (E, { $Y'[3]$, $Y'[4]$}) and (D, { $X'[3]$, $X'[4]$}).♦

Note that the definition above is easily extendable to the situation where there are not only edges going from the producers to the consumers (.e., from left to right in Figure 3.19) but also in the reverse direction. We need such edges to support not only infinite-capacity buffers but also for finite-capacity buffers, which we introduce later. To extend this definition, we can use the same patterns as in Figure 3.19, but for convenience, one has to flip every pattern horizontally, preserving the structure of the graph.

Now we are ready to start the discussion on finite-capacity buffers. As mentioned before, transforming the initial buffer to the prototype buffer gives us a convenient tool for doing that.

We are still considering the case where all token sizes in a complex buffer are the equal. In this case, it is meaningful to specify the capacity in terms of the number of tokens.

Definition. (Modeling the buffer capacity using the backward edges: equal token-size case)

Let b be the buffer capacity specified in the number of data tokens that can fit in the given buffer. Let m be the number of initial tokens in that buffer. Then, it is an obvious requirement that: $b \geq m$.

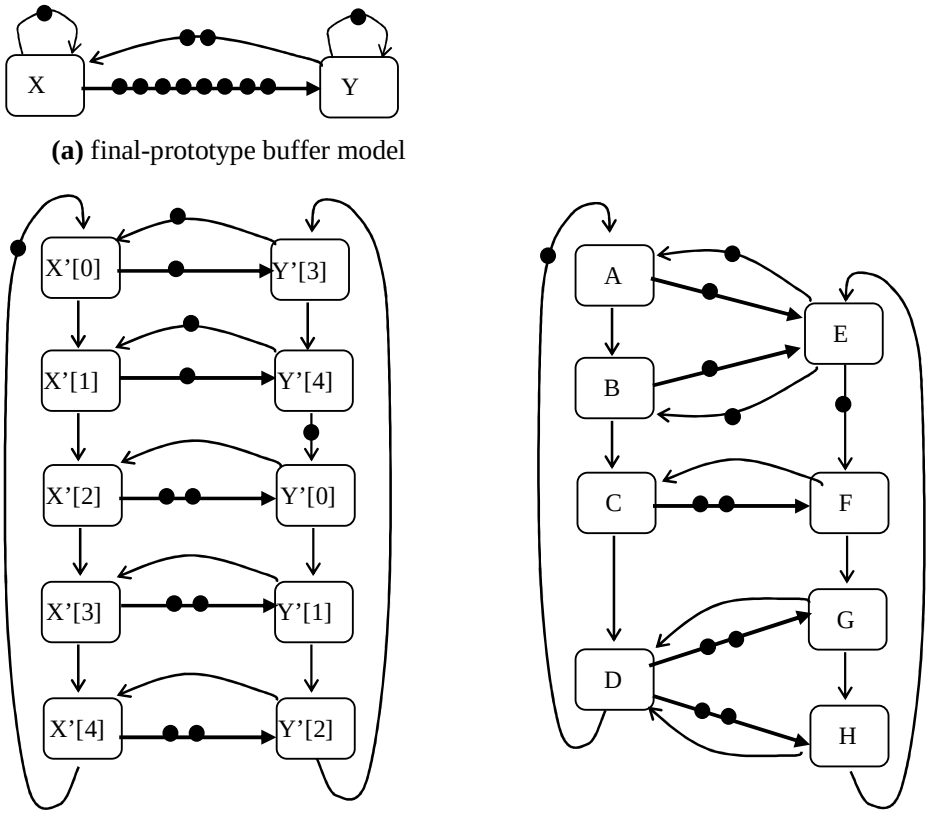
To model the limited capacity of a buffer, we first obtain the 'usual' (infinite-capacity) prototype graph of that buffer. Then, we introduce an extra edge to the prototype graph. That edge goes in the reverse direction: from the consumer to the producer; we call that edge a *backward edge*. It carries $(b-m)$ initial tokens.[24] After introducing the backward edge, the prototype buffer is called *final-prototype buffer*, because it acts as the prototype buffer model final buffer model; recall that the latter is the modified version of the initial model where the finite buffer capacity is taken into account.

Having obtained the final-prototype buffer, we translate it into the final buffer by first unfolding it and then by grouping the actors that need to be grouped; hereby we use the groupings that have been created in the beginning, when the initial was being translated into the prototype graph.

The final buffer model models the complex buffer with capacity b tokens. The edges that result from unfolding the backward edge are also called backward edges. They go from the consumers to the producers. To distinguish the default data edges of the buffer (i.e. the edges going from the producers to the consumers) we refer to them *forward edges*. ♦

For example, consider the complex buffer modeled in Figure 3.18(c). If we decide to limit the capacity of that buffer to $b = 10$ tokens, then, taking into account that $m = 8$, we have to introduce a backward edge with 2 initial tokens in the prototype buffer, as illustrated in

[24] Note that although this simple model only works under assumption that the tokens are consumed at the end of actor execution, it can be modified to support the case where the tokens are consumed in the beginning of actor execution, by adding another actor (with zero delay) into the consumer cycle of the prototype buffer



(a) final-prototype buffer model

(b) unfolded final-prototype buffer model

(c) final buffer model

Figure 3.20 Limiting the buffer capacity by introducing backward edges

Figure 3.20(a). After unfolding the final-prototype graph, we get 5 backward edges (see Figure 3.20(b)), which are inherited by the final buffer model (see Figure 3.20(c)).

Remark (The semantics of the backward edges: communicating the free slots for data tokens from the consumers to the producers). The tokens carried by the backward edges can be seen as free slots for the data tokens on the forward edges. At the start, there are $(b-m)$ free slots, so the backward edge of the prototype graph has that number of initial tokens. At each execution, the producer of the final-prototype buffer consumes one free slot from the backward edge and produces one token at the forward edge. If there are no free slots (the backward edge is empty), then, in line with the definition of actor behavior, the producer blocks until a token appears on the backward edge, which will signal that the consumer has released a free slot. In the context of the HSDF graph, the producer actor behaves in the same way with respect to all incoming edges; it does not 'know' that one of the incoming edges is a backward edge. Note that because the backward edges do not model the carrying of any real data, we choose to denote them as *sequence edges*, in contrast to the forward edges, which are data edges. This serves for better visual understanding of the graphs modeling the buffers.♦

So far, in this subsection we have studied the equal token-size case and answered the question on how to translate the initial buffer model, modeling an infinite-capacity buffer, to the final

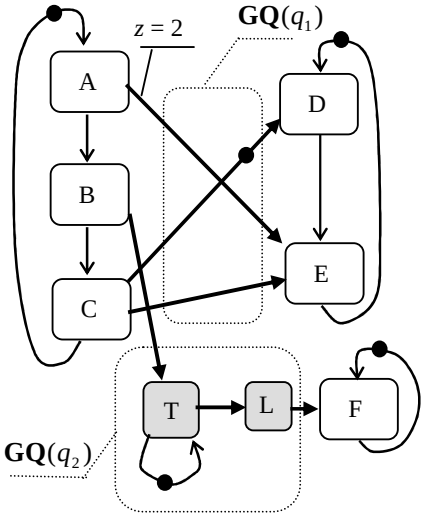
buffer model, representing a finite-capacity buffer, using the prototype graph as an intermediate point of translation. To complete the description on finite-capacity buffer modeling, we need to answer two more questions:

1) How to waive the requirement that the token sizes of the communication transfers are equal? We answer this question by using different rules to construct the initial buffer model for that kind of buffers.
2) How to re-include the final buffer model we have obtained after limiting the buffer capacity in the overall HSDF model of the application? Recall that we have extracted the initial buffer model from the context of the application's total HSDF.

Definition (Initial model buffer for infinite-capacity complex buffers: unequal token sizes) For the case with unequal token sizes, the initial buffer model is in fact not a graph in the usual meaning, but a *multigraph*, i.e., an extension of the notion 'graph' by a possibility to have more than one edge between two vertices. The set of vertices and the set of sequence edges of the extended original graph are defined the same way as for the 'usual' initial buffer model in the equal-token-size case. The set of data edges is, however, defined differently. Let g be the greatest common divisor of all token sizes in the channel. Then every communication transfer belonging to the given complex channel has a token size $h \cdot g$, where h is an integer. We represent the given transfer by h data edges joining the producer and the consumer. ♦

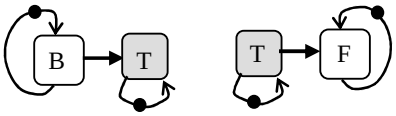
This way we reduce this case to a case where the token sizes of all data edges are equal. Now we can answer question 1) above by applying the same rules to the initial buffer model for this case as we have defined so far in this subsection. When we re-include the final buffer model into the application HSDF graph, we remove multiple edges between the same pair of actors, so that the application's HSDF graph remains to be a usual graph. Hereby, if different edges between a given pair of actors have different numbers of initial tokens, we remove the edges with more initial tokens. We do that because the edges with more initial tokens are less restrictive for the timing behavior of the graph.

Figure 3.21(a) gives an extended example of imposing finite capacity values to the buffers of the communication channels, including an unequal-capacity case. In our example, channel q_1 contains two transfers with a token size of one unit and one transfer with a token size of two units. Channel q_1 is local and thus it consists of one buffer, which is in this example a complex buffer. Figure 3.21(b), among other things, shows the initial buffer model for the buffer of channel q_1. We see that the transfer with the token size two is represented by two edges between the same pair of actors. In Figures 3.21(c) and 3.21(d), the initial buffer undergoes the transformations we defined in this section: splitting the complex producers/consumers and folding the graph with a factor equal to the number of data edges. Afterwards, we introduce the backward edge into the prototype buffer model. For the buffer of channel q_1, we arbitrarily assume a capacity of five units, and every initial token at the data edge and backward edge models one capacity unit. Finally, we unfold the final-prototype buffer and group the actors back. From the resulting modified extended graph, in Figure 3.21(e), we remove two superfluous replicas of the same edge, and then we re-include the modified graph back into the HSDF model in Figure 3.21(f).

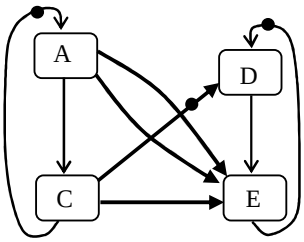


every data edge, except for one, assumes token size 1 unit

(a) An HSDF after channel merging

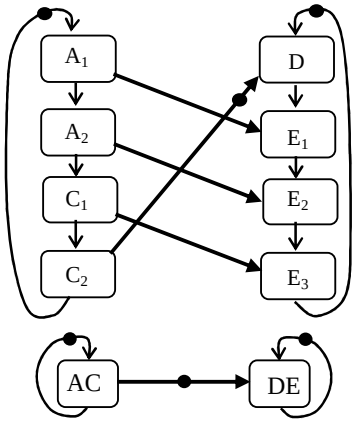


the initial buffers in q_2 (coincide with the prototype buffers)

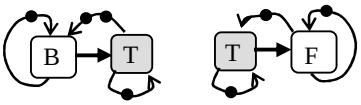


the initial buffer for q_1 (multigraph)

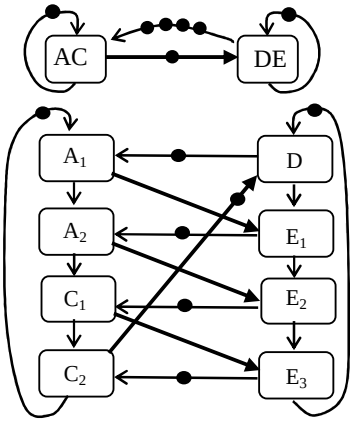
(b) Initial buffer models, obtained from the HSDF graph



(c) The unfolded prototype and the prototype buffers for q_1

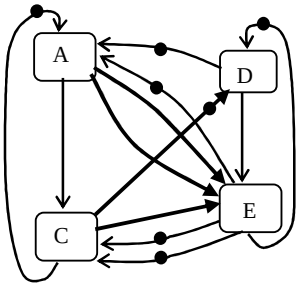


the final-prototype buffers for q_2 (capacity 2 and 1 unit)

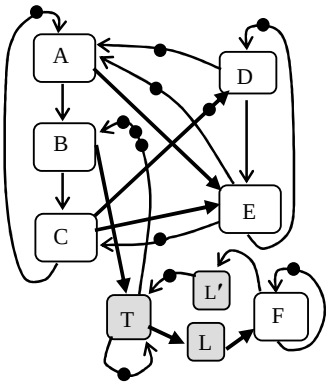


the final-prototype buffer for q_1 and its unfolded version (capacity 5 units)

(d) Introducing the backward edges



(e) The final buffer for q_2



(f) Final HSDF graph = IPC graph

Figure 3.21 Introducing finite buffer capacity modeling into the HSDF model

When answering question 2), the simplest case is the buffer of a local channel. In this case, we just import the backward edges of the modified original graph in the application HSDF graph. The same applies to the *producer* buffer of a network channel. Note that in that case the original graph sees the channel producers as producers and the transfer actors as consumers. In Figure 3.21, we demonstrate it using a simple network channel, with a simple producer buffer, being modeled by a data edge from actor B to actor T.

The only case of translation from infinite- to finite-capacity model that is left to be studied in this subsection is the case of the consumer buffer, joining a transfer cycle to the channel consumer cycle. Recall from Figure 3.17, that the data edges of that buffer are actually split into two parts by the latency actors. When forming the initial buffer model of a consumer buffer, we temporarily remove the latency actors; for example, see the original graph with actors T and F in Figure 3.21(b). Having obtained the initial buffer model, we perform the usual transformations to obtain the final buffer model – see e.g. the top-right part of Figure 3.21(d)). When including the final buffer into the HSDF model, we re-introduce the latency actors back by splitting the data edge again (e.g. see path T→ L→F in Figure 3.21(f)). It is important to note that we also need to split the backward edges and introduce actors that are analogous to the latency actors (e.g. see actor L′) – we call those *credit actors*, for the reason explained below. Hereby, if there are initial tokens at the backward edge, then we place the initial tokens at the edge closest to the transfer actor (e.g. edge (L′,T) in Figure 3.21(f)), as motivated below. Together with the transfer actors and the latency actors, the credit actors form the set of communication actors. They are all part of the channel macro $\mathbf{GQ}(q_j)$ for the network channel.

In fact, *the backward edges* of the consumer cycle *model the flow control mechanism* of the communication network because they carry the information on the number of free slots available from the network consumers to the transfer actors that push the data into the network at the producer side of the channel. The tokens carried by the backward edges are in fact the credits that propagate through the flow control connection. The credit actors have delay equal to the upper bound given in Equality (3.23). Initially, credit tokens are placed at the backward edge closest to the transfer actor because at start of the execution, the channel is ‘aware’ of how many free slots are available at the consumer side and thus this information does not need to be communicated through the network.

After importing the backward edges of all the channels, the application HSDF graph, in fact, reaches a final form, which we call the IPC graph. Recall that it is the main purpose of this chapter to explain how the IPC graphs for the network-on-chip are constructed, and at this point we have reached that goal. Recall that we define IPC graphs as HSDF graphs modeling the final implementation process network in the end of the flow, where all budgets are set to realistic finite values. From this, it follows that the IPC graphs are the HSDF graphs that model the finite capacities of the communication buffers and the finite bandwidths of the network channels. Thus, they justify their name – *inter-processor communication graphs*, or the graphs where the inter-processor communication is (conservatively) modeled with the highest accuracy that the given mapping methodology can deliver.

In the remainder of this chapter, we explain the usage of the IPC graphs for buffer capacity minimization – the last step of our preferred mapping flow – and we also mention some miscellaneous properties of the IPC graphs.

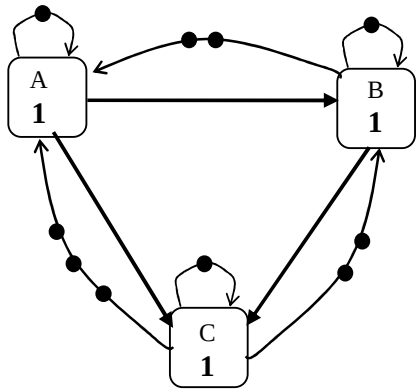


Figure 3.22 An IPC graph showing a feasible solution

3.6.4 Buffer Capacity Minimization Using IPC Graphs

In the previous subsection, we have seen that the capacities assigned to the buffers influence the initial marking of the backward edges. For the complex channels, they also influence which consumer-producer pairs are joined by the backward edges. For example, if in Figure 3.21(d), we increased the buffer capacity from 5 to 6, then the final-prototype graph would get one more initial token at the backward edge and when we unfold that graph the backward edges would be (D, A_2), (E_1, C_1), (E_2, C_2), and (E_3, A_1). Therefore, the buffer capacity assignment step has a direct impact on the structure and the number of initial tokens at the edges of the IPC graph, and thus in general it influences the MCM of the graph and the throughput of the loop of interest.

If a critical cycle of an IPC graph contains at least one backward edge, then a large enough increase in the capacity of the corresponding buffer will always eliminate this critical cycle, which can only either leave the maximum cycle mean unchanged or lead to a favorable reduction of that value (and thus to the favorable increase in the throughput).

Let us define a *buffer capacity vector* as a vector of the buffer capacities of all buffers, including $Q_{\text{prod-buffer}}, Q_{\text{cons-buffer}}$ of all network channels and Q_{buffer} of all local channels. A buffer capacity vector is called *feasible* if at least one of the conditions below is satisfied:

- the critical cycles of the corresponding IPC graph have cycle mean λ that does not exceed the required iteration interval: $\lambda \leq \Lambda_{\text{allowed}}$

OR:

- there is a critical cycle in the corresponding IPC graph that does not include any backward edge of any buffer.

A non-feasible IPC graph can be made feasible, by increasing the buffer capacities of the buffers contributing to the critical cycle until the maximum cycle mean goes below Λ_{allowed} or some critical cycles appear that cannot be changed by any buffer capacity increase. For a feasible solution as defined above, it holds that no increase in any buffer capacity can provide the necessary throughput improvement anymore.

The *buffer minimization problem* can be defined as a search for a feasible buffer capacity vector that minimizes the sum of the capacities of all buffers.

Thus, the IPC graphs proposed in this thesis enable defining the buffer minimization problem in the context of HSDF-modeled applications having simple or complex channels implemented either in the local memory or using an on-chip network with guaranteed bandwidth of point-to-point connections.

In [68] it is shown that the minimization of buffer capacities in HSDF graphs is an NP-complete problem, therefore we consider heuristic approaches. One can construct an interative heuristic using the optimization scheme we explained in Figure 1.8(b), with the analysis of IPC graph used to generate guidelines for iterative improvement of the solution candidate. As we show in this subsection, the guidelines can be generated based on the analysis of the critical cycles in the graph.

Figure 3.22 gives an IPC graph, modeling a feasible solution with minimum total capacity. It considers three single-actor processes communicating via three local channels with equal token sizes. The optimal buffer capacities are 2 tokens for the channel from A to B, 2 tokens for the channel from B to C and 3 tokens for the channel from A to C. We assume that, in this example, $\Lambda_{\text{allowed}} = 1$.

All cycles in that IPC graph have a cycle mean at most 1. Thus, the given solution satisfies the throughput constraint. Moreover, from the point of view of buffer capacity minimization it is a feasible solution, because there is at least one cycle, e.g. (A)* that has a maximum cycle mean of 1 and does not include backward edges. Two other such cycles are (B)* and (C)*. Thus improving the throughput of this solution would require an increase in the computation budgets of those three actors.

This solution uses minimum total capacity, because it has the property that removing one token from the capacity of any channel would lead to an increase of the MCM above the current value 1. For example, removing 1 initial token from edge (C, A) would leave only two initial tokens in cycle (C, A, B)*, leading to the cycle mean of 3/2 = 1,5, and removing 1 initial token from edge (B, A) would leave only one initial token in cycle (A, B)*, leading to the cycle mean of 2.

Now let us consider an example that uses a complex network channel, namely our JPEG decoding application example. The communication channel of that example has two buffers: the producer buffer and the consumer buffer. We determine a feasible buffer capacity vector by gradually increasing the capacities and changing the IPC graph respectively. This is repeated to the point when iteration interval λ does no longer change, where λ is computed analytically as the MCM of the IPC graph.

Table 3.2 illustrates the results of this exercise. The third column of the table shows, for comparison purposes, the iteration intervals Λ_N measured using a particular input data stream fed to a multiprocessor simulator modeling two processors communicating via a network channel. Λ_N is computed as Δ_N / N, where Δ_N is the time it took the multiprocessor to decode the image and N is the total number of tokens in the sequence. The results of the third column considerably differ from the results of the middle column because in that case the 'VLD' actor has variable processing times, as shown in Figure 3.5, and thus it has also variable execution delays. However, a mapping flow cannot make use of such measurements because they can only be made a posteriori at run time, whereas our preferred intra-application mapping flow needs a priori estimates at design time, made using the typical delay timing mode, in which all actors have constant delays.

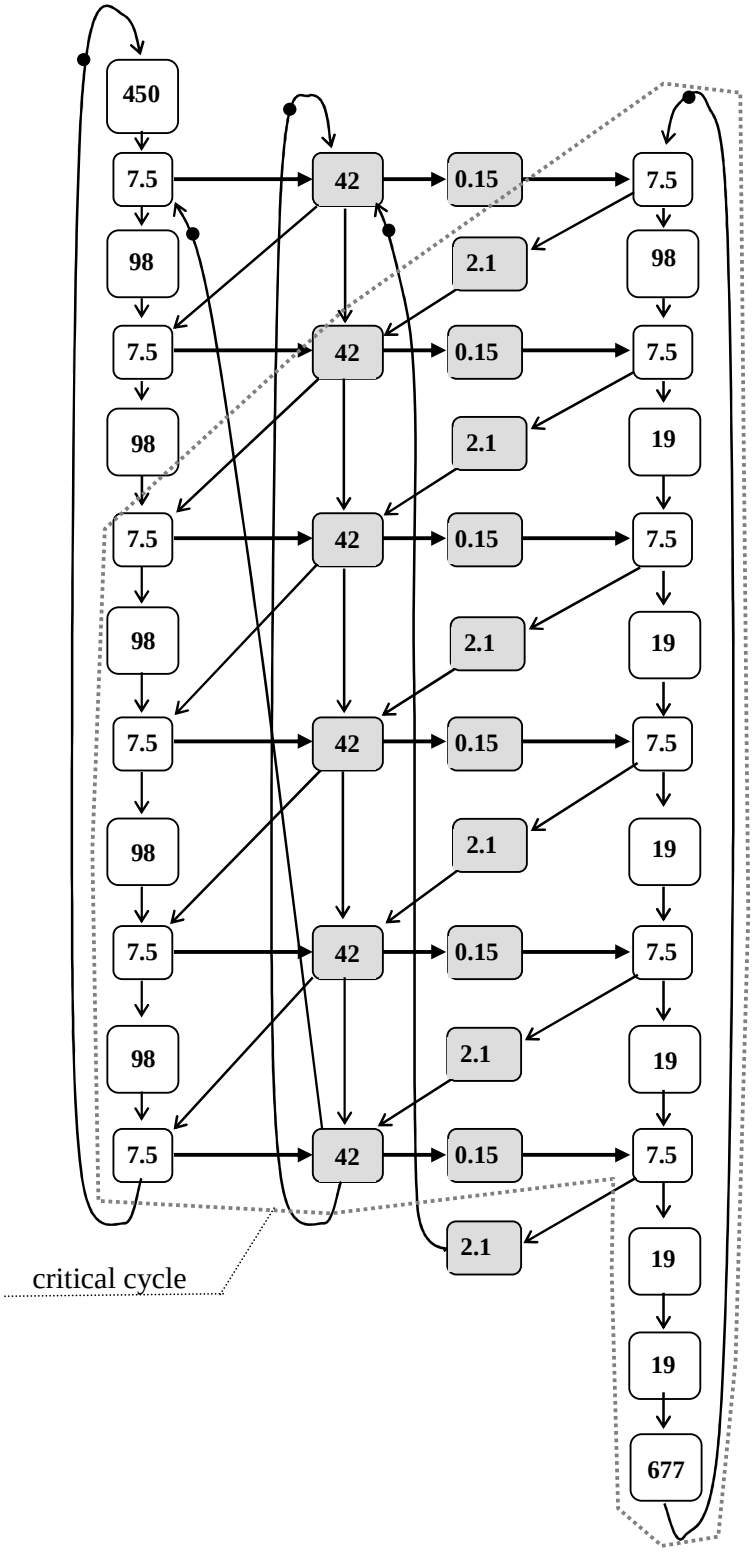


Figure 3.23 JPEG IPC Graph

Table 3.2 Arriving at a feasible buffer capacity vector

The capacity vector, $(Q_{\text{prod-buffer}}, Q_{\text{cons-buffer}})$	λ, μs	Λ_N, μs, measured for the 'PHILIPS logo' input sample
1,1	1140	1201
1,2	1035	1150
2,2	985	1116
2,3	no change	1093
3,3		1076
3,4		no change

From Table 3.2, we see that the typical-delay based predictions are more optimistic than the measured iteration intervals, i.e., $\lambda < \Lambda_N$. This is, in fact, not necessarily the case for any input data sequence, but it can happen. Thus, although the mapping flow assumes that the throughput constraint is met with buffer capacity vector (2, 2), in reality, for the given input data stream, the constraint is not satisfied: $\Lambda_N > \Lambda_{\text{allowed}}$, which is 1000 μs. Recall from the previous chapters that it is not possible to avoid throughput constraint violation unless one can afford to design for the worst case[25]. Recall also that a remedy for this is run-time adaptation. It is for the purpose to support the run-time adaptation that, in the later chapters, we develop the run-time performance analysis that is based on the IPC graphs with variable actor execution delay. Because our run-time adaptation approach gives guaranteed performance estimates, one can expect that for this input data sequence it would predict a throughput constraint violation and signal to the adaptation manager about the need to e.g. increase the processor clock frequency and/or scale down the quality of the output image[26].

Figure 3.23 shows the IPC graph that corresponds to the solution with capacity vector (1,1). Compared to Figure 3.13, we see that the transfer actors now reflect the channel number minimization decision (the actors are joined into a single transfer cycle) and the bandwidth minimization decision (the actors have delay 42 μs, as calculated earlier). We also see the backward edges and the credit actors. The credit actor delay 2.1 μs is calculated using Equality (3.23), where we again assume a maximum network path of 20 routers.

In Figure 3.23, we see that the critical cycle uses the backward edges of the channel and the actors of both processes. That critical cycle results in entry '1140 μs' in the table. When we increase the capacity of the consumer buffer and use the capacity vector (1,2), the destinations of the backward edges in the consumer buffer model in Figure 3.23 shift by one actor lower and as a result the critical cycle takes one '7.5' and one '98 ' actor less in the first process cycle, which explains the decrease of the iteration interval by roughly 105 μs – see entry '1035 μs' in the table.

[25] We tried to use the worst-case actor delay for the VLD actor and got such a large process cycle delay for process p_1 that the conclusion was that satifying the throughput constraint in the worst-case is only possible using much faster processors.

[26] We demonstrate this approach in Chapter 6, using a different application case study, where the actor delay variation is similar to this application

This example demonstrates the fact that our modeling technique for finite-capacity channels allows to obtain guidelines for improving a given solution when solving the buffer capacity minimization problem under throughput constraints. To make a decision on how to improve the current solution, one can calculate the critical cycles of the IPC graph – which can be done using polynomial algorithms ([19]). To improve the throughput, one has to touch every critical cycle by increasing the capacity of one of the buffers whose backward edge is contained in the cycle. These guidelines can be used to develop iterative buffer minimization algorithms.

In the related literature, [2] studies dataflow buffer capacity minimization, but does not take the throughput constraint into account. In [33], [86], [87], [97] we can find a few alternative approaches for solving this problem under the throughput constraint. All mentioned works consider the SDF model of computation, which is more general than HSDF (see Section 2.1.1); in fact the last two references consider cyclo-static data-flow, which is even more general than SDF. However, all these algorithms do not support buffer space sharing between several simple buffers, and thus they do not support complex channels. Trying to extend these algorithms to support complex channels or to develop a heuristic algorithm based on iterative breaking of the backward edges in the critical cycles are interesting subjects for future work.

3.7 The Properties of the Proposed IPC Graphs

3.7.1 Strong-connectedness, Liveness and the FIFO Property

In Section 2.2.4 we postulate the necessary requirements an HSDF graph must satisfy to be an IPC graph. Recall that our postulate says that a generic IPC graph:

1) is strongly-connected,

2) live, and

3) has the FIFO property.

Recall also that the IPC graphs of earlier related work, modeling bus-based architectures, can be shown to satisfy all those properties. In this subsection, we briefly show that also for the proposed IPC graphs, modeling network-based architectures, these properties hold as well for a quite general case.

Let us first assume that the original computation graph fed to our mapping flow in the beginning is connected. Then the implementation process network is connected as well and stays connected during the whole mapping flow. From the connectedness of the implementation process network the strong-connectedness of the IPC graph follows, because in an IPC graph every process macro is a cycle and every channel macro contains not only the data edges, which go in the same direction as the channel, but also the backward edges, which go in the opposite direction.

Only if the original graph is not connected, but consists of several disconnected subgraphs, may the IPC graph ever appear to be a disconnected graph. Nevertheless, in this case, due to the reasons that we have just discussed, every connected component of the IPC graph will by itself be a strongly-connected graph that can be considered independently from the other components as a separate IPC graph.

The liveness property means that any actor in the HSDF graph can eventually always fire again and implies the absence of deadlock, i.e. the absence of a situation where no actor will

eventually fire again. For HSDF graphs, liveness is equivalent to the requirement that every cyclic path contains at least one initial token, according to a well-known property of so-called event graphs [4 - §1]. In our implementation trajectory, the intra-application mapping flow makes sure that that the MCM of the HSDF graph is not larger than Λ_{allowed}, and this implies that every IPC graph cycle should contain at least one initial token; otherwise the average iteration interval would be infinite.

Recall that Lemma 2.1 in Section 2.2.4 states that an HSDF graph possesses the FIFO property if all actors having dynamic execution delays are enclosed within cycles having only one initial token. In our IPC graphs, only computation actors can have dynamic delays, but they are all enclosed into the process cycles. In fact, the only actors in our IPC graphs that are not enclosed into such a cycle are the actors modeling the network latency. Those actors have static delays even in the timing modes where the computation actor delays vary. The static delay of those actors models the fact that the data tokens cannot overtake each other as they propagate via a network connection.

3.7.2 An Upper Bound on the Number of Initial Tokens in any Cycle

In this subsection, we give an upper bound $\hat{\mu}_{\text{C}}$ on the number of initial tokens of any simple cycle (i.e. the cycle depth) of an IPC graph generated by our preferred mapping flow. We need this result in Chapter 4 to bound the algorithmic complexity which depends on the initial token count.

The upper bound proposed in this subsection is such that if all cycles contain at least $\hat{\mu}_{\text{C}}$ tokens, no increase of the capacity of any buffer can lead to throughput improvements. This bound helps to limit the maximum 'reasonable' number of initial tokens on the backward edges, and it is relevant only for *mixed cycles*, i.e., cycles containing the computation actors of different processes or computation actors and communication actors. Note that this definition includes the artificial cycles, as defined in Section 3.5.5. All the other cycles have only one initial token.

Let **C** be an arbitrary simple mixed cycle in IPC graph **G**. Let the depth of cycle **C** be $\hat{\mu}_{\text{C}}$. The main question we have to answer now is how to ensure that cycle **C** is not critical by setting $\hat{\mu}_{\text{C}}$ high enough. Let us represent the cycle mean of **C** as:

$$CM(\mathbf{C}) = \frac{\sum_i a_i + \sum_i b_i}{\hat{\mu}_{\text{C}}}, \tag{3.29}$$

where a_i are the typical delays of all computation actors and transfer actors in **C**, and b_i are the typical delays of all latency and credit actors in **C**.

Before we proceed in finding an appropriate $\hat{\mu}_{\text{C}}$, let us introduce a few notations. Let $\hat{a}$ and $\hat{b}$ be the maximum computation/transfer and latency/credit actor delay in graph **G** respectively. Also, let $V_{\text{C}a}$ and $V_{\text{C}b}$ be the number of process/transfer and latency/credit actors in **C**.

Because any computation and transfer actor belongs to a cycle with only one initial token, we have $MCM(\mathbf{G}) \geq \hat{a}$. Therefore, we can ensure that **C** is not critical by the following requirement:

$$\sum_i a_i + \sum_i b_i \leq \hat{\mu}_{\text{C}} \cdot \hat{a}, \tag{3.30}$$

which, in turn, can be ensured by:

$$V_{Ca} \cdot \hat{a} + V_{Cb} \cdot \hat{b} \le \hat{\mu}_C \cdot \hat{a}\,, \tag{3.31}$$

Now we can distinguish two major cases.

In case $\hat{b} < \hat{a}$, we can obtain a valid inequality by replacing $\hat{b}$ by $\hat{a}$ in Formula (3.31). This yields the following requirement: $\hat{\mu}_C \ge V_{Ca} + V_{Cb}$, i.e. $\hat{\mu}_C \ge V_C$, where V_C is the total number of actors in cycle **C**.

In case $\hat{b} \ge \hat{a}$, we can make use of the observation that at most half of the actors in cycle **C** are latency or credit actors because every latency or credit actor is joined to a process and a transfer actor. Therefore, we can replace V_{Cb} and V_{Ca} by $V_C/2$ in Formula (3.31). Hereby, we get:

$$\hat{b} \ge \hat{a} \Rightarrow \hat{\mu}_C \ge \frac{V_C}{2} \cdot \left(1 + \hat{b}/\hat{a}\right) \tag{3.32}$$

In both cases $\hat{\mu}_C$ depends on V_C. Now, to obtain an upper bound that does not depend on the number of actors in the cycle, we can use the observation that any simple cycle contains at most V actors, where V is the total number of actors in the graph. Combining the two major cases together, replacing V_C by V, and replacing inequality by equality we get the final expression for the upper bound on the number of tokens:

$$\hat{\mu}_C = \frac{V}{2} \cdot \left(1 + \max(1, \hat{b}/\hat{a})\right) \tag{3.33}$$

This formula can be motivated as follows. If the communication network latency is considerably higher than the computation delay, we need larger buffers in order to 'hide' the network latency and thus more initial tokens are introduced in the IPC graphs. If, however, the network latency is very small, only the computation and transfer actors influence the throughput. Then we need to put at most one token per one such actor in the mixed cycle to avoid that that cycle is critical.

Note that Formula (3.33) only limits the depth of the cycles where the mapping flow introduces at least one initial token. It does not limit the number of tokens that are inherited by the IPC graph from the computation graph. However, we would consider it as an extremely unlikely situation where computation graphs would be heavily 'saturated' by initial tokens, such that there are more initial tokens than actors in the graph. Only in that case could they make our upper bound invalid. Assuming that such untypical and unpractical saturation do not take place, we conclude that Formula (3.33) holds for a quite general case.

3.8 Notes

We conclude Chapter 3 by highlighting and summarizing the achievements of this chapter and mentioning some important related work and the sources of inspiration without which this chapter would never have been possible.

We place the main novelty claim of this chapter in the treatment of network communication channels. The novel parts of this treatment are summarized below:

1) Models for complex finite-capacity buffers, based on HSDF graph unfolding;

2) Models for network-on-chip (NoC) channels, including the transfer, latency and credit actors, the transfer cycle and the bounded-capacity buffer models mentioned in 1).

These contributions were first presented in our paper [75]. The value of those contributions is that they enable guaranteed throughput analysis for the streaming applications running on NoCs, even when the network connections are shared by multiple primitive channels of the application that carry data tokens of different type. Contribution 1) is essential for minimizing the amount of communication buffer memory, which is an important cost factor for systems-on-chip (SoC). Contribution 2) enables extension of the buffer minimization problem formulation to the realm of NoCs. Our contributions are described in Section 3.7 and the supported hardware architectures are described in Section 3.4. Section 3.5 puts our contributions in the overall context of multiprocessor mapping techniques.

An important part of this chapter is the following relatively new idea. For the data-dependent HSDF/SDF applications, it is useful to aid the mapping flow with an application preparation part, which precedes the mapping flow and detects the actor-level parameters and their coefficients for linear parameter functions modeling the actor delays. That idea is not our contribution, but it comes due to the work of Milan Pastrnak, [72], with whom we worked in a close cooperation.

In this chapter, the application preparation part is systematically described in Sections 3.1 through 3.3. Those sections contribute to the previous related work by an original discussion of linear parameter functions and by highlighting the less commonly known possibilities in this field – the use of confidence intervals in the linear regression approach to generate linear functions that are conservative from a probabilistic point of view.

Note that, for this thesis, the main importance of the application preparation part lies not in determining the typical actor delays for the mapping flow but in our run-time performance prediction method for dynamic-delay HSDF graphs. We describe that method in Chapter 5 after building the necessary basis for that in Chapter 4.

The first idea for the use of backward edges for modeling buffer capacities is coming from the discussion on strictly bounded Kahn process networks discussed in the PhD thesis of Thomas Parks, [73]. The formula for the delay in TDMA scheduling has been adopted from the Master Thesis of Rob Hoes [38] and [8]. The examples in Figure 3.10 and 3.22, explaining the peculiarities of the different steps in the mapping flow have been borrowed from the discussions in the 'HIJDRA' research project for soft-real-time multiprocessor streaming applications carried out in Philips Research Laboratories Eindhoven (nowadays NXP Semiconductors).

In his Master Thesis [60] and in [63], Arno Moonen introduces more elaborate dataflow models for the network channels than our channel macros. His models are based on the general SDF model of computation, and they include more powerful and less pessimistic models of network scheduling of data packets, using transfer cycles with multiple initial tokens. However, those models reflect the events in the network channels at fine-grain hardware-specific level of granularity, i.e., at the level of network data words. On the contrary, our models work at the application-specific level of granularity, which allows us to model a block of data communicated through the network channels as one single token. Therefore, when the application uses data blocks, our models are simpler and performance analysis complexity is reduced. This is particularly favorable for doing the performance analysis at run time for variable actor delays.

The work of Arno Moonen, as well as our own, provides a way to represent the FIFO memory buffers of the network channels using (H)SDF graphs. As already mentioned, provided an

(H)SDF representation, several works propose the algorithms to minimize the required buffer capacity, e.g. [2], [33], [86], [87], [97], but none of them can handle complex FIFO buffers, i.e. buffers shared by multiple producer-consumer actor pairs. For such cases, our modeling approach enables finding bottlenecks in a problem solution and thus can be potentially used in iterative-improvement optimization algorithms.

Last but not the least, M. Coenen *et al* [15] propose a FIFO buffer capacity calculation method for TDMA-scheduled on-chip networks. Using the details of the behavior of such networks, it has a potential to produce more optimal buffer capacity allocation than dataflow-oriented network models, which are based on a more abstract view of the scheduling. However, this work would not fit well into our design methodology for two reasons. First, it requires allocating network hardware resources at design time, whereas, in order to provide enough flexibility for dynamic run-time combinations of streaming applications it is better to postpone allocation of physical resources to run time. Second, it assumes that the producers and the consumers are ‘well-behaved’, in the sense that they communicate data at a constant rate. This assumption works well e.g. for video processing hardware blocks, which produce and consume video data samples following a (multi-)periodic pattern. However, the software tasks in applications like video/audio de-coding do not necessarily satisfy this assumption.

4

4 Analysis of Static-delay HSDF Graphs

In the previous chapter we have described how one can create an HSDF timing model of an application executable running on a multiprocessor SoC with network-on-chip communication that supports predictable timing. An important advantage of this model is that it combines both the computation and communication using the same basic primitives, namely the actors, the edges and the initial tokens. This opens up a possibility to ignore in this chapter the details on what is being modeled and rather to focus on the model itself.

In this chapter, we consider the fundamentals of static-delay HSDF graphs, in order to establish the facts that we use in the performance analysis approach proposed in this thesis. Hereby, the major analysis goal is to obtain tight and conservative bounds on the performance of the multiprocessor system being modeled by the graph.

In this and the following chapters, we use the term *algorithmic rule* to refer to all the algorithms contributing to the performance analysis. The outcome of this chapter is the algorithmic rule that calculates the conservative bounds on the performance of the static-delay graph. We refer to it as the *major algorithmic rule for static-delay analysis*. The major rule consists of the smaller rules that are established throughout this chapter.

In the whole chapter, we assume that the HSDF graph in question satisfies the basic IPC graph properties. Recall that those properties are:

- strong-connectedness,
- liveness, and
- FIFO property.

Given those properties, we focus on the theoretical results that lead to the *main theorem* on the periodic timing behavior of an HSDF graph. That theorem serves as a foundation for the major algorithmic rule for static-delay analysis. A very convenient mathematical apparatus for expressing the main theorem and the supporting lemmas is *max-plus algebra*. The theoretical results of max-plus algebra are studied in much detail in [4]. However this book studies a different model of computation, namely *event graphs*.

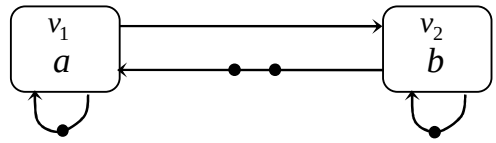
Our major algorithmic rule re-uses certain event-graph results such as max-plus algebra, certain algorithms and the main theorem. Therefore, to be able to introduce the major algorithmic rule unambiguously in the second part of this chapter, in the first part of this chapter – Sections 4.1 - 4.3 – we extend the results of [4] by building a thorough connection between HSDF graphs on one side and event graphs on the other side. This connection is not new, but, to the best of our knowledge, never worked out in enough detail in the literature. Once the connection is clear, for consistency, we also reintroduce the main theorem.

In the second part of this chapter – Section 4.4 – we use the main theorem to introduce a new characteristic of the HSDF graph called 'lateness' – denoted 'σ' – which is important for our performance analysis approach. Having introduced the main-theorem, the event-graph results and 'σ', we are ready to present the major algorithmic rule, which is also done in the same section. Section 4.5 summarizes this chapter and mentions some related literature.

Note that so-called *state-space exploration* is an approach to static-delay HSDF performance analysis that is alternative to max-plus algebra and event graphs. As shown by A.H. Ghamarian *et al* in [23], using that approach, the main theorem can be proven for more general graphs – i.e. SDF graphs (i.e. multirate dataflow graphs) and, for such graphs the performance analysis can be done more efficiently than using a translation from SDF to event graphs. However, it is not yet clear whether this advantage is still present when this technique is applied to HSDF graphs. Although the analysis techniques of [23] are focused on the calculation of throughput, they have some similarities with a technique we propose in this chapter to calculate lateness. We discuss these similarities in Section 4.5. In this chapter, we prefer using event graphs over state spaces, because, as already mentioned, for the event graphs efficient performance analysis algorithms are known to us, making a working set to construct the major algorithmic rule. Construction of similar algorithms using state space exploration, in order to provide better support for general SDF graphs is a subject for future work.

4.1 HSDF Graphs and Max-plus Algebra

In this section, we build a relation between HSDF graphs and max-plus algebra. First, using max-plus algebra, we give an overview of the basic steps towards the main theorem. Then, we make the first step in that direction by expressing the timing behavior of the HSDF actors using the max-plus algebra notations.



a, b are delays of actors v_1 and v_2

(In this chapter, we do not distinguish between communication and computation actors and between the sequence and data edges.)

Figure 4.1 'Producer-consumer' HSDF graph

4.1.1 Overview of the Steps towards the Main Theorem

Our starting point is a formal description of an HSDF graph's timing behavior. Suppose we have an HSDF graph **G**(**V**, **E**, *m*) as defined in Chapter 2, where **V** is the set of actors, **E** is the set of edges and *m* – the edge marking – gives the number of initial tokens on each edge.

Suppose that graph **G** satisfies the basic properties mentioned in the introduction to this chapter. Recall from Chapter 2 that the timing of the HSDF graphs possessing the FIFO property can be described using variables $x_k(n)$, which, for each actor v_k, give the completion time of the execution with index *n*, $n \geq 0$. We also refer to $x_k(n)$ as the time of the *completion event* of actor v_k in iteration *n*. For $n < 0$, we assume, for convenience reasons, that all variables $x_k(n)$ take value 0.

Recall also that, to define the timing behavior of an HSDF graph, recurrent equations are constructed, relating the future completion events to the past events. In Chapter 2, Lemma 2.2, we introduced those equations as the *evolution equations*.

Consider, for example HSDF graph **G** in Figure 4.1. Using Lemma 2.2, we can write the following evolution equations for this graph:

$$\begin{aligned} x_1(n) &= a + \max(x_1(n-1), x_2(n-2)) \\ x_2(n) &= b + \max(x_1(n), x_2(n-1)) \end{aligned} \tag{4.1}$$

The right-hand side of each equation is the sum of the actor delay and a 'max' operation. The 'max' operation, in effect, gives the earliest moment when the actor can capture the input tokens and start the execution in iteration *n*. The 'max' has one entry per incoming edge of the actor. The entry gives the time of the completion event that produces the required input token. The index of this event equals *n* minus the number of initial tokens on the corresponding edge.

In max-plus algebra, one can rewrite these equations as follows:

$$\begin{aligned} x_1(n) &= a \otimes (x_1(n-1) \oplus x_2(n-2)) = a \otimes x_1(n-1) \oplus a \otimes x_2(n-2) \\ x_2(n) &= b \otimes (x_1(n) \oplus x_2(n-1)) = b \otimes x_1(n) \oplus b \otimes x_2(n-1) \end{aligned} \tag{4.2}$$

where in the middle part of Equality (4.2) we have replaced 'max' by '$\oplus$' and '+' by '$\otimes$'.

Operators '$\oplus$' and '$\otimes$' are the scalar 'addition' and 'multiplication' operations in terms of max-plus algebra. Just as for the 'normal' addition and multiplication, the distributive law applies to them: $u \otimes (v \oplus t) = u \otimes v \oplus u \otimes t$. We apply the distributive law to the middle part of Equality (4.2) to obtain the right-hand part.

Just as the 'normal' algebra, both the 'addition' and the 'multiplication' in max-plus algebra have one neutral element, or an element that, when combined with any other element in

'addition' or 'multiplication' does not change that element. Thus, in the 'normal' algebra, the neutral element for addition is 0, 'zero', and the neutral element for the multiplication is 1, or 'unit'. In max-plus algebra, we also use the names 'zero' and 'unit element', but they have a different numeric value.

'Zero' in max-plus algebra is denoted 'ε' and its numeric value is $-\infty$. We see that for this 'zero' element and the '$\oplus$' the neutral-element property holds: $u \oplus \varepsilon = \max(u,-\infty) = u$.

Similarly, the unit element in max-plus algebra is denoted 'e'. By definition $e = 0$. Thus, $u \otimes e = u + 0 = u$.

The scalar Equalities (4.2) can be re-written in matrix form:

$$\begin{bmatrix} x_1(n) \\ x_2(n) \end{bmatrix} = \begin{bmatrix} \varepsilon & \varepsilon \\ b & \varepsilon \end{bmatrix} \cdot \begin{bmatrix} x_1(n) \\ x_2(n) \end{bmatrix} + \begin{bmatrix} a & \varepsilon \\ \varepsilon & b \end{bmatrix} \cdot \begin{bmatrix} x_1(n-1) \\ x_2(n-1) \end{bmatrix} + \begin{bmatrix} \varepsilon & a \\ \varepsilon & \varepsilon \end{bmatrix} \cdot \begin{bmatrix} x_1(n-2) \\ x_2(n-2) \end{bmatrix}, \quad (4.3)$$

where 'normal' matrix multiplication '$\cdot$' rules apply, i.e. matrix vector-rows are multiplied with vector-columns. However the role of *scalar* operations '$\cdot$' and '+' are played by max-plus operations '$\otimes$'and '$\oplus$'. The reader can verify that by applying those rules to (4.3) we arrive at the equalities in (4.2).

In this chapter we use the following mathematical notations. All algebraic expressions involving *matrices* (including *vectors*, seen as special cases of matrices) are max-plus algebra expressions. Thus, a matrix product, although denoted '$\cdot$', involves a linear combination of matrix elements using '$\otimes$'and '$\oplus$'. A matrix addition, although denoted '+', means element-by-element application of '$\oplus$' (maximization); for example '+' in Equality (4.3) refers to matrix element-by-element maximization.

To express the periodic behavior of HSDF graphs, also scalar-by-matrix product, '$\cdot$', is used in this chapter, and it means element-by-element application of '$\otimes$' (addition). If, in a scalar-by-matrix product, the scalar is in some power N, then the 'power' means N times application of '$\otimes$', being equivalent to 'normal' multiplication by N. Thus, in expression '$a^N \cdot \mathbf{A}$' where '$\mathbf{A}$' is a matrix, 'a^N' corresponds to $a \otimes a \otimes \ldots \otimes a$ (N times) and refers to 'normal' algebra's: '$N \cdot a$'. Also, the division of a matrix by a scalar, e.g. '$\mathbf{A}/a$', should be interpreted differently, namely as element-wise subtraction. At the end of this subsection, we summarize our max-plus notations in the form of a table.

The main rationale of using max-plus algebra in our context is that one can interpret the iterations of the HSDF graph as applications of the matrix multiplication to the vector of completion times of the HSDF actors. As shown in [4], many matrix multiplication properties of the 'normal' linear algebra have analogies in max-plus algebra. Therefore, one can re-apply the powerful linear-algebra apparatus to explore the properties of the HSDF graphs, which appear to be 'linear' systems in the context of max-plus algebra.

In a compact form, Equality (4.3) can be rewritten as follows:

$$\mathbf{x}(n) = \mathbf{A_0} \cdot \mathbf{x}(n) + \mathbf{A_1} \cdot \mathbf{x}(n-1) + \mathbf{A}_2 \cdot \mathbf{x}(n-2) \quad (4.4)$$

where the $\mathbf{A}_i$ are above matrices and the $\mathbf{x}(n-m)$ are vectors of variables $x_k(n-m)$.

The main theorem, which we develop in the first part of this chapter, requires that the evolution equations should be represented in *canonic form*:

$$\mathbf{x}'(n) = \mathbf{B} \cdot \mathbf{x}'(n-1) \quad (4.5)$$

where $\mathbf{x}'(n)$ is an vector of variables which is, in general, different from the original vector of variables, $\mathbf{x}(n)$. The translation into canonic form is always possible, and one can compute the new vector from the original one and vice versa. The basic idea is that any variable occurrences of the form $x(n-k)$ for $k > 1$ in the evolution equations (Equality (4.2)) are replaced by a new variable.

For example, in Equality (4.2), we do the following variable replacement:

$$x_1'(n) = x_1(n), \qquad x_2'(n) = x_2(n), \qquad x_3'(n) = x_2(n-1)$$

Note that these replacements are defined for all integer values of n, e.g. one can derive that $x'_3(n) = x'_2(n-1) = x'_2(n-1)$. Using the variable replacement and substituting the first equality into the second one in (4.2), the following equivalent canonic-form system of can be obtained:

$$\mathbf{B} = \begin{bmatrix} a & \varepsilon & a \\ a \otimes b & b & a \otimes b \\ \varepsilon & e & \varepsilon \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} x_1'(n) \\ x_2'(n) \\ x_3'(n) \end{bmatrix} = \mathbf{B} \cdot \begin{bmatrix} x_1'(n-1) \\ x_2'(n-1) \\ x_3'(n-1) \end{bmatrix} \tag{4.6}$$

We have:

$$\mathbf{x}'(n) = \mathbf{B}^{n+1} \cdot \mathbf{x}'(-1) \tag{4.7}$$

$\mathbf{B}$ is called *the canonic matrix* of graph $\mathbf{G}$. Thus, if the canonic matrix and the initial state, $\mathbf{x}'(-1)$, are known then one can compute the completion times of any actor in any iteration n. Recall that under our assumptions, the initial state is such that all elements of $\mathbf{x}'(-1)$ are equal to 0, and this holds for any negative iteration index n.

For a certain class of HSDF graphs, the main theorem for static actor delay executions implies that $\mathbf{B}^n = \lambda \cdot \mathbf{B}^{n-1}$, where scalar λ is a max-plus eigenvalue of $\mathbf{B}$ and where n is large enough. In the context of max-plus algebra, the definition of the matrix eigenvalue is different from the eigenvalue definition of the 'normal' algebra and will be given later. From $\mathbf{B}^n = \lambda \cdot \mathbf{B}^{n-1}$, it follows that $\mathbf{x}(n) = \lambda \cdot \mathbf{x}(n-1)$, which is equivalent to $x_k(n) = \lambda \otimes x_k(n-1)$ (for all actors v_k). Because 'multiplication' $\otimes$ in max-plus algebra means 'normal' addition, we conclude that, for the class of HSDF graphs referred to above, the actor completion times are strictly periodic with period λ: $x_k(n) = \lambda + x_k(n-1)$.

As we see later, for *all* static-delay HSDF graphs possessing the basic IPC graph properties, a generalized version of expression $\mathbf{B}^n = \lambda \cdot \mathbf{B}^{n-1}$ applies, namely: $\mathbf{B}^n = \lambda^W \cdot \mathbf{B}^{n-W}$, where W is an integer and n is large enough. This again means that the graph's behavior is strictly periodic, whereby the period spans W iterations and has duration $\lambda \cdot W$ (recall that that is how we should interpret 'λ^W' in all expressions where matrices are involved).

If we divide the period, i.e. $\lambda \cdot W$, by the number of iterations in the period, i.e. W, we get the average iteration interval 'λ'. Thus we see that 'λ', introduced here as a max-plus matrix eigenvalue, has the same meaning as in the previous chapters. In accordance to what we stated in the previous chapters, the main theorem, studied in this chapter, states that the max-plus matrix eigenvalue is computed as the maximum cycle mean of the HSDF graph. Therefore, the *graph cycles*, i.e. the cyclic paths in the graph, determine the graph's long-run steady-state timing behavior, which is, in fact, strictly periodic. Therefore, the graph cycles play a central role in this chapter.

The summary of the max-plus algebra notations used in this chapter is given in Table 4.1.

Table 4.1 A summary of max-plus algebra notations

Notation contents	Example	Example meaning in 'normal' algebra
Scalar operation '⊕'	$a \oplus b$	$\max(a,b)$
Scalar operation '⊗'	$a \otimes b$	$a+b$
Max-plus 'unit' element	e	0
Max-plus 'zero' element	ε	$-\infty$
Scalar-by-matrix multiplication	$\lambda \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix}$	$\begin{bmatrix} \lambda+a \\ \lambda+b \\ \lambda+c \end{bmatrix}$
Scalar in power N by matrix multiplication	$\lambda^N \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix}$	$\begin{bmatrix} N\cdot\lambda+a & N\cdot\lambda+b \\ N\cdot\lambda+c & N\cdot\lambda+d \end{bmatrix}$
Matrix division by a scalar	$\begin{bmatrix} a \\ b \end{bmatrix} / c$	$\begin{bmatrix} a-c \\ b-c \end{bmatrix}$
Multiplication of two matrices	$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} f & g \\ h & i \end{bmatrix}$	$\begin{bmatrix} \max(a+f,b+h) & \max(a+g,b+i) \\ \max(c+f,d+h) & \max(c+g,d+i) \end{bmatrix}$
Scalar expression with 'normal' operations	$a \cdot b + c \cdot d$	$a \cdot b + c \cdot d$

Table 4.2 Table of variable contributions to the evolution equations, initially containing only default values (max-plus 'zero' element)

		column 1	…	column j	…	column V
row 1	$x_1(n)$ =	ε	…	ε	…	ε
…	…	…	…	…	…	…
row i	$x_i(n)$ =	ε	…	ε	…	ε
…	…	…	…	…	…	…
row V	$x_V(n)$ =	ε	…	ε	…	ε

4.1.2 Evolution Equations of an HSDF Graph in Max-plus Algebra

In this section, we describe how to obtain the matrix form of the HSDF evolution equations of graph **G**, e.g. Equality (4.3), in general.

Let us consider a square table, with V rows and columns, where $V = |\mathbf{V}|$ is the number of actors in graph **G**. We define that table as follows. Every row 'i' of the table corresponds to the

evolution equation for variable $x_i(n)$. Every column *j* corresponds to the *contribution* by variables $x_j(n-m)$, $m = 0, 1, 2, ...$ to the evolution equations. Note that at most one of those variables can be a contributor to any given evolution equation, because there can be at most one edge going from actor v_j to actor v_i.

Let us first initialize all entries of the table with ε, as shown in Table 4.2.

Before we update the table, let us answer the question how the $x_j(n-m)$ variables contribute to the evolution equation of variable $x_i(n)$. An evolution equation, as given by Lemma 2.2, expresses variable $x_i(n)$ as the maximum of a subset of variables $x_j(n-m)$ plus the delay of actor v_i. In max-plus algebra, we can write it as:

$$x_i(n) = d(v_i) \otimes \left(\bigoplus_s x_{j(i,s)}(n - \mu(i,s)) \right) \tag{4.8}$$

where index *s* enumerates the incoming edges in an arbitrary order, $j(i,s)$ identifies the producer actor of the incoming edge with index *s* and $\mu(i,s)$ gives the number of initial tokens on that edge.

In 'normal' algebra, multiplication distributes over the addition – i.e. $a(b+c) = ab + ac$; as notices before, the same holds for operations '$\otimes$' and '$\oplus$' of max-plus algebra, namely $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$. Applying the distributive law to Equality (4.8), we conclude that every edge (v_j, v_i) with index *s* contributes the following term to the evolution equation of variable $x_i(n)$:

$$d(v_i) \otimes x_{j(i,s)}(n - \mu(i,s))$$

whereby the individual contributions are summed, using max-plus operator '$\oplus$'.

Now let us update the table with the terms contributed by the edges. For each edge, (v_j, v_i), we replace the ε element at row *i*, column *j* by a non-ε element, equal to the contribution of edge (v_j, v_i), which is $d(v_i) \otimes x_j(n - m(i,j))$, where $m(i,j)$ is the number of initial tokens on edge (v_j, v_i). We insert the contribution of edge (v_j, v_i) into the table row '*i*' column '*j*', as shown in Table 4.3.

Eventually, the table contains as many non-ε elements as the number of edges in graph **G**.

Before we explain how to derive matrix-form equations from the table, consider the HSDF graph example in Figure 4.2. The contributions of completion-time variables for that example are shown in Table 4.4. For example, element '$3 \otimes x_1(n)$' in row 2 column 1 corresponds to edge (v_1, v_2), whereby index 'n' or 'n – 0' shows that the edge carries 0 initial tokens and coefficient '3' shows that the consumer actor has delay 3. Element 'ε' in row 3 column 3 shows that no edge joins actor v_3 with itself.

Having constructed the table, it is straightforward to derive the matrix-form evolution equations. Hereby, for the matrices, we use the same notations as in Equality (4.4): $\mathbf{A_0}, \mathbf{A_1}, \mathbf{A}_2, \ldots$ In general, the system of evolution equations can be expressed in max-plus algebra in the following form:

$$\mathbf{x}(n) = \mathbf{A_0} \cdot \mathbf{x}(n) + \mathbf{A_1} \cdot \mathbf{x}(n-1) + \ldots + \mathbf{A}_M \cdot \mathbf{x}(n-M) \tag{4.9}$$

where *M* is the maximum number of initial tokens of any edge in **G**.

Table 4.3 Table of variable contributions: non-‘zero’ values

		column 1	…	column j	…	column V
row 1	$x_1(n)$ =	…	…	…	…	…
…	…	…	…	…	…	…
row i	$x_i(n)$ =	…	…	$d(v_i) \otimes x_j(n - m(i,j))$	…	…
…	…	…	…	…	…	…
row V	$x_K(n)$ =	…	…	…	…	…

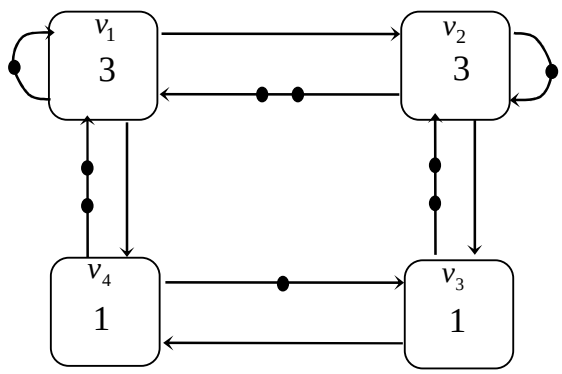


Figure 4.2 An HSDF graph with 4 actors

Table 4.4 Table of variable contributions for the example in Figure 4.2

		1	2	3	4
1	$x_1(n)$ =	$3 \otimes x_1(n-1)$	$3 \otimes x_2(n-2)$	ε	$3 \otimes x_4(n-2)$
2	$x_2(n)$ =	$3 \otimes x_1(n)$	$3 \otimes x_2(n-1)$	$3 \otimes x_3(n-2)$	ε
3	$x_3(n)$ =	ε	$1 \otimes x_2(n)$	ε	$1 \otimes x_4(n-1)$
4	$x_4(n)$ =	$1 \otimes x_1(n)$	ε	$1 \otimes x_3(n)$	ε

It is straightforward to obtain the matrices $\mathbf{A_0}, \mathbf{A_1}, \mathbf{A}_3, \ldots$ from the table. All of them have the same dimensions as the table: $V \times V$. For matrix $\mathbf{A}_0$, consider all the table elements that have variables with index ‘n’ or ‘$n-0$’. All the coefficients of those table elements go into the same positions of $V \times V$ matrix $\mathbf{A}_0$ as they have in the table. All the other elements in matrix $\mathbf{A}_0$ get value ‘ε’. For instance, based on the table example above, we have:

$$\mathbf{A_0} = \begin{bmatrix} \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ 3 & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & 1 & \varepsilon & \varepsilon \\ 1 & \varepsilon & 1 & \varepsilon \end{bmatrix}$$

Similar rules holds for any $\mathbf{A}_m$, $m = 0,\ldots M$; one just needs to pick the coefficients of variables having index ‘$n-m$’ instead of ‘$n-0$’. For our example, we have $M = 2$ and:

$$\mathbf{A}_1 = \begin{bmatrix} 3 & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & 3 & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \varepsilon & 1 \\ \varepsilon & \varepsilon & \varepsilon & \varepsilon \end{bmatrix} \qquad \mathbf{A}_2 = \begin{bmatrix} \varepsilon & 3 & \varepsilon & 3 \\ \varepsilon & \varepsilon & 3 & \varepsilon \\ \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \varepsilon & \varepsilon \end{bmatrix}$$

Matrix $\mathbf{A}_m$ is, in fact, the so-called *adjacency matrix* with respect to the maximum subgraph of **G** that contains only edges with *m* initial tokens. An adjacency matrix of a graph is, in general, a matrix that contains a non-zero element in row *i* and column *j* if and only if there is an edge from the *j*-th node to the *i*-th node of the graph.

We call $\mathbf{A}_m$ the adjacency matrix of order *m* with respect to graph **G**.

To be able to apply the evolution equations in practice for the derivation of $\mathbf{x}(n)$, one needs to define the *initial conditions*, because the evolution equations are recurrent equations. Under the assumption of Section 2.2.1 that all initial tokens are available at time 0, we have:

$$\mathbf{x}(-1) = \mathbf{x}(-2) = \ldots = \mathbf{x}(-M) = \mathbf{e}$$

where **e** is the vector of max-plus unit elements *e*, numerically equal to 0. Because these particular initial conditions correspond to the simultaneous release of all initial tokens, we call them *synchronous initial conditions.*

The evolution equations are different from the canonic equations, given in Equality (4.5), in two ways.

Firstly, the maximum order of adjacency matrices in (4.9) is *M*, whereas the system of canonic equations has order 1.

Secondly, the evolution equations are, in general, not constructive, i.e., they cannot be directly applied to compute $\mathbf{x}(n)$ from $\mathbf{x}(n-1)$, $\mathbf{x}(n-2)$, etc., because, as we see in Equality (4.9), $\mathbf{x}(n)$ is present both in the left and the right side of the equations. Max-plus algebra does not have an analogy to the standard subtraction operation '–', which would help to resolve this situation in 'normal' algebra.

To derive the canonic equations from the evolution equations, we first reduce the maximum order of equations to 1 and then make them constructive. This is done through a series of transformations presented in the next section, following the same method as described in [4 - §2], now considering HSDF graphs rather than event graphs and filling in some details that were skipped in [4 - §2].

4.2 Transformation into Canonic Form

In this section, we study the derivation of canonic equations and the accompanying theoretical results about the canonic matrix, which are important for applying the main theorem given in Section 4.3 in the context of HSDF graphs and, in particular, for the major algorithmic rule, obtained at the end of this chapter.

4.2.1 Low-order Variant of Graph G

To reduce the maximum order of evolution equations, one can transform graph **G** into graph $\mathbf{G}'$ by splitting each edge with marking *m* into *m* edges with marking 1, inserting new actors in between that have delay 0. We call the new HSDF graph $\mathbf{G}'$ the *low-order variant of graph* **G** or just low-order graph. For example, the low-order graph corresponding to the graph in Figure 4.2

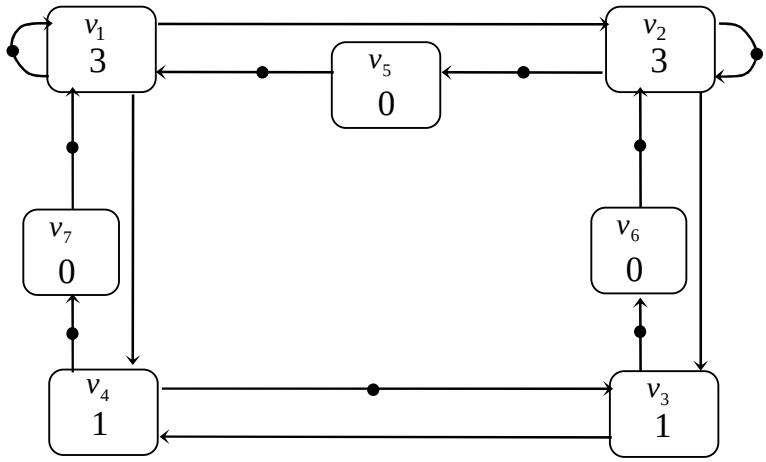


Figure 4.3 The low-order variant of the graph in Figure 4.2

is shown in Figure 4.3. For a low-order graph, the evolution equations can be expressed as follows:

$$\mathbf{x}'(n) = \mathbf{A}'_{\mathbf{0}} \cdot \mathbf{x}'(n) + \mathbf{A}'_{\mathbf{1}} \cdot \mathbf{x}'(n-1) \tag{4.10}$$

with initial conditions $\mathbf{x}'(-1) = \mathbf{e}$,

where

$$\mathbf{x}'(n) \equiv \begin{bmatrix} \mathbf{x}(n) \\ \tilde{\mathbf{x}}(n) \end{bmatrix} \tag{4.11}$$

which means that it is a concatenation of the completion time vector of the original graph **G** with the vector of completion times of new actors $\tilde{\mathbf{x}}(n)$.

Example The evolution equations for the low-order graph shown in Figure 4.3 look as follows:

$$\begin{bmatrix} x'_1(n) \\ x'_2(n) \\ x'_3(n) \\ x'_4(n) \\ x'_5(n) \\ x'_6(n) \\ x'_7(n) \end{bmatrix} = \begin{bmatrix} \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ 3 & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & 1 & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ 1 & \varepsilon & 1 & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \end{bmatrix} \cdot \begin{bmatrix} x'_1(n) \\ x'_2(n) \\ x'_3(n) \\ x'_4(n) \\ x'_5(n) \\ x'_6(n) \\ x'_7(n) \end{bmatrix} + \begin{bmatrix} 3 & \varepsilon & \varepsilon & \varepsilon & 3 & \varepsilon & 3 \\ \varepsilon & 3 & \varepsilon & \varepsilon & \varepsilon & 3 & \varepsilon \\ \varepsilon & \varepsilon & \varepsilon & 1 & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & e & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & e & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \varepsilon & e & \varepsilon & \varepsilon & \varepsilon \end{bmatrix} \cdot \begin{bmatrix} x'_1(n-1) \\ x'_2(n-1) \\ x'_3(n-1) \\ x'_4(n-1) \\ x'_5(n-1) \\ x'_6(n-1) \\ x'_7(n-1) \end{bmatrix} \blacklozenge$$

The adjacency matrices of graph $\mathbf{G}'$, $\mathbf{A}'_{\mathbf{0}}$ and $\mathbf{A}'_{\mathbf{1}}$, are needed for the derivation of the canonic-form evolution equations of graph $\mathbf{G}$. They can be constructed from $\mathbf{G}'$ using the same rules for adjacency matrices as defined earlier in this section.

Algorithmic Rule (Construction of low-order graph adjacency matrices $\mathbf{A}'_{\mathbf{0}}$ and $\mathbf{A}'_{\mathbf{1}}$) Let K be the number of actors in graph $\mathbf{G}'$. We would like to construct matrices $\mathbf{A}'_{\mathbf{0}}$ and $\mathbf{A}'_{\mathbf{1}}$ of size $K \times K$ represented as conventional 2-dimensional arrays.

The first pass of the algorithm computes the size K of the matrices and initializes them with elements 'ε'. We have:

$$K = V + R - E_{m>0} \tag{4.12}$$

where K is the number of actors in $\mathbf{G}'$, V is the number of actors in **G**, R is the total number of initial tokens in **G**, and $E_{m>0}$ is the number of edges containing at least one initial token.

The second pass constructs graph $\mathbf{G}'$ by splitting the edges of graph **G** and introducing auxiliary actors if m is greater than 1.

The third pass creates the adjacency matrices of graph $\mathbf{G}'$ as described in the previous subsection.

The complexity of the algorithm is dominated by the matrix initialization, and it amounts to $O(K^2)$, where K is defined by Equality (4.12).♦

Potentially, the complexity of this rule could be exponential in the size of the specification of the original graph **G**. The point is that it is quadratic in the number of initial tokens m on any edge in **G**, whereas to express this number in the specification of **G** takes log(m) digits. In practice, this means that adding just a few decimal digits to the marking**,** e.g. changing from 2 to 200 initial tokens, can lead to a big increase in the number of elements in $\mathbf{A}_0'$ and $\mathbf{A}_1'$, up to a factor of 10^4 in the given example.

Fortunately, one can anticipate that IPC graphs will be characterized by a polynomially bounded number of initial tokens per edge. We have shown in Section 3.7.2 that for practical IPC graphs one can bound the number of initial tokens in any cycle by an expression that is linear in the total number of actors V. Obviously, the same upper bound applies to the marking m of any edge in **G**. Therefore, for practical IPC graphs, K is at most $O(VE)$, and the complexity of the algorithmic rule given above is at most $O(V^2E^2)$.

4.2.2 From the Low-order Graph to the Canonic Form

To obtain the canonic form from the low-order graph, one has to get rid of the dependency of $\mathbf{x}'(n)$ on itself in Equality (4.10). This is done by applying the following lemma.

Lemma 4.1 If matrix **A** is an adjacency matrix of a graph that contains no cycles and K is the number of nodes in the graph, then equation $\mathbf{x} = \mathbf{A} \cdot \mathbf{x} + \mathbf{b}$ has is a unique solution which is given by $\mathbf{x} = \mathbf{A}^* \cdot \mathbf{b}$, where $\mathbf{A}^*$ is defined by

$$\mathbf{A}^* = \mathbf{A}^0 + \mathbf{A}^1 + \ldots + \mathbf{A}^{K-1} \tag{4.13}$$

♦

Here $\mathbf{A}^0$ is a diagonal matrix with unit elements on the diagonal:

$$\mathbf{A}^0 \equiv \begin{bmatrix} e & \varepsilon & \varepsilon & \\ \varepsilon & e & \varepsilon & \ldots \\ \varepsilon & \varepsilon & e & \\ & \ldots & & \ldots \end{bmatrix} \tag{4.14}$$

Proof. See e.g. [4 - §3.2.3.1] ♦

To apply this lemma to Equation (4.10), where $\mathbf{A} = \mathbf{A}_0'$ and $\mathbf{b} = \mathbf{A}_1' \cdot \mathbf{x}'(n-1)$, we should check whether the lemma conditions hold. Recall that $\mathbf{A}_0'$ is the adjacency matrix of the maximum subgraph of $\mathbf{G}'$ that contains only edges without initial tokens. That subgraph cannot be cyclic because $\mathbf{G}$, and hence $\mathbf{G}'$ as well, cannot contain a cycle without initial tokens. The point is that we assume that $\mathbf{G}$ is an IPC graph, and an IPC graph is live, so it is free of such cycles.

Thus from Equation (4.10) and Lemma 4.1 we obtain an equation, which is, in fact, a canonic-form expression:

$$\mathbf{x}'(n) = \mathbf{B} \cdot \mathbf{x}'(n-1)$$

where

$$\mathbf{B} = (\mathbf{A}_0')^* \cdot \mathbf{A}_1' \tag{4.15}$$

is the canonic matrix for graph $\mathbf{G}$.

Note that the canonic expression can be rewritten as follows:

$$\mathbf{x}'(n) = \mathbf{B}^{n+1} \cdot \mathbf{x}'(-1) \tag{4.16}$$

Thus, the timing behavior of an HSDF graph at any iteration can be derived from the initial conditions.

In the next subsection, based on Equality (4.15), we revisit the construction of the canonic matrix, giving more insight into the meaning of the matrix multiplication expression $(\mathbf{A}_0')^* \cdot \mathbf{A}_1'$, which helps us to see an efficient algorithmic rule for that purpose. There, we also derive the canonic matrix for the example in Figure 4.2.

4.2.3 HSDF Graph Paths and Their Representation in the Canonic Matrix

The canonic expression (4.5) is used to reason about the periodicity and, consequently, about the throughput of an HSDF graph **G**. Recall from the previous chapters that it is the cycle with maximum *cycle mean* that determines the throughput, where the 'cycle mean' is the ratio between the *length* of the cyclic path, i.e., its total delay, and the *depth* of the cyclic path, i.e., the number of initial tokens on the cycle. In this subsection, we highlight the relation between the canonic matrix **B** on the one side and the paths through the low-order graph $\mathbf{G}'$, on the other side. This gives us an efficient algorithm to compute matrix **B**, which is based on a longest path algorithm.

We start by defining an HSDF graph path and the path length formally, and then we show how one can calculate the canonic matrix efficiently using the longest path calculation algorithms.

Definition. (A path/cycle and its length and depth) A *non-empty path* in an HSDF graph is an ordered sequence of edges, whereby, for every two subsequent edges in the order, it holds that the consumer of the first edge is the producer for the second edge. A non-empty path has a *source* – which is the producer of the first edge – and a *destination* – which is the consumer of the last edge. For example, in Figure 4.3, there is path $((v_1, v_1), (v_1, v_1), (v_1, v_4), (v_4, v_7))$, whose source is v_1 and destination is v_7.

The set of edges of an *empty path* is empty. Nevertheless, an empty path has a source and a destination, which are always the same actor. We say that any actor is joined to itself by such a path.

A *cycle* is a non-empty path joining an actor to itself.

The *depth* of path *P*, $\mu(P)$, is the sum of initial markings of all edges in the path.

The *length* of path *P*, denoted $l(P)$, is the sum of the delays of the consumer actors of all edges in the path. ♦

Intuitively, the path length is a minimum time interval between the completion events of the source actor and the destination actor.

Having defined paths, let us come back to the main topic of this subsection – the relationship between matrix **B** and the paths in graph $\mathbf{G}'$. First of all, let us consider matrix $(\mathbf{A}_0')^*$, the first multiplier in Equality (4.15). The meaning of this matrix is explained in the following lemma.

Lemma 4.2 In matrix $(\mathbf{A}_0')^*$, the element in row *i* and column *j* of this matrix represents the length of the longest path through graph $\mathbf{G}'$ from the source actor *j* (by actor *j* we mean actor v_j) to the destination actor *i* consisting only of 0-marking edges. We call such a path a *0-mark* path.

Therefore, the entries in matrix $(\mathbf{A}_0')^*$ can be filled by inserting at position '*i,j*' the length of the longest 0-mark path from actor *j* to actor *i* containing any number of edges. If the path is empty we insert '*e*', which can happen only at the matrix diagonal. If no paths exist between the actors, we insert 'ε'. ♦

Before giving a proof, we provide an example.

Example (Computation of matrix $(\mathbf{A}_0')^*$) In Figure 4.3, there are only two 0-mark paths going from actor v_1 to actor v_4. One of them consists of edges (v_1, v_2), (v_2, v_3), and (v_3, v_4). The other one consists of edge (v_1, v_4). The first path has a larger length, equal to $d(v_2) + d(v_3) + d(v_4) = 3 + 1 + 1 = 5$. Therefore, the value of matrix element $\{(\mathbf{A}_0')^*\}_{4,1}$ is 5. Having computed all the elements of matrix $(\mathbf{A}_0')^*$ in the same way, we obtain:

$$(\mathbf{A}_0')^* = \begin{bmatrix} e & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ 3 & e & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ 4 & 1 & e & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ 5 & 2 & 1 & e & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \varepsilon & \varepsilon & e & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & e & \varepsilon \\ \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon & e \end{bmatrix} \qquad (4.17)$$

♦

Proof of Lemma 4.2. The lemma can be proven by showing that the *k*-th power of matrix $(\mathbf{A}_0')^k$ gives the lengths of all 0-mark paths that have exactly *k* edges and then using the definition of unary matrix operator '*' (See Lemma 4.1), which applies the max-plus algebra matrix operation '+' – i.e. maximization of the individual elements – to all the interesting powers of matrix $\mathbf{A}_0'$, thereby finding the longest possible paths. Note that the powers accumulated in $(\mathbf{A}_0')^*$ are in range 1..*K*–1, which covers all possible numbers of edges on a path through an acyclic graph that has *K* nodes.

The equality between the elements of the *k*-th power of matrix $\mathbf{A}_0'$ and the longest 0-mark path length can be shown by mathematical induction. By Equality (4.14), it holds for power 0, because the paths with zero edges start and end at the same actor and have length *e*, which corresponds to the *e*'s on the diagonal. For power 1, this property follows from the definition of the adjacency matrix, which contains the delays of the destination actors of the edges. Let us assume that the property is proven for power *k* and let us prove it for power $(k+1)$. Let's consider the multiplication of $(\mathbf{A}_0')^k$ by $\mathbf{A}_0'$. Element '*i,j*' of the resultant matrix is obtained by applying operation '⊕' – i.e. maximization – to all possible combinations $\{(\mathbf{A}_0')^k\}_{j,p} \otimes \{\mathbf{A}_0'\}_{p,i}$

for all *p*, whereby the first element in the combination is the length of the longest path from actor *j* to actor *p* having exactly *k* edges and the second element is the length of the longest path from actor *p* to actor *i* having exactly one edge. Therefore, the result of operation '⊕' contains the length of the longest path containing exactly $(k+1)$ edges. **qed**♦

When considering the product of $(\mathbf{A}_0')^*$ and $\mathbf{A}_1'$, resulting in matrix **B**, now it is not difficult to see the relation between **B** and the low-order live graph $\mathbf{G}'$. The element in row *i* and column *j* of matrix **B** represents the length of the longest of all the paths having the following properties:

1) the path has actor *j* as the source and actor *i* as the destination;

2) the path is not empty;

3) the first edge of the path has marking 1;

4) the rest of the edges, if any, have marking 0.

We call such a path a *special path*. In a special path, the first edge is contributed by $\mathbf{A}_1'$ and the rest are contributed by $(\mathbf{A}_0')^*$. If no such path exists, matrix element '*i, j*' contains ε.

Example (Computation of matrix B) In Figure 4.3, there are two special paths from actor 7 to actor 4. The first path consists of edges (v_7, v_1), (v_1, v_2), (v_2, v_3), and (v_3, v_4). The second path consists of edges (v_7, v_1) and (v_1, v_4). The first path has a larger length, equal to 8. Therefore element $\{\mathbf{B}\}_{4,7}$ has value '8'. The other elements in matrix **B** can be computed in the same way. As a result, we obtain:

$$\mathbf{B} = \begin{bmatrix} 3 & \varepsilon & \varepsilon & \varepsilon & 3 & \varepsilon & 3 \\ 6 & 3 & \varepsilon & \varepsilon & 6 & 3 & 6 \\ 7 & 4 & \varepsilon & 1 & 7 & 4 & 7 \\ 8 & 5 & \varepsilon & 2 & 8 & 5 & 8 \\ \varepsilon & e & \varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & e & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \varepsilon & e & \varepsilon & \varepsilon & \varepsilon \end{bmatrix} \tag{4.18}$$

♦

Every multiplication of matrices of size *K*x*K* has complexity $O(K^3)$. Therefore, direct application of Equalities (4.13) and (4.15) to derive a canonic matrix from the low-order graph adjacency matrices, $\mathbf{A}_0'$ and $\mathbf{A}_1'$, by $(K+1)$ multiplications has complexity $O(K^4)$.

Let us consider faster algorithms for this problem, which is, by the way, well-studied and classical. It is not difficult to show that matrix $(\mathbf{A}_0')^*$ is the *K*-th power of matrix $((\mathbf{A}_0')^0 + \mathbf{A}_0')$. [16 - §25] explains an $O(K^3 \cdot \log K)$ algorithm for computing the *K*-th matrix power, which is faster than $O(K^4)$. However, [16] also shows that due to the relationship between the max-plus matrix powers and the path lengths, there exists a faster algorithm. Rather then directly computing the matrix powers, the fast algorithm computes longest 0-mark path lengths between all pairs of actors in graph $\mathbf{G}'$. We mention that algorithm in the following rule.

Algorithmic Rule (Construction of the canonic matrix) To construct matrix **B**, one can, as a first step, compute $(\mathbf{A}_0')^*$ via the Floyd-Warshall algorithm [16 - §25] to solve the all-pair

longest path problem in the maximal subgraph of $\mathbf{G}'$ containing all 0-mark edges. Then, one can find the product of matrices $\mathbf{A}_1'$ and $(\mathbf{A}_0')^*$. The algorithmic complexity of both steps is $O(K^3)$.♦

4.2.4 The Canonic Graph and its Relation to the HSDF Graph

The canonic equation, Equality (4.5), might suggest that a canonic matrix, i.e. **B**, itself could be seen as a first-order adjacency matrix of an HSDF graph. However, it is not true in general. In an HSDF adjacency matrix, all the non-zero (non-ε) elements of any row *i* must be identical, because they all contain the delay of actor *i*. This property is not necessarily true for any canonic matrix. For instance, it does not hold for rows 2, 3 and 4 of matrix **B** in the example given in Equality (4.18).

Thus, in general case, for a given canonic matrix, one cannot build a supplementary HSDF graph such that matrix **B** would be its adjacency matrix, although that would be useful for characterization of that matrix. Nevertheless, another graph-theoretic model of the canonic matrix has proven to be a very handy tool for characterization and analysis of the canonic equation. In this subsection, we build a so-called *canonic graph*, which serves these purposes and which is a representative of a model of computation that is different from HSDF. Below we also establish an important relation between the canonic graph and the original HSDF graph **G**.

Just as in the related work, [4 - §1-3], we use the '*event graph*' model of computation, which can be seen as a modified version of the HSDF graph model of computation. In the event graphs, the graph nodes behave like actors. However, for simplicity, without loss of generality, we assume that the event graph nodes do not have delays (or we may say that they have delay 0). Instead, unlike HSDF graph edges, event graph edges do have delays[27] [4 - §2]. As this model executes, after a token production on an edge, the consuming node can capture the token only after the edge delay.

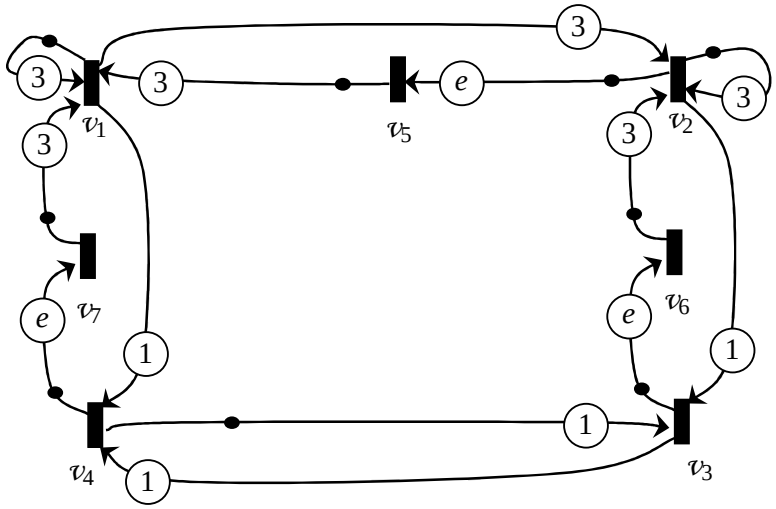
Definition. An *event graph* $\mathcal{G}$ is a tuple $(\mathcal{V}, \mathcal{E}, \mu, d)$, where $\mathcal{V}$ is the set of nodes, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of (directed) edges, $\mu(e)$ is a function defines an non-negative integer number of initial tokens at edge e, and $d(e)$ is function that gives a non-negative real number defining the delay of edge e. ♦

One can always translate an HSDF graph **G** into an *equivalent event graph* $\mathcal{G}_E$, preserving the same graph structure. One just has to change each HSDF actor into an event graph node, shifting the actor delay annotation to its incoming edges; see the example in Figure 4.4(a).

Definition. The *precedence graph* $\mathcal{G}(\mathbf{A})$ of matrix **A** with dimensions $K \times K$ is an event graph with *K* nodes: $v_1, \ldots, v_K$, whose edges (v_j, v_i) correspond one-to-one to non-zero (non-ε) elements '*i, j*' of **A**, getting the delay annotations $d(v_j, v_i) = \{\mathbf{A}\}_{i,j}$ and identical markings $\mu(v_j, v_i) = 1$. ♦

Definition (Canonic graph). If **B** is the canonic matrix of HSDF graph **G**, then its precedence graph $\mathcal{G}(\mathbf{B})$ is called the *canonic graph* of HSDF graph **G**.♦

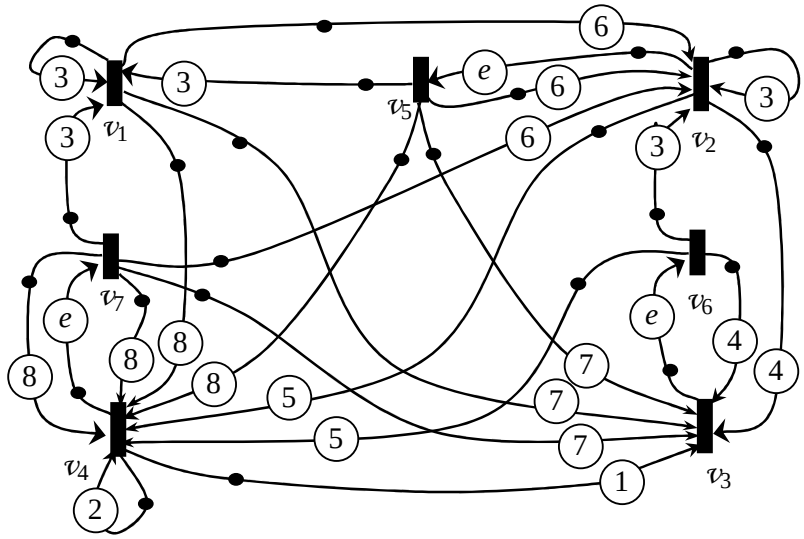
[27] Event graphs have two more features that we, however, do not need to use. Namely, they also allow multiple 'edges' (in our terminology) between the same pair of nodes, and they also may allow nodes to have delays, but we do not need and do not include this into our definition of event graphs.



(a) Event graph $\mathcal{G}_E$ equivalent to the low-order HSDF graph in Figure 4.3

v_k ▮ - event graph node, similar to an HSDF actor, but with delay e
(this drawing style is borrowed from Petri nets, a closely related model)

(1) - the delay of an event graph edge



(b) The canonic graph $\mathcal{G}$ obtained for the HSDF graphs in Figures 4.2 and 4.3
This graph is the precedence graph of matrix **B** in Equality (4.18)

Figure 4.4 The equivalent event graph and its canonic graph

The canonic graph $\mathcal{G}(\mathbf{B})$ is the graph-theoretic model that we refer to in the beginning of this subsection. The structure of this model is, in general, quite different from the structure of the original HSDF graph. For instance, look at the canonic graph shown in Figure 4.4(b) and compare it to the original HSDF graph in Figure 4.2.

For an event graph model, one can reuse the same definition for path, cycle, path/cycle length $l(P)$ and path/cycle depth $\mu(P)$ as given in the previous subsection, with one exception: it is the delays of the edges themselves that contribute to the path length, not the delays of the edge consumers.

The whole purpose of obtaining the canonic graph is to use the theoretical results that apply for such graphs. We start doing that in the next subsection, but first we need to establish an important relation between an HSDF graph and its canonic graph.

Lemma 4.3 (Canonic cycles and HSDF cycles) Let **G** be a live HSDF graph and $\mathcal{G}$ its canonic graph. Then:

1) For any cycle **C** in **G**, there is cycle $\mathcal{C}$ in $\mathcal{G}$ such that $l(\mathbf{C}) \leq l(\mathcal{C})$ and $\mu(\mathbf{C}) = \mu(\mathcal{C})$.
2) For any cycle $\mathcal{C}$ in $\mathcal{G}$, there is cycle **C** in **G** such that $l(\mathbf{C}) = l(\mathcal{C})$ and $\mu(\mathbf{C}) = \mu(\mathcal{C})$. ♦

Proof Instead of proving this lemma for cycles **C** in **G**, we do that for the correspondent cycles $\mathcal{C}_E$ in graph $\mathcal{G}_E$ that is equivalent to low-order graph $\mathbf{G}'$. In terms of the example we use in this chapter, this means that we prove the relationship between the cycles of the graphs shown in Figure 4.4(a) and 4.4(b).

The replacement of graph **G** by graph $\mathcal{G}_E$ in the proof is justified as follows. Firstly, $\mathcal{G}_E$ has the same structure as the low-order variant of graph **G**, i.e. graph $\mathbf{G}'$. Moreover, the paths/cycles in $\mathbf{G}'$ and $\mathcal{G}_E$ have identical lengths and depths. Secondly, there obviously exists a one-to-one relationship between the cycles in $\mathbf{G}'$ and the cycles in **G** that preserves lengths and depths. Therefore, such a one-to-one relationship exists between graphs **G** and $\mathcal{G}_E$, and one graph can be replaced by the other.

Before we start with the actual proof, we need to observe a certain property of graph $\mathcal{G}$. Graph $\mathcal{G}$ contains an edge (v_j, v_i) if and only if there is at least one non-empty path through $\mathcal{G}_E$ from node v_j to node v_i whose first edge has marking 1 and the other edges, if any, have marking 0. By analogy to the terminology of the previous subsection, let us call such a path a *special path*. The delay of edge (v_j, v_i) is equal to the length of the longest special path from v_j to v_i. In the remainder, we refer to this property of graph $\mathcal{G}$ as the *special property*.

The special property follows from the definition of $\mathcal{G}$ as the precedence graph of the canonic matrix **B**, and the corresponding property of matrix **B** has been already shown in the previous subsection. To give an example, edge (v_5, v_2) in Figure 4.4(b) corresponds to special path (v_5, v_1), (v_1, v_2) in Figure 4.4(a).

(a) Special path $\mathcal{P}$ in $\mathcal{G}_E$ **(b)** The corresponding tree in $\mathcal{G}$

Figure 4.5 Transformation of the event graph into the canonic graph

Let us prove part 1) of the lemma. Let $\mathcal{C}_E$ be a cycle in $\mathcal{G}_E$. That cycle can be split into special paths $\mathcal{P}_1$, $\mathcal{P}_2$, … $\mathcal{P}_P$, where $P = \mu(\mathcal{C}_E) \geq 1$. This can be done by splitting the sequence of edges in cycle $\mathcal{C}_E$ into subsequences, whereby every first edge in each subsequence has an initial token and the following edges, if any, do not have initial tokens. Now let $\mathcal{P}_p$ be one of those paths. We illustrate this path in Figure 4.5(a). Just as in that figure, let us denote the starting and final node of this path as $v_{k(1)}$ and $v_{k(L)}$. By the special property, graph $\mathcal{G}$ must contain an edge $(v_{k(1)}, v_{k(L)}$ that joins the starting and the final nodes. That edge has marking 1 and its delay should be at least the length of path $\mathcal{P}_p$: $d(v_{k(1)}, v_{k(L)}) \geq l(\mathcal{P}_p)$. We can find such edges in $\mathcal{G}$ for all paths $\mathcal{P}_p$ and all those edges together form a cycle in $\mathcal{G}$. Let us denote this cycle $\mathcal{C}$. We get $l(\mathcal{C}) \geq l(\mathcal{P}_1) + l(\mathcal{P}_2) + \ldots + l(\mathcal{P}_P) = l(\mathcal{C}_E)$, whereas $\mu(\mathcal{C}) = \mu(\mathcal{C}_E) = P$. **qed**

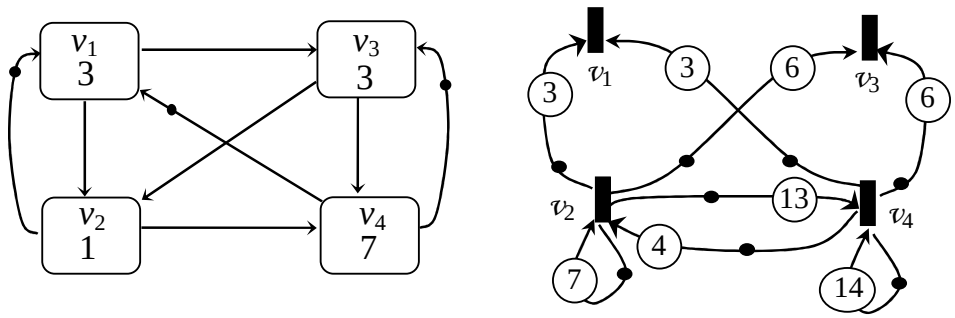
For example, in graph $\mathcal{G}_E$ in Figure 4.4(a), consider cycle $\mathcal{C}_E$ with edge sequence (v_4, v_7), (v_7, v_1), (v_1, v_4). We have $l(\mathcal{C}_E) = 4$ and $\mu(\mathcal{C}_E) = 2$. Cycle $\mathcal{C}_E$ can be split into two special paths: $\mathcal{P}_1$ with edge (v_4, v_7) and $\mathcal{P}_2$ with edges (v_7, v_1), (v_1, v_4). In Figure 4.4(b) these two paths are translated into two edges: (v_4, v_7) and (v_7, v_4), which form cycle $\mathcal{C}$ with a larger length and the same depth: $l(\mathcal{C}) = 8$ and $\mu(\mathcal{C}) = 2$.

Let us prove part 2) of the lemma. Let $\mathcal{C}$ be a cycle consisting of P edges in $\mathcal{G}$. By the special property of graph $\mathcal{G}$, each edge (v_j, v_i) in $\mathcal{C}$ corresponds to at least one special path $\mathcal{P}_p$ in $\mathcal{G}_E$ from v_j to v_i, whereby $d(v_j, v_i) = l(\mathcal{P}_p)$. All paths $\mathcal{P}_p$ combined together make a cycle $\mathcal{C}_E$. We have: $l(\mathcal{C}_E) = l(\mathcal{P}_1) + l(\mathcal{P}_2) + \ldots + l(\mathcal{P}_P) = \sum d(v_j, v_i) = l(\mathcal{C})$, and $\mu(\mathcal{C}) = \mu(\mathcal{C}_E) = P$. **qed**

For example, in Figure 4.4(b), consider the cycle $\mathcal{C}$ with edge sequence (v_4, v_7), (v_7, v_1), and (v_1, v_4). Edges (v_4, v_7) and (v_7, v_1) correspond to the same edges in $\mathcal{G}_E$. Edge (v_1, v_4) corresponds in $\mathcal{G}_E$ to the path with edge sequence (v_1, v_1), (v_1, v_2), (v_2, v_3), (v_3, v_4). As a result, we obtain in $\mathcal{G}_E$ a cycle that visits node v_1 twice. ♦

The last issue considered in this subsection is the issue of *strong connectedness*. We study this issue because, as mentioned in Chapter 2, the strong connectedness of an HSDF graph is essential so that the behavior of that graph can reach its steady state (i.e., eventually it becomes periodic). The question we look at in this subsection is to which extent the strong connectedness of HSDF graph **G** is preserved in its canonic graph $\mathcal{G}$.

To understand the concern about the preservation of strong connectedness, let us examine how certain parts of graph $\mathcal{G}_E$ are represented in graph $\mathcal{G}$. Consider a special path $\mathcal{P}$ in $\mathcal{G}_E$, see Figure 4.5(a). Let the sequence of edges of path $\mathcal{P}$ be $(v_{k(1)}, v_{k(2)})$, $(v_{k(2)}, v_{k(3)})$, …, $(v_{k(L-1)}, v_{k(L)})$, where L is the total number of edges in $\mathcal{P}$. Then, in graph $\mathcal{G}$, this path will be represented by a tree of L edges $(v_{k(1)}, v_{k(2)})$, $(v_{k(1)}, v_{k(3)})$, …, $(v_{k(1)}, v_{k(L)})$, see Figure 4.5(b). This is due to the fact that there exists at least one special path (a prefix of $\mathcal{P}$) from $v_{k(1)}$ to every other node in $\mathcal{P}$.



(a) HSDF graph **G** **(b)** The corresponding canonic graph $\mathcal{G}$
v_1 and v_3 are dangling nodes

Figure 4.6 An IPC graph with dangling nodes in the canonic graph

Thus, nodes $v_{k(2)}$, $v_{k(3)}$, …, $v_{k(L)}$ are directly connected with each other in $\mathcal{G}_E$ and not anymore directly connected in $\mathcal{G}$; which means that the strong connectedness of graph $\mathcal{G}$ is compromised. This happens due to the nodes in $\mathcal{G}_E$ that do not have any outgoing edges with initial tokens. Those nodes end up in graph $\mathcal{G}$ as *dangling nodes*. They are 'dangling' in the sense that they do not have any outgoing edges, they have only incoming edges.

Our canonic graph example in Figure 4.4(b) does not have any dangling nodes. To illustrate dangling nodes, we give an example in Figure 4.6. In Figure 4.6(a), an IPC graph is shown that models two processes communicating through a local channel having a capacity of one token. In Figure 4.6(b), we show the canonic graph of that IPC graph; we see there that nodes v_1 and v_2 are dangling.

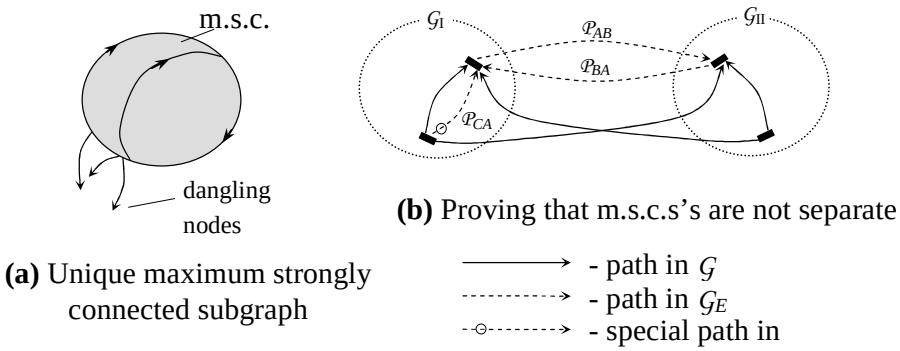
Remark (Dangling nodes and simplification of the canonic graph) One can simplify the canonic system of equalities by excluding the variables of the dangling nodes from the system [4]. This simplification reduces the computational complexity of the algorithmic rules applied to the canonic graph later. However, this simplification step is not essential and has certain implications. Therefore, we skip it in this thesis to avoid overloading it with details.♦

Coming back to the strong-connectedness issue, we conclude that in general the canonic graph is not strongly connected. Nevertheless, there is one essential property that the canonic graph still inherits from its HSDF graph, namely, it has exactly one maximal strongly connected subgraph (*m.s.c.s.*). For example, in Figure 4.6(b), the subgraph formed by nodes v_2 and v_4 is the only *m.s.c.s.* of the given canonic graph. The uniqueness of the *m.s.c.s.* is stated below as a lemma.

Lemma 4.4 (Strong connectedness) Let **G** be a live strongly-connected HSDF graph and $\mathcal{G}$ its canonic graph. Then graph $\mathcal{G}$ has exactly one maximal strongly-connected subgraph (*m.s.c.s.*). ♦

In other words, if all nodes of graph **G** can be clustered together into one strongly-connected graph, then at least part of the nodes of $\mathcal{G}$ can also be clustered such that the nodes outside that cluster are dangling nodes, which by definition cannot form another cluster. This generic structure is illustrated in Figure 4.7(a).

Proof In this proof, we again represent graph G by graph $\mathcal{G}_E$. First of all, it is straightforward to show that the number of *m.s.c.s.*'s of graph $\mathcal{G}$ is at least one. We can make that statement because



(a) Unique maximum strongly connected subgraph

(b) Proving that m.s.c.s's are not separate

Figure 4.7 The generic structure of the canonic graph

$\mathcal{G}_E$ is a strongly connected graph, thus it contains at least one cycle and therefore, by Lemma 4.3, graph $\mathcal{G}$ also contains at least one cycle and thus it has at least one *m.s.c.s.*

Let us prove the uniqueness of the *m.s.c.s.* in graph $\mathcal{G}$ by contradiction. Consider Figure 4.7(b). Suppose there are two distinct *m.s.c.s.*'s in $\mathcal{G}$, namely $\mathcal{G}_{\mathrm{I}}$ and $\mathcal{G}_{\mathrm{II}}$. Then each of them must contain at least one cycle. Let edge (C, A) belong to a cycle in subgraph $\mathcal{G}_{\mathrm{I}}$. Similarly, let edge (D, B) belong to a cycle in subgraph $\mathcal{G}_{\mathrm{II}}$.

Because $\mathcal{G}_E$ is a strongly connected graph, any pair of nodes is joined in that graph by a path. In this proof, we make use of the fact that in graph $\mathcal{G}_E$ there exists a path from A to B, denoted $\mathcal{P}_{AB}$, and also a path in the reverse direction, denoted $\mathcal{P}_{BA}$. In Figure 4.7(b), we illustrate paths in graph $\mathcal{G}_E$ by dashed arcs, in order to distinguish them from the paths in canonic graph $\mathcal{G}$, shown as solid arcs.

By the special property of the canonic graph, edge (C, A) corresponds to a special path $\mathcal{P}_{CA}$ from C to A in $\mathcal{G}_E$. Paths $\mathcal{P}_{CA}$ and $\mathcal{P}_{AB}$ combined together form path $\mathcal{P}_{CB}$, whose first edge carries one initial token due to the fact that $\mathcal{P}_{CA}$ is special. For that reason, path $\mathcal{P}_{CB}$ can be split into P special subpaths $\mathcal{P}_1, \mathcal{P}_2, \ldots \mathcal{P}_P$, where P is the depth of path $\mathcal{P}_{CB}$. Those subpaths correspond to edges in $\mathcal{G}$; therefore canonic graph $\mathcal{G}$ too has a path from C to A, and we illustrate this in Figure 4.7 by a solid arc.

By a similar reasoning, we can show that D is connected to A. Therefore, we see that a node in $\mathcal{G}_{\mathrm{I}}$ is connected to a node in $\mathcal{G}_{\mathrm{II}}$ and vise versa; therefore the maximal strongly connected subgraphs $\mathcal{G}_{\mathrm{I}}$ and $\mathcal{G}_{\mathrm{II}}$ are equal, which contradicts the original assumption that they are different. **qed** ♦

4.3 Main Theorem

In this section, we first state the main theorem, from which it directly follows that event graphs, when their execution starts, after a certain number of iterations, by themselves reach a mode in which they execute according to a periodic schedule with the period equal to the graph's MCM (see Section 2.2.5). Then we use the relationship between canonic event graphs and HSDF graphs to prove a theorem stating that HSDF graphs behave in the same way. In Section 4.4, we use that Theorem to establish the major algorithmic rule for static-delay analysis, which plays an important role in the performance analysis approach of this thesis.

Theorem 4.5 ('Main theorem' – Cyclicity of matrix B). Suppose that the precedence graph $\mathcal{G}$ of matrix **B** has exactly one maximum strongly connected subgraph. Let λ be the MCM of graph $\mathcal{G}$. Then there are integers $T \geq 0$ and $W > 0$ such that:

$$n \geq T \Rightarrow \quad \mathbf{B}^{n+W} = \lambda^W \cdot \mathbf{B}^n \tag{4.19}$$

where 'λ^W', like the rest of the expression, is written in terms of max-plus algebra, in normal notation 'λ^W' equals $\lambda \cdot W$.

Proof. This theorem follows from theorem 3.112 in [4]. ♦

Definition (Matrix cyclicity. *(W,T)*-cyclic matrix.) The property expressed by Formula (4.19) is referred to as the *cyclicity* of matrix $\mathbf{B}$. Matrix $\mathbf{B}$ satisfying Formula (4.19) for some $T \geq 0$ and $W > 0$ is called *cyclic* or, in particular, *(W,T)-cyclic*. ♦

Remark (The periodic behavior of strongly-connected event graphs). Consider an arbitrary strongly-connected event graph $\mathcal{G}$ whose edges have positive static delays and contain one initial token each. Then, using Theorem 4.5, we can state that graph $\mathcal{G}$ is a precedence graph of some *(W,T)*-cyclic matrix $\mathbf{B}$, for which Formula (4.19) applies.

Let vector $\mathbf{x}'(-1)$ give for every node in $\mathcal{G}$ the time when the initial token at every output edge of that node is released at the start of the execution. Then the completion times of the node executions in graph $\mathcal{G}$ satisfy Equality (4.16).

Multiplying the left and right part of the equality in Formula (4.19) by $\mathbf{B} \cdot \mathbf{x}'(-1)$, we get:

$$n \geq T \Rightarrow \quad \mathbf{B}^{(n+W)+1} \cdot \mathbf{x}'(-1) = \lambda^W \cdot \mathbf{B}^{n+1} \cdot \mathbf{x}'(-1) \tag{4.20}$$

Finally, applying Equality (4.16) to the left and the right part of this equality, we obtain:

$$n \geq T \Rightarrow \quad \mathbf{x}'(n+W) = \lambda^W \cdot \mathbf{x}'(n) \tag{4.21}$$

Because, in max-plus algebra, multiplying a vector by expression 'λ^W' means adding constant $\lambda \cdot W$ to every element of the vector, Formula (4.21), in fact, states that, after the first *T* iterations, the event graph is characterized by a periodic execution schedule that spans *W* iterations of the graph and has a period equal to $\lambda \cdot W$. ♦

Definition (The periodic and transient power of a matrix) For a cyclic matrix $\mathbf{B}$, we refer to the *minimum* integer values of positive *W* and non-negative *T* such that Formula (4.19) holds as the *periodic power* and the *transient power* of matrix **B** accordingly. ♦

Theorem 4.6 (Periodicity of HSDF graphs). Let **G** be a strongly connected live HSDF graph with static delays that contains *V* actors, and let $\mathbf{x}(n)$ be the completion time vector of $\mathbf{G}$. Then there are integers $T \geq 0$ and $W > 0$ such that:

$$n \geq T,\ k \in [1..V] \Rightarrow \quad x_k(n+W) = \lambda \cdot W + x_k(n) \tag{4.22}$$

where $\lambda = MCM(\mathbf{G})$ and $x_k(n)$ are the elements of $\mathbf{x}(n)$. ♦

This theorem can be interpreted as follows: after the time it takes the graph to go through the transient phase (the first *T* iterations), we can split the time axis into intervals of duration $\lambda \cdot W$; let us call them the *periods*. Within each period, every actor executes *W* times, and the completion times of those executions relative to the period boundaries are exactly the same in all periods. Thus λ is the average interval between the subsequent executions of each actor, which means that it is the iteration interval of the given HSDF graph.

Proof. Let $\mathcal{G}$ be the canonic graph of HSDF graph **G**. Let **C** be a cycle with the maximum cycle mean in graph **G** (a critical cycle). Recall that the *cycle mean* of cycle **C** is the ratio $l(\mathbf{C})/\mu(\mathbf{C})$. From Lemma 4.3, part 1, it follows that there is a cycle $\mathcal{C}$ in graph $\mathcal{G}$ with at least the same cycle mean. Thus, $MCM(\mathcal{G}) \geq MCM(\mathbf{G})$.

On the other hand, for a critical cycle $\mathcal{C}$ in graph $\mathcal{G}$, we can find a cycle **C** in **G** with the same cycle mean, as follows from Lemma 4.3, part 2. Thus, $MCM(\mathbf{G}) \geq MCM(\mathcal{G})$. This relation between the MCMs and the previously obtained relation imply that they are equal. Let's denote their value as λ.

By Lemma 4.4, graph $\mathcal{G}$ has only one m.s.c.s., thus by Theorem 4.4 the canonic matrix **B** is (W,T)-cyclic for some integers W and T and Equality (4.19) holds for (MCM) λ. Let $\mathbf{x}'(n)$ be the state vector of graph $\mathcal{G}$, then Formula (4.21). Because, according to Equality (4.11), $\mathbf{x}(n)$ is a sub-vector of vector $\mathbf{x}'(n)$, Formula (4.21) is also correct if you replace $\mathbf{x}'(n)$ by $\mathbf{x}(n)$. After such a replacement, Formula (4.21) becomes just a vector form of scalar equality (4.22). **qed** ♦

Definition (A periodic depth of an HSDF graph) We refer to *any* positive integer W such that Formula (4.22) holds as a *periodic depth* of an HSDF graph. It is the number of HSDF iterations in a period. ♦

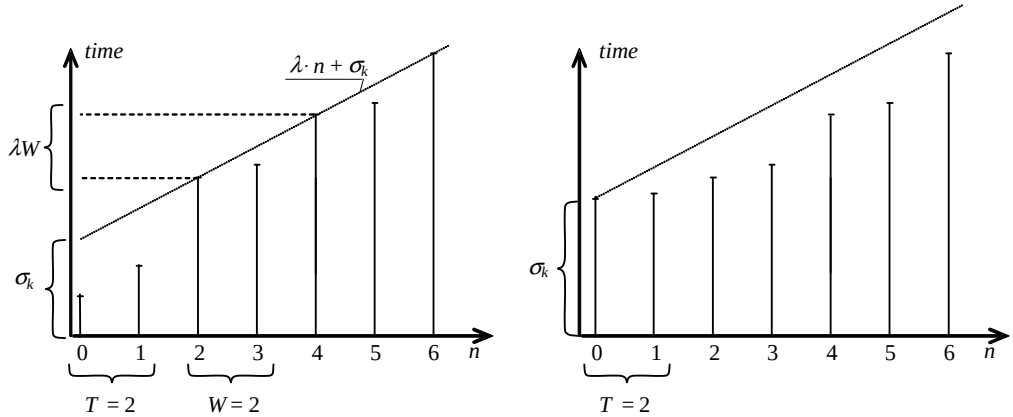
Definition (The transient depth of an HSDF graph) We refer to the *minimum* value T such that Formula (4.22) holds as the *transient depth* of an HSDF graph. It is the number of HSDF iterations between the start of the execution and the first period. ♦

Note that, unlike the transient and periodic powers of matrix **B**, the transient and minimum periodic depths of an HSDF graph depend on the initial conditions $\mathbf{x}'(-1)$. The powers of the matrix give us an upper bound on the depths, because Formula (4.22) always holds if W and T are the powers of the canonic matrix. However, the HSDF graph depths might be smaller for certain initial conditions. In fact, Theorem 3.28 in [4] implies that for quite general case there exist initial conditions such that $W = 1$ and $T = 1$.

4.4 The Lateness and the Major Algorithmic Rule for Static-delay HSDF

4.4.1 The Upper Bound on the Execution Time and the Lateness

The periodicity result shown in Theorem 4.6 is a refinement on the facts about the steady-state behavior of the static-delay HSDF graph that we considered in Section 2.2.5. Theorem 4.6 gives us a key to solve a major problem raised in Chapter 2: finding a conservative bound on the execution time of an HSDF graph for the first N iterations. Recall that, in Chapter 2, we come to a conclusion that, for the static-delay case, the upper bound is equal to $\lambda \cdot N$ plus some *additional component*, which is less significant for large N. Nevertheless, that component cannot be ignored if one wants to obtain conservative results for smaller N, and, most importantly, this component takes into account the *transient phase*. Recall from Section 1.5 that the transient phase is an important factor for extending the static-delay analysis to the dynamic delay case. Therefore, in this section, we consider the additional component again.



(a) Graphical representation of $x_k(n)$, HSDF periodicity and the asymptote

(b) Possible $x_k(n)$ where the lateness is determined by the transient phase

σ_k – lateness of actor v_k; the graph lateness σ is maximum σ_k

T – transient depth
W – periodic depth

Figure 4.8 Graphical illustration of lateness

Figure 4.8(a) illustrates the growth of the completion time variable $x_k(n)$ for actor v_k in **G** as characterized by Formula (4.22). In Figure 4.8(a), we assumed that both the transient depth and the minimum periodic depth are 2 iterations. As also shown in the figure and implied from Formula (4.22), for any actor v_k, the diagram of the actor completion time function, $x_k(n)$, has an asymptote, which is a linear function on n with slope equal to λ. Let us express that function as $\lambda \cdot n + \sigma_k$, where σ_k is a constant that only depends on which actor we select. If we choose σ as the maximum σ_k for all actors of the HSDF graph, then we can give an *upper bound* on the execution time of N graph iterations, which can be expressed as $\lambda \cdot (N-1) + \sigma$, because the last iteration has index $n = N-1$.

Now let us establish that upper bound formally.

Definition (Execution time of static- or dynamic-delay HSDF) The execution time of the first N iterations of a first-in-first-out (FIFO) HSDF graph with V actors and completion variables $x_k(n)$ is defined by:

$$\Delta_N \equiv \max_{k=1..V}\left(x_k(N-1)\right) \tag{4.23}$$

Note that this definition applies for both static and dynamic-delay HSDF. This definition is in line with the informal definition given in Chapter 2, where we define the execution time as the latest completion of any actor execution in the first N iterations. Note that we included only the last iteration index – i.e. $n = N-1$ – because the graph is a FIFO graph and hence all the previous iterations complete earlier. ♦

Lemma 4.7 (Lateness and an upper bound on HSDF execution time) For the execution of a live strongly connected static-delay HSDF graph **G** with V actors and any $N > 0$ holds:

$$N \geq 0 \Rightarrow \ \Delta_N \leq \hat{\Delta}_N \tag{4.24}$$

where:

$$\hat{\Delta}_N = \lambda \cdot (N-1) + \sigma \tag{4.25}$$

and σ is the HSDF graph *lateness*, defined by:

$$\sigma \equiv \max_{k=1..V} \max_{n=1..T+W} x_k(n-1) - \lambda \cdot (n-1) \tag{4.26}$$

where T is the transient depth and W is a periodic depth of HSDF **G**, which have finite integer values. ♦

Proof For convenience, let us introduce the following expression: $\sigma_k(n) \equiv x_k(n-1) - \lambda \cdot (n-1)$. From the definition of T and W, it follows that the infinite sequence $\sigma_k(T+1)$, $\sigma_k(T+2)$,... repeats itself periodically every W entries. Therefore, the maximization over $n = 1..T + W$ is equivalent to the maximization over the entire range $1..+\infty$. After putting this range into Equality (4.26) and substituting (4.26) into (4.25), we get:

$$\begin{aligned}
\hat{\Delta}_N &= \max_{k=1..V} \left(\lambda \cdot (N-1) + \max_{n=1..+\infty} \sigma_k(n)\right) = \\
&= \max_{k=1..V} \left(\lambda \cdot (N-1) + \max_{n=1..+\infty} x_k(n-1) - \lambda \cdot (n-1)\right) \geq \\
&\geq \max_{k=1..V} \left(\lambda \cdot (N-1) + x_k(N-1) - \lambda \cdot (N-1)\right) = \\
&= \max_{k=1..V} \left(x_k(N-1)\right) \equiv \Delta_N
\end{aligned}$$

qed ♦

Thus, we have refined the upper bound given in Chapter 2: $\hat{\Delta}_N = (\lambda + \varepsilon_N) \cdot N$. In fact, we now assume that ε_N is equal to $(\sigma - \lambda)/N$.

Therefore, using the results of Section 2.2.5, for HSDF graphs with synchronous initial conditions, we can write:

$$\lambda \cdot N \leq \Delta_N \leq \lambda \cdot (N-1) + \sigma \tag{4.27}$$

Remark. (The upper bound on the execution time is tight) From Equality (4.27), it follows that the maximum difference between the upper bound and the real value of the execution time is equal to $\sigma - \lambda$. This means that the error of $\hat{\Delta}_N$ in estimating the execution time is bounded by a constant, and therefore we say that $\hat{\Delta}_N$ is a *tight* bound. ♦

To calculate our upper bound on the graph execution, one needs to calculate two constants, characterizing the HSDF graph as a whole, namely the average iteration period, λ, and the graph lateness, σ. In this thesis, we use this upper bound as a conservative estimate of the performance of static-delay HSDF graphs. Therefore, we need to provide algorithmic rules to calculate λ and σ.

Efficient polynomial algorithms exist to compute λ, see a survey in [19]. For example, one can compute it using Karp's algorithm [44], having complexity $\Theta(K^3)$. Note that in [23] the state-space exploration techniques have been experimentally shown to be more run-time efficient than the graph analysis algorithms, when applied to more general graphs – SDF (multi-rate) graphs.

As for the lateness, to the best of our knowledge, no related work studies the derivation of any characteristic of event or HSDF graphs that is related to it. We dedicate the next subsection to the problem of calculating σ.

4.4.2 The Algorithmic Rule to Calculate Lateness

Before we consider an exact algorithm, let us mention a polynomial algorithm to calculate a lower bound on lateness.

Lemma 4.8 (A lower bound on the lateness) Under the synchronous initial conditions, the graph lateness σ is larger or equal to the length of a longest special path through **G**.

Hereby a special path through an HSDF graph is defined in a similar way we defined before for the event graphs; namely, the first edge in such a path should contain some initial tokens and the other edges (if any) should contain no initial tokens. ♦

Proof Substituting $n = 1$ into Equality (4.26), we get:

$$\sigma \geq \max_{k=1..V} x_k(0)$$

From the canonic equations, Equality (4.7), it follows that:

$$x_k(0) = \max_i (B_{k,i})$$

Now let us observe that $B_{k,i}$ is, by definition, equal to the longest special path in the equivalent graph. Using this observation and the inequalities established in this proof it is easy to prove the statement of the lemma.♦

One can calculate the lower bound by applying the topological sorting algorithm, e.g. see [16], to the maximum acyclic subgraph of **G** that does not contain initial tokens. However, it is an *upper* bound that is necessary for a conservative estimate, whereas a lower bound can be used to evaluate how tight the upper bound is.

Example (A lower bound on the lateness) A longest special path in Figure 4.2 is (v_1, v_1), (v_1, v_2), (v_2, v_3), (v_3, v_4) and the length of this path is 8. Therefore, for this graph holds that $\sigma \geq 8$. ♦

Unlike the value of average period λ, which depends only on the structure and delay values of HSDF graph **G**, the value of lateness σ also depends on the initial conditions. In the remainder of this subsection, we study an exact algorithm to calculate lateness, which takes the initial conditions as an input. It computes not only σ, but also λ, as a byproduct. It repeats the same 'step' until a stop criterion is satisfied. We index the steps with symbol N, N = 1, 2, etc.

Algorithmic Rule (Calculation of σ and λ of graph G) Suppose that graph $\mathbf{G}'$, the low-order variant of graph **G**, and **B**, the corresponding canonic matrix, are derived as explained earlier in this chapter. At step N, the algorithm calculates completion time vector $\mathbf{x}'(N-1)$ of graph $\mathbf{G}'$ recursively as the product of matrix **B** and vector $\mathbf{x}'(N-2)$. Hereby, vector $\mathbf{x}'(N-2)$ is either derived at the previous step if $N > 1$ or it is equal to the initial conditions if $N = 1$.

Then the algorithm checks whether N has passed the end of the first period, i.e. whether $N = T + W + 1$, see Equality (4.26). Because the values of T and W are not known in advance, this is done as follows. At every step, after obtaining $\mathbf{x}'(N-1)$ we compute the *vector of differences* $\mathbf{y}(N-1)$, whose elements are defined by expression: for any integer n holds $y_k(n) = x'_k(n) - x'_1(n)$. Note that the choice of actor v_1 to be the reference actor is arbitrary. Afterwards, the algorithm compares $\mathbf{y}(N-1)$ with the difference vectors computed at the previous steps: $\mathbf{y}(N-2)$, $\mathbf{y}(N-2)$, $\mathbf{y}(N-3), \ldots \mathbf{y}(-1)$. If there is a $\mathbf{y}(N-W-1)$ that is equal to $\mathbf{y}(N-1)$, then this means that a period $n = N-W-1..N-2$ of length W has been detected. Afterwards, the algorithm computes λ as $\left(x'_1(N-1) - x'_1(N-W-1)\right)/W$. Finally, to compute σ,

the algorithm applies Formula (4.26), whereby the maximization goes over $n = 1..N-1$ instead of $1..W + T$ (which is legal, because we show in a lemma below that $N - 1 = W + T$).

Let us determine the complexity of this algorithm. At the N-th step, one applies Equality (4.5) to compute $\mathbf{x}'(N-1)$ – which costs $O(K^2)$ of computation time – and performs the comparison of the difference vectors. The comparison can be done efficiently by keeping the previous difference vectors in a max-heap – i.e. a data structure that efficiently implements a sorted list (see [16 - §6]). To store vectors in a max-heap one needs to define comparison between two vectors, which can be done e.g. using lexicographic ordering (i.e. first compare the first element of the two vectors, if they are equal then compare the next two, etc.). The search operation for an existing element as well as the insertion of a new element in a max-heap performs $O(\log N)$ comparisons of heap elements. Because the complexity of one comparison is $O(K)$, the total cost for searching an existing vector is $O(K \cdot \log N)$ per each step of our algorithm.

In total, H steps are executed; H is defined by:

$$H \equiv T + W + 1 \tag{4.28}$$

where T and W are the transient and minimum periodic depths of HSDF graph **G** for the given initial conditions.

Therefore the complexity of this algorithm is $O(K^2 \cdot H + K \cdot H \cdot \log H)$.♦

The correctness of this algorithm follows from the following lemma. It shows that the values for W, T, and λ calculated by the above algorithmic rule satisfy Formula (4.21). Having shown this, we in fact show that the interval $n = 1..N-1$ where the algorithmic rule applies the maximization defined in Formula (4.26) is in fact interval $n = 1..T + W$, where T and W are transient and periodic depths of the HSDF graph, as required by that formula.

Lemma 4.9 (Correctness of the calculation of σ and λ) Let $\mathbf{x}'(n)$ satisfy equation $\mathbf{x}'(n) = \mathbf{B} \cdot \mathbf{x}'(n-1)$ for $n \geq 0$. Let $\mathbf{y}(n)$ be defined as $\mathbf{y}(n) = \mathbf{x}'(n) / x_1'(n)$ and λ' be defined as $\lambda' = \left(x_1'(N-1) - x_1'(N-W'-1)\right)/W'$. If there are some integers $W' > 0$ and $N > W'$ such that $\mathbf{y}(N-1) = \mathbf{y}(N-W'-1)$, then Formula (4.21) applies, with $W = W'$, $\lambda = \lambda'$ and $T = N - W' - 1$.♦

Proof Let $k \geq 0$. Then, we have:

$$\begin{aligned}
&\mathbf{x}'(N-1+k) = \mathbf{B}^k \cdot \mathbf{x}'(N-1) = \mathbf{B}^k \cdot \mathbf{y}(N-1) \cdot x_1'(N-1) = \{\text{using}\, \mathbf{y}(N-1) = \mathbf{y}(N-W'-1)\} = \\
&= \mathbf{B}^k \cdot \mathbf{y}(N-W'-1) \cdot x_1'(N-1) = \mathbf{B}^k \cdot \left(\mathbf{x}'(N-W'-1) / x_1'(N-W'-1)\right) \cdot x_1'(N-1) = \\
&= \left(\mathbf{B}^k \cdot \mathbf{x}'(N-W'-1)\right) \cdot (\lambda')^{W'} = (\lambda')^{W'} \cdot \mathbf{x}'(N-W'-1+k)
\end{aligned}$$

To summarize, if W' and N satisfy the lemma conditions, we have:

$$k \geq 0 \Rightarrow \mathbf{x}'(N-1+k) = (\lambda')^{W'} \cdot \mathbf{x}'(N-W'-1+k), \tag{4.29}$$

which is equivalent to Formula (4.21) (when substituting n for $T + k$). ♦

Example (Calculation of σ and λ) Let us compute the iteration interval and the lateness of the graph in Figure 4.2. For that purpose, we use matrix **B** from Equality (4.18). We assume that $\mathbf{x}'(-1) = [e \quad e \quad \ldots]^{\mathbf{T}}$. From this follows: $\mathbf{y}(-1) = [e \quad e \quad \ldots]^{\mathbf{T}}$.

Step 1 (N = 1)

$$\mathbf{x}'(0) = \mathbf{x}'(N-1) = \mathbf{B} \cdot \mathbf{x}'(N-2) = \mathbf{B} \cdot \mathbf{x}'(-1) = [\ 3\ \ 6\ \ 7\ \ 8\ \ 0\ \ 0\ \ 0\ ]^{\mathbf{T}}$$

$$\mathbf{y}(0) = \mathbf{x}'(N-1) / x_1'(N-1) = [\ 3\ \ 6\ \ 7\ \ 8\ \ 0\ \ 0\ \ 0\ ]^{\mathbf{T}} / 3 = [\ 0\ \ 3\ \ 4\ \ 5\ \ -3\ \ -3\ \ -3\ ]^{\mathbf{T}}$$

Step 2 (*N* = 2)

$$\mathbf{x}'(1) = \mathbf{x}'(N-1) = \mathbf{B} \cdot \mathbf{x}'(N-2) = \mathbf{B} \cdot \mathbf{x}'(0) = [\ 6\ \ 9\ \ 10\ \ 11\ \ 6\ \ 7\ \ 8\ ]^{\mathbf{T}}$$

$$\mathbf{y}(1) = \mathbf{y}(N) = \mathbf{x}'(N-1) / x_1'(N-1) = [\ 6\ \ 9\ \ 10\ \ 11\ \ 6\ \ 7\ \ 8\ ]^{\mathbf{T}} / 6 = [\ 0\ \ 3\ \ 4\ \ 5\ \ 0\ \ 1\ \ 2\ ]^{\mathbf{T}}$$

Step 3 (*N* = 3)

$$\mathbf{x}'(2) = \mathbf{x}'(N-1) = \mathbf{B} \cdot \mathbf{x}'(N-2) = \mathbf{B} \cdot \mathbf{x}'(1) = [\ 11\ \ 14\ \ 15\ \ 16\ \ 9\ \ 10\ \ 11\ ]^{\mathbf{T}}$$

$$\mathbf{y}(2) = \mathbf{y}(N) = \mathbf{x}'(N-1) / x_1'(N-1) = [\ 11\ \ 14\ \ 15\ \ 16\ \ 9\ \ 10\ \ 11\ ]^{\mathbf{T}} / 11 = [\ 0\ \ 3\ \ 4\ \ 5\ \ -2\ \ -1\ \ 0\ ]^{\mathbf{T}}$$

Step 4 (*N* = 4)

$$\mathbf{x}'(3) = \mathbf{x}'(N-1) = \mathbf{B} \cdot \mathbf{x}'(N-2) = \mathbf{B} \cdot \mathbf{x}'(2) = [\ 14\ \ 17\ \ 18\ \ 19\ \ 14\ \ 15\ \ 16\ ]^{\mathbf{T}}$$

$$\mathbf{y}(3) = \mathbf{y}(N) = \mathbf{x}'(N-1) / x_1'(N-1) = [\ 14\ \ 17\ \ 18\ \ 19\ \ 14\ \ 15\ \ 16\ ]^{\mathbf{T}} / 14 = [\ 0\ \ 3\ \ 4\ \ 5\ \ 0\ \ 1\ \ 2\ ]^{\mathbf{T}}$$

At this step, the value of $\mathbf{y}(N-1)$ for the first time repeats a value calculated earlier, namely, the one calculated at Step 2. Thus we have identified a period, and $W = 2$.

We have:

$$T = N - W - 1 = 1$$

$$\lambda = \left(x_1'(N-1) - x_1'(N-W-1)\right) / W = (14-6)/2 = 4$$

To calculate σ, we can, for $N = 1, 2, 3$ (thus covering the transient phase and the first period), calculate the difference between the loop execution time Δ_N (obtained from Equality (4.23) and expression $\lambda \cdot (N-1)$. The graph lateness is equal to the maximum value of this difference for $N = 1, 2, 3$ (see Formula (4.26)).

For $N = 1$, $\Delta_N = \max(3,\ 6,\ 7,\ 8,\ 0,\ 0,\ 0) = 8$; $\lambda \cdot (N-1) = 0$. The difference is 8.

For $N = 2$, $\Delta_N = \max(6,\ 9,\ 10,\ 11,\ 6,\ 7,\ 8) = 11$; $\lambda \cdot (N-1) = 4$. The difference is 7.

For $N = 3$, $\Delta_N = \max(11,\ 14,\ 15,\ 16,\ 9,\ 10,\ 11) = 16$; $\lambda \cdot (N-1) = 8$. The difference is 8.

Therefore, $\sigma = \max(8, 7, 8) = 8$. ♦

4.4.3 The Major Algorithmic Rule for Static-delay Analysis

Recall that the major goal of this chapter is to give a tight and conservative upper bound on the performance of static-delay HSDF graphs. In Section 4.4.1, we have chosen $\lambda \cdot (N-1) + \sigma$ as such a bound (for *N* iterations) in terms of the graph execution time. This bound uses the σ and λ characteristics of the graph. In the previous subsection, we have given an algorithm to calculate them, but that algorithm starts from the pre-calculated canonic matrix, **B**. In this subsection, we summarize the algorithmic rules used to obtain these characteristics starting from scratch. Those rules combined together constitute the major algorithmic rule for static-delay analysis.

Algorithmic Rule (The major rule for static-delay analysis)

1. Construct the adjacency matrices $\mathbf{A}_0'$ and $\mathbf{A}_1'$ of the low-order variant of graph **G**. Recall that the complexity is $O(K^2)$, where K is the number of actors in the low-order graph, as given by Equality (4.12).
2. From $\mathbf{A}_0'$ and $\mathbf{A}_1'$, we obtain the canonic matrix $\mathbf{B} = (\mathbf{A}_0')^* \cdot \mathbf{A}_1'$. Recall that the complexity is $O(K^3)$.
3. Compute λ and σ using the algorithm described earlier in this subsection. The complexity is $O(K^2 \cdot H + K \cdot H \cdot \log H)$.

Summing up the complexities of all parts together, we conclude that the total complexity is $O(K^3 + K^2 \cdot H + K \cdot H \cdot \log H)$. ♦

The two major complexity parameters here are thus K and H.

As for parameter K, as already mentioned in the discussion about Equality (4.12), although this parameter includes the total number of initial tokens in the graph, which is a numeric value that can grow exponentially with the size of the HSDF graph specification, for IPC graphs this parameter is polynomial.

As for parameter H, the sum of the transient depth and the minimum periodic depth of the HSDF graph, the situation is more complex. Recall that those depth values depend on the initial conditions and that they are bounded from above by the periodic power and the transient power of the canonic matrix, **B**. Let us consider those upper bounds.

Let us first consider the periodic power, W. From Lemma 4.3, it follows that if all critical cycles in the HSDF graph have the same depth μ, then this also holds for all the critical cycles in the canonic graph. From the results presented in [4 - §3.7], it follows that, in that particular case, W is equal to μ. For example, the graph in Figure 4.2 has only one critical cycle (v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_1). It contains two initial tokens, and we have seen, in our example above, that W is also equal to 2.

However, in the worst case, when different critical cycles of the HSDF graph have different depths, W can be equal to the *product* of the critical cycle depths. This can lead to a high overhead in the calculation of σ using our algorithm. To avoid this situation, one can make a conservative assumption that the delay of one of the actors belonging to one of the critical cycles is slightly higher than it is in reality (one can use Karp's algorithm to find not only λ, but also a critical cycle of the graph in $O(K^3)$ runtime). This will make the resultant estimates of λ and σ slightly pessimistic, but it will ensure W is limited by the depth of one of the critical cycles.

Unfortunately, we are not aware of any general method to limit the upper bound on the other component of H, the transient power, T. In [4 - §3.7] a small example of a 2x2 canonic matrix is shown, whose T can be made arbitrarily high by slightly changing the value of one of the matrix entries. Using that example, our algorithm could be shown to have at least exponential complexity. Fortunately, the same example also brings the good news that in some cases a slight change in one of the delays in the graph can also make T significantly smaller, and therefore one can sometimes use an approach to limit T similar to the one we can use to limit W, but it is not known how to limit T in general.

Nevertheless, in our experiments, we never face the problem of high complexity of calculation of σ. Some of these experiments are reported in Chapter 6. Anyway, this problem deserves investigation in future work.

4.5 Summary and Notes

In this chapter, we use the mathematical apparatus of max-plus algebra and event graphs to give an algorithm to calculate an upper bound on the execution time of static-delay HSDF graphs. The upper bound is tight in the sense that it has a constant maximum error.

In fact, an essential part of our major algorithmic rule that calculates the required upper bound can be seen as doing a simulation of an HSDF graph execution run and letting the simulation continue until a periodic regime is detected. A similar idea is exploited in [23] for performance analysis of SDF graphs, however using a finite-state machine model rather than event graphs. We believe their work can also be adapted to calculate not only λ but σ as well.

In our publications, this approach was first briefly outlined in [75] and then thoroughly described in [76]. In itself, this investigation in the well-studied area of periodic behavior of the HSDF graphs is not a significant contribution, because giving an upper bound on the execution time of a periodic schedule is more or less trivial. Nevertheless, this investigation provides a basis for our performance analysis method for the HSDF graphs with dynamic actor delays, as shown in the next chapter.

5

5 The Dynamic-Delay Analysis

5.1 Delay Quantization

5.1.1 Basic Idea

In Chapter 4, we have considered the timing behavior of HSDF graphs with static delays. The objective of this chapter is to accurately characterize the conservative (i.e. maximal) execution time for an execution run of an HSDF graph whose actor delays $d(v_k, n)$ are not static, but rather variables changing with *n*. Note we still assume that basic IPC-graph properties are satisfied by the HSDF graph – i.e. strong-connectedness, liveness and FIFO order of the token transfers, we only change our assumption about the actor delays. Hereby our purpose is to treat the data-dependent behavior of the application.

We approach the problem as follows. First, we introduce a set of quantization levels for actor delays. We apply quantization to $d(v_k, n)$ for all actors and obtain functions $\hat{d}(v_k, n)$, which we call quantized delay functions. Those are stepwise functions that approximate but stay above $d(v_k, n)$, so they constitute a conservative actor delay model, called the *multi-scenario delay (MSD-) mode*.

The MSD mode of an HSDF graph is one of the timing modes, as introduced in Chapter 3. This particular mode can be described as a state transition system with a scenario space, where the states are called *scenarios*. HSDF evolution from iteration to iteration corresponds to transitions between the scenarios. For each actor, one scenario corresponds to one quantization

level. Thus, as long as the HSDF graph stays in the same scenario, the actor delays are approximated with static values, and thus the static-delay theory applies.

We reuse the static-delay results to characterize the HSDF graph behavior within each scenario. The new part that we introduce here is a method to analyze the timing properties of *transitions* between different scenarios. This method makes it possible to derive a conservative estimate of the execution time and throughput of the HSDF graph with dynamic delays.

Recall that to handle the data-dependent execution delays, in this thesis we use so-called parameters, which are variables counting the data-dependent number of times the application executes the given calculation. Whereas, in Chapter 3, we described the parameters determining the execution delays at the actor level of granularity, the MSD mode opens up a possibility to define the parameters that determine the delays at the level of the whole graph, which we call the *loop-level parameters*. Using an MSD mode we can conservatively express the loop execution time – and hence also the throughput – in terms of linear functions on the loop-level parameters.

Because the MSD mode plays a central role in this chapter, in the following subsections we first introduce the MSD mode formally, and then we outline the contents of the rest of this chapter.

5.1.2 The Multi-scenario Delay (MSD) Timing Mode of an HSDF Graph

Let us introduce actor delay quantization and scenarios, using Figure 5.1 for illustration purposes. In that figure, we assume that the graph has just two actors; for each of them, the figure shows the actor delay as function of iteration index n. For a given execution run, the delay functions are defined for $n = 0..N-1$, where N is the total number of HSDF graph iterations in the execution run. For better illustration, in Figure 5.1 the delay functions, despite being discrete, are shown as solid curves.

As we see in the figure, an MSD mode splits the iteration axis n into *execution intervals*: $\mathbf{I}_1$, $\mathbf{I}_2$, $\mathbf{I}_3$, etc. The number of iterations in one interval is called the *interval depth*, denoted $N_p = |\mathbf{I}_p|$. Thus, axis n is split as follows: $\mathbf{I}_1 = [0..N_1-1]$, $\mathbf{I}_2 = [N_1..N_1+N_2-1]$, etc. Note that the splitting of axis n into intervals $\mathbf{I}_m$ has a similar purpose as the similar splitting of axis x into intervals Δx_i in the definition of the integral in calculus; we are going to characterize the whole execution run by applying a summation over the intervals.

Remark (Relationship between scenarios and execution intervals). To each execution interval $\mathbf{I}_p$, the MSD mode assigns a certain *scenario*, identified by the scenario number $s=s(p)$, where $s(p)$ is a positive integer. We say that interval $\mathbf{I}_p$ *belongs* to scenario $s(p)$. The scenario $s(p)$ is said to be *active* in interval $\mathbf{I}_p$.

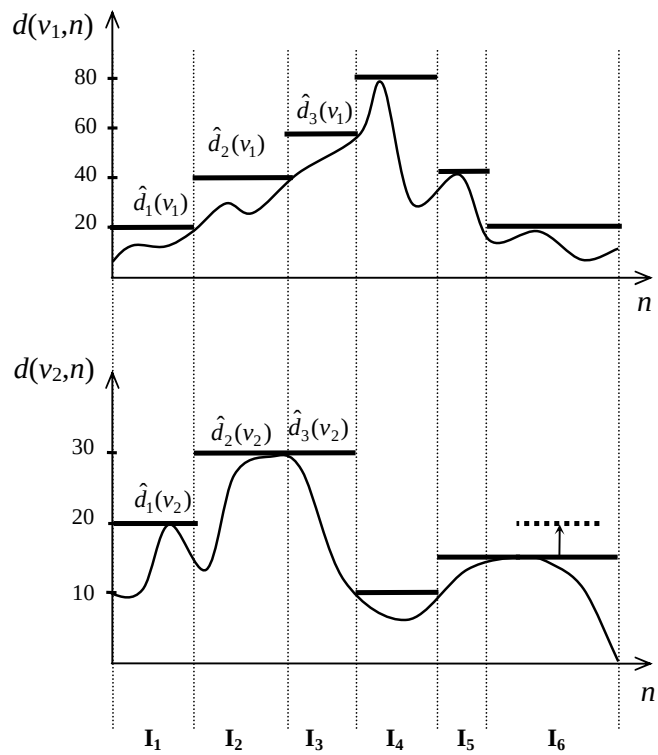
Every scenario s is distinguished by a unique vector of actor *quantization levels*:

$\hat{d}_s = (\hat{d}_s(v_1), \hat{d}_s(v_2), ...)$ ♦

For example, in Figure 5.1 we enumerate a few scenarios identically to the execution intervals, i.e. $s(1) = 1$, $s(2) = 2$, …, $s(6) = 6$, and we have:

$\hat{d}_1 = (20, 20)$; $\hat{d}_2 = (40, 30)$; $\hat{d}_3 = (60, 30)$; $\hat{d}_4 = (80, 10)$; $\hat{d}_5 = (40, 15)$; $\hat{d}_6 = (20, 15)$;

Without loss of generality, we assume that any two subsequent execution intervals $\mathbf{I}_p$ and $\mathbf{I}_{p+1}$ always belong to different scenarios, otherwise we can merge them into one execution interval. Therefore, the scenario transitions take place exactly at the interval boundaries: $s(p) \neq s(p+1)$.



$\hat{d}_s(v_k)$ show the delay quantization levels of scenario *s*, the scenarios here turn out to have the same numbering as the intervals, i.e. $\mathbf{I}_1$, $\mathbf{I}_2$, …

↑ - shows the modification of scenario 6 that would make it equivalent to scenario 1

Figure 5.1 Quantization of actor delays for a two-actor graph

Definition (Actor QD-function). The *quantized-delay function* of the given actor is a stepwise function that in each execution interval takes the value equal to the quantization level of that actor in that interval:

$\hat{d}(v_k, n) = \hat{d}_s(v_k)$, if $s=s(p)$ and $n \in \mathbf{I}_p$ ♦

Definition (Conservative MSD mode). An MSD mode being *conservative* means that the QD-functions of all actors have at least the same magnitude as the real actor delay:

$\hat{d}(v_k, n) \geq d(v_k, n)$, for any $n \in \{0..\, N-1\}$ and for all actors ♦

For example, the MSD mode illustrated in Figure 5.1 is conservative. In the remainder, we consider by default only conservative MSD modes.

Different scenarios correspond to different execution intervals, but not vise versa. For example, in the MSD mode of Figure 5.1, we can change $\hat{d}_6 = (20, 15)$ to $(20, 20)$ – as shown by an arrow. In that case, we can consider 1 and 6 as the same scenario. Thereby, we make the given MSD mode less accurate, but still conservative.

In fact, for practical reasons, we would like the number of scenarios to be limited, such that many execution intervals belong to the same scenario. In the extreme case we have only one scenario and only one interval covering the whole iteration axis. In that case, one can use the worst-case actor delays as the quantization level of that scenario. That is the simplest MSD mode

one can construct for the given execution run. However, for highly dynamic data-dependent applications, the accuracy of a worst-case model can be quite low due to high overestimation of actor delays. The basic intention of an MSD mode is to have an improved accuracy when the number of scenarios is increased. In other words, we achieve *scalability in accuracy*, i.e. one can achieve the required accuracy by selecting the proper number of scenarios.

5.1.3 The Contribution of this Chapter

Recall that the whole purpose of the dynamic-delay graph performance analysis is being able to predict the execution times (i.e. the total duration) of the execution runs using a-priori run-time values of application parameters, characterizing the processing complexity of the input data. In our approach, those parameters characterize the curves of the delay functions of all the actors, like the curves we have seen in Figure 5.1. In our method, the delay quantization realized by the MSD mode helps us reduce the overhead of parameters by letting them characterize the step-wise QD functions instead of the delay functions.

In fact, our concept of MSD timing mode uses the scenario-based paradigm for the characterization of the application behavior [27], [26], [99], [29]. This paradigm is based on the observation that the dynamic behavior of an application is typically composed of a limited number of sub-behaviors, i.e., scenarios, that have similar resource requirements, i.e., similar actor execution delays in the context of this thesis. An extensive overview on scenario-based paradigm can be found in the paper of S. V. Gheorghita *et al* [29].

A scenario-based performance analysis method estimates the execution time via an *algebraic expression* in terms of scenario coefficients, i.e. the contributions of a scenario to the execution time, and scenario parameters, typically variables counting the number of invocations of the scenario. Scenario-based performance analysis has three basic tasks:

- the derivation of the algebraic expression for the execution time,
- *scenario identification*, i.e. defining the set of scenarios and scenario parameters,
- *characterization*, i.e. calculating the scenario coefficients.

In this chapter we develop a scenario-based performance analysis method based on the MSD timing mode, sketched in the previous section. As explained in Section 1.5.1, our method introduces the support of certain essential streaming application features, which is, to the best of our knowledge, not supported in other work on multiprocessor scheduling, such as scenario-based scheduling work of Zhe Ma *et al* [55], [56]. We discuss the closely related work in more detail in the end of this chapter.

Because our analysis works at the level of the application's loop of interest, represented by the HSDF graph, we say that those tasks are carried out at the *loop level*, and we speak of *loop-level coefficients* and *loop-level parameters*, and also of *loop-level characterization* and *loop-level identification*. The corresponding loop-level algebraic expression can be represented as a linear combination of the parameters and coefficients, as we already introduced in a generic form in Section 2.3.3. Recall that the loop-level identification is done in the beginning of our design flow – because it does not depend on the mapping. The loop-level characterization is done partly in the end of the flow and partly at run time (see our flow overview in Section 2.3.4).

In Section 5.2, we present the derivation of the execution time expression and loop-level characterization. We assume that an MSD mode is already given, in terms of quantization levels

and execution intervals. Under that assumption, we apply static-delay performance analysis to every execution interval $\mathbf{I}_p$ and develop a new technique to analyze the boundaries of the execution intervals, where the scenario transitions take place. This helps us to obtain an algebraic loop execution time estimate and to establish an *algorithmic rule for dynamic-delay graphs*, which calculates the loop-level coefficients. Provided that the MSD mode quantization levels are defined and the loop-level parameter values are known, this defines a complete method for run-time estimation of the execution time. The practical usage of this method is demonstrated by our application study in the next chapter.

To be able to apply this method for a given application, first one has to define an MSD mode for any possible execution run. This is the task of loop-level identification, considered in Section 5.3. Although the quantization levels are in general data-dependent, we can still define them at design time, using *subspaces*, i.e. certain intervals of actor-level parameter values. In Section 5.3, we introduce the scenario subspaces in more detail.

Finally, Section 5.4 summarizes and concludes this chapter. There we also discuss the closely related publications of the other researchers as well as our own publications.

Before we continue with the main topic, it is worthwhile to briefly discuss similarity between our loop-level and actor-level timing models. Recall that, in Chapter 3, we consider actor-level linear parameter functions, similar to the loop-level expression we derive in this chapter. Compared to our loop-level analysis, the actor-level analysis may deal with much more complex control flow that a plain loop. On the other hand, our actor-level analysis is restricted to sequential execution, as opposed to the parallel HSDF graph execution studied in this chapter. Thus, it is interesting to notice that, in this chapter, we show how the execution time of certain class of parallel programs such as HSDF graphs can be expressed in a similar linear form as the processing time of sequential programs such as actors.

5.2 Using an MSD Mode for Performance Analysis

5.2.1 Basic Execution Time Estimate as a Parametric Function

Let us consider a loop of interest executing iterations $0..N-1$ of an HSDF graph. In this subsection, we give a algebraic loop execution time estimate using a given MSD model. That estimate does not yet take the scenario transitions properly into account. We call it the *basic expression* for the execution time estimation.

The basic expression is considered here for two main reasons: to reintroduce the concept of the loop-level parametric function (which we introduced in Chapter 2) and to use the basic expression as foundation for obtaining our final execution time expression, used in the algorithmic analysis rule for dynamic-delay HSDF graphs.

In the basic expression, we assume that at the beginning of every execution interval the HSDF graph starts a new execution run and at the end of the execution interval the HSDF graph stops and waits until all actors finish the given execution interval before the next interval may start. Because in reality the actors do not wait for all other actors at the boundaries of the execution intervals, the basic expression is pessimistic, thus giving us a conservative estimate of the loop execution time.

In the basic expression, we use the results of Chapter 4 to estimate the execution time of every interval and then add the results.

Definition. *The instantaneous period* λ_s and *instantaneous lateness* σ_s are the period and the lateness of the HSDF graph with static delays equal to the levels $\hat{d}_s$ of scenario *s*. ♦

According to the given MSD mode, let us split the loop iteration interval into sub-intervals with depth values $N_1, N_2, \ldots, N_P$, $\sum N_p = N$. Then the basic expression is obtained by summation of Equality (4.25) over all intervals:

$$\hat{\Delta}_{N\,basic} = \sum_{p\in[1..P]} (\lambda_{s(p)} \cdot N_p + (\sigma_{s(p)} - \lambda_{s(p)})) \qquad (5.1)$$

where $\lambda_{s(p)}$ and $\sigma_{s(p)}$ are instantaneous period and lateness in scenario $s(p)$.

For practical reasons, it is convenient to group the intervals that belong to the same scenario together and rewrite this summation respectively. Define J_s as:

$$J_s = \sum_{p\in[1..P]:s(p)=s} N_p$$

Thus, J_s is the total number of loop iterations in scenario *s*. Define L_s as:

$$L_s = \left|\{p \in [1..P] \mid s(p) = s\}\right|,$$

Thus, L_s is the total number of execution intervals belonging to scenario *s*.

Grouping the terms of Equality (5.1) by scenario number *s*, we obtain the basic expression the execution time estimation:

$$\hat{\Delta}_{N\,basic} = \sum_{s} (\lambda_s \cdot J_s + (\sigma_s - \lambda_s) \cdot L_s) \qquad (5.2)$$

In this algebraic expression, we see data-dependent parameters (J_s and L_s) and constant coefficients (based on λ_s and σ_s). They are, in fact, examples of the loop-level parameters and loop-level coefficients that we referred to earlier in this chapter. They satisfy the definition given in Section 2.3.3, where we said that the loop-level parameters count the number of loop iterations that have certain properties and the coefficients give the contribution of those loop iterations.

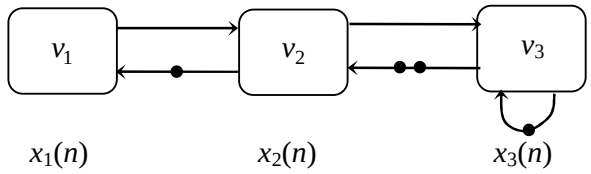
5.2.2 Scenario Transitions: Basic Considerations

In this subsection, we start an investigation of the scenario transitions taking place at the boundaries of execution intervals. Eventually, this investigation leads to an algorithm that estimates the execution time with an essentially improved accuracy compared to the basic expression.

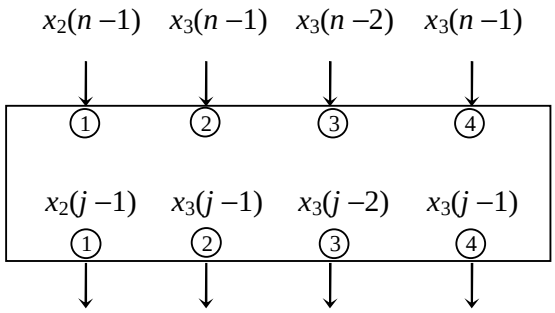
The transitions take place between the iterations of the HSDF graph. Therefore, inter-iteration dependencies play an important role there. Those dependencies are reflected in the graph as initial tokens.

Consider the HSDF graph in Figure 5.2 (a). It has three actors, whose completion time variables are x_1, x_2, x_3. Consider an arbitrary execution interval $[n...j-1]$, where $j > n$, represented in Figure 5.2 (b) as a box with multiple inputs and outputs.

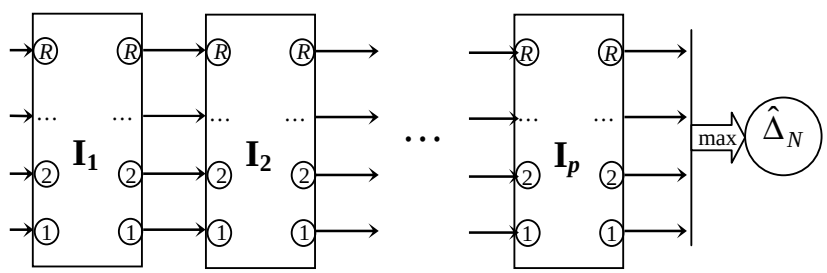
The inputs of this box correspond to dependencies of the actor executions within the box on the iterations before *n*. Let us consider those dependencies one-by-one for the example given in Figure 5.2(a). Consider actor v_1. Its evolution equation depends on $x_2(n-1)$ due to the initial token on edge (v_3, v_2). So, we label the first input of the box as $x_2(n-1)$. Look at actor v_2 and



(a) HSDF graph example and actor completion time variables



(b) Execution range box for range n .. $j-1$ for the graph under (a)



(c) Combining the execution range boxes together
R is the number of initial tokens in the graph

Figure 5.2 The framework for taking the scenario transitions into account in $\hat{\Delta}_N$

consider its executions $x_2(n)$ and $x_2(n+1)$. They both may fall within the interval of the box[1], and they consume tokens produced by events $x_3(n-2)$ and $x_3(n-1)$ respectively[2] due to two initial tokens on edge (v_2, v_1). Therefore, we introduce two more inputs with the corresponding labels. Finally, the initial token on the self-edge of v_3 introduces a new box input labeled $x_3(n-1)$. Thus, we have two inputs labeled by the same event $x_3(n-1)$, but we separate them because they represent the dependencies of two different consumer actors, namely v_2 and v_3. In general, one initial token in a graph corresponds to one dependency at the box input. The same can be said about the box outputs, where we can just replace n by j in the labels (see Figure 5.2(b)). We enumerate the initial tokens – and hence the box inputs and outputs as well –

[1] In fact, $x_2(n+1)$ only falls into this range, in case $j>n+1$, and it does not in case $j=n+1$. But considering all possible dependencies makes our model conservative, because, as we will see later, taking any extra dependency into consideration can only make our execution time estimate more pessimistic.
[2] Those dependencies are there only if $n>0$ and $n>1$ respectively, but the same remark as before applies here.

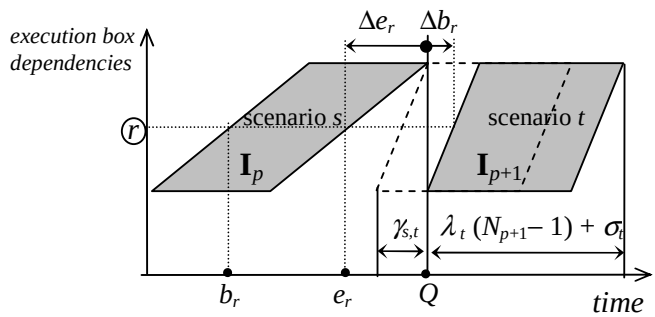


Figure 5.3 Minimum overlap of the time shapes

in an arbitrary order by index r with values 1..R, where R is the total number of the initial tokens in the graph. For example, in Figure 5.2 (a), $R = 4$.

Such an 'execution interval box' is similar to an actor, because it consumes tokens at the inputs and produces them at the outputs. However, unlike an actor, a box does not have to capture all input tokens simultaneously, neither does it have to simultaneously produce all output tokens. The tokens are produced at the times defined by the labels assigned to the box dependencies.

Now suppose we have execution intervals $\mathbf{I}_1$, $\mathbf{I}_2$, $\mathbf{I}_3$,… $\mathbf{I}_P$ for the given execution run of the loop of interest. Then we can introduce a separate box for each interval and connect the boxes in one chain, as shown in Figure 5.2 (c). According to the synchronous initial conditions, we put zeros at the box inputs of $\mathbf{I}_1$ and calculate its outputs. This way, one can propagate the results through the chain from $\mathbf{I}_1$ to $\mathbf{I}_P$.

To give a conservative estimate of the loop execution time $\hat{\Delta}_N$, we have to provide an upper bound on the latest output of $\mathbf{I}_P$. If there was always only one interval, $P = 1$, the answer would be: $\lambda_{s(1)} \cdot (N_1 - 1) + \sigma_{s(1)}$. For more intervals, we consider each pair of subsequent intervals – $\mathbf{I}_p$, $\mathbf{I}_{p+1}$. The behavior of $\mathbf{I}_{p+1}$ depends on when the execution interval box $\mathbf{I}_p$ produces a token at each output. If $\mathbf{I}_p$ produced all tokens simultaneously at a certain time Q_p, then it would hold that $\mathbf{I}_{p+1}$ would complete all its iterations by time $Q_{p+1} = Q_p + \lambda_{s(p+1)} \cdot (N_{(p+1)} - 1) + \sigma_{s(p+1)}$. Then making a conservative assumption that all output tokens of the box $\mathbf{I}_{p+1}$ are produced at time Q_{p+1}, we could apply similar reasoning and calculate Q_{p+2} from Q_{p+1} , Q_{p+3} from Q_{p+2} , etc.

However, this kind of calculation, would again yield the basic expression, (5.1). The disadvantage of that calculation is that we assume that the execution interval boxes produce all output tokens simultaneously, in an actor-like manner. In general, tokens at different outputs can be produced at different moments of time. As a consequence, when box $\mathbf{I}_p$ releases a token at some outputs and still continues running, box $\mathbf{I}_{p+1}$ may pick up a token at the corresponding inputs and start running before interval $\mathbf{I}_p$ finishes its execution. In other words, there is a *timing overlap* between subsequent execution intervals.

An idea of how to conservatively estimate that overlap is illustrated in Figure 5.3. Two filled parallelograms shown in the figure represent so-called *time shapes* of two subsequent execution intervals. The vertical axis corresponds to index r that enumerates the R dependencies between the execution intervals. Thus, the vertical axis is discrete, although our drawings ignore this fact. The horizontal axis corresponds to physical time. As shown in the figure, a horizontal section of a time shape is a time interval between two events: the begin event b_r, which corresponds to the capturing of a token at box input r, and the end event e_r, corresponding to the production of a

token at the box output *r*. In general, time shapes are not necessarily parallelograms; we drew them like that for illustrative purposes.

Figure 5.3 shows how the time shapes are arranged according to the assumptions of the basic expression, (5.1). The main property of this arrangement is that we can distinguish a reference time point *Q* where one can draw a vertical line through the latest e_r in execution interval $\mathbf{I}_p$ and the earliest b_r in execution interval $\mathbf{I}_{p+1}$. In such an arrangement, there is a gap between the time shapes. If we redrew these shapes in Figure 5.3 for the real HSDF execution, there would be no gap at all. We implicitly introduced such a gap in the basic execution time estimate. This was a legal thing to do, because delaying the events b_r of $\mathbf{I}_{p+1}$ in a monotone timing model like ours leads to conservative results. In return, our basic expression was relatively simple.

We reduce the gap between the time shapes by shifting $\mathbf{I}_{p+1}$ to the left as far as we can such that the shapes do not overlap each other. The shifted position of the time shape $\mathbf{I}_{p+1}$ is shown in Figure 5.3 with dashed lines. The absence of overlaps between the shapes implies that we can shift the shape safely, i.e. without violating the dependencies between the shapes, because all possible dependencies are represented in the time shapes. This way, the execution interval $\mathbf{I}_{p+1}$ does not 'notice' the shifting and its time shape remains intact.

We denote the shift distance γ. Let us see how it can be calculated. For dependency *r*, let Δe_r be the position of $\mathbf{I}_p$'s right border relative to the reference point *Q*. Let Δb_r be the relative position of $\mathbf{I}_{p+1}$'s left border. From Figure 5.3, it is obvious that time shape $\mathbf{I}_{p+1}$ can be shifted to the left by at most:

$$\gamma = \min_{r \in 1..R} (\Delta e_r + \Delta b_r) \tag{5.3}$$

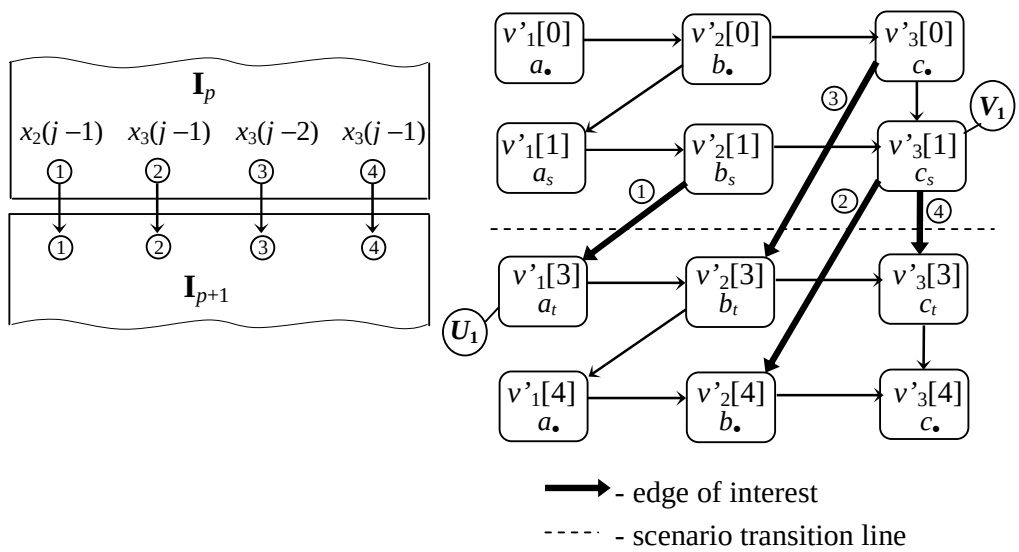
The value of γ has the meaning of time overlap between the time shapes. Because one can overlap the time shapes by at least[28] γ, we call it the *minimum overlap at the scenario transition*. It plays a key role in our execution time estimation for the dynamic-delay case. Just as λ and σ, this characteristic can be derived by analyzing the paths through the HSDF graph, as presented in the next subsection.

5.2.3 Minimum Overlap: Graph Analysis

The minimum overlap of a scenario transition depends on the current scenario and the next scenario; let us denote the scenarios as *s=s(p)* and $t = s(p+1)$. We calculate the overlap directly, by applying Equality (5.3); therefore we need to calculate the relative positions of the time shape borders Δb_r and Δe_r for each *r*.

It is important to stress here that in this subsection we assume that the HSDF graph executes with the actor delays defined by the quantized-delay function $\hat{d}(v_k, n)$, not the function $d(v_k, n)$ that represents the actual actor delays. This means that the shape borders, Δb_r and Δe_r, and the minimum overlap, γ, are calculated in a conservative way – i.e. small enough – with respect to the quantized-delay execution. Note that when considering the actual execution delay, the overlap between the execution intervals may be even much smaller than the γ that we obtain here. This is still legitimate, because our final purpose is to give an algorithmic rule to calculate

[28] In general, one can shift $\mathbf{I}_{p+1}$ even further, but then it will 'notice' the change in the initial conditions and its time shape will deform.



(a) Border between execution range boxes **(b)** Transition graph, $\mathbf{G}'$

Figure 5.4 Transition analysis example for the HSDF in Figure 5.2(a)

the *loop execution time*, and our estimate, being obtained for the quantized-delay execution, is still conservative for the actual-delay execution.

We have already explained the basic idea of the time shape borders using Figure 5.3. In this subsection, to calculate the relative positions of the borders Δb_r and Δe_r, we give more rigorous definitions of those values. Because they in fact represent the time distances between certain events, they correspond to the lengths of certain paths through the HSDF graph. We show that by first applying the so-called *trimmed unfolding* to the HSDF graph and then doing a longest path analysis on the result of the trimmed unfolding.

Let us first define and illustrate the trimmed unfolding and then explain its use in our context. The trimmed unfolding of HSDF graphs is based on the conventional unfolding, defined in Section 3.6.3.

Recall that the graph resulting from the conventional graph unfolding is called the *unfolded representation*. Let $\mathbf{G}'$ be the unfolded representation of graph $\mathbf{G}$ and H the chosen unfolding factor. Then every actor v_k in $\mathbf{G}$ is represented by H actors in the unfolded representation: $v'_k[0]$, $v'_k[1]$, **…,** $v'_k[H-1]$. Those actors are joined by edges according to the rules defined in Lemma 3.3. For example, the graph in Figure 3.18(b) is the unfolded representation of the graph in Figure 3.18 (a) with unfolding factor $H = 5$.

The *trimmed unfolded representation* can be obtained from the 'conventional' unfolded representation by removing all the edges containing initial tokens. As a result we obtain an acyclic graph where all edges have marking 0. For example, Figure 5.4(b) shows the trimmed unfolded representation (with unfolding factor 4) of the HSDF graph in Figure 5.2(a).

We give the name *transition graph* to the trimmed unfolded representation of the HSDF graph obtained for the purpose of analyzing the time shape boundaries Δb_r and Δe_r. Comparing Figures 5.4(a) and (b), we see that the transition graph gives a detailed view of the dependencies between boxes $\mathbf{I}_p$ and $\mathbf{I}_{p+1}$. The transition graph can be partitioned into two parts, the 'upper' part, representing execution interval $\mathbf{I}_p$ and the 'lower' part, representing execution interval $\mathbf{I}_{p+1}$. The upper part contains the nodes that model the actor executions before the transition and the

lower part models the actor executions after the transition. Therefore, we refer to the cut-line that partitions the graph as the *transition line* (a dashed line in Figure 5.4(b)).

Let M be the maximum number of initial tokens on any edge of the HSDF graph $\mathbf{G}$. The output tokens of $\mathbf{I}_p$ can be produced up to M iterations backwards from the transition, and the box inputs of $\mathbf{I}_{p+1}$ can be consumed up to M iterations forwards[29]. Therefore, to capture all possible dependencies, the transition graph has unfolding factor $H = 2M$, covering the interval $n = j - M \ldots j + M - 1$, where j is the first execution index in $\mathbf{I}_{p+1}$. For our example in Figure 5.2(a), $M = 2$, and therefore every actor v_k is represented in the transition graph in Figure 5.4(b) by four actors $v'_k[0]$, $v'_k[1]$, $v'_k[2]$, and $v'_k[3]$. We use the trimmed unfolding because we can ignore the dependencies of the actor executions in the specified index interval on the executions outside that interval, still being conservative.

Note that the transition graph construction implicitly assumes that $M > 0$. However, this property always holds for the graphs that satisfy basic IPC properties, because strong-connectedness and liveness imply that the HSDF graph contains at least one cycle having at least one initial token.

So far, we have introduced only the *structure* of the transition graph, but not yet the *actor delays* in that graph. We also need to specify them before we can use the transition graph to calculate Δe_r, Δb_r, and γ.

The delays in the transition graph have to be either equal to the values of the quantized-delay function $\hat{d}(v_k, n)$ in the interval $n = j - M \ldots j + M - 1$ or represent those values in a conservative way. In fact, we do not fill exact values of $\hat{d}(v_k, n)$ into the transition graph, because this would require information on not only which pair of scenarios, *s* and *t*, is involved in the transition, but also which scenarios come before scenario *s* and which scenarios come after scenario *t* within the inspected interval of index *n*. That would complicate the use of the transition graph for the calculation of time shape borders, whereby considerable overhead would be involved in terms of the required input information. Therefore we choose to abstract from the detailed delay values by rather using their conservative estimates.

Let us define the delays of the transition graph in a conservative way. We denote them as $d_{trans}(v'_k[f])$, where $v'_k[f]$ is an actor in the transition graph and $f = 0 .. 2M - 1$ is its *unfolding index*, representing the loop iteration with index $n = j - M + f$. Note that the transition line separates the transition graph actors with $f < M$ from the actors with $f \geq M$. When assigning the delay values to d_{trans} we need to remember that we can only be sure that the loop iteration with unfolding index $f = M - 1$ belongs to scenario *s* and iteration $f = M$ belongs to scenario *t*, because the depths of intervals $\mathbf{I}_p$ and $\mathbf{I}_{p+1}$ are at least one. It is not known which scenarios are active further than one iteration away from the transition line, so we have to fill in conservative values there. Remember that the goal of model d_{trans} is the calculation of Δe_r and Δb_r. For Δe_r and Δb_r, 'conservative' means 'small enough'. Therefore in that case we fill in the minimal quantization delay values from all scenarios. So, we have:

$$d_{trans}(v'_k[f]) = \hat{d}_{\bullet}(v_k), \; f < M - 1 \quad \text{or} \quad f > M \tag{5.4.1}$$

[29] In case the depth of the ranges $\mathbf{I}_p$ or $\mathbf{I}_{p+1}$ is less than *M*, the number of actual dependencies between them is less than *R*, but filling this number always up to *R* in Equality (5.3) keeps the estimate of γ conservative, because this can make the value of γ only smaller (and thus more conservative)

where $\hat{d}_{\bullet}(v_k) = \min_q \left(\hat{d}_q(v_k)\right)$

$$d_{trans}(v'_k[M-1]) = \hat{d}_s(v_k) \tag{5.4.2}$$

$$d_{trans}(v'_k[M]) = \hat{d}_t(v_k) \tag{5.4.3}$$

The notations used for actor delays in the example of Figure 5.4(b) refer to the values of $d_{trans}(v'_k[f])$ complying with equalities (5.4), e.g. $a_{\bullet} = \hat{d}_{\bullet}(v_1)$, $a_s = \hat{d}_s(v_1)$, $a_t = \hat{d}_t(v_1)$.

Based on these delay values, let us calculate the time shape borders Δb_r and Δe_r. First of all, we observe that there is a one-to-one correspondence between each edge crossing the transition line and dependency *r*. We call such edges the *edges of interest* and show them in Figure 5.4(b) using bold arrows. We index them with index *r* as well.

Value Δb_r is the 'as soon as possible' (*asap*) time when the consumer node of edge *r* is ready to consume a token. The *asap* time is relative to the time when the lower part of the graph starts its first actor execution. To calculate this *asap* time, we find the nodes in the lower part of the graph that are the first to start, referring to them as the *sources of interest* U_i. A source of interest is recognized by the property that it has solely edges of interest as incoming edges. For example, in Figure 5.4(b) the only source of interest is $U_1 \equiv v'_1[2]$. The *asap* time of a (consumer) node $v'_k[f]$ in the lower part of the transition graph is equal to the largest delay of a graph path from any source of interest to node $v'_k[f]$, not including the delay of that node. For example, the *asap* time of node $v'_2[3]$ is $a_t + b_t + a_{\bullet}$, and it is equal to Δb_2, because $v'_2[3]$ is the consumer of edge 2.

The right boundary Δe_r can be calculated by the same line of reasoning, except that we look at the upper part of the graph, we calculate the 'as late as possible' (*alap*) relative times, we use *sinks of interest* V_l, which have solely edges of interest as outgoing edge, and the paths propagate from the producer node of the edge of interest (not including its delay) to a sink of interest. In our example, node $V_1 \equiv v'_3[1]$ is the only sink of interest and, for example, Δe_1 is the *alap* time of node $v'_2[1]$, which is equal to c_s.

Having calculated all relative time shape boundaries Δe_r and Δb_r, it is straightforward to calculate the minimum overlap value, γ, using Equality (5.3). For the example in Figure 5.4, we examined each edge of interest in Figure 5.4(b) and calculated *alap* value Δe_r for its producer and *asap* value Δb_r for its consumer. Table 5.1 summarizes the results.

Table 5.1 *Asap* and *alap* values for the transition graph in Figure 5.3

r	**From**	**To**	Δe_r	Δb_r
1	$v'_2[1]$	$v'_1[2]$	c_s	0
2	$v'_3[1]$	$v'_2[3]$	0	$a_t + b_t + a_{\bullet}$
3	$v'_3[0]$	$v'_2[2]$	c_s	a_t
4	$v'_3[1]$	$v'_3[2]$	0	$a_t + b_t$

We see that table rows 2 and 3 cannot influence the minimum overlap in Equality (5.3) because rows 4 and 1 have smaller sums of columns Δe_r and Δb_r. The result of the minimum overlap analysis is thus: $\gamma = \min(c_s, a_t + b_t)$.

In the next two subsections, we summarize the minimum overlap analysis and update the loop execution time estimate, which directly yields the algorithmic rule for the performance analysis in the dynamic-delay case, being the major purpose of this section.

5.2.4 Calculating the Minimum Overlap Values for a Multi-scenario-delay Mode

In the previous subsection, we have, in fact, proposed a method to conservatively calculate the minimum overlap between the execution intervals of the given MSD mode in a way that is independent of the positions of the intervals in the execution run and the depths of the intervals. This is so, because γ only depends on the value of delays filled into the transition graph and those delays only depend on which pair of scenarios – *s*, *t* – is being considered. To be more precise, for a transition from scenario *s* to scenario *t*, the *asap* values Δb_r depend only on scenario *t* and the *alap* values Δe_r depend only on scenario *s*. Therefore, as already mentioned, the minimum overlap defined by Equality (5.3) depends only on *s* and *t*. Thus, it is valid to speak of the *minimum overlap* between scenario *s* and scenario *t*, denoted $\gamma_{s,t}$.

For a given MSD mode, one can rewrite Equality (5.3) as follows:

$$\gamma_{s,t} = \min_{r \in 1..R} (\Delta e_r(s) + \Delta b_r(t)) \qquad (5.5)$$

The fact that this estimate of the timing overlap depends only on the pair of subsequent scenarios simplifies the performance analysis, because, to calculate all the overlaps between the execution intervals in a given execution run, one only needs to consider all pairs of scenarios rather than all the transition points of that run. This can be very beneficiary from the calculation complexity point of view, especially if the execution run is long enough and the number of scenarios is small.

To calculate *asap* and *alap* values in the given transition graph, one can apply a longest path calculation algorithm. Because the transition graph is acyclic, the algorithm can be based on topological sorting [16]. Because the number of edges and actors in the transition graph is at most $2ME$ and $2MV$, the topological sorting algorithm complexity is $O(M(V+E))$, where *V* and *E* are the number of actors and edges of the main HSDF graph, and *M* is the maximum number of intial tokens on any edge of the HSDF graph **G**.

Therefore, the total algorithmic complexity to calculate all overlap values in the given multi-scenario-delay timing mode of an HSDF graph is $O(S\,M\,(V+E) + S^2R)$, where *S* is the number of scenarios . This expression includes the complexity of first calculating the *asap* and *alap* values for all the scenarios and then applying Equality (5.5) for every pair of scenarios. If required, one can reduce this complexity by excluding the scenario pairs for which transitions occur only rarely or never at all and assuming that $\gamma_{s,t}$ for such pairs is equal to 0 (which would lead to only a limited loss of accuracy).

Note that a disadvantage of our current approach is the possibility of an estimate of the timing overlap that is too conservative when the HSDF graph has multiple sinks of interest, because of the (implicit) assumption that all sinks of interest complete their executions simultaneously. We consider an improvement of our technique for such cases as a topic of future work.

5.2.5 The Algorithmic Rule for the Dynamic-delay Analysis

To obtain our final version of the loop execution time estimate for a given MSD mode, we subtract the overlaps of the transitions from the basic execution time estimate, given by Equality (5.2). As the result, we obtain:

$$\hat{\Delta}_N = \sum_{s \in \mathbf{S}} (\lambda_s \cdot J_s + (\sigma_s - \lambda_s) \cdot L_s) - \sum_{(s,t) \in \overline{\mathbf{S}}} \gamma_{s,t} \cdot K_{s,t} \qquad (5.6)$$

where **S** = { 1 .. S } is the set of scenario indices, $K_{s,t}$ is the number of transitions from scenario s to scenario t, and $\overline{\mathbf{S}}$ is the set of distinct scenario pairs: $\overline{\mathbf{S}} = \{(s,t) \mid 1 \le s,t \le S,\ s \ne t\}$. Just as J_s and L_s, parameter $K_{s,t}$ is a loop-level parameter, depending on the application input data, obtained from the frame headers of the application. Therefore, we can see that $\gamma_{s,t}$ is a loop-level coefficient. Just as the other loop-level coefficients, σ_s and λ_s, this coefficient is calculated by applying an analysis algorithm to the HSDF graph.

Parameters L_s depend on parameters $K_{s,t}$ and therefore parameters L_s do not need to be provided explicitly. We have:

if $s \ne s(1)$ => (i.e. if s is not the scenario of the first interval)

$$L_s = \sum_{q \in \mathbf{S} \setminus \{s\}} K_{q,s} \qquad (5.7.1)$$

if $s = s(1)$ =>

$$L_s = 1 + \sum_{q \in \mathbf{S} \setminus \{s\}} K_{q,s} \qquad (5.7.2)$$

At this point, we can formulate the algorithm for estimating the loop execution time.

Algorithmic Rule (Dynamic-delay Analysis Rule)

Given:

- the set of delay vectors for each scenario: $\hat{d}_s = (\hat{d}_s(v_1), \hat{d}_s(v_2), ...)$;
- the scenario of the first interval: $s(1)$;
- loop-level parameters: $K_{s,t}$ and J_s.

Then, one can estimate the loop execution time as follows:

1. Use equalities (5.7.1) and (5.7.2) to obtain L_s.
2. Apply the major static-delay rule to calculate loop-level coefficients σ_s and λ_s – see Section 4.4.3.
3. Apply the minimum overlap calculation algorithm to calculate loop-level coefficients $\gamma_{s,t}$.
4. Apply Equality (5.6)

The algorithmic complexity of this rule is:

$$O\left((K^3 + K^2 H_S + K\, H_S \log H_S) + SM(V+E) + S^2 R\right)$$

where H_S is the maximum value of parameter H in all scenarios of the given MSD mode; parameters K and H are defined in Section 4.4.3 as the number of rows (or columns) in the canonic matrix **B** and the number of the lateness calculation algorithm iterations respectively. ♦

Although, in theory, the obtained worst-case algorithmic complexity may in some cases indicate a large calculation overhead, in practice the overhead can often be kept limited. We already discussed the overhead related to parameters K and H (and thus H_S) in Chapter 4. For any IPC graph, parameters M (the maximum number of initial tokens on a single edge) and R (the total number of initial tokens on the graph edges) are polynomial in the number of vertices in the graph, V, because, as discussed in Section 3.7.2, the total number of initial tokens in any cycle of an IPC graph is, in the worst case, linear in V. As for the number of scenarios S, it depends on the construction of the MSD mode, discussed in Section 5.3, where we argue that S also can be limited in practice.

5.2.6 The Throughput of the Dynamic-delay HSDF Graph

Before we finish this section, let us use the obtained execution time estimate to give a conservative estimate for the *throughput*, an important performance metric for streaming applications. Recall from Section 2.2.5 that, in general, for HSDF graphs with dynamic delays, one cannot give a practical way to derive a conservative throughput estimate for an infinite HSDF execution run from the statistical characteristics of the execution delays of actors. In other words, for streaming applications in general, it is an open problem to give an accurate lower bound on the mathematical expected value of the throughput from the characteristics of the probability distribution of the actor delays, whereby the calculation overhead to compute that bound should be reasonable.

To contribute to the research in that direction, in this subsection, we give a conservative throughput estimate under the assumption that the dynamic delays can be accurately characterized by an MSD mode with known parameter values and quantization delay levels.

Just as in Section 2.2.5, let us denote the number of output data bytes produced by the application per HSDF iteration as $z(\mathbf{G})$. Then, for a finite execution run of N iterations, the throughput in bytes per second, denoted $\langle \theta_N \rangle$, is equal to $z(\mathbf{G}) \cdot N / \Delta_N$, where Δ_N is the execution time. Using the conservative execution time estimate given in Equality (5.6), we obtain the following estimate of the throughput for a finite execution run:

$$\left\langle \hat{\theta}_N \right\rangle = z(\mathbf{G}) \cdot \frac{N}{\sum_{s \in \mathbf{S}} \lambda_s \cdot J_s + (\sigma_s - \lambda_s) \cdot L_s - \sum_{(s,t) \in \bar{\mathbf{S}}} \gamma_{s,t} \cdot K_{s,t}} \tag{5.8}$$

This estimate is conservative, i.e. the real throughput is at least $\left\langle \hat{\theta}_N \right\rangle$.

We can extend this finite-execution-run expression to the case of an infinite execution run provided that the following characteristics of the actor delays are known:

$p_J(s) = \lim_{N \to \infty} J_s / N$, i.e., the probability that the application runs in scenario 's'

$p_L(s) = \lim_{N \to \infty} L_s / N$, i.e., the probability that a transition occurs from scenario 's' to another scenario.

$p_K(s,t) = \lim_{N \to \infty} K_{s,t} / N$, i.e. the probability of a transition from scenario 's' to scenario 't'.

If these characteristics are known, then an lower bound on the long-run application throughput is:

$$\left\langle \hat{\theta} \right\rangle = z(\mathbf{G}) \cdot \frac{1}{\sum_{s \in \mathbf{S}} \lambda_s \cdot p_{\mathrm{J}}(s) + (\sigma_s - \lambda_s) \cdot p_{\mathrm{L}}(s) - \sum_{(s,t) \in \overline{\mathbf{S}}} \gamma_{s,t} \cdot p_{\mathrm{K}}(s,t)} \tag{5.9}$$

This equality can be seen as a generalization of Equality (2.7) for multiple scenarios. It gives a fundamental relationship between the timing properties of the individual actors and the performance of the whole graph. Unfortunately, in general, it does not give a complete answer to the problem of obtaining a practical long-run throughput estimate for streaming applications, due to two main limitations:

1. This approach assumes that probabilities $p_{\mathrm{J}}(s)$, $p_{\mathrm{L}}(s)$, and $p_{\mathrm{K}}(s,t)$ exist (in the sense that the mathematical limits defining them exist) and that they can be calculated beforehand.
2. This approach can only be accurate if the MSD mode being employed in the calculation is a good approximation of the real actor delays. In general, this can only be true if the number of scenarios is large enough, whereas the calculation complexity grows quadraticaly in the number of scenarios, which leads to an increase in the calculation overhead.

However, as we already stressed in Section 2.2.6, the main focus of this thesis is a simpler problem: to analyze the performance of finite executions runs whose characteristics of interest are provided a-priory, in the frame headers of the input data streams. Thus, we assume that the exact values of the probabilities (which, for finite runs, are always defined) are provided a-priori, hereby overcoming the first limitation mentioned above. Also, in many practical situations, we can also overcome the second limitation because we do not have to define the actor delay quantization levels statically, but can do that dynamically, based on a-priori information; therefore a working set of scenarios can be constantly updated at run time to only include a limited number of scenarios needed for the current frame. The determination of the working scenario set is discussed in the next subsection.

5.3 Loop-level Identification of Scenarios

5.3.1 The Goals

In the previous section, we assumed that an MSD mode is already defined. In this section, we consider the problem of defining an MSD mode. This means specifying the number of scenarios, S, and, giving an algorithm that calculates the quantization levels $\hat{d}_s(v_k)$ and execution intervals $\mathbf{I}_p$ for the any possible application input data processed by the execution run.

As we already mentioned earlier, in our design flow, the step that addresses this problem is referred to as loop-level identification, which belongs to the Application Preparation part of the flow. What is exactly meant under defining the scenarios for the given application's loop of interest is explained later in this section. However, informally, we can interpret this step as splitting the set of possible loop iteration behaviors into subsets and identifying every subset as a distinct scenario s. We also say that this step (indirectly) identifies all the loop-level parameters, by giving each of them an application-specific meaning. For example, the meaning of parameter J_s is that it counts the number of times a behavior in subset s occurs in the execution run.

In our work, the behavior of every scenario is defined as a subspace of values of the actor-level parameters, ξ_ω. We refer to our scenario definition approach as *scenario subspace approach*. Although having a way to define the scenario behavior, we do not have an own algorithmic solution method for efficient identification of scenarios. However, the research work of S. V. Gheorghita *et al*, [28], [24], proposes an automated solution method for this problem that can be adapted for our design flow, and we are not aware of any alternative solutions. They also develop a scenario subspace approach that is very similar to ours.

In Sections 5.3.2 – 5.3.4 we explain the main idea of the scenario subspace approach, which is essential for the practical use of our performance analysis work. Differently from [28], which is designed for single-task (i.e. single-actor) applications, we also discuss the issues relevant to multi-task (i.e. multi-actor) applications. In Section 5.3.5 we explain how the scenario identification method of [28] can be adapted for our design flow. In Section 5.3.6, we make a summary on how the scenarios are defined and used in our run-time performance analysis framework.

5.3.2 Basic Issues

Before we explain the scenario subspace approach, let us first discuss the basics of defining an MSD timing mode of an HSDF graph. Instead of first determining the execution intervals and then defining the quantization levels for them, as we did in Figure 5.1 for better illustration, we specify the MSD mode the other way around. We introduce the scenarios and their quantization levels first, and then, from the quantization levels, the execution intervals can be determined. In this way, the number of scenarios is determined directly, which makes it easier to ensure that the model does not have too many scenarios, thus being usable in practice. The number of scenarios should be large enough to provide for enough performance analysis accuracy and small enough to keep the overhead of scenarios and scenario transitions limited, (The overhead includes the graph analysis for calculating the loop-level coefficients and bit overhead for encoding the parameters in the frame headers.) In this section, we provide some hints on how to reach that goal in many cases.

A simple quantization method is to enforce the quantization step for the delay of all actors to be a constant value. By selecting the magnitude of that constant, one can directly control the quantization error. Unfortunately such an MSD mode would apply to only one particular hardware architecture and scheduling, because it relies on physical time values of the delays. As a consequence, to specify the values of loop-level parameters J_s and $K_{s,t}$ to be placed in the frame headers of the input data stream, one would have to measure all actor delays of the given execution run for the given implementation and round them up to the predefined quantization levels, this way identifying the scenarios and scenario transitions. This would make the loop-level parameters only valid for the given hardware architecture and scheduling, making them useless for the other implementations, which is not acceptable.

What also should be avoided is defining a separate scenario for each possible combination of the quantization levels. In this case, if we denote the number of quantization levels per actor as Q, then the number of scenarios would be expressed as $Q^{V_{\text{DATA-DEP}}}$, where $V_{\text{DATA-DEP}}$ is the number of HSDF graph actors with data-dependent delays. For example, having 5 levels in a 3-actor graph would result in $5^3 = 125$ scenarios, and $5^3(5^3 - 1) = 15500$ scenario transitions. Although the loop-level coefficients could be calculated at design time, the amount of calculations and the

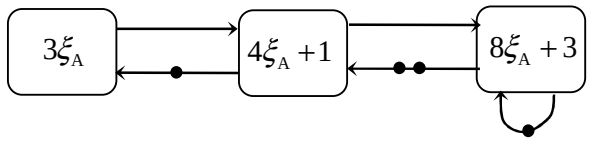


Figure 5.5 Mutually dependent actor delays: an ideal case

sheer number of coefficients could easily go beyond the practical limits. In the situation where different actors have mutually independent delays, it is not possible to avoid the exponential growth of the number of scenarios. Fortunately, in practice, we can expect that the delays of different actors depend on each other because all the actors are involved in processing of the same stream of data. Our 'scenario subspace' approach relies on the use of actor-level parameters to make the dependency between the actor delays explicit.

5.3.3 The Advantage of Using the Actor Parameters in Delay Quantization

Let us consider a simple example illustrating how the actor-level parametric functions can expose the correlation between actor delays. We illustrate it by considering the idealistic case where different delays depend on the same parameter. An example is shown in Figure 5.5. There, all actor delays are linear in the same parameter ξ_A – in practice that can be the case if each actor consists of a loop that executes ξ_A iterations. Suppose that [0, $\xi_{A\text{-max}}$] is the interval of possible values of ξ_A. Introducing five quantization levels on ξ_A means splitting this interval into five integer intervals: $[0, \xi_{A1}]$, $[\xi_{A1}+1, \xi_{A2}]$, $[\xi_{A2}+1, \xi_{A3}]$, $[\xi_{A3}+1, \xi_{A4}]$, $[\xi_{A4}+1, \xi_{A\text{-max}}]$. Considering the evolution of parameter ξ_A in different iterations, each time the parameter value falls into a different interval, the MSD mode can round it up to the interval's upper boundary, and assign the result as the (quantized) actor delay. Because all actors depend on the same parameter, such a quantization leads to just 5 scenarios, not to 125, as in the previous example. This example might be a bit idealistic, but what can be expected in practice more often is that the parameters of different actors are statistically correlated, which also helps to reduce the number of scenarios.

To give a practical example, the number of AC symbols per MCU – denoted ξ_{AC} – to a large extent determines the execution delay of the VLD actor in the JPEG application considered in Chapter 3 (see Figures 3.9 and 3.3). At the same time, ξ_{AC} can also have a considerable impact on the execution delays of the IDCT actors for a certain implementation of the IDCT algorithm in the actor body. In that case, the execution delays of VLD and IDCT are correlated, which is favorable for making the number of scenarios as small as possible.

5.3.4 Using the Actor Parameters to Define the MSD-mode Scenarios

The scenario subspace approach defines the MSD mode based on the actor parameters, ξ_ω. In this way the resulting model is applicable to a wide range of hardware architectures and can exploit the mutual correlation between the delays of different actors.

In the scenario subspace approach, we quantize the actor parameters, ξ_ω, and, consequently, the actor delays are also quantized, indirectly. Through the quantization levels of parameters the scenarios are defined. Not all the actor parameters are involved in the scenario definition, but

only a subset of them. These parameters are called *primary parameters*. Those are, by preference, parameters that have a considerable impact on the execution delays of several actors.

Before providing more in-depth definitions and explanation of the scenario subspace approach, let us give an example. For the JPEG application example mentioned in the previous subsection, let us choose ξ_{AC} as the primary parameter. The range of possible values of ξ_{AC} is 0..384. We can split that range into five sub-ranges, e.g. [0, 6], [7, 24], [25, 48], [49, 96], [97, 384], thus defining five scenarios, for this application (s = 0, 1, 2…). According to this definition, the scenario of the application in every iteration n of the loop of interest is determined by the sub-range in which parameter ξ_{AC} is located in that iteration.

The fact that the scenarios are defined in terms of the quantization levels of actor parameters makes the loop-level parameters, i.e., J_s, L_s, and $K_{s,t}$, independent of the hardware architecture. Consequently, to compute the run-time values of i.e. J_s, L_s and $K_{s,t}$ for the given execution run, one can first instrument the application source code with the actor parameter counters (as explained in Chapter 3) and then perform the execution run on a high-performance workstation, this way getting the trace of all actor parameter values in that run. From that trace, one can find the execution ranges $\mathbf{I}_p$ of different scenarios and can calculate the values of J_s, L_s, and $K_{s,t}$. For example, for the JPEG application, one can compute J_1 for the given JPEG image as the number of MCU blocks of that image for which ξ_{AC} lies in interval [7, 24] (as defined in our MSD mode example given above).

In case there are multiple primary parameters, the scenarios are defined in terms of their combination. For convenience, we put the primary parameters in a vector, $\boldsymbol{\xi}_{\mathrm{PRIM}}$. The multidimensional space of possible values of $\boldsymbol{\xi}_{\mathrm{PRIM}}$ is split into subspaces, corresponding one-to-one to scenarios.

Example (Scenario subspaces). Suppose there are two primary parameters, $\xi_1 \in [0,2]$ and $\xi_2 \in [0,8]$. We have $\boldsymbol{\xi}_{\mathrm{PRIM}} = (\xi_1, \xi_2)^{\mathbf{T}}$. Below we give an example of a possible division into subspaces:

$s = 0$ if $(0, 0)^{\mathbf{T}} \le \boldsymbol{\xi}_{\mathrm{PRIM}} \le (1, 3)^{\mathbf{T}}$,

$s = 1$ if $(0, 4)^{\mathbf{T}} \le \boldsymbol{\xi}_{\mathrm{PRIM}} \le (1, 8)^{\mathbf{T}}$,

$s = 2$ if $(2, 0)^{\mathbf{T}} \le \boldsymbol{\xi}_{\mathrm{PRIM}} \le (2, 8)^{\mathbf{T}}$ ♦

Given the scenario definition in terms of subspaces, for any iteration n of the loop of interest we can identify the scenario s to which it belongs. This is done by extracting the elements of $\boldsymbol{\xi}_{\mathrm{PRIM}}$ in iteration n from the trace of actor parameter values and then identifying the subspace to which the obtained $\boldsymbol{\xi}_{\mathrm{PRIM}}$ belongs. Recall from Chapter 3 that the extraction of actor parameter values can be done offline, when the input stream frame headers are being prepared for the application. Therefore, any necessary information about the scenarios, including J_s and $K_{s,t}$, can be prepared offline and stored in the frame headers.

Let us answer the question why we define the scenarios in terms of subspaces and how that helps us to keep the error of the MSD mode limited. The MSD mode accuracy is limited because it assumes that the actor delays are constant in every scenario. Therefore, to keep the error limited, one has to define the scenarios such that the actor delay variation within each scenario is limited. The scenario subspace approach achieves this by trying to keep the variation of all actor parameters limited within each scenario. It is obvious that, in every scenario, the variation of all

the primary parameters is limited because, by definition, the primary parameters stay within the scenario subspace. If the subspace is small enough, the error caused by variation of the primary parameters is also small enough. To ensure small enough subspaces, one has to make their number large enough. Therefore, there is certain trade-off between the number of scenarios and the error of the MSD mode.

As for all the other actor parameters, which we call *secondary parameters*, one can ensure that also their contribution to the total error is limited by using a proper way for classification of the actor parameters as primary and secondary.

One has the most control over the quantization error in case all the parameters are classified as primary parameters. However, the number of scenarios grows exponentially in the number of primary parameters, and in practice one can expect that most parameters should be classified as secondary ones.

We can propose two rules of thumb for identification of secondary parameters.

Firstly, a parameter can be classified as secondary if its *contribution* to the actor delays is comparatively small. The *contribution* of an actor parameter to the delay of an actor at the given target multiprocessor platform can be defined as the parameter coefficient multiplied by the difference between the maximum and minimum parameter value. If for every actor and every representative target platform holds that the contribution of the given parameter is considerably smaller than the contribution of some other parameters, then the given parameter is a good secondary parameter candidate. Such a parameter can bring only a small contribution to the quantization error.

Secondly, a parameter is a good secondary parameter candidate if it is statistically dependent on one or a few primary parameters. In that case, limiting the variation of the primary parameters within a subspace leads to the phenomenon where the parameter dependent on the primary parameters stays with high probability within a certain limited interval. A typical example is the number of bits used to encode the MCU block of a JPEG image – denoted ξ_b, see Figure 3.3 – which depends on the number of AC symbols per MCU block, ξ_{AC} as sketched in the following way. Almost all bits used to encode an MCU block of a JPEG image are dedicated to AC symbols. For the JPEG images of a certain quality level, one can give a narrow interval for the most typical number of bits per AC symbol, e.g. from 4 to 12 bits. Therefore, for most MCU blocks with ξ_{AC} lying in subinterval [25, 48], we can be sure that ξ_b stays within interval $[4 \cdot 25, 12 \cdot 48] = [100, 576]$ with high probability.

We know two alternative methods to define the scenarios, namely, a manual one and an automatic one. The manual one first finds the primary parameters (possibly, using the above rules of thumb) and then splits the range of possible values of each primary parameter into large enough number of sub-ranges such that the distance between the quantization levels is small enough and the required accuracy of the MSD mode is obtained. An example of an automated method is considered in the next subsection.

5.3.5 Applying an Automatic Scenario Identification Technique

The automated scenario identification technique proposed by S. V. Gheorghita *et al* in [28] consists of three major steps:

1. Identification of actor-level control variables.

2. Scenario selection.

3. Scenario analysis.

The first step identifies the control variables of the actor source code that can have an impact on the actor execution time. The actor-level control variables are similar to our actor-level parameters, but they do not necessarily contribute to the actor execution time linearly. In our flow, we prefer actor-level parameters over actor-level control variables, because any combination of parameter values can be translated into actor processing times using linear actor-level parameter functions (see Section 3.2), whereas [28] perform table lookup from the control variables to the processing times and it may happen that some combinations are not available in the table. Therefore, in our flow, we replace this step by identification of actor-level parameters, i.e. actor-level identification, explained in Section 3.2.

Step 2, the scenario selection, is profiling-driven. Therefore, it requires a training stream, i.e. a representative input stream of application data used to generate the results. It also requires the corresponding trace of actor processing times, measured using profiling tools. Because their work assumes single-actor applications, during the scenario selection, we assume a single actor, whose processing time is the sum of the processing times of all actors. Based on the actor processing time trace, the scenario selection step defines multiple MSD mode *candidates* for the loop execution run that processes the training stream. Every candidate is identified by a given set of quantization levels and the corresponding set of execution intervals generated for the training stream. The scenario selection step produces MSD mode candidates with various numbers of quantization levels, trying to exploit the accuracy-overhead trade-off. When generating a candidate with a given number of scenarios, it tries to minimize the quantization error and the number of scenario transitions.

An important task of Step 3, the scenario analysis, is to imply, for every MSD mode candidate, the primary parameters and subspaces of primary parameter values. As proposed and automated in [28], the basic goal is to detect a set of primary parameters and the corresponding subspaces of primary parameter vector ξ_{PRIM} such that the cardinality of this vector is minimized and the value of this vector goes from one subspace to another at every boundary between two execution intervals in the given MSD mode candidate. As a result, for every MSD mode candidate, this scenario identification technique maps the quantization levels into subspaces of the primary parameter values. Every subspace becomes a definition of a scenario. Effectively, this step translates every MSD mode candidates into a scenario set candidate.

As explained in [28], another important task of Step 3 is *scenario set evaluation*. For evaluation, one needs to measure the impact of different scenario set candidates on the run-time optimization goal, such as quality/resource budgets/energy, whatever is intended for this application. In a multiprocessor-oriented implementation trajectory like ours, for evaluation, it is recommended to run the design flow for several representative instances of the application-domain specific platforms. In the end of every run of the design flow, for the given intra-application mapping, one can run a simulation of the run-time adaptation manager, evaluating the gain in the optimization goal for every candidate set of scenarios. One has to weigh this gain against the overhead in terms of the processor clock cycles consumed by the run-time adaptation manager and the number of bits required to encode the parameters in the frame headers.

In the end of this subsection, we can conclude that the considered automation technique fits well into our implementation trajectory, with some minor modifications. In this context, the last

important thing to mention is the fact that, unlike [28], we do not assume static (conservative) quantization levels for each scenario. As mentioned before and explained in the next subsection, to ensure good accuracy and guaranteed performance, we calculate the quantization levels dynamically, for every execution run of the loop of interest. We do this based on the run-time a-priori actor-level parameter values calculated (and encoded in the frame headers) offline for every scenario subspace, as explained in the next subsection.

5.3.6 Scenario Subspaces and Run-time Execution Time Prediction

Having explained the foundations of the scenario subspace approach, in this subsection, we define the MSD mode based on scenario subspaces. After this, we can explain how our performance analysis method uses the scenarios to predict the loop execution times at run time.

Definition (A subspace-based MSD mode) is an MSD mode where the scenarios are defined as subspaces of the actor-level parameter values. The MSD mode is defined by:

1. the set of scenario subspaces;
2. given an execution run of the loop of interest, the method to determine to which scenario each iteration *n* belongs;
3. given an execution run, the method to calculate the quantization levels of the actor delays.♦

Below we define these three notions one-by-one.

Definition (Scenario subspaces) Let ξ_{PRIM} denote the vector of primary parameters. The *subspace* of scenario *s* can be specified by a predicate (i.e., a Boolean function) $\pi_s(\xi_{\mathrm{PRIM}})$, which defines a subspace in the space of possible values of vector ξ_{PRIM} – namely the subspace where predicate π_s evaluates to true.

For the predicates of the scenario set should hold:

a. $s \neq t \;\Rightarrow\; \pi_s \wedge \pi_t = 0$, meaning that different subspaces do not overlap

b. $\pi_1 \vee \pi_2 \vee \pi_3 \vee \ldots = \pi_{\mathrm{possible}}$, where π_{possible} is the predicate specifying all possible values of the primary parameters; this way the subspaces cover the complete space of values of ξ_{PRIM}.

♦

For instance, in the scenario subspace example given in the previous subsection, the predicate for $s = 0$ can be expressed as follows:

$\pi_0\,(\xi_{\mathbf{PRIM}}) = (0, 0)^{\mathbf{T}} \leq \xi_{\mathrm{PRIM}} \leq (1, 3)^{\mathbf{T}}$

Definition (The method to determine to which scenario iteration *n* belongs.) Let $\xi_{\mathrm{PRIM}}(n)$ give the values of the primary actor parameters in iteration *n* of the given execution run. Due to properties A and B in the definition of the scenario subspaces, there should be exactly one predicate π_s taking Boolean value 1, which identifies scenario *s* to which iteration *n* belongs. ♦

Definition (The method to calculate the quantization levels of actor delays.) The purpose of this method is to give a quantization level of the delay of every actor v_k in each scenario *s*, denoted $\hat{d}_s(v_k)$. To make the quantization levels conservative, for each parameter ξ_ω in each scenario *s* we use the maximum value of that parameter encountered within all iterations

belonging to scenario s. We call that value the *characteristic parameter value* and denote it as $\hat{\xi}_{\omega,s}$:

$$\hat{\xi}_{\omega,s} = \max\,(\xi_{\omega}(n) \mid n = 0..N-1,\ \pi_s(\boldsymbol{\xi}_{\mathrm{PRIM}}(n)) = 1) \tag{5.10}$$

Given the characteristic parameter values, we can calculate the characteristic value of the processing time of every actor, using the actor parameter function (see Section 3.2):

$$\hat{t}_s(v_k) = C_{k,0} + C_{k,1}\cdot\hat{\xi}_{1,s} + C_{k,2}\cdot\hat{\xi}_{2,s} + \ldots + C_{k,\Omega}\cdot\hat{\xi}_{\Omega,s} \tag{5.11}$$

Finally, using the computation delay relation, we obtain the quantization level of the actor delay (see Section 3.1):

$$\hat{d}_s(v_k) = R_{\mathrm{comp}}\left(\hat{t}_s(v_k), v_k\right) \tag{5.12}$$

♦

This definition completes the formal definition of the multi-scenario timing mode of the HSDF graph. We finish this subsection by summarizing the representation of the MSD mode in the implementation trajectory for streaming applications adopted in this thesis. It consists of three parts:

a. when designing the application, the application designer identifies the loop-level scenarios during the Application Preparation part of our implementation trajectory;
b. the application generating the input data stream (e.g. the video encoder) fills the frame headers with the loop-level parameter values – i.e. the scenario of the first interval, $s(1)$, the loop-level parameters J_s and $K_{s,t}$ (recall from Section 5.2.5 that L_s can be derived from $K_{s,t}$ and $s(1)$) – and characteristic actor parameter values – $\hat{\xi}_{\omega,s}$, once per every frame;
c. at run time of the user application, the quality adaptation manager decodes J_s, $K_{s,t}$ and $\hat{\xi}_{\omega,s}$, and applies Equalities (5.11) and (5.12) to find the quantization levels; finally, the manager executes the algorithmic rule for dynamic-delay analysis (see Section 5.2.5). Note that the quantization levels cannot be pre-encoded in the frame headers, because that would make the input data stream dependent on the target hardware platform and mapping.

Hereby, the major goal of performance analysis is achieved – i.e. accurate prediction of the important performance metrics for timely adaptation of quality/energy consumption/resource budgets. In the next chapter we apply our performance analysis method in an application case study.

5.4 Summary and Notes

This chapter introduces a performance analysis framework leading to an algorithmic rule for predicting the data-dependent execution time of an application based on run-time workload characteristics given in the application headers. In line with the goals of this thesis, the estimates provided by our framework are guaranteed, i.e. conservative and giving good expectations on the achieved accuracy. Our framework is based on the integration of contributions of multiple execution intervals that are parts of the total execution run. The contribution of intervals is estimated in a conservative way, based on the technique proposed in Chapter 4. The key idea of this chapter is the *minimum overlap analysis* technique. This technique does not follow a naïve

method of integration of contributions by a plain addition. Instead, it takes into account that, in an HSDF graph, different relevant events of the given execution interval occur at different times. This leads to tighter combination of contributions during integration than a plain addition could give, which suggests a good accuracy improvement. For special cases, indicated in the description of this technique (i.e. multiple sinks in an HSDF graph), this technique may require some refinement in future work, to make the combination of contributions tighter. In our implementation trajectory, the minimum overlap analysis contributes to the loop-level characterization, i.e. the run-time calculation of coefficients of the loop-level parameter function.

Our framework requires only a limited a-priori information overhead in the application header, because we define the execution intervals in terms of a set of so-called scenarios. Each scenario is specified by a conservative assignment of an actor delay level to every actor. The overhead of our analysis framework depends on the number of scenarios, because for every scenario our approach needs to know the values of certain parameters and also for every scenario transition it needs to know the number for transitions between the given pair of scenarios. The number of scenarios can be kept small using our scenario-subspace approach, described in Section 5.3.

Our first publication on static-delay analysis and integration of execution intervals for multiprocessor streaming applications is [77]. In [76], [78], [79] we introduced the minimum overlap technique and worked out the static-delay analysis of execution intervals in more detail.

As we mentioned in Section 1.5.1, among the related work, the only closely comparable work is the work of Zhe Ma *et al* [55], [56]. In [55], they introduce an analogue of our MSD mode for a streaming application case study. Although the focus of that work is run-time adaptation for minimization of energy consumption, they also implicitly introduce an accompanying run-time performance analysis technique that has much in common with ours. However, because their static-delay analysis only supports limited-length execution runs, their execution intervals include a constant number of loop iterations. Our analysis technique exploits steady-state analysis of HSDF graphs, which allows it to be more flexible, because it can extend every interval until it encounters a scenario transition. This allows us to minimize the number of execution intervals, which helps to reduce the possible prediction error introduced at interval boundaries. Note also that [55] does not take into account the timing overlap between the execution intervals, which makes their technique inefficient when significant timing overlap is present. Later on, in [56], they proposed ‘interleaving’ of different static schedules being combined together on the same multiprocessor resources. In that paper, they exploit – independently – a similar idea as we do, namely, the shifting the static schedules until the best timing overlap is reached. However, their timing overlap calculation does not support dependencies between the schedules being combined together, so, to support streaming application, it requires extensions.

In the next chapter, we support our claims on good accuracy, low overhead and usefulness for run-time adaptation by doing an application case study.

6

6 The Practical Use of Performance Analysis

In Chapter 1, we indicated that the practical use of our performance analysis framework is to provide optimization guidelines in two major areas:

a. mapping streaming applications to a multiprocessor platform,
b. dynamic adaptation of the application and implementation parameters to the application workload.

Providing the performance analysis guidelines for mapping – point ‘a’ above – is described and illustrated in Chapter 3, using a JPEG image decoder as an application example. In this chapter, we do a case-study supporting point ‘b’, and we use an MPEG-4 decoding application for that. Hereby, we show how the techniques described in Chapters 3, 4 and 5 together can be used in a practical quality adaptation framework.

In this case study, we investigate the costs and benefits that the performance analysis ultimately yields for the end user. For this purpose, we constructed a multiprocessor simulation environment, modeling the hardware timing at the processor instruction level. This allows us to evaluate the most important aspects without realization in the real hardware and with reasonable accuracy.

This chapter is organized as follows. Section 6.1 describes the case-study application and the accompanying quality-adaptation manager. We explain why the latter needs accurate performance analysis. Section 6.2 reports on the design-time part of the performance analysis,

involving the framework presented in Chapters 3-5. In Section 6.3, we present the results of dynamic execution-time prediction in terms of benefits and overhead. Section 6.4 summarizes this chapter.

6.1 Application: an MPEG-4 Video Object Shape Decoder

6.1.1 Application Overview

As our application case study, we use the same application example as the one briefly introduced in Chapter 1, see Figure 1.7. This application performs the shape decoding for an arbitrary-shape video object, defined in the MPEG-4 standard [42], [12]. According to the standard, a video presentation may consist of multiple video objects that can be opened and closed dynamically by the user or by a remote system. Therefore, in terms of this thesis, one video object corresponds to one active application. The resource requirements per application can considerably change at run time, because the objects can change in size and in shape. MPEG-4 video objects are thus good representatives of the streaming applications with dynamic workload.

In the MPEG-4 standard, the video frames are referred to as '*video object planes*' (VOPs). The VOPs are grouped in so-called *groups of pictures* (*GOP*s). Every VOP in a GOP except the first one, needs the previous VOP to be decoded first, because it uses the results of the decoding. VOPs in different GOPs do not depend on each other.

Figure 6.1 shows the top-level functional diagram of one video object decoding application. The application contains a few core subroutines, shown as ovals. Those are GOP Decoder, VOP Decoder, and QoS Manager. Some of those subroutines contain functionality that can be distributed between a few processors. The application also uses a few peripheral modules such as Timer, VOP Presenter, the input memory queue and the output memory queue, shown as boxes.

The application presents VOPs on the video screen at a constant rate, which is set by the Timer module. The Timer signals the moments of time when the presentation of each video frame should start. These equidistant time moments at which the frames should be ready are the *deadlines* that should be met by the application. In our case study, missing some deadlines is still acceptable (although discouraged). Thus this application is a *soft real-time* (SRT) application.

As it is illustrated in Figure 1.7 in Chapter 1, each VOP is a variable-size two-dimensional array of *macroblocks* (*MB*s). The macroblocks are 16x16 pixels each. For simplicity, we assume that every VOP in the output queue takes a fixed portion of memory, equal to the size of the largest possible macroblock array for a VOP.

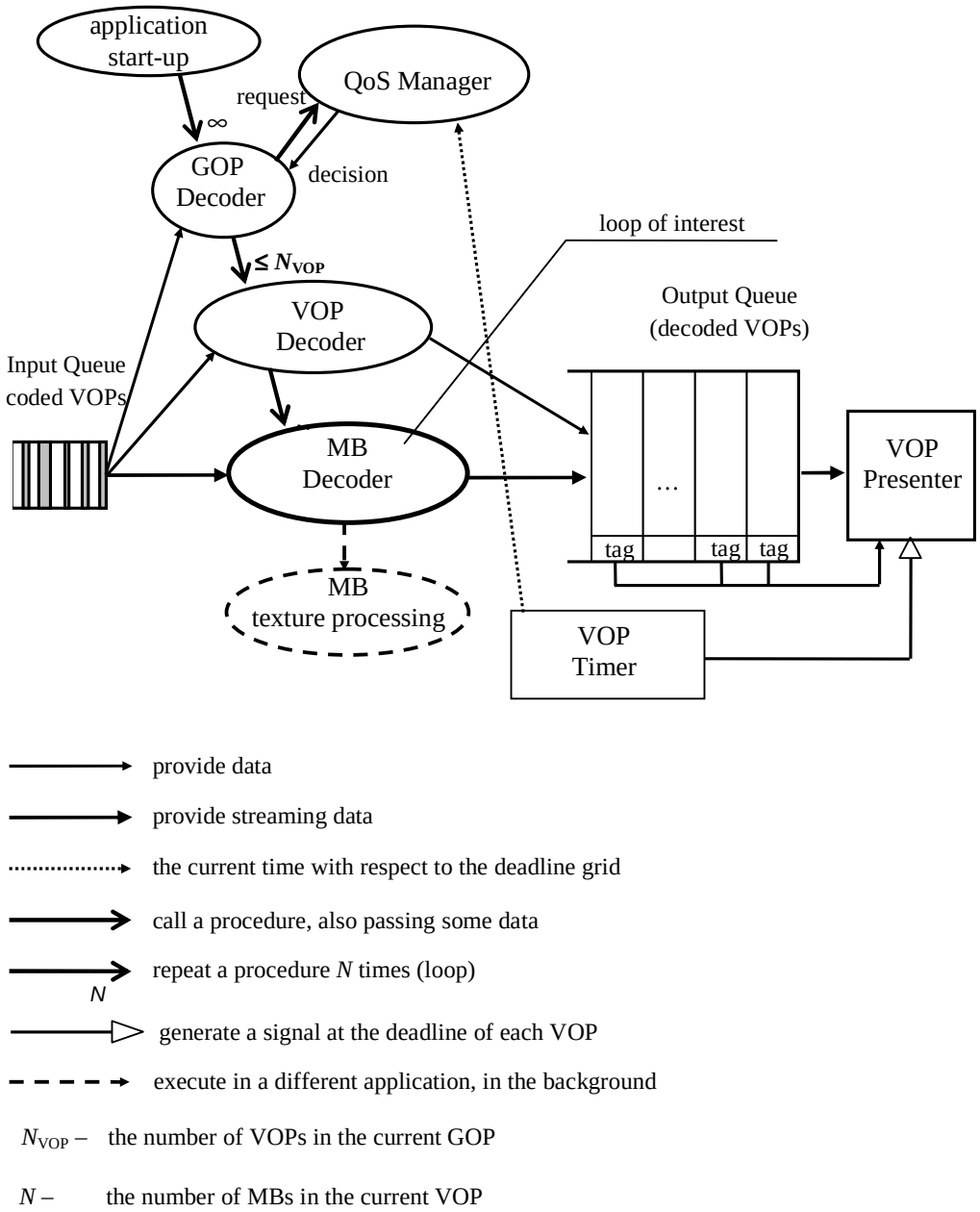


Figure 6.1 MPEG-4 video object shape decoding application

As also illustrated in Figure 6.1, each VOP in the output queue has a tag that indicates the deadline of that VOP. The VOPs with earlier deadlines are in front in the queue. At each deadline, the VOP Presenter looks in the queue for the first VOP in the queue whose deadline is not later than the current deadline. That VOP is selected as the next one to be presented. Ideally, that would be the VOP whose deadline tag coincides with the current deadline, but it is possible that that VOP was not decoded on time, and a later VOP is selected. The selected VOP of choice is kept in the queue at least until the next deadline. All the VOPs in front of that VOP are removed from the queue as they are no longer needed.

As illustrated in Figure 6.1, the whole application can be seen as a nested loop. The top-level loop is the GOP-decoding loop, containing the VOP-decoding loop, which, in turn, contains the MB-decoding loop. The latter produces one MB per iteration at the output, and it is expected to

have produced all the MBs of the current VOP by the deadline. Therefore, we identify the MB-decoding loop as the loop of interest, which means that we directly analyze the performance of that loop at run time to see if the loop can meet its deadlines.

Recall that our performance analysis techniques operate under the assumption that the input data of the loop of interest is always provided on time. In our case study, this requirement means that the input queue should be filled fast enough such that the loop of interest always finds the required input data when it is needed. This requirement is realistic because the input video data is coded (compressed) and therefore it involves much less off-chip and on-chip communication traffic than the decoded (uncompressed) video stream at the output.

A similar requirement holds for the output queue. The environment should consume the output data of the loop of interest fast enough so that the loop always has the space to store the output data. This requirement is satisfied automatically, because, as explained later, the VOP Decoder subroutine starts the MB-decoding loop only when there is space enough for the whole VOP.

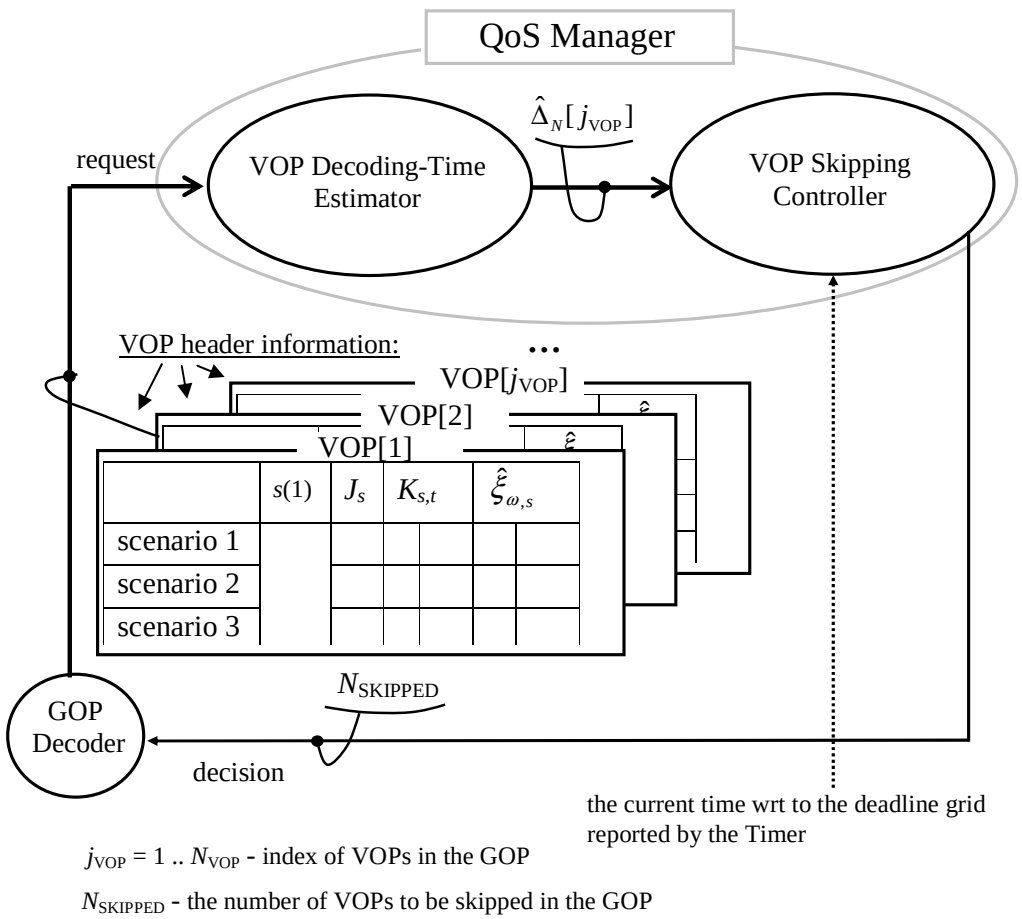
Our case study covers only part of the VOP decoding procedure defined in the MPEG-4 standard. Each VOP consists of three color planes describing the VOP '*texture*' (the colored image) and one plane with the VOP '*shape*' (defining the contour of the image). All four planes require that the input bitstream be parsed, but the texture planes also require some extra processing (such as inverse quantization, inverse discrete cosine transform, etc). Parsing the input bitstream and producing the shape plane can be done independently from the texture processing. Therefore, as also illustrated in Figure 6.1, we put the texture processing into a separate application and focus only on the bitstream parsing and shape decoding. We assume that the texture decoding runs in the background fast enough so that it does not influence the timing of our case-study application. This requirement is realistic because the texture-processing is characterized by better-predictable performance and can be implemented efficiently on high-performance domain-specific processors or hardware [9], [20].

Now let us consider the subroutine called *QoS Manager*. It is the part of application that is responsible for treating the situation when the computational workload grows dynamically to such a high magnitude that not all the deadlines can be met. As shown in Figure 6.1, QoS Manager is called once per GOP with a request to generate a decision. The Manager decides how many VOPs of the current GOP are to be skipped due to lack of computational resources. (Those have to be the last VOPs in the GOP because of the chain dependency of every VOP on the previous VOP.) The algorithm used by the Manager is rather simple, but it relies on a complex performance analysis technique to predict the decoding times of the VOPs contained in the GOP.

Because the QoS Manager is the part where we apply our run-time performance analysis techniques, in the next two subsections, we describe the Manager in more detail.

6.1.2 QoS Manager

The role of the QoS Manager subroutine can be classified as 'local quality adaptation manager' as we define it in Chapter 1 (see Figure 1.1). We call the QoS Manager local because it controls only one application. Recall that the role of quality adaptation in general is to treat the computational overload by scaling down some computational complexity parameters of the application such that the processor workload decreases. Recall that we refer to the application



j_{VOP} = 1 .. N_{VOP} - index of VOPs in the GOP

$N_{SKIPPED}$ - the number of VOPs to be skipped in the GOP

Figure 6.2 QoS manager
(being an instance of the quality optimization framework – see Figure 1.8(b))

complexity parameters that can be scaled by the Manager as *active parameters* whereas the parameters that cannot be scaled are called the *passive parameters*.

Unfortunately, Macroblock Decoder subroutine has only passive parameters. Although some of those parameters are active in a broader context, they cannot be scaled by a local manager; only the video encoder (i.e., the remote application that produces the input data stream for this application) can scale those parameters [12].

Therefore, our QoS Manager works at a higher level of the loop nesting, namely, at the level of GOPs, where there is one active parameter – the number of VOPs to be skipped. Under the resource overload conditions, our manager decides to skip some VOPs in the GOP, thus accepting some unavoidable loss in the quality of the video content presented to the user.

Figure 6.2 shows the basics of the Manager's implementation. As we illustrate in the figure, the QoS Manager subroutine can be expanded into two subroutines: the VOP Decoding-Time Estimator and the VOP Skipping Controller. Those two subroutines correspond to two blocks in Figure 1.8(b) in Chapter 1. The Decoding-Time Estimator acts here as the 'performance analyzer' and the Skipping Controller acts as the 'optimization unit' for optimizing the visual quality. The Controller sets the number of VOPs to be skipped, $N_{SKIPPED}$, trying to make it as small as possible. The QoS Manager communicates $N_{SKIPPED}$ back to the requestor as the 'decision' taken by the Manager. Note that, in our application case study, every VOP in GOP,

except the first one, depends on the previous VOP and, therefore, it is the *last* N_{SKIPPED} VOPs in a GOP that get skipped.

The Estimator provides the Controller with the execution time estimates for every VOP in the current GOP. Using our notation for the execution time, we denote the result of estimation as $\hat{\Delta}_N[j_{\text{VOP}}]$, where $j_{\text{VOP}} = 1\ldots N_{\text{VOP}}$ is the VOP index and N_{VOP} is the number of VOPs in the GOP. To obtain the estimates, the Estimator uses the algorithmic rule for dynamic-delay analysis introduced in Chapter 5.

The required input information for the dynamic-delay analysis is encoded in the VOP headers. Recall from Section 5.3.6 that the inputs are the first scenario identifier, $s(1)$, loop-level parameters – J_s, $K_{s,t}$ – and characteristic values of actor-level parameters – $\hat{\xi}_{\omega,s}$.

In this case study, the only part of the application whose timing behavior is taken into account in our models is the loop of interest. Thus we ignore the delays of subroutines GOP Decoder, VOP Decoder and QoS Manager. This can be justified by the fact that the delays of GOP/VOP Decoder subroutines, which are responsible only for the header parsing, are significantly smaller than the VOP decoding times. As for the overhead of the QoS Manager itself, it consists of the Estimator overhead and the Controller overhead. Later in this chapter we see that, in this case study, the most complex estimation work can be done at design time, so that the run-time overhead to calculate $\hat{\Delta}_N$ is negligible. If necessary, it is relatively straightforward to take into account the worst-case delay introduced by these three subroutines.

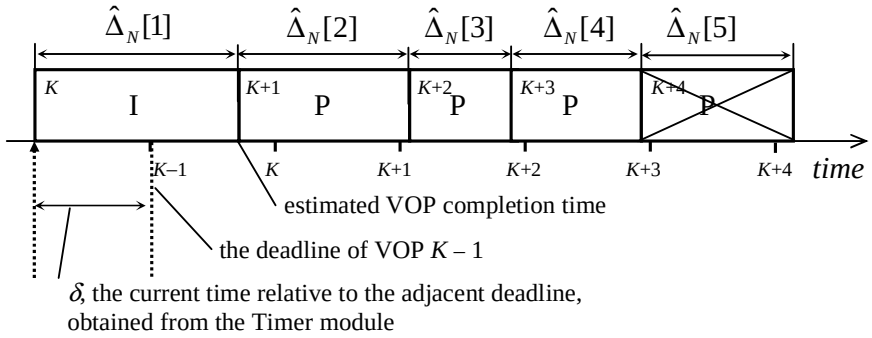
The Skipping Controller is also fast enough to be neglected, because it uses an algorithm which is much less complex and computation-intensive than the Estimator. The Skipping Controller is the direct user of the $\hat{\Delta}_N[j_{\text{vop}}]$ estimates generated by our run-time performance analysis framework. We use it in this chapter to evaluate the impact of performance analysis accuracy on the video quality. Therefore, in the next subsection we focus on the Controller. As for the Estimator, we describe it in Section 6.3, after we have introduced the necessary details on the design flow for this application in Section 6.2.

6.1.3 The Frame-skipping Algorithm

The skipping algorithm is illustrated in Figure 6.3(a). The VOPs are shown as indexed rectangles, where index *'K'* is an absolute VOP index, counting the VOPs from the beginning of the input video stream. Each GOP covers only a certain range of indices: K, $K+1$, $K+2$,…, $K + j_{\text{VOP}}$, …, $K + N_{\text{VOP}} - 1$. The horizontal dimension of each rectangle shown in the figure corresponds to the time span of VOP decoding, from the starting time to the completion time.

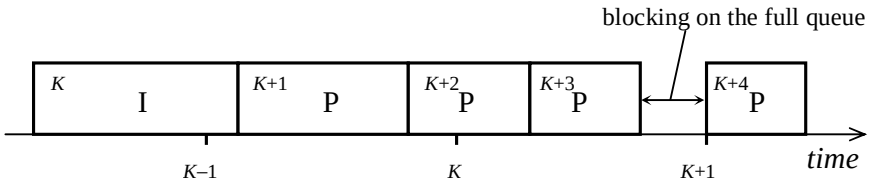
The figure illustrates the generic structure of a GOP. The first VOP in a GOP is called I-VOP, the rest are called P-VOPs. (MPEG-4 coding also knows so-called B-VOPs, but we do not support them is this case study.) An I-VOP is decoded independently of the previous VOPs. On the contrary, the decoding of a P-VOP depends on the previous VOP. Consequently, if the Manager decides to skip a VOP, it has to skip all the VOPs that come later in the same GOP. Thus, as already mentioned, parameter N_{SKIPPED} specifies the *last* N_{SKIPPED} VOPs in the GOP which are to be skipped.

The Skipping Controller estimates the VOP completion times in order to check them against the grid of equidistantly placed deadlines to predict which VOPs will miss their deadlines. The completion time estimates are calculated by adding $\hat{\Delta}_N[j_{\text{VOP}}]$ to the estimated starting times –



VOP $K + 4$ is estimated to miss its deadline; therefore, it is skipped thus, the decision is $N_{\text{SKIPPED}} = 1$.

(a) an example GOP with 5 VOPs and the VOP skipping decision



This example assumes output queue capacity 4

(b) Influence of the limited capacity of the queue on the timing of VOP decoding

Figure 6.3 Skipping management for an example GOP with 5 VOPs

see Figure 6.3(a). Typically, the starting time of VOP 'j_{VOP}' is equal to the completion time of VOP '$j_{\text{VOP}} - 1$', but this rule has an exception due to a limited capacity of the output queue, as explained later.

In Figure 6.3(a), the grid of deadlines is shown using marks on the time axis. We see that VOPs K+1 and K+4 miss their deadlines, because their completion times are later than the deadlines.

Let us explain how the Skipping Controller makes the skipping decision. At the tail of the current GOP, the Controller identifies the longest sequence of VOPs that miss their deadlines. The Controller assigns this sequence to be skipped. If there are any VOPs in the middle of the GOP that miss their deadlines, they are kept, so that the VOPs that follow them and do meet their deadlines can be kept as well. For example, in Figure 6.3(a), only $K + 4$ is skipped because it is in the tail. $K + 1$ is kept, because $K + 2$ and $K + 3$ meet their deadlines.

Now let us come back to the estimation of the VOP starting times. For the I-VOP (the first VOP in the GOP, $j_{\text{VOP}} = 1$), the starting time is known. It is the current time, reported by the VOP Timer, which measures it directly and reports it to the Skipping Controller. At the start of every GOP, the Timer module reports the current position at the time axis with respect to the deadline grid; for the example in Figure 6.3(a), the Timer would report that the current time is in front of the $K - 1$-th deadline and that the distance to the deadline is δ. Knowing the exact position at the time axis is important for the Skipping Controller to build a realistic estimate of which VOPs in the current GOP will meet their deadlines.

For $j_{VOP} = 2, 3\ldots$, the starting times, are estimated recursively, first for $j_{VOP} = 2$, then for $j_{VOP} = 3$, etc. The estimated starting times depend on $\hat{\Delta}_N[j_{VOP}]$ and the number of VOPs that can fit in the output queue – i.e. the output-queue capacity.

Let us first assume that the output queue has an infinite capacity. In that case, the GOP Decoder never postpones decoding a VOP due to the full output queue. Therefore, the starting time of every P-VOP is equal to the completion time of the previous VOP.

In practice, the output-queue capacity is finite. Therefore, the Controller takes into account the fact that, due to the full output queue, the GOP Decoder postpones starting a new VOP until at least one place is freed in the queue by the Presenter. Figure 6.3(b) shows an example. Suppose that the output queue capacity is 4 VOPs. Then, after the completion of VOP $K + 3$, all four places are filled, containing VOPs K, $K + 1$, $K + 2$, $K + 3$. Since VOP K is needed in the decoding of VOP $K + 1$, the Presenter keeps VOP K in the queue until the next deadline, deadline $K + 1$. Only at deadline $K + 1$, the Presenter removes VOP K from the queue, and then there is place for $K + 4$, and the decoding for that VOP starts. This is taken into account by the Controller when it estimates the starting times.

The Controller depends on the accuracy of the $\hat{\Delta}_N[j_{VOP}]$ estimates made by the Estimator. In the next subsection, we show that accurate execution time estimations are very important for the Controller to produce good results.

6.1.4 The Sensitivity of Visual Quality to the Accuracy of the Performance Analysis

To evaluate the need for accurate VOP decoding time estimates, we simulated our application with the Skipping Controller enabled and with the Decoding-Time Estimator temporarily replaced by an artificial ‘oracle’ estimator, predicting the decoding times of all VOPs exactly and adding a random error to them when producing the estimations. In the random error generator, we assumed that the relative prediction error is normally distributed, being conservative with a probability of 95%. This model of prediction error reflects the fact that for soft-real-time applications, our performance analysis technique can produce predictions that are not 100%-guaranteed to be conservative. Allowing the results to be occasionally non-conservative saves analysis effort and contributes to better accuracy.

Note that the interval between the deadlines was chosen small enough to create significant computational overload, such that one could not avoid dropping at least 20% of the VOPs. The output queue capacity was set to three VOPs (the minimum required for a smooth performance).

Figure 6.4 shows the dependency of the quality (as the percentage of VOPs presented to the user) on the average VOP-decoding-time estimation error. This dependency is obtained for one of the sample video streams from our case study. It illustrates that prediction errors may lead to an unacceptable drop in quality, the reason for that being that the higher the overestimation the more VOPs are skipped by the Skipping Controller.

In the next section, we apply our performance analysis techniques to this case study with the purpose to achieve a small enough prediction error, and a close to 100% guarantee of conservative prediction results.

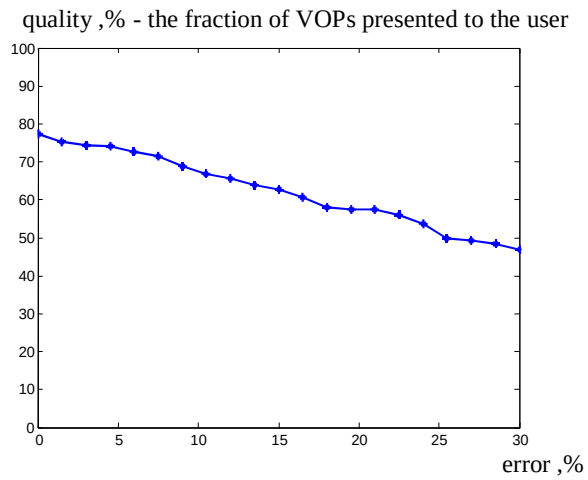


Figure 6.4 The sensitivity of quality to the VOP decoding time estimation error

6.2 Design-time Performance Analysis of the MPEG-4 Decoder

6.2.1 Overview and Recapitulation

In this section, we take the MPEG-4 shape decoder application through the design flow introduced in Chapter 2. We focus on the performance analysis aspects of the design flow.

Let us give a detailed overview of this section. In the beginning, we specify two starting points for the design flow: the description of the target system-on-chip platform (Section 6.2.2) and the HSDF graph **G** with mapping constraints (Section 6.2.3). Recall that the graph nodes are computation subroutines – actors. The mapping of graph **G** to the target platform is expressed in terms of the implementation process network, **PQ**, which encloses the actors and the edges of graph **G** into the processes and the channels, mapped to the processors and the communication network.

Sections 6.2.4 and 6.2.5 describe Part I of our design flow, namely the Application Preparation. This part identifies the application's complexity parameters, which are key aspects for dealing with the data-dependency in our performance analysis method. Recall that the parameters are defined at two granularity levels: the actor level (the finer granularity) and the loop level (the coarser granularity).

The front-end of the Application Preparation is the analysis of the application timing at the actor level. Section 6.2.4 reports on the actor-level analysis for our case study, based on the methodology described in Sections 3.2 and 3.3. For the given application, we identify the actor-level parameters – denoted ξ_ω. The linear combination of actor-level parameters and constant hardware-dependent coefficients models the *actor processing times*, i.e. the processor cycle counts of actor executions. The actor processing times are the main ingredients of the *detailed dynamic timing mode*, which is our highest-accuracy actor timing model. That model assumes that for every loop iteration n the values of all actor-level parameters – $\xi_\omega(n)$ – are known exactly, so that the processing times can be accurately calculated for each iteration of every actor.

In Section 6.2.5, we transform the detailed timing mode into a less thorough actor timing model, reducing the performance analysis overhead at the cost of giving up some accuracy. This timing model is referred to as the *multi-scenario delay (MSD) mode*. The main idea of this mode is to reduce the amount of information in the actor timing model by applying quantization to it. Recall that the core of the MSD mode is a set of *scenarios*, each scenario defining a quantization level for the actor processing times and for all actor-level parameters. Unlike the detailed mode, the MSD mode requires only one set of actor-level parameter values per scenario s – $\hat{\xi}_{\omega,s}$ – and not per loop iteration n.

The MSD mode brings us closer to our goal, which is to enable the run-time estimation of the execution times. Recall that our performance analysis method estimates the execution times as a linear combination of loop-level parameters and loop-level coefficients, where the loop-level parameters count the frequency of different scenarios (parameters J_s) and the transitions between them (parameters $K_{s,t}$). As for the loop-level coefficients, they are determined later in the design flow.

The main result of the Application Preparation, is thus the definition of the actor and loop-level parameters to be encoded in the VOP headers: $\hat{\xi}_{\omega,s}$, J_s, $K_{s,t}$ – see Figure 6.2.

Section 6.2.6 is refers to the first and major step of Part II of our design flow, i.e., the intra-application mapping flow. Intra-application mapping is a complex problem solved in multiple mapping steps (see Figure 3.7). In our case study, due to the mapping constraints, only the processing and communication budget assignment steps are left to be done, which is the topic of Section 6.2.6.

Section 6.2.7 considers the second step of Part II (mentioned in the flow overview in Section 2.3.4). This step is optional and it is dedicated to finding the analytical formulas for the loop-level coefficients, to reduce the overhead of the Decoding Time Estimator.

After this step, the VOP Decoding Time Estimator is well-equipped for the run-time estimation of the VOP decoding times (see Figure 6.2). The results of the VOP decoding time estimation are postponed until Section 6.3.

6.2.2 Target Platform

In this subsection we describe not only the target platform, but also the environment we used to simulate it.

For this case study, the MPEG-4 application was written in C++ and then compiled and tested for the ARM7TDMI processor architecture, which is a RISC core. To simulate the application, we used the C++ programming environment and the ARMulator simulator provided by ARM Ltd [3], modeling the system timing at the level of processor instructions.

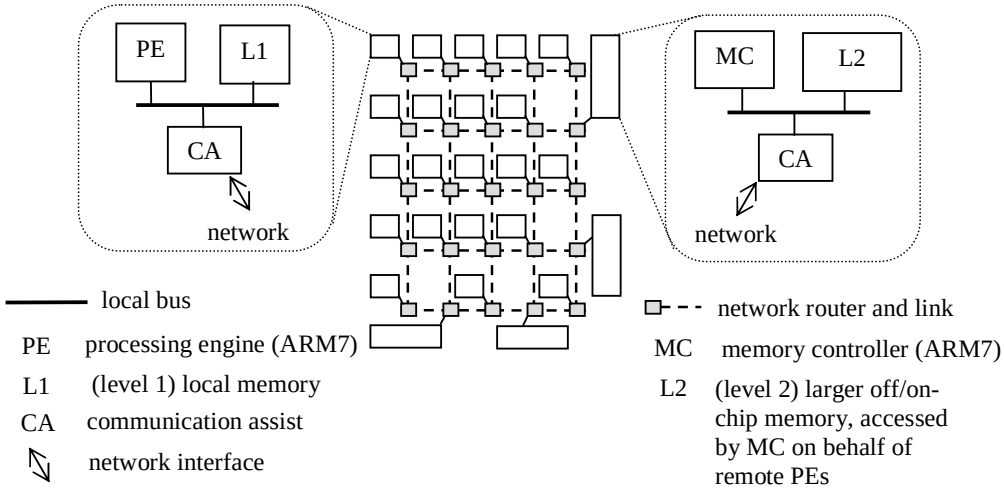


Figure 6.5 Target platform

(the number of PEs and MCs, and the topology are chosen arbitrarily)
(every dynamically created application instance is mapped to one PE and one MC)

We extended the ARMulator single-processor simulation environment to handle the multiprocessor case. Our environment can run several ARMulators in parallel together with a simulation model of the ÆTHEREAL on-chip network for the communication between them. We simulate the network as a set of abstract channels, characterized by fixed bandwidth, latency, input buffer capacity and output buffer capacity.

In this case study, we assume a multiprocessor consisting of a 5x5 matrix of processing tiles, as shown in Figure 6.5. Each processing tile is an ARM7TDMI core running at 100MHz with a certain amount of local memory, enough to accommodate the instruction code and the local data of several computation actors. As explained below and illustrated in Figure 6.5, the tile processors may play one of the two possible functions and their local memory architectures differ respectively. Each processing tile is connected to a local network router. All the routers together make a 5x5 grid topology. Note, that in our implementation, every active MPEG-4 shape decoding application takes only two processing tiles out of 25; multiple such applications can be running on different tiles of the platform if multiple video objects are active in the video presentation. Which two tiles are assigned to which application is decided by the run-time mapping manager, which is outside the scope of this thesis.

Some processors must have direct and fast access to a large memory storage. We call such processors *memory controllers*. Not every processor can afford to have a large local memory resource, and therefore the memory stores and retrieves blocks of data on behalf of the other processors. The other processors, called *processing engines*, use comparatively small local memories and run compute-intensive parts of the application with a highly localized range of memory accesses (delegating the wide-range accesses to the memory controllers). Typically, the hardware architecture of a memory controller would be optimized for efficient transfers of data blocks, handling multiple data transfers in parallel and performing memory address calculations efficiently. Also, typically a processing engine would be an application-domain specific processor, optimized for the given application domain. However, due to limitations in our experimental setup, we use the general-purpose ARM7TDMI architecture for both the processing engines as well as the memory controllers.

Table 6.1 Network constants used in this chapter

Notation	Meaning	Value
$T_{\text{T Æ}}$	TDMA period of the network routers	192 μs
$B_{\text{LINK-Æ}}$	Link bandwidth in one direction	$8 \cdot 10^6$ byte/s
$d_{\text{ROUTER-Æ}}$	Latency per router	0.75 μs
$Z_{\text{min-Æ}}$	Network data size granularity	6 bytes
$B_{\text{min-Æ}}$	Network bandwidth granularity	$31.25 \cdot 10^3$ byte/s

Note that the described multiprocessor falls into the generic template we described in Section 3.4, and is similar to the multiprocessor network-on-chip architectures described by other researchers, e.g. by Sander Stuijk *et al* in [88].

In Section 3.4, we referenced a sample set of realistic (i.e., feasible for modern microelectronics technology) characteristics of the ÆTHEREAL on-chip network. In our case study, we assume a network that is functionally the same, but running at 100 times lower clock frequency, i.e., not 400MHz, but only 4 MHz. Therefore, some technological constants we use for the ÆTHEREAL network (see Section 3.4) need to be scaled by a factor of 100. Some of the scaled values are given in Table 6.1.

Later on, in Section 6.2.6, we use this table to calculate the communication delays of our case-study application. The purpose of scaling is to make the communication delays large enough compared to the computation delays of the ARM7TDMI@100MHz architecture; otherwise, the communication would be too much underutilized. Note also that, unlike Section 3.4, where we assume ARM processor clock frequency of 133MHz, in the examples of this chapter we assume a frequency of 100 MHz.

To program the multiprocessor using our methodology, one needs the means to express the processes and channels of the application, as well as to express the resource binding and resource budgets of the processes and channels. For this purpose, we used the YAPI C++ parallel-process simulation library, which gives the basic infrastructure to express the processes and channels [45]. We extended YAPI with the following features:

- running the processes on different instances of the ARMulator;
- modeling the TDMA scheduling of different processes on the same processor;
- modeling the guaranteed-bandwidth ÆTHEREAL network channels.

Using this infrastructure, we simulated the implementation process network of the MPEG-4 shape decoder application. This process network is described in the next subsection.

6.2.3 IPC Graph and Implementation Process Network

In this subsection, we specify the HSDF graph **G** – describing the parallelism and synchronization inside the application – and the *implementation process network* **PQ**, describing the mapping constraints. In this case study, the mapping constraints impose a complete binding of all computation actors to the processes and of all data transfers to the channels. Recall that, in our design flow, every binding decision is reflected in graph **G** by transformations of the graph

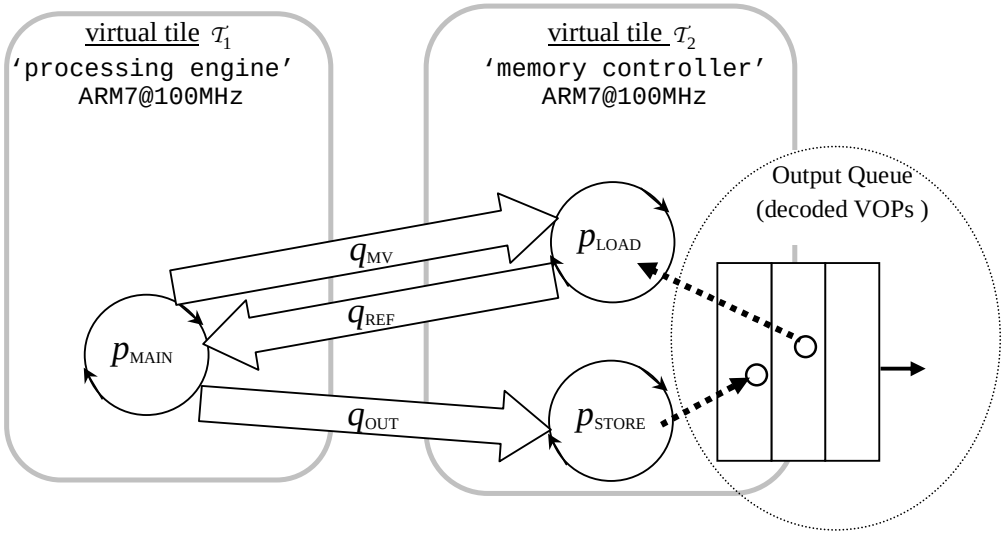


Figure 6.6 Implementation process network of the MB-decoding loop

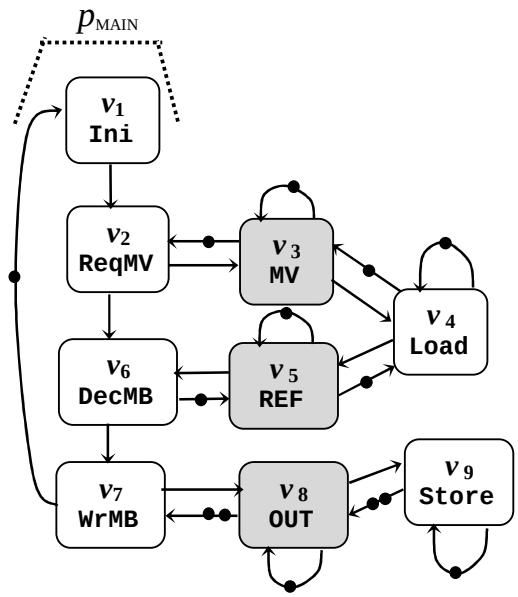
structure. Having a complete binding means having the final structure of graph **G**, and we refer to the final graph structure as the IPC graph (inter-process communication graph). In this case study, the final graph structure is already defined in the beginning of the design flow.

Let us first consider the implementation process network, **PQ**. Recall that it defines a graph-like structure whose nodes are *processes* in set **P**, joined by *channels* in set **Q**. In addition, **PQ** also specifies the process and channel resource budgets, using a data structure called a *budget descriptor*, denoted **B**. Just as for graph **G**, in this case study, the mapping constraints also enforce a certain structure on process network **PQ** and a certain contents on budget descriptor **B**.

Recall from Section 3.5.3 that a budget descriptor specifies, among others, the following design decisions: the set of the virtual tiles { τ_1, τ_2, etc.}, the mapping of processes to the virtual tiles, and the channel capacities (in bytes). For all these settings, we also enforce certain decisions as mapping constraints. The only settings that remain to be decided by the mapping flow are the budgets of the processes *BP*, (in processor cycles per second), the bandwidth of the channels *BQ* (in bytes per second) and the scheduler settings (TDMA periods) of the virtual tiles.

Figure 6.6 shows the structure of the process network of the MPEG-4 shape decoding application. We see three processes – p_{MAIN}, p_{LOAD}, and p_{STORE} – mapped to two virtual tiles – τ_1 and τ_2. Recall that each virtual tile is characterized by its processor type and TDMA period, *T*. Each process p_i executes a cyclic sequence of actors and is only active within a certain TDMA time slot, proportional to the budget, $BP(p_i)$.

The processes are joined by three channels – q_{MV}, q_{REF}, and q_{OUT}. Note that each of them is a network channel because it joins different virtual tiles. Recall that a network channel q_j is characterized by the number of initial tokens in the channel – $m(q_j)$, a set of data transfers mapped to the channel, the producer and consumer buffer capacities – $Q_{\text{prod-buffer}}(q_j)$ and $Q_{\text{cons-buffer}}(q_j)$, and the reserved network bandwidth – $BQ(q_j)$. The data transfers through the channel are defined by triplets 'producer actor, consumer actor, token size' – $\mathbf{TQ}(q_j) = \{(v_{\text{prod}\,k}, v_{\text{cons}\,k}, z_k)\}$.



The communication actors are marked with grey color

Figure 6.7 The IPC graph of the VL/S MB decoding loop (the loop of interest)

Let us consider the contents and the meaning of the processes and channels, at the same time also referring to graph **G**, the IPC graph of the MPEG-4 application. Graph **G** reflects the internal behavior of the processes and channels in the corresponding subgraphs, called *process macros* and *channel macros*. Graph **G** is shown in Figure 6.7.

Let us start a detailed description by process p_{MAIN}. In terms of the MPEG-4 shape decoding algorithm, the role of this process is to parse the bit-fields from the input queue of the application and to decode the shape information, MB by MB. This is a compute-intensive process whose range of memory accesses is limited to a relatively small address space. Therefore, this process is mapped to a virtual tile of type 'processing engine' (and not 'memory controller'). For better illustration of process p_{MAIN}, we refer to its macro contained in the IPC graph **G** in Figure 6.7. That macro consists of a cyclic path in **G** containing actors v_1, v_2, v_6, and v_7.

In the beginning of the process, actor v_1, `Ini` initializes the decoding data structures to start decoding a new MB. For the MB decoding purposes, sometimes an MB from the previous VOP – a reference MB – is needed. Because the decoded VOPs are kept in a large memory module, managed by a remote memory controller, process p_{MAIN} sends a request to process p_{LOAD} running on that controller. To send a request, actor v_2, `ReqMV`, produces a token to channel q_{MV}, thus acting as a channel producer. Note that in case no reference MB is needed, actor v_2 still has to send a token through the channel, because in the HSDF graph, the actors communicate tokens at every iteration. In the case when no reference MB is needed, we assume that actor v_2 produces an 'empty' token containing no useful information, but having the same size as the other tokens communicated through this channel.

The token comes through the channel to the consumer, v_4, `Load`. If the token is non-empty, it contains a motion vector (MV), i.e., the pair of relative coordinates of the reference MB. The corresponding data-transfer triplet is $TQ_{MV} = (v_2, v_4, z_{MV})$, where the token size is $z_{MV} = 8$ bytes (two 32-bit words specifying horizontal and vertical positions). TQ_{MV} is the only transfer in channel q_{MV}, i.e., $\mathbf{TQ}(q_{MV}) = \{ TQ_{MV} \}$.

Process p_{LOAD} runs on a memory controller tile, which has fast physical access to the large memory module where the output queue is contained – see Figure 6.6. p_{LOAD} is represented in graph **G** by one actor, v_4, with a self-edge, i.e., an edge joining an actor with itself. In every execution, that actor consumes one token from channel q_{MV}. If the token is non-empty, the process uses the motion vector coordinates contained in the token to locate the reference MB and to fetch it from the one-but-last VOP in the output queue – i.e., from the previously decoded VOP, see Figure 6.6. The fetched MB goes into channel q_{REF} as a data token. If the incoming token is empty, actor v_4 also produces an empty token in channel q_{REF}. Channel q_{REF} is also described by just one data transfer triplet: $\mathbf{TQ}(q_{\text{REF}}) = \{TQ_{\text{REF}}\}$, $TQ_{\text{REF}} = (v_4, v_6, z_{\text{REF}})$ and $z_{\text{REF}} = 256$ bytes (16x16 pixels of shape data).

After receiving the reference MB (or an empty token) from q_{REF}, process p_{MAIN} decodes the entropy-encoded information contained in the MB. This task is performed by actor v_7, '`DecMB`'. Finally, the main process sends the decoded MB through channel q_{OUT} back to the memory controller, to the process called p_{STORE}. Channel q_{OUT} is described by one transfer triplet: $\mathbf{TQ}(q_{\text{OUT}}) = \{TQ_{\text{OUT}}\}$, where $TQ_{\text{OUT}} = (v_7, v_9, z_{\text{OUT}})$ and $z_{\text{OUT}} = 256$ bytes (16x16 pixels of shape data).

Process p_{STORE} – represented by actor v_9 and a self-edge – picks the output MBs from channel q_{OUT} and copies them to the corresponding (x, y) position inside the current VOP, see Figure 6.6.

Once all MBs of a VOP are decoded, the MB-decoding loop finishes the current execution and a new VOP is released in the output queue.

So far, we mostly focused on the processes; now let us turn our attention to the channels. Two important settings specified for the channels are the number of initial tokens, m (as specified by the application designer) and buffer capacities, BQ (as specified in the mapping constraints). As we see below, these specifications are reflected in the channel macros of graph **G**, according to the methodology described in Section 3.6 (see Figures 3.14 and 3.17).

Because every channel has just one data transfer TQ, each channel macro contains just one transfer actor, shown in Figure 6.7 in gray color. For example, the transfer actor of channel q_{OUT} is v_8, `OUT`.

In addition to the transfer actors, the channel macros would normally contain actors modeling the network latency. However, for the given application and target platform, the latencies can be ignored. The reason is that they are much smaller than the typical processing delay per data token. It can be shown as follows. Recall that in the ÆTHEREAL network, the latency is proportional to the number of the network routers on the network path. The longest 'reasonable' path in a 5x5 grid topology includes at most 10 network routers (the maximal Manhattan distance). Based on Table 6.1 and Equality (3.22), it is equal to $10 \times 0.75\ \mu s = 7.5\ \mu s$. On the other hand, according to the profiling performed on our processor simulator, the minimum time to copy one token (256 bytes) from channel q_{OUT} to the background memory using an ARM7TDMI@100MHz is approximately 1000 μs. This is much larger than the maximal latency. Therefore, for simplicity, the network latency actors are not included into the channel macros in our example.

As we know from Section 3.6.3, the producer and the consumer buffers of the network channel are modeled by so-called forward and backward edges. For the producer buffers, the forward edges go from the channel producers to the transfer actors. For the consumer buffers, they go from the transfer actors to the channel consumers. The backward edges go in the reverse

direction. For example, the forward edges of channel q_{OUT} are (v_7, v_8) and (v_8, v_9) and the backward edges are (v_9, v_8) and (v_8, v_7).

The initial tokens of a channel, if any, are located at the forward edges of the producer buffers. In this case study, when the MB-decoding loop starts, the channels are initially empty: $m(q_j) = 0$. Therefore, the forward edges are also free from the initial tokens.

The channel buffer capacities, $Q_{\text{prod-buffer}}(q_j)$ and $Q_{\text{cons-buffer}}(q_j)$, (i.e., the numbers of tokens that can fit in the producer and consumer queues of the channel) are modeled in the IPC graph using the backward edges. In this case study, all the channel buffers are so-called simple channels; they have only one producer and consumer. Therefore (also given the fact that these channels do not have initial tokens), the following simple rule applies: the number of initial tokens at the backward edge is equal to the buffer capacity when specified in tokens.

For channels q_{MV} and q_{REF}, we fix the capacities of the producer and the consumer buffers to one token. For channel q_{OUT}, we fix them to two tokens. As we see in Figure 6.7, this choice is reflected in the number of tokens on the backward edges.

Let us motivate the chosen capacities. Because process p_{MAIN} first waits for a response to each reference-MB request before it issues another request, no two requests can be pending at the same time. Therefore, channels q_{MV} and q_{OUT} only need one token place at the producer and consumer buffers. For channel q_{OUT}, the situation is different, because processes p_{MAIN} and p_{STORE} can run in parallel to each other and in parallel to the data transfers in channel q_{OUT}. To ensure that this parallelism is possible, both producer and consumer buffers need a place for at least two tokens. If the buffers could fit only one token each, actors v_7 and v_8 would always execute sequentially, and actors v_8 and v_9 as well.

In this and the previous subsection, we covered the input specification for implementing our case-study application. Hereby, we pre-constrained most of the mapping decisions. Therefore, in the following subsections, we can put more emphasis on the performance analysis aspects than on the mapping aspects, as intended for this case study.

6.2.4 Detailed Actor Timing

In this subsection, for our case study, we report on the *actor-level parameter identification* (i.e., defining the parameters as functions of the input data) and *actor-level characterization* (i.e., finding the contributions of the actor-level parameters to the actor processing times).

The goal of this section is, for every actor, to obtain actor parameter functions which closely estimate the real actor processing times. The parameter functions constitute the detailed timing mode, which is the most accurate actor delay model in our modeling approach. The parameter functions give the linear relationship between the actor processing time $t(v_k, n)$ and the parameters:

$$t(v_k, n) = C_{k,0} + \sum_{\omega=1}^{\Omega} C_{k,\omega} \cdot \xi_\omega(n) \tag{6.1}$$

where $C_{k,\omega}$ are the actor-level coefficients and $\xi_\omega(n)$ are the actor-level parameter values in loop iteration n. Typically, each actor v_k is only influenced by a small subset $\mathbf{\Omega}_k$ of parameters, and only the parameters from that subset have non-zero coefficients $C_{k,\omega}$. Note that, for a conservative performance analysis, the coefficients have to be conservative.

The goal of the parameter identification is to identify the parameters in subset $\mathbf{\Omega}_k$ for each actor v_k, ensuring that as few parameters as possible are introduced, but at the same time that the required level of accuracy is achieved. The parameter identification requires analyzing the actor execution algorithms. One technique for doing the parameter identification is sketched in Section 3.2.2, using the VLD actor of the JPEG decoder as an example; however that technique is not automated. In this case study, we performed the parameter identification manually, based on the knowledge of the application algorithm. This resulted in a grand total of 17 actor-level parameters. We do not report the details of parameter identification, but, later in this subsection, we summarize the ten most influential parameters in a table. Note that we do not mention all of them in order to not overload the text with too detailed application-specific information, but nevertheless we use all 17 parameters in the performance analysis.

In the rest of this subsection we focus mostly on the actor-level characterization, i.e. the calculation of $C_{k,\omega}$. In see Section 3.3, we described two alternative methods for that, namely, the direct measurement and the linear regression.

The *linear-regression method* gives certain probabilistic guarantees on the accuracy and the conservativity of the processing time estimate. Recall that the control setting $p_{\text{coef}} < 1$ controls the degree of pessimism in estimating the coefficients, and the closer to 1 this setting is, the higher the probability that the parameter function time $t(v_k, n)$ defined in Equality (6.1) gives a higher estimate than the real processor cycle count. However, we do not want p_{coef} to be too close to 1, because then the accuracy of estimating the processor cycle count by $t(v_k, n)$ will suffer. In many use cases of linear regression, the usual practical setting used for p_{coef} is 0.95. We also use this default setting in our experiments. As we saw in practice, sometimes, for achieving a better accuracy, it helps to split the actor into a few subroutines and to apply linear regression to them separately. The advantage of the linear-regression method is that quite often it reduces the manual effort compared to the other method, especially if the number of parameters in $\mathbf{\Omega}_k$ is three or more (i.e., $|\mathbf{\Omega}_k| \geq 3$). See Section 3.3 for more details on the linear-regression method.

The other method, i.e., the *direct-measurement method*, is closely related to so-called worst-case execution time (WCET) calculation, giving much stricter guarantees on the conservativity of than the linear regression. This is important for hard real-time applications, especially for the safety-critical ones. This method is easier to apply if the control flow is relatively simple, especially when the processing can be characterized by less that three parameters (i.e., for actors where $|\mathbf{\Omega}_k| < 3$).

As we see from Figure 6.7, the graph contains six computation actors and three transfer actors. The Application Preparation part of the design trajectory (which we are doing now) focuses only on the computation actors. The transfer delays are constant and independent of the parameters; they are calculated later on in the trajectory.

In this case study, we use both the mentioned methods. For the actors with $|\mathbf{\Omega}_k| \geq 3$, we use linear regression. Hereby, we sometimes split an actor into a few subroutines, mostly because these subroutines have clearly different functionality and sometimes because this improves the accuracy of the linear-regression results. For the actors with $|\mathbf{\Omega}_k| < 3$, we use direct measurement, to achieve better guarantees on conservativity.

Because the MPEG-4 decoder is a soft real-time application, not a safety-critical one, we do not need the parameter functions to be strictly conservative. Instead, we use coefficients that are conservative with a high probability, and we obtain them empirically from measurements based on a *representative stream* of input data samples.

The chosen representative stream consists of the first 30VOPs of the 'singer' sample stream for MPEG-4 shape coding developed in the MoMuSys project [74]. Each VOP in this 30-VOP substream contains 60 to 96 MBs, leading to a grand total of 1891 MBs.

In both methods, we use the same representative input stream. Both methods use profiling, i.e., running the application code in the simulator and measuring actor processing times.

The accuracy of the actor parameter functions depends on whether the representative stream and the parameter subsets $\mathbf{\Omega}_k$ are chosen correctly. To evaluate this, we use two kinds of metrics:

1) linear regression quality metrics – $err(\mathbf{c})_g$ and $R^2{}_g$;

2) estimation error metrics: average overestimation, maximum overestimation, probability of overestimation ($e_{\text{t-avg-over}}$, $e_{\text{t-max-over}}$, p_{over}) and the same for underestimation ($e_{\text{t-avg-under}}$, $e_{\text{t-max-under}}$, p_{under}).

Let us explain all these metrics before we proceed to reporting them for our case study.

The regression quality metrics are measured for each actor subroutine, g, for which linear regression is applied. Metric $err(\mathbf{c})_g$ estimates the uncertainty in the calculation of the coefficients; the ideal value for this metric is 0%. Metric $R^2{}_g$ is introduced because the parameters typically do not capture all the sources of dynamic variations of subroutine processing times. This metric gives the best percentage of the processing time variability that the parameters still can capture; the ideal value for this metric is 100%. More details on the linear regression quality metrics can be found in Section 3.3.

The other metrics are based on the *estimation error* e_t. It is defined as follows:

$$e_t(v_k, n) = \frac{t(v_k, n) - t_{\mathbf{prof}}(v_k, n)}{\mathbf{E}(\, t_{\mathbf{prof}}(v_k, n)\,)} \tag{6.2}$$

where $t(v_k, n)$ is the actor parameter function (see Equality (6.1)), $t_{\mathbf{prof}}(v_k, n)$ is the actor cycle count measured from the profiling run, and '$\mathbf{E}(\ldots)$' calculates the average value of the argument over a range of different n, which is, in our experiments, the range of all MBs in the given sample input stream.

Based on e_t, our estimation error metrics are defined as shown in Table 6.2.

Unlike the linear-regression metrics, which are measured on the representative stream, the estimation-error metrics are measured on the *sample stream*, i.e., an input data stream (longer than the representative stream) used for the evaluation of the accuracy of our performance analysis approach. We have two such streams: 'singer' and 'dancer', from the MoMuSys project [74]. We use them throughout this case study to evaluate the accuracy and conservativity of different performance analysis stages. Their basic characteristics (e.g., total number of VOPS) are summarized in Table 6.3. Note that the size of the complete 'singer' stream is much larger than the size of the sub-stream we used as reference stream (30 VOPs, 1891 MBs). Therefore, we consider it fair to use the complete stream to evaluate the accuracy of the coefficient calculation done based on such a small sub-stream.

The rest of this subsection is built around the four tables with results. First, Table 6.4 summarizes actor-level parameter identification. It is followed by Tables 6.5 and 6.6, where the estimation error is evaluated for two sample streams. Finally, Table 6.7 explains the parameter and subroutine names mentioned in this section.

Table 6.2 Estimation error metrics.

Metric	**Definition**	**Explanation**
$e_{\text{t-avg-over}}$	$^{*}\mathbf{E}(e_t \mid e_t > 0)$	Average e_t for overestimation
$e_{\text{t-max-over}}$	$\text{Max}(e_t \mid e_t > 0)$	The maximum e_t for overestimation
p_{over}	$\mathcal{P}r(e_t > 0)$	The probability of overestimation
$e_{\text{t-avg-under}}$	$\mathbf{E}(-e_t \mid e_t < 0)$	Average e_t for underestimation
$e_{\text{t-max-under}}$	$\max(-e_t \mid e_t < 0)$	The maximum absolute value of e_t for underestimation
p_{under}	$\mathcal{P}r(e_t < 0)$	The probability of underestimation

$^{*}\mathbf{E}(X \mid C)$ is the average X for the cases where condition C holds and $\mathcal{P}r(C)$ is probability of condition C.

Table 6.3 Sample input streams.

Name	**#VOPs**	***N*, #MBs per VOP**	**#MBs**
'singer'	250	45-204	24708
'dancer'	250	77-255	32915

Recall that we split some actors into subroutines. Table 6.4 summarizes the parameter identification statistics for every subroutine. The first column identifies the actor owning the subroutine. The second column gives the mnemonic name of the subroutine together with the subroutine's average processing time and the call count per actor execution. For example, '1 k x 6' means that the average processing time is 1 kilocycles (of the ARM7TDMI core) and that the subroutine is called 6 times per actor execution. The third column gives the total number of parameters (Ω_{sub}) influencing the given subroutine and the number of new parameters ($\Delta\Omega$), i.e., the ones that are not shared with the subroutines reported earlier. It also mentions one or two most influential parameters (whose meaning is explained in Table 6.7). Hereby we measure how 'influential' a parameter is as product of the measured dynamic range of parameter values and the parameter coefficient. Note that we do not mention the less influential parameters not because we do not use them, but to avoid overloading our reporting with application-specific information. The fourth column reports which characterization method was used. The last two columns report the linear-regression metrics, in the cases where linear regression was used.

From Table 6.4, we see that for all cases where the linear regression was used, the parameters accounted for at least 93% of the processor cycle count variations, and that the uncertainty of the actor-level coefficients was at most 14%. This indicates a reasonable accuracy of the actor processing time estimates.

We also used graphical plots to evaluate the accuracy visually. For example, Figure 6.8, shows the curves for subroutine '`TextMV`', for one of the VOPs of sample sequence 'singer', from which we see that the detailed timing mode accurately models the processing cycle counts, sometimes underestimating them and sometimes overestimating them. Note that the VOP we

Table 6.4 Parameter identification and actor characterization

Actor	**Subroutine**	$\mathbf{\Omega_{sub}}_g$	**Method**	$err(\mathbf{c})_g$	$R^2{}_g$
v_1 `Ini`	`Ini` 2 k x 1	$\Omega_{sub} = 0$	direct	-	-
v_2 `ReqMV`	`ReqMV`(*) 1 k x 1	$\Omega_{sub} = 4$ $\Delta\Omega = 4$ $\xi_{ref}, \xi_{bbMV} \in \mathbf{\Omega_{sub}}$	lin-reg	14 %	97 %
v_4 `Load`	`Load` 120 k x 1	$\Omega_{sub} = 1$ $\Delta\Omega = 0$ $\xi_{ref} \in \mathbf{\Omega_{sub}}$	direct	-	-
v_6 `DecMB(1)`	`DecodeCAE` 116 k x 1	$\Omega_{sub} = 2$ $\Delta\Omega = 1$ $\xi_{ref}, \xi_{CAE} \in \mathbf{\Omega_{sub}}$	lin-reg	1 %	99 %
v_6 `DecMB(2)`	`CBP` 3 k x 1	$\Omega_{sub} = 2$ $\Delta\Omega = 2$ $\xi_{bnd}, \xi_{empty} \in \mathbf{\Omega_{sub}}$	lin-reg	3 %	~100%
v_6 `DecMB(3)`	`TextMV` 3 k x 1	$\Omega_{sub} = 3$ $\Delta\Omega = 3$ $\xi_{P}, \xi_{Nmv} \in \mathbf{\Omega_{sub}}$	lin-reg	12 %	99 %
v_6 `DecMB(4)`	`VLD` 1k x 6	$\Omega_{sub} = 9$ $\Delta\Omega = 9$ $\xi_{ne}, \xi_{bbVLD} \in \mathbf{\Omega_{sub}}$	lin-reg	11 %	~100 %
v_6 `DecMB(5)`	`EndMB` 6 k x 1	$\Omega_{sub} = 1$ $\Delta\Omega = 1$ $\xi_{tex} \in \mathbf{\Omega_{sub}}$	lin-reg	3 %	93 %
v_7 `WrMB`	`WrMB` 98 k x 1	$\Omega_{sub} = 0$	direct	-	-
v_9 `Store`	`Store` 102 k x 1	$\Omega_{sub} = 0$	direct	-	-

(*) – all acronyms are explained in Table 6.7

have chosen to illustrate the accuracy in Figure 6.8 is chosen such that we can illustrate possible extreme deviations observed for this actor, and this VOP is outside the range of the first 30 VOPs used to calculate the coefficients

The accuracy and the conservativity of the processing times is directly measured by the estimation error metrics, presented in the next two tables, Table 6.5 and 6.6, for the two sample streams. In those tables, we skip actors v_4(`Load`), v_7(`WrMB`) and v_9(`Store`), because the estimation error for those actors is negligible and can be considered zero.

Table 6.5 Detailed-mode estimation error for ‘singer’ stream

Actor	$e_{\text{t-avg-over}}$	p_{over}	$e_{\text{t-max-over}}$	$e_{\text{t-avg-under}}$	p_{under}	$e_{\text{t-max-under}}$
v_1 `Ini`	136 %	1.0	158 %	-	0.0	-
v_2 `ReqMV`	4 %	0.87	107 %	10 %	0.13	152 %
v_6 `DecMB`	1 %	0.71	7 %	2 %	0.29	21 %

Table 6.6 Detailed-mode estimation error for ‘dancer’ stream

Actor	$e_{\text{t-avg-over}}$	p_{over}	$e_{\text{t-max-over}}$	$e_{\text{t-avg-under}}$	p_{under}	$e_{\text{t-max-under}}$
v_1 `Ini`	147 %	1.0	165 %	-	0.0	-
v_2 `ReqMV`	5 %	0.87	104 %	15 %	0.13	152 %
v_6 `DecMB`	1 %	0.80	6 %	2 %	0.20	33 %

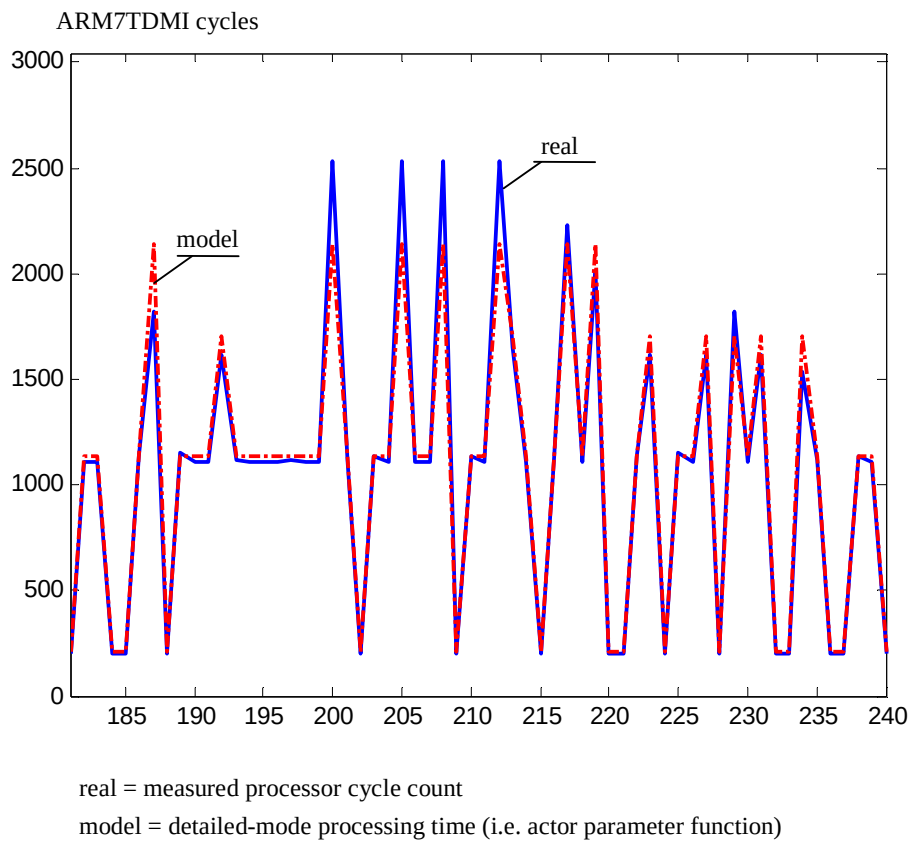


Figure 6.8 Subroutine `TextMV`: cycle count curve for a VOP in ‘singer’ stream

In Tables 6.5 and 6.6 we see similar results for the two different streams. As expected, the probability of conservative estimation (i.e. overestimation) is in all cases much larger than the probability of underestimation. For actor v_4 (`Ini`), we tolerate the huge average error because this actor has a comparatively small contribution of 2 k cycles per actor execution to the total processor cycle count. For the other actors, all analyzed using linear regression, the average

errors are reasonably small. We also see that considerable underestimation can occur. (Note that > 100% underestimation means that, for some MBs, the value of $t_{prof}(v_k, n) - t(v_k, n)$ is larger than $\mathbf{E}(t_{prof}(v_k, n))$.) Nevertheless, the multi-scenario-delay mode (described in the next subsection), brings the underestimation down considerably. Also, the fact that the underestimation has a low probability makes it less of a concern, because the final performance analysis (i.e., VOP decoding time estimation in this case) integrates the actor processing times over multiple iterations, whereby frequent overestimation is likely to cancel the occasional underestimation.

The last table, Table 6.7, explains the meaning of some key actor-level parameters and subroutines. The purpose of that table is to give an example of what the actor-level parameters can mean in practice. Note, however, that it is not necessary to understand all the details in this table.

Table 6.7 The meaning of actor-level parameters and subroutines

Parameter/ subroutine	**Meaning**
`Ini`	Initialization of MB decoding.
`ReqMV`	If required, sends a request to load a reference MB, hereby decoding and sending the motion vectors (i.e., the relative coordinates of that MB)
ξ_{ref}	Boolean, equal to ‘1’ if and only if a reference MB is required.
ξ_{bbMV}	The number of times a new byte is shifted into the 64-bit word used as a bit-stream parsing cache when decoding the shape motion vectors.
`Load`	Loading of the reference MB from the previous VOP.
`DecodeCAE`	Decodes the current MB shape, using the reference MB; often involves the decoding of the context-arithmetic-encoded (CAE) data
ξ_{CAE}	Boolean, equal to ‘1’ if and only if the context-arithmetic decoding is involved.
`CBP`	Decodes the ‘code-bit-pattern’, i.e., determines which of the six sub-blocks of the MB are empty (transparent).
ξ_{bnd}	Boolean, equal to ‘1’ if and only if the MB is located at the contour of a video object (i.e., determines whether shape decoding is active for this MB).
ξ_{empty}	Number of zero (i.e., transparent) pixels, traversed when scanning the MB sub-blocks in search for non-zero pixels.
`TextMV`	Decodes the motion vectors for reference texture blocks
ξ_{P}	Boolean, equal to ‘1’ if and only the current VOP is a P-VOP
ξ_{mv}	Number of texture motion vectors encoded in the current MB.
`VLD`	Performs variable-length decoding for the texture information.
ξ_{ne}	Number of AC symbols encoded by so-called ‘non-escape’ codes.
ξ_{bbVLD}	Same as ξ_{bbMV}, but for VLD decoding.

`EndMB`	Finishes the processing of an MB.
ξ_{tex}	Boolean, equal to '1' if the MB block contains any non-uniform texture, and equal to '0' if the MB block is transparent or if all non-transparent pixels have the same color.
`WrMB`	Writes the decoded MB to the memory.
`Store`	Stores the MB into the current VOP memory in the output queue

Having obtained the detailed-timing model of the application, we proceed with reducing the amount of detail in the timing models and finalizing the implementation of the application. In the next subsection, we step from the detailed mode to the multi-scenario delay mode.

6.2.5 Scenario-based Actor Timing

Recall that scenarios are defined as *subspaces* of the space of possible values of the most influential actor-level parameters, referred to as *primary parameters*. The subspaces define the conditions for the quantization of detailed-timing actor delay functions. As we mentioned in Section 5.3, subdividing the space of parameter values into scenarios can be done either manually or using automated technique, as the one proposed in [28], [24].

To define scenarios in this case study, we followed the manual approach. First, we identified eight major types of macro-blocks defined in the decoding algorithm, considering each type as a scenario candidate as it showed similar decoding delays for different blocks and corresponded to a certain combination of values of the most influential actor-level parameters. The candidates with similar contribution to execution time were merged, yielding three scenarios in the end, which are defined using two primary parameters: ξ_{ref} and ξ_{CAE} (see Table 6.7 for their definition).

The scenarios are defined as follows:

Example Scenario 1: (s = 1) $\xi_{ref} = 0$ ♦

Example Scenario 2: (s = 2) $\xi_{ref} = 1$, $\xi_{CAE} = 0$ ♦

Example Scenario 3: (s = 3) $\xi_{ref,} = 1$, $\xi_{CAE} = 1$ ♦

Recall that the delay quantization works as follows: for all the data tokens that satisfy the scenario condition, maximum values of all actor parameters are found and used to calculate the delay quantization level. In the given case study, the quantization affects actors v_1, v_2, and v_6, because the other actors have delays that are constant in every scenario.

Figure 6.9 demonstrates the effect of quantization on the delay of actor v_6 (`DecMB`). As we see from that figure, the quantization removes some delay variations, making the timing model less detailed and more conservative.

Similarly to our experiments on the detailed timing mode, we evaluated the multi-scenario mode for the two video streams and summarized the results in Tables 6.8 and 6.9 below.

Comparing these results to Tables 6.5 and 6.6, for actor `Ini`, we see the same behavior as in the detailed timing model, because the delay of that actor does not depend on any of the primary parameters. For the other two actors, we see that the probability and the magnitude of overestimation have increased considerably. This is in line with our expectations that the multi-scenario delay mode is more conservative than the detailed timing mode.

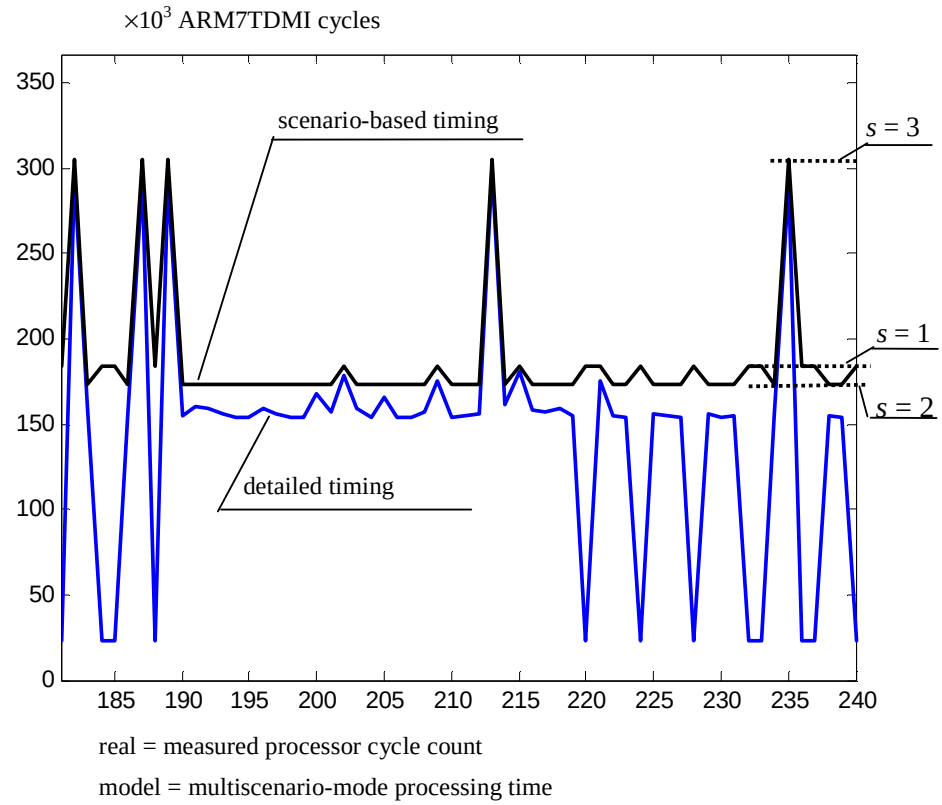


Figure 6.9 Actor v_6 (`DecMB`) – processing time in multiscenario mode

Table 6.8 Estimation error of MSD mode for the 'singer' stream

Actor	$e_{\text{t-avg-over}}$	p_{over}	$e_{\text{t-max-over}}$	$e_{\text{t-avg-under}}$	p_{under}	$e_{\text{t-max-under}}$
v_1 `Ini`	136 %	1.0	158 %	-	0.0	-
v_2 `ReqMV`	67 %	0.98	279 %	27 %	0.02	143 %
v_6 `DecMB`	75 %	0.99	199 %	4 %	0.01	19 %

Table 6.9 Estimation error of MSD mode for the 'dancer' stream

Actor	$e_{\text{t-avg-over}}$	p_{over}	$e_{\text{t-max-over}}$	$e_{\text{t-avg-under}}$	p_{under}	$e_{\text{t-max-under}}$
v_1 `Ini`	147 %	1.0	165 %	-	0.0	-
v_2 `ReqMV`	62 %	0.98	250 %	34 %	0.02	152 %
v_6 `DecMB`	56 %	0.995	149 %	3 %	0.005	15 %

We also see that the average error has reached a level of over 50%, which, as it seems, threatens to affect the accuracy of our performance analysis. In reality, it is not a problem. Let us ignore actor `ReqMV`, which, according to Table 6.4, has two orders of magnitude smaller average processing time than `DecMB`. For `DecMB`, as we see from Figure 6.9, we could have considerably reduced the scenario overestimation by introducing one extra scenario that would distinguish the

smallest actor delay values, which occur on a regular basis. According to our extra experiments with stream 'dancer', this would take the MSD-mode overprediction error down to 13% on average and 41% maximum. Nevertheless we did not introduce that scenario, because we observed that for the final results – the estimation of the VOP decoding times – that scenario appears to bring no extra improvement. The reason for that lies in the structure of the HSDF graph, see Figure 6.7, and we can explain it in an informal way as follows. In general, the cyclic paths of the HSDF graph determine its performance. The cyclic paths containing `DecMB` compete with the other cyclic paths for the impact on the graph throughput. The processing time of actor `DecMB` is comparatively large on average (see Table 6.4), so it is a 'major contributor' for all its cyclic paths. Therefore, when actor `DecMB` gets a very small value, the other cyclic paths (in particular, those that contain actor `Store`) win the competition, become the performance bottleneck and hide the influence of `DecMB`. Thus, reducing the MSD mode error is not necessary in this case.

What we also see when we compare MSD-mode results to the detailed-mode results, is that the probability of underestimation has decreased. It is interesting to observe that the value of underprediction has increased, but this is due to the fact that we measure underprediction relative to the average processing time of the samples with underprediction, whereas this average value has decreased.

At this point, the actor delay model has given up some detailed information that was present there originally, but still stays accurate enough to obtain reasonable accuracy in the end (as we see later in the results of VOP decoding time estimation).

6.2.6 Budget Assignment

Whereas in the previous two subsections we considered the first part of the design flow, i.e., the Application Preparation, in this subsection we turn our attention to the second part, the Intra-application Mapping Flow, which optimizes the mapping of the application to the target platform resources.

Recall that, due to the mapping constraints, the only intra-application mapping decisions left to be made in this case study are the processor cycle budgets allocated to the processes and the amount of the communication bandwidth assigned to the channels. We refer to these decisions as the *budget assignment*. Referring back to Figure 3.7, we note that, in the generic design flow, the budget assignment is part of the processing assignment and communication assignment steps. Before this point of the design flow, we reasoned about the actor delays using the actor processing times, $t(v_k, n)$, measured in processor clock cycles. The budget assignment effectively translates the processing times into actor delays, $d(v_k, n)$, measured in the real time units. By taking the budget assignment into account, the performance analysis can analyze the application performance in the real-time domain.

As an optimization problem, budget assignment for this case study can be described by the following subtopics:

a. problem instance,
b. controllable variables (i.e., the structure of a problem solution),
c. objectives and constraints,
d. problem solution.

Table 6.10 The structure of a problem solution

Objects (processes, channels)	**Primary variables**	**Secondary variables**
p_{MAIN}, p_{LOAD} and p_{STORE}	$BP(p_i)$ – processor cycle budget (clock cycles per second) $\mathcal{T}_{\text{sched}}(\mathcal{T}(p_i))$ – the vector of local scheduler variables for the processing tile where process p_i is running (depends on the scheduling method being used)	$T_{\text{B}}(p_i)$ – TDMA time slot reserved for process p_i (in seconds) (see Section 3.1.3) $T_{\text{T}}(\mathcal{T}(p_i))$ – the TDMA period of the local scheduler (in seconds)
q_{MV}, q_{REF} and q_{OUT}	$BQ(q_i)$ – channel bandwidth (bytes per second)	$n_{\text{slots}}(q_i)$ – the number of TDMA slots of the ÆTHEREAL network-on-chip connection allocated for the given channel (see Section 3.4)

Problem instance. The problem instance consists of the implementation process network (recall Figure 6.6) and IPC graph (recall Figure 6.7). For every computation actor, we specify a typical (i.e., average) processing time. We use the average times reported in the first column of Table 6.4 (in kilocycles of the ARM7 processor):

$t(v_1) = 2\ \texttt{k}$; $t(v_2) = 1\ \texttt{k}$; $t(v_4) = 120\ \texttt{k}$; $t(v_6) = 134\ \texttt{k}$;
$t(v_7) = 98\ \texttt{k}$; $t(v_9) = 102\ \texttt{k}$;

For every transfer actor, we specify the size of the data tokens, in accordance to Section 6.2.3:

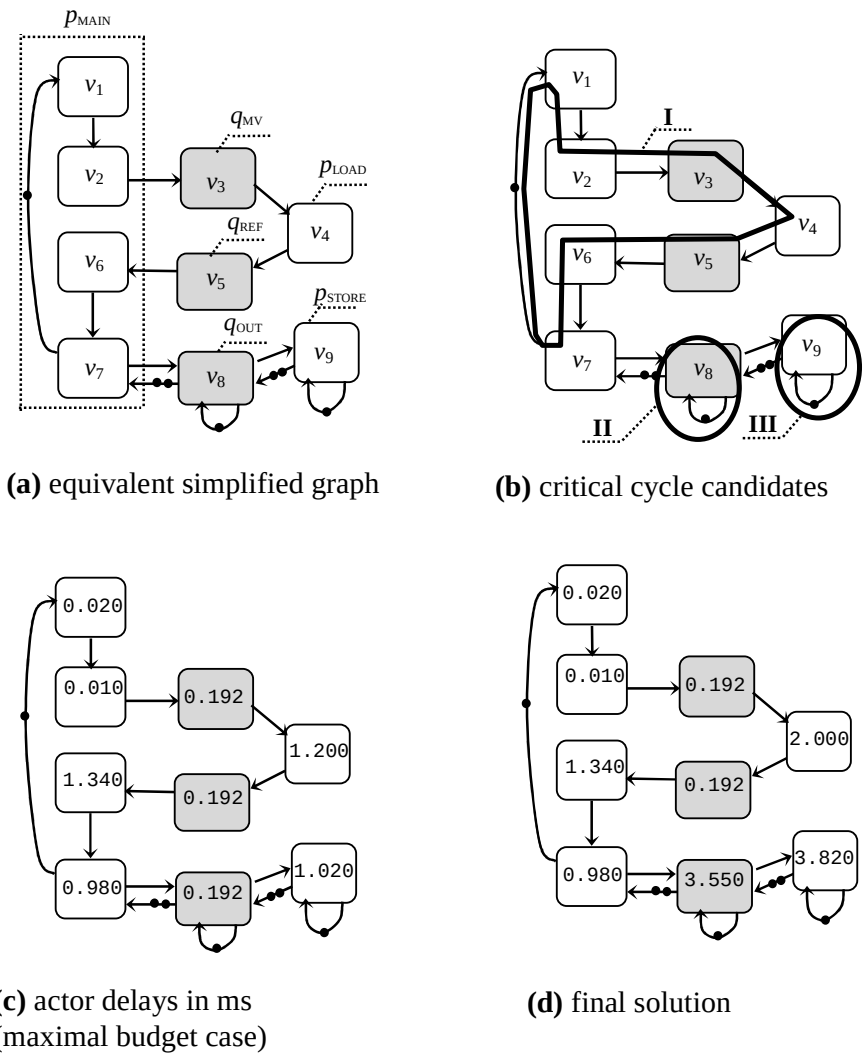
$z(v_3) = 8\ \texttt{bytes}$; $z(v_5) = 256\ \texttt{bytes}$; $z(v_4) = 256\ \texttt{bytes}$;

Note that hereby we partly specify the typical static timing mode – see Section 3.1.1 for an overview of the timing modes.

Controllable variables. Referring back to Figure 6.6, we see that there are three processes and three channels in this case study. For each channel and process, the budget assignment determines certain budget variables. We refer to these variables as primary budget variables. They form a *solution* to the budget assignment problem. The primary variables are functions of a few platform-dependent variables, which we call secondary variables. Given the target platform, from the secondary variables we can determine the primary variables and vise versa. We use the secondary variables for convenience of explanation. Recall that $\mathcal{T}(p_i)$ is the notation for the tile to which process p_i is mapped.

The relationship between the primary and the secondary variables for our case study can be expressed as follows:

$$BP(p_i) = \begin{cases} F_{\text{clock}}(\mathcal{T}(p_i)) \text{ if processing tile } \mathcal{T}(p_i) \text{ does not use scheduling} \\ F_{\text{clock}}(\mathcal{T}(p_i)) \times T_{\text{B}}(p_i) / T_{\text{T}}(\mathcal{T}(p_i)) \text{ if the TDMA scheduling is used} \end{cases}$$



(a) equivalent simplified graph

(b) critical cycle candidates

(c) actor delays in ms (maximal budget case)

(d) final solution

Figure 6.10 Analyzing and solving the budget assignment problem

$$\mathcal{T}_{\text{sched}}(\mathcal{T}(p_i)) = \begin{cases} (\text{'NO\_SCHEDULING'}) \text{ if processing tile } \mathcal{T}(p_i) \text{ does not use scheduling} \\ (\text{'TDMA'}, T_{\text{T}}(\mathcal{T}(p_i))) \text{ if the TDMA scheduling is used} \end{cases}$$

$BQ(q_j) = n_{\text{slots}}(q_i) \cdot B_{\text{min-Æ}}$

where F_{clock} is processor clock frequency and $B_{\text{min-Æ}}$ is the granularity of network bandwidth allocation, which, for the ÆTHEREAL network, is defined in Table 6.1.

Objectives and constraints. The budget assignment problem requests to find the budget variables (either primary or secondary) such that the minimum required throughput constraint for the given application is satisfied at the minimum resource usage.

We assume the minimum throughput constraint is indirectly specified by the maximum allowed iteration interval: $\lambda_{\text{allowed}} = 5$ ms.

The minimum resource usage objective requests to use as small as possible values of the control variables $BP(p_i)$ and $BQ(q_j)$, seen as multiple cost functions. A possible problem formulation would request to find the so-called set of Pareto points in the space of control variables, yielding a set of alternative solutions that have a property that each solution is better than any other feasible solution by at least one control variable $\{BP(p_i)\}$ or $\{BQ(q_j)\}$. Another

problem formulation would minimize a weighted sum of $\{BP(p_i)\}$ and $\{BQ(q_j)\}$. However, since it is not the purpose of this case study to evaluate any budget assignment algorithm, we do not adopt any of those formulations, and try to find a solution using as small budgets as possible relying on logical reasoning. Note that algorithms for closely related optimization problems in the same context were proposed by Sander Stuijk *et al* in [88] and Orlando Moreira *et al* in [66].

Problem solution. To solve the budget assignment problem, we first simplify the problem instance. Figure 6.10(a) shows an equivalent simplified version of the IPC graph in Figure 6.7, also showing how the graph is partitioned into the process and channel macros. A few edges have been removed as superfluous. For each removed edge it holds that the graph contains a path that joins the same pair of actors as the edge and contains the same number of initial tokens. For example, edge (v_3, v_2), containing one initial token, is superfluous, because v_3 and v_2 are joined by a path – (v_3, v_4, v_5, v_6, v_7, v_1, v_2) – that contains one initial token too.

After the simplification, the graph has five simple cyclic paths, and only three of them – highlighted and indexed in Figure 6.10(b) by Roman numbers I, II and III – are potential critical cycle candidates (the two simple cycles which are not included have two initial tokens each and thus, in this particular case, cannot become critical).

The purpose of this exercise is to assign as small as possible budgets to the processes and channels such that none of the three critical cycles gets a total delay exceeding `5 ms`, which is our maximal iteration-interval constraint, λ_{allowed}.

Let us first assume that all processes and channels get the maximum budgets. That means that all computation actors get typical delays equal to the processing times divided by the processor clock frequency (100 MHz for our target platform). Given the processing times as specified in the problem instance, we get the actor delays as shown in Figure 6.10(c). The delays of the transfer actors are all equal to `0.192 ms`, i.e., the network TDMA period, $T_{\text{T}Æ}$, according to Table 6.1. Assuming maximal bandwidth allocation, this is the delay within which the network is guaranteed to transfer one token that fits within one TDMA period. Recall that within one TDMA period of the ÆTHEREAL has 256 data slots, with 6 bytes per slot (in our reduced version of the network, see Section 3.4), which means that the tokens with the sizes mentioned in the problem instance easily fit within one period.

Examining the graph in Figure 6.10(c), we see that the critical cycle is cycle 'I' with delay `3.934 ms`, i.e., there is a slack of (`5.000 ms - 3.934 ms`) = `1.066 ms`. The other critical cycle candidates have much larger positive slack values. The positive slack means that the real-time constraints are met. However, for the final solution we need to take two remarks into account:

1) the processes of actors v_4 and v_9 share the same processing tile – i.e., the memory controller; therefore, they cannot have both a 100% budget;
2) although we need to keep the slack values positive, as small as possible budgets should be assigned to the processes.

Therefore, we split the processor clock cycle budget of the memory controller between p_{LOAD} – the process of actor v_4 – and p_{STORE} – the process of actor v_9. We assume that the TDMA period of the local scheduler is $T_{\text{T}}(\tau_2)$ = `1.000 ms`, where τ_2 is the processing tile of the memory controller. We give 60% of the budget to p_{LOAD} and 30% of the budget to p_{STORE} (leaving 10% of the budget for the local scheduler overhead). This means that the time slots for these processes are $T_{\text{B}}(p_{\text{LOAD}})$ = `0.600 ms` and $T_{\text{B}}(p_{\text{STORE}})$ = `0.300 ms` respectively.

Now, to calculate the typical delays of actors v_4 and v_9, we can use the equality specified in Section 3.1.2, which we, for convenience, reproduce here using convenient notations:

for $v_k \in \{ v_4, v_9 \} \Rightarrow d(v_k) = D(v_k) + \lceil D(v_k) / T_B(p(v_k)) \rceil \cdot (T_T(\mathcal{T}_2) - T_B(p(v_k)))$
where:

$$D(v_k) = t(v_k) \big/ F_{\text{clock}}(\mathcal{T}_2)$$

which results in the following actor delays:

$d(v_4)$ = `2.000 ms` and $d(v_9)$ = `3.820 ms`

These actor delays are filled in into Figure 6.10(d). From that figure, we see that the resulting delay of cycle 'I' is `4.734` ms, and the remaining slack of that cycle is (`5.000 ms - 4.734` ms) = `0.266 ms`, which is very small, and therefore we do not change the budgets of any actors belonging to that cycle anymore. Also, the delay setting of actor v_9 results in the slack of (`5.000 ms - 3.820` ms) = `1.180 ms` in cycle 'III'.

At this point, the delay of actor v_8 is as annotated in Figure 6.10(c), and the slack of the critical cycle candidate to which that actor belongs is very relaxed and amounts to (`5.000 ms - 0.192` ms) = `4.808 ms`. We make use of this slack to reduce the bandwidth allocated to channel q_{OUT}.

The delay of a transfer actor of a channel is calculated using Equality (3.21), given in Section 3.4.2, which we, again for convenience, reproduce here using convenient notations:

for $v_k \in \{ v_8 \} \Rightarrow d(v_k) = D(v_k) + \lceil D(v_k) / T_B(q(v_k)) \rceil \cdot (T_{\text{TÆ}} - T_B(q(v_k)))$
where:

$D(v_k) = \left\lceil z(v_k) \big/ Z_{\text{min-Æ}} \right\rceil \cdot \left(Z_{\text{min-Æ}} \big/ B_{\text{link-Æ}} \right)$ is the transfer delay the token would experience without TDMA scheduling and without sharing the network links with other channels (see Table 6.1 for an explanation of the notations, and paragraph 'Problem instance' for the values of $z(v_k)$), and $T_B(q(v_k)) = T_{\text{TÆ}} \cdot (n_{\text{slots}}(q(v_k)) \cdot B_{\text{min-Æ}} / B_{\text{link-Æ}})$ is the time interval allocated for the given channel per TDMA period with $n_{\text{slots}}(q(v_k))$ being the number of link slots allocated for the channel (see Table 6.10).

We set $n_{\text{slots}}(q_{\text{OUT}})$ as small as possible, but such that $d(v_k)$ stays within a small interval below `5.000 ms`, to ensure some positive slack. We chose $n_{\text{slots}}(q_{\text{OUT}})$ = `2`. This results in: $d(v_8)$ = `3.550 ms` , as annotated in Figure 6.10(d).

Note that for the other network channels we assumed $n_{\text{slots}}(q_{\text{MV}})$ = `2` and $n_{\text{slots}}(q_{\text{REF}})$ = `44`, which corresponds to 12 resp. 262 bytes, thus being enough to fit the corresponding token sizes $z(v_3)$ = `8 bytes` resp. $z(v_5)$ = `256 bytes` within one TDMA period. As we see Figure 6.10(d), this allows us to keep the transfer delays for actors v_3 and v_5 equal to the minimal possible conservative value of a transfer delay – `0.192` ms, i.e., one network TDMA period.

Hereby we determined all the secondary controllable variables that we needed to determine at this design flow step. With this step, we finished the implementation issues of the main application functionality and start considering the implementation of the run-time performance analysis.

6.2.7 Loop-level Characterization

In this section, we consider the last step of the design flow (see the second step in part II in Section 2.3.4). This step is responsible for the loop-level characterization of the application. This design-flow step is optional, because it takes upon itself part of the tasks of the dynamic-delay analysis algorithmic rule (defined in Section 5.2), whereas that rule can also be completely

executed at run time. Doing the loop-level characterization at design time reduces the run-time overhead. Note that the main difficulty of this design-flow step is that it operates with mathematical values that are only known at run time, therefore working with *symbolic expressions* rather than numbers. Due to that difficulty, we do not have a general algorithm perform this step completely, the method proposed by Amir Hossein Ghamarian *et al* in [22] can automate this step partially, as explained below. Therefore, in our application case study, we perform this step manually.

The purpose of the loop-level characterization is the calculation of the loop-level coefficients: λ_s, σ_s, and $\gamma_{s,t}$. The end result represents the loop-level coefficients in symbolic form as functions of some variables. For practical reasons, those variables should be defined such that it is straightforward to calculate their values at run time, using the application header data. Then these values can be assumed to be given.

The set of input variables of the loop-level characterization step contains the delay quantization levels of all actors in all scenarios: $\{\hat{d}_s(v_k)\}$, where s is the scenario index and k is the actor index. The values of delay quantization levels are recalculated at run time for every execution run of the loop of interest, using the data contained in application headers as described in Section 5.3.6.

Based on those input variables, the dynamic-delay analysis step defines some intermediate variables using symbolic equalities. The final goal is to express the loop-level coefficients using the intermediate and/or input variables. For convenience, we refer to the set of symbolic equalities used in the loop-level characterization as *rules*.

To derive the rules for this case study, let us first simplify the IPC graph, reducing the number of actors and edges. We start from the simplified graph structure shown in Figure 6.10(a) and simplify it even further. First of all, in Figure 6.10(a), we observe a chain of actors v_1, v_2, v_3, v_4, v_5, v_6, where every actor has only one incoming edge and one outgoing edge (except for the last actor). The subsequent actors are joined by an edge with zero initial tokens. This chain can be replaced by one actor whose delay is the sum of the delays of the actors in the chain. The graph structure after the replacement is shown in Figure 6.11. We denote the new actor as v'_1 (see Figure 6.11(a)). The other three actors of the original graph, v_7, v_8, and v_9, are copied into the new graph as actors v'_2, v'_3, and v'_4.

We denote the delay of actor v'_1 in scenario s as η_s (see Figure 6.11(b)). This value can be expressed using the following rule:

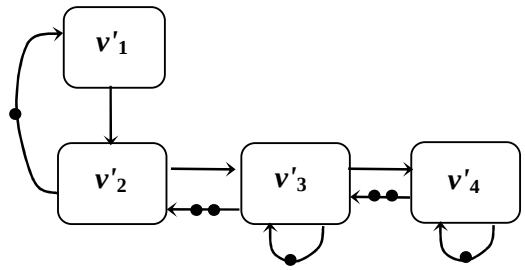
Rule 1: $\eta_s = \sum_{k=1}^{6} \hat{d}_s(v_k)$ ♦

The delay of actor v'_2 in scenario s is denoted as φ_s. Because v'_2 is equivalent to v_7 (see Figure 6.11), we have:

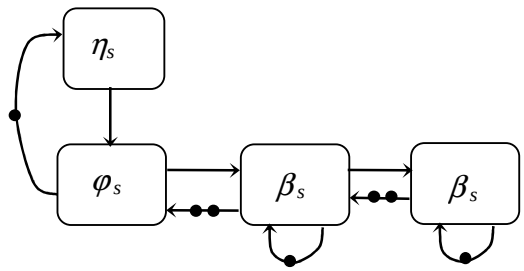
Rule 2: $\varphi_s = \hat{d}_s(v_7) = 0.980$ ms ♦

The constant numeric value for this symbol is based on the observation that that actor has constant delay (it has zero actor-level parameters – see Table 6.4) and on delay calculations for the given target platform (see Figure 6.10(d)).

As for actors v'_3 and v'_4, they are equivalent to actors v_8 and v_9 (see Figure 6.11). They both have constant delays, because v_9 has zero actor parameters (see Table 6.4) and v_8 is a transfer actor, transferring data tokens of constant size. From Figure 6.10(d), we see that their delays are



(a) The IPC graph structure obtained from further simplification of the IPC graph of Figure 6.10(a)



(b) The (intermediate) variables representing the actor delays

Figure 6.11 Simplified IPC graph, for the loop-level characterization

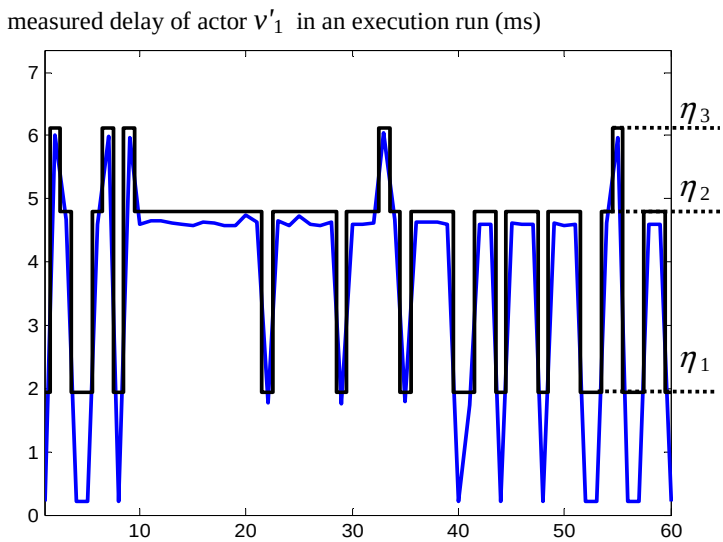


Figure 6.12 The scenario levels of actor v'_1 – η_1, η_2, and η_3

almost equal. Therefore, to keep our symbolic expressions as simple as possible, we use the same symbol, β_s, for both actors, and choose a conservative value for it:

Rule 3: $\beta_s = \max(\hat{d}_s(v_8), \hat{d}_s(v_9)) = 3.820$ ms♦

Although φ_s and β_s are constant values, we still use index 's' in their notations, to stress that in general this step has to work with variables that take different values in different scenarios. To illustrate this fact, the reasoning in the rest of this section does not exploit the fact that those values are constants. Note that our application example is still interesting, because, unlike φ_s and

β_s, the value of η_s is not constant. That value is considerably different in different scenarios, as illustrated in Figure 6.12. Because it also differs in different execution runs, it can only be calculated at run time.

As also indicated in the caption of Figure 6.11(b), values η_s, φ_s, and β_s are intermediate variables. In the rest of this subsection, we derive the loop-level coefficients from these variables.

In this application case study, the easiest loop-level coefficient to derive is λ_s, i.e., the iteration interval. Recall that λ_s is equal to the maximum cycle mean of all cycles in the graph, whereby the scope can be reduced to only simple cycles. There are only five simple cycles in the IPC graph in Figure 6.11(b). Recall also that the cycle mean is the sum of actor delays divided by the number of the initial tokens along the path. Therefore, we have:

$\lambda_s = \max((\eta_s+\varphi_s),\ \ (\varphi_s+\beta_s)/2,\ \ \beta_s,\ \ (\beta_s+\beta_s)/2,\ \ \beta_s)$

This expression can be simplified, and we obtain the following rule:

Rule 4: $\lambda_s = \max((\eta_s+\varphi_s),\ \beta_s)$♦

Note that, in [22], an efficient method is proposed that can generate such a symbolic expression for λ_s automatically for a general class of HSDF graphs, and therefore that work can be used for automation of this design flow step, although this automation so far is only partial, because [22] only considers the maximum cycle mean (i.e., λ_s), but not the lateness and the minimum overlap (i.e., the other loop-level coefficients).

We calculate coefficient σ_s, i.e., the graph lateness, from its definition, which, for convenience, can be rewritten as follows (see also Section 4.4.1):

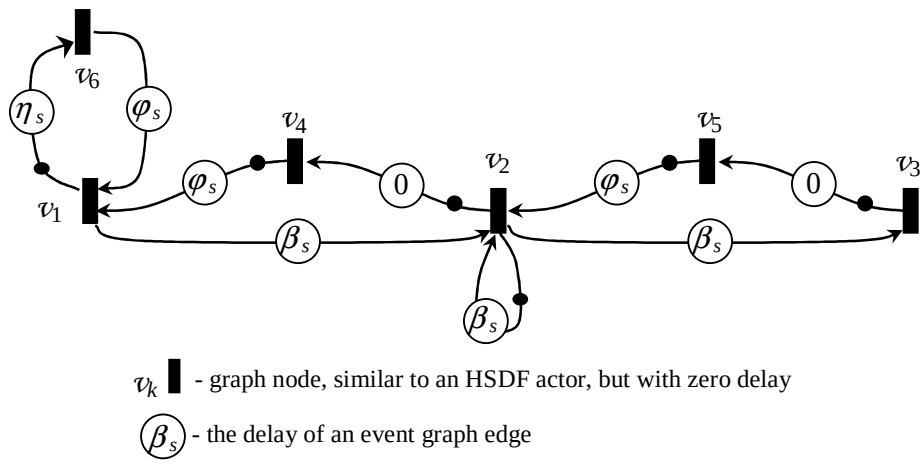
$$\sigma \equiv \max_{k=1..V} \ \max_{n=0..+\infty} x_k(n) - \lambda \cdot n \qquad (6.3)$$

where $x_k(n)$ is the completion time of actor v_k in iteration n, assuming synchronous initial conditions (where all initial tokens are released at time zero). Note that we skipped the scenario index s in this formula for convenience, but not only σ_s but also $x_k(n)$ and λ depend on the scenario. Note also that, again for convenience, we have replaced a finite range for n by an infinite range, but any assignment of static delays to the actors of IPC graph implies a finite range of n where it is enough to apply this formula, because, as follows from Theorem 4.6, the argument of Expression (6.3) is periodic.

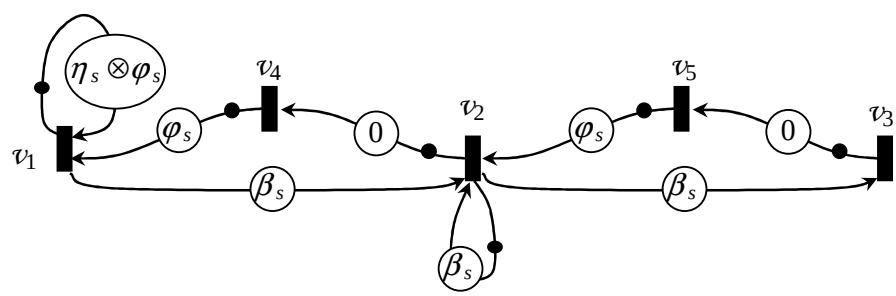
This definition relies on a comparison between symbolic expressions, due to the need to detect the periodic pattern in the argument of 'max' and due to 'max' itself, because $x_k(n)$ and λ can be only expressed using symbolic expressions on unknown actor delays. Therefore, applying this definition directly in practice is a challenging task. Nevertheless, for this case study, we overcome this difficulty. We do that by first using a lower bound on lateness (which is easy to calculate in symbolic form) and then using the definition to prove that it is also an upper bound, thus being equal to the lateness.

Lemma 4.8 implies that a lower bound on σ_s is the largest total delay of a chain of actors joined by edges that do not contain initial tokens. From Figure 6.11(b) we see that the chain of actors v'_1, v'_2, v'_3, v'_4 is such a chain of actors, and therefore a lower bound on σ_s is $\eta_s + \varphi_s + 2\beta_s$.

For convenience reasons, to calculate σ_s, we follow the same mathematical conventions as summarized in Table 4.1 in Section 4.1.1. Recall that we use notations '$\oplus$' and '$\otimes$' for conventional 'max' and 'plus' operations on scalar values, and 'ε' and e for '$-\infty$' and '0'. We



(a) The equivalent event graph $\mathcal{G}_E$ for the IPC graph of Figure 6.11(b)



(b) Graph $\mathcal{G}_E$ after simplification

Figure 6.13 The equivalent event graph

also use the power operation enclosed in round parentheses as an alternative notation for product, e.g., $a^{(b)} = a \cdot b$.

Thus, we may use alternative notations for some 'normal' mathematical operations between scalars and for some scalar values; for the rest, the expressions on scalars are as usual. However, for the expressions involving matrices and vectors, we make a special exception, meaning that usual notations '$\cdot$', '+', and a^b are used to denote max-plus operations. For example, '+' for two matrices means element-wise application of '$\oplus$'. Multiplying two matrices means an inner matrix product where '$\otimes$' and '$\oplus$' are applied for matrix elements instead of 'normal' multiplication and addition. If a constant is multiplied by a matrix/vector or is a matrix/vector element, we do not use '$\otimes$', '$\oplus$', and $a^{(b)}$ to express the constant, instead we use '$\cdot$', '+', and a^b. For example, if constant $\eta_s \otimes \varphi_s \otimes (\beta_s)^{(2)}$ is a member of a matrix/vector or is multiplied by a matrix/vector, this constant is written simply as $\eta_s \varphi_s \beta_s^2$.

To find an upper bound on lateness, we reason along the lines of the static-delay analysis algorithm of Section 4.4.2. Hereby we apply the algorithm in the following steps:

1. Construct the equivalent event graph $\mathcal{G}_E$.
2. Use the equivalent event graph to derive the canonic matrix, denoted **B**.
3. Assume **x(-1)** = **e** (where **e** is a vector filled with values *e*, i.e., with '0'). Calculate max-plus products:

$$\mathbf{x}(0) = \mathbf{B}\,\mathbf{x}(-1), \quad \mathbf{x}(1) = \mathbf{B}\,\mathbf{x}(0), \ \ldots\ , \mathbf{x}(n) = \mathbf{B}\,\mathbf{x}(n-1), \ldots$$

4. Now we deviate from the original algorithm such that we can stop calculating $\mathbf{x}(n)$ for some n. In this case study, we can find an upper bound on $\mathbf{x}(n)$ that can be used to replace the elements of $\mathbf{x}(n)$ in Equality (6.3) and to get an upper bound on the lateness in symbolic form.

Now let us realize this plan.

In step 1, the structure of the equivalent event graph $\mathcal{G}_E$ is closely related to the IPC graph. It is shown in Figure 6.13(a). How an equivalent event graph can be obtained from an HSDF graph in general is explained in Section 4.2.

Note that in this example the equivalent graph has two extra nodes compared to the IPC graph: v_4 and v_5. Note also that for convenience we use a different enumeration of nodes in graph $\mathcal{G}_E$: v_6 corresponds to v'_1, v_1 corresponds to v'_2, v_2 corresponds to v'_3, and v_3 corresponds to v'_4.

Before we proceed to the next step to calculate the canonic matrix, we simplify graph $\mathcal{G}_E$ by merging nodes v_6 and v_1 into one node, v_1 (see Figure 6.13(b)).

Recall from Section 4.2.3 that to calculate matrix **B**, one needs to analyze the largest-length (i.e., longest) special paths between all pairs of nodes in $\mathcal{G}_E$. A special path is a path where only the first edge carries an initial token. In matrix **B**, entry $\{\mathbf{B}\}_{i,j}$ contains the length of the longest special path from v_j to v_i. For example, the longest (and, in fact, the only) special path from node v_1 to node v_2 is $((v_1, v_1), (v_1, v_2))$; its length is $\eta_s \otimes \varphi_s \otimes \beta_s$ and this value is placed as entry $\{\mathbf{B}\}_{2,1}$ in matrix **B**. In a similar way, one can calculate all the other entries in **B** and the result is given below.

$$\mathbf{B} = \begin{bmatrix} \eta_s\varphi_s & \varepsilon & \varepsilon & \varphi_s & \varepsilon \\ \eta_s\varphi_s\beta_s & \beta_s & \varepsilon & \varphi_s\beta_s & \beta_s \\ \eta_s\varphi_s\beta_s^2 & \beta_s^2 & \beta_s & \varphi_s\beta_s^2 & \beta_s^2 \\ \varepsilon & e & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & e & \varepsilon & \varepsilon \end{bmatrix}$$

Now, according to the algorithm, we have:

$$\mathbf{x}(-1) = \begin{bmatrix} e \\ e \\ e \\ e \\ e \end{bmatrix}; \ \mathbf{x}(0) = \mathbf{B}\cdot\mathbf{x}(-1) = \begin{bmatrix} \eta_s\varphi_s & \varepsilon & \varepsilon & \varphi_s & \varepsilon \\ \eta_s\varphi_s\beta_s & \beta_s & \varepsilon & \varphi_s\beta_s & \beta_s \\ \eta_s\varphi_s\beta_s^2 & \beta_s^2 & \beta_s & \varphi_s\beta_s^2 & \beta_s^2 \\ \varepsilon & e & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & e & \varepsilon & \varepsilon \end{bmatrix} \bullet \begin{bmatrix} e \\ e \\ e \\ e \\ e \end{bmatrix} = \begin{bmatrix} \eta_s\varphi_s \\ \eta_s\varphi_s\beta_s \\ \eta_s\varphi_s\beta_s^2 \\ e \\ e \end{bmatrix};$$

$$\mathbf{x}(1) = \eta_s\varphi_s \begin{bmatrix} \eta_s\varphi_s \\ \beta_s(\eta_s\varphi_s + \beta_s) \\ \beta_s^{\,2}(\eta_s\varphi_s + \beta_s) \\ \beta_s \\ \beta_s^{\,2} \end{bmatrix};$$

$$\mathbf{x}(2) = \eta_s\varphi_s \begin{bmatrix} \varphi_s(\eta_s^{\,2}\varphi_s + \beta_s) \\ \beta_s(\eta_s\varphi_s + \beta_s)^2 \\ \beta_s^{\,2}(\eta_s\varphi_s + \beta_s)^2 \\ \beta_s(\eta_s\varphi_s + \beta_s) \\ \beta_s^{\,2}(\eta_s\varphi_s + \beta_s) \end{bmatrix} \le \eta_s\varphi_s \begin{bmatrix} \eta_s\varphi_s(\eta_s\varphi_s + \beta_s) \\ \beta_s(\eta_s\varphi_s + \beta_s)^2 \\ \beta_s^{\,2}(\eta_s\varphi_s + \beta_s)^2 \\ \beta_s(\eta_s\varphi_s + \beta_s) \\ \beta_s^{\,2}(\eta_s\varphi_s + \beta_s) \end{bmatrix} = (\eta_s\varphi_s + \beta_s)\cdot\mathbf{x}(1) = \lambda_s \cdot \mathbf{x}(1)$$

Now, let us multiply the left and right part of this inequality by $\mathbf{B}^{\boldsymbol{j}}$, where $j \ge 0$:

$$\mathbf{x}(2) \le \lambda_s\ \mathbf{x}(1)$$

$$\mathbf{B}\mathbf{x}(2) \le \lambda_s\ \mathbf{B}\cdot\mathbf{x}(1) \Rightarrow \mathbf{x}(3) \le \lambda_s\ \mathbf{x}(2) \Rightarrow \mathbf{x}(3) \le \lambda_s^{\,2}\ \mathbf{x}(1)$$

$$\mathbf{B}^2\mathbf{x}(2) \le \lambda_s\ \mathbf{B}^2\cdot\mathbf{x}(1) \Rightarrow \mathbf{x}(4) \le \lambda_s\ \mathbf{x}(3) \Rightarrow \mathbf{x}(4) \le \lambda_s^{\,3}\ \mathbf{x}(1)$$

...

$$\mathbf{x}(2+j) \le \lambda_s^{\,j+1}\ \mathbf{x}(1) \tag{6.4}$$

The definition of σ_s in Equality (6.3) can be rewritten as follows:

$$\sigma_s = \mathbf{e}^{\mathbf{T}}\cdot\left(\mathbf{x}(0) + \lambda_s^{-1}\cdot\mathbf{x}(1)\ + \lambda_s^{-2}\cdot\mathbf{x}(2)\ + \lambda_s^{-3}\cdot\mathbf{x}(3) + \ldots\right)$$

where $\mathbf{e}^{\mathbf{T}}$ is a row-vector filled with values e.

Using Formula (6.4), we can write:

$$\sigma_s \le \mathbf{e}^{\mathbf{T}}\cdot\left(\mathbf{x}(0) + \lambda_s^{-1}\cdot\mathbf{x}(1)\ + \lambda_s^{-2}\cdot\left(\lambda_s\cdot\mathbf{x}(1)\right)\ + \lambda_s^{-3}\left(\lambda_s^{\,2}\cdot\mathbf{x}(1)\right) + \ldots\right)$$

So, we have:

$$\sigma_s \le \mathbf{e}^{\mathbf{T}}\cdot\left(\mathbf{x}(0) + \lambda_s^{-1}\cdot\mathbf{x}(1)\right)$$

$$\sigma_s \le \begin{bmatrix} e & e & e & e & e \end{bmatrix}\cdot\left(\begin{bmatrix} \eta_s\varphi_s \\ \eta_s\varphi_s\beta_s \\ \eta_s\varphi_s\beta_s^{\,2} \\ e \\ e \end{bmatrix} + \lambda_s^{-1}\cdot\eta_s\varphi_s\begin{bmatrix} \eta_s\varphi_s \\ \beta_s\lambda_s \\ \beta_s^{\,2}\lambda_s \\ \beta_s \\ \beta_s^{\,2} \end{bmatrix}\right)$$

From this, it follows that the upper bound is the maximum element of the two vectors being added within the brackets. We list the elements from top to bottom and from left to right:

$$\begin{aligned}\sigma_s \le\ & (\eta_s\otimes\varphi_s) \oplus (\eta_s\otimes\varphi_s\otimes\beta_s) \oplus \left(\eta_s\otimes\varphi_s\otimes\beta_s^{\,(2)}\right) \oplus e \\ & \oplus \left(\eta_s\otimes\varphi_s\otimes\beta_s^{\,(2)}\right) \oplus \left(\lambda_s^{\,(-1)}\otimes(\eta_s\otimes\varphi_s)^{(2)}\right) \oplus (\eta_s\otimes\varphi_s\otimes\beta_s) \\ & \oplus \left(\eta_s\otimes\varphi_s\otimes\beta_s^{\,(2)}\right) \oplus \left(\lambda_s^{\,(-1)}\otimes\eta_s\otimes\varphi_s\otimes\beta_s\right) \oplus \left(\lambda_s^{\,(-1)}\otimes\eta_s\otimes\varphi_s\otimes\beta_s^{\,(2)}\right)\end{aligned}$$

Comparing different elements to each other, we conclude that the maximum is $\eta_s\otimes\varphi_s\otimes(\beta_s)^{(2)}$. Because this value is both a lower bound and an upper bound on σ_s, we can conclude:

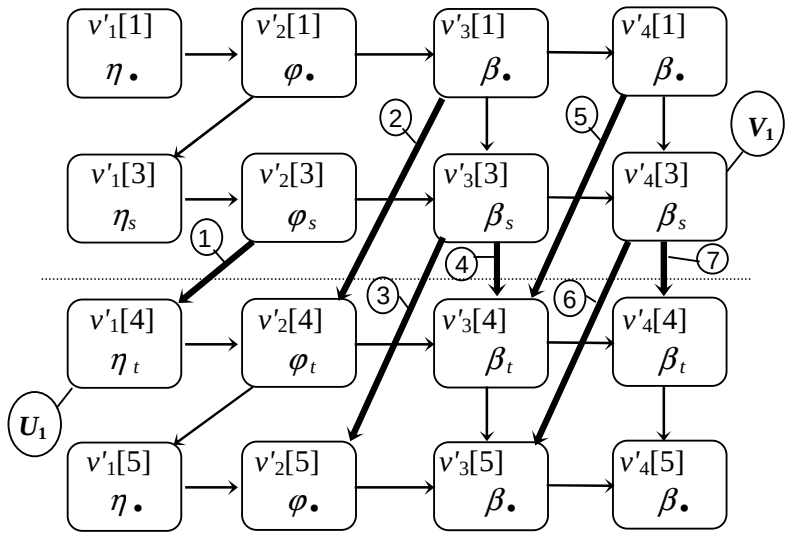


Figure 6.14 The transition graph for the IPC graph of Figure 6.11

Rule 5: $\sigma_s = \eta_s + \varphi_s + 2\beta_s$♦

The only loop-level coefficient left to be characterized is $\chi_{s,t}$, i.e., the minimum overlap between scenario *s* and scenario *t*. To calculate that coefficient, we use the IPC graph analysis algorithm described in Section 5.2.3. For that, we unfold the IPC graph in Figure 6.11 with unfolding factor four (which is twice the maximum number of initial tokens per edge).

As a result, we obtain the transition graph, shown in Figure 6.14. The actor delays drawn in the first and in the last row have index '•', e.g. $\eta_\bullet$, which refers to the minimum value in all scenarios, e.g. $\eta_\bullet = \min \eta_s$. Recall that the transition graph is partitioned into two subgraphs by the transition line.

Recall that the edges that cross the transition line are called the *edges of interest*. They are shown by bold arcs in the figure and numbered with index $r = 1..7$. Above the transition line there are special nodes, which are involved in the calculation of $\chi_{s,t}$. Those are so-called *sinks of interest* V_l, which are defined as the nodes that have only edges of interest as outgoing edges. In our example, there is only one such node, $v'_4[2]$, which is therefore labeled as V_1. Similarly, there are special nodes also below the transition line. Those are the so-called *sources of interest* U_i, defined as the nodes that have only edges of interest as incoming edges. In our example, such a node is node $v'_1[3]$, labeled as U_1.

Before we can calculate $\chi_{s,t}$, for each edge of interest with index *r* we calculate:

- Δe_r, as-late-as-possible production time on edge *r* (*alap production time*). It is equal to the largest sum of the delays of the consumer actors on a path from the producer of edge *r* to a sink of interest V_l. For example, for the edge indexed as edge '2', we take path ($v'_3[1]$, $v'_3[2]$, $v'_4[2]$). Therefore, we have $\Delta e_2 = d(v'_3[2])) + d(v'_4[2])) = 2\beta_s$. An alternative path is ($v'_3[1]$, $v'_4[1]$, $v'_4[2]$), but it has a smaller sum of consumer delays.

- Δb_r as-soon-as-possible consumption time on edge *r* (*asap consumption time*). It is equal to the largest sum of the delays of the producer actors on a path from a sink of interest U_i to the producer of edge *r*. For example, for the edge indexed as edge '6', two paths are candidates: the first one is ($v'_1[3]$, $v'_2[3]$, $v'_1[4]$, $v'_2[4]$, $v'_3[4]$) and the second one is ($v'_1[3]$, $v'_2[3]$, $v'_3[3]$,

Table 6.11 *Alap* and *asap* values for the transition graph in Figure 6.14

r	**From**	**To**	Δe_r	Δb_r
1	$v'_2[2]$	$v'_1[3]$	$2\beta_s$	0
2	$v'_3[1]$	$v'_2[3]$	$2\beta_s$	η_t
3	$v'_3[2]$	$v'_2[4]$	β_s	$\eta_t + \varphi_t + \eta_\bullet$
4	$v'_3[2]$	$v'_3[3]$	β_s	$\eta_t + \varphi_t$
5	$v'_4[1]$	$v'_3[3]$	β_s	$\eta_t + \varphi_t$
6	$v'_4[2]$	$v'_3[4]$	0	$\eta_t + \varphi_t + \max(\eta_\bullet + \varphi_\bullet, \beta_t)$
7	$v'_4[1]$	$v'_4[3]$	0	$\eta_t + \varphi_t + \beta_t$

$v'_3[4]$). Taking the maximum sum of the producer delays, we obtain:

$$\Delta b_6 = d(v'_1[3])) + d(v'_2[3])) + \max\,(d(v'_1[4])) + d(v'_2[4])), d(v'_3[3])) =$$
$$= \eta_t + \varphi_t + \max(\eta_\bullet + \varphi_\bullet, \beta_t)$$

The values of Δe_r and Δb_r for the edges of interest in this case study are given in the table below Table 6.11.

Recall that $\chi_{s,t}$ is equal to the minimum sum of Δe_r and Δb_r in a row of the table. To find that value, let us first exclude the rows that have sums larger than or equal to other rows. Row 2 has a larger sum than row 1; row 3 has a larger sum than row 4; row 5 has a sum equal to row 4; row 6 has a sum larger or equal to row 7.

This leaves us with only three candidate table rows: row 1, row 4 and row 7. Finding the minimum value of those three rows yields the following expression:

Rule 6: $\chi_{s,t} = \min\,(\,2\beta_s\,, \eta_t + \varphi_t + \min(\beta_s, \beta_t)\,)$♦

We see that Rules 4-6 express the loop-level coefficients in terms of the intermediate variables, which, in turn, can be calculated from the actor delays using Rules 1-3. Thus, the loop-level characterization is completed.

6.3 Run-time Performance Analysis Results for the MPEG-4 Decoder

6.3.1 The Goals of the Experiments

Whereas in the previous section we considered the design steps performed at design time, in this section, we evaluate the quality adaptation, performed at run time. The adaptation is the task of the QoS manager, which we introduced in Sections 6.1.2 and 6.1.3. In this section, we apply our performance-analysis approach to realize an important part of the QoS manager: the VOP Decoding-Time Estimator. Recall that, using Figure 6.2, we already explained the task of the estimator and showed that it can be realized with the dynamic-delay analysis algorithm introduced in Chapter 5. In this section, we evaluate the Decoding-Time Estimator experimentally. Note that the estimator uses the results of the previous section, such as the actor-

level coefficients, the definition of scenarios, the budget assignment and the loop-level characterization rules.

The purpose of our experiments with the VOP Decoding-Time Estimator is to evaluate the run-time component of the performance-analysis approach proposed in this thesis. The central issue is the accuracy of the execution time estimation, which is very important to reach good results in practice, as we showed in Section 6.1.4.

In particular, the goals of the experiments are:

- to measure the accuracy and to check conservativity;
- to explore the overhead-accuracy trade-off;
- to measure the impact of analysis error on visual quality;
- to compare our performance analysis to the worst-case analysis.

Thus, as the evaluation reference, we use the *worst-case throughput* approach, which is the traditional approach to handle the applications with dynamic data-dependent delays. This approach avoids the analytical difficulties in handling the dynamic delay variations by replacing the dynamic delays with static maximum values. With static delays, it is relatively easy to analyze the throughput of the dataflow graph and thus also to calculate the execution time. As it can be concluded from our earlier related work overviews in Sections 1.5 and 2.2.6, the worst-case throughput approach is the only alternative to our method known to us provided that conservativity needs to be ensured and arbitrarily long execution runs need to be supported.

We model the worst-case throughput approach by a special case of our approach where the multi-scenario delay (MSD) mode has only one scenario. Thus, our approach takes advantage of more scenarios (three in this case study) and of our 'dynamic-delay analysis' algorithm to handle multiple scenarios.

6.3.2 Estimator Implementation

For the experiments with the estimator, we used the two sample input video streams, 'singer' and 'dancer', which we introduced in Section 6.2.4.

First of all, for both streams, we obtained the data to be encoded in the VOP headers as input for the estimator. As indicated in Figure 6.2, three kinds of data are necessary, the characteristic values of actor-level parameters, $\hat{\xi}_{\omega,s}$, the values of loop-level parameters, J_s and $K_{s,t}$, and the scenario index of the first macroblock, $s(1)$. All these data are obtained based on the scenario definition in Section 6.2.5.

Given this data, the run-time task of the estimator for the given VOP is as follows. From $\hat{\xi}_{\omega,s}$, the estimator calculates the delay quantization levels of all actors, $\hat{d}_s(v_k)$. Using Rules 1-6 from Section 6.2.7, the estimator obtains the loop-level coefficients of the given VOP: λ_s, σ_s, and $\gamma_{s,t}$.

Afterwards, the estimator calculates the loop-level parameters L_s, using Equalities (5.7). At this point, the estimator has all the loop-level coefficients and parameters values. After that, it calculates the predicted execution time as a linear combination of loop-level coefficients and parameters, as given in Equality (5.6), which we reproduce here for convenience:

$$\hat{\Delta}_N = \sum_s (\lambda_s \cdot J_s + (\sigma_s - \lambda_s) \cdot L_s) - \sum_{s,t,s\neq t} \gamma_{s,t} \cdot K_{s,t} \qquad (6.5)$$

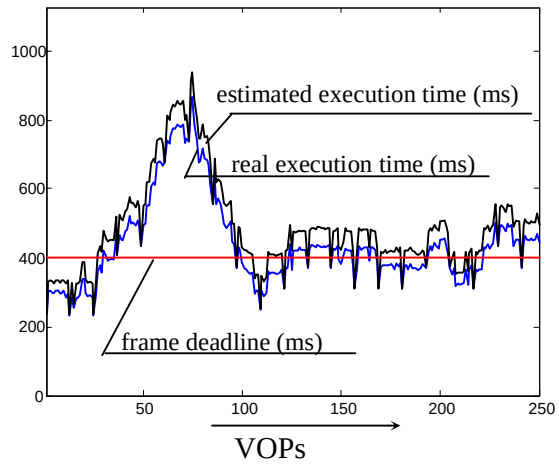


Figure 6.15 VOP decoding time estimation results

Table 6.12 Estimator evaluation results

	stream 'singer'				**stream 'dancer'**			
scenario count	**ideal**	**3**	**2**	**1**	**ideal**	**3**	**2**	**1**
avg error, %	0	11	25	56	0	10	18	60
max error, %	0	17	36	77	0	14	24	89
quality, %	77	64	46	28	77	70	65	33
overhead (bytes)	-	20	12	5	-	19	13	6
overhead, %	-	5	3	1.3	-	1	0.7	0.3

To evaluate the estimator accuracy, in our experiments, we compare the results of the estimation from the execution times measured on our multiprocessor simulator.

To evaluate the impact of estimation error on the visual quality of the video decoder, we feed the estimation results to the VOP Skipping Controller, being used in our QoS manager and described in Section 6.1.2.

6.3.3 Experimental Results

The results are summarized in Table 6.12. The columns show per stream the results for ideal estimation and the estimation using a certain number of scenarios. The first two lines show the average and maximum error with respect to simulations. For our default setting (three scenarios), our method yields 11% and 10% average error for the two sample streams. Figure 6.15 shows execution-time curves for stream 'singer'. Our estimation turned out to be strictly conservative (although this is theoretically not guaranteed), which is in line with our objectives. The measured estimation error can be mostly explained by overestimation due to multi-scenario actor delay mode (Figure 6.12).

The more scenarios are used in the estimation, the larger the overhead, because the more actor-level and loop-level parameter values need to be encoded in the VOP header. Note that we need to encode 15 characteristic actor-level parameters per scenario (and two primary parameters are implied by the scenario itself) and one loop level parameter per scenario and scenario transition. We measure the overhead needed to encode the difference between the parameter values in the current and the previous VOP using Shannon's entropy metric. The results are shown in the last two rows of Table 6.12. The relative overhead differs considerably between the streams because they have different average VOP size, 400 bytes and 2000 bytes. The absolute overhead is almost independent of the VOP size, because it determined mainly by the probability distributions of variations of parameter values from one VOP to another. The results show that the relative overhead is limited if VOP sizes are not too small. Note that overhead can be reduced further (compared to the application of entropy coding), e.g., by applying quantization to the least sensitive parameters.

We see that using less scenarios reduces the overhead but leads to poorer accuracy. As we see in Table 6.12, reducing the number of scenarios from 3 to 2 (by merging the two highest delay levels in Figure 6.12) results roughly in twice the error. Having only 1 scenario leads to an even larger error increase (by a factor of 5 to 6). The one-scenario approach models the worst-case throughput approach, which shows the big advantage of our scenario-based approach over that technique.

Note that 11% accuracy is a good result compared to the related work which is closest to our work in this area – [6] – where similar accuracy results were achieved, using, however, much less data encoded in the header (only three parameter values versus around 50 in our case). Nevertheless, the advantage of our technique is that, in return to a larger overhead, it gives conservative results and that it supports multiple processors.

Due to the fact that we can expect a similar IPC graph structure from any video decoding algorithm, we expect that in practice the same accuracy can be achieved in this application domain with our technique. It is definitely an interesting future work subject to evaluate our technique for other application domains and for dataflow graphs in general.

Differences in estimation accuracy have a big impact on the visual quality, because more VOPs are skipped if the overestimation grows. For stream 'singer', we set the VOP deadline to 400 ms, which produces significant processor overload. For stream 'dancer', we set the deadline such that similar overload was achieved. This choice is intentional, showing a situation where the QoS manager has to actively control the quality. For our VOP-skipping QoS manager, Table 6.12 shows the quality results, measured in the percentage of VOPs presented to the user. From the table, for stream 'singer', we observe a significant quality drop, from 64% for our approach to only 28% for the worst-case approach. A similar observation holds for stream 'dancer'.

Note that the video frame-skipping is not typical for advanced video decoders; neither is the frame rate of 2.5 frames per second (i.e., a 400 ms deadline). We made those assumptions due to practical limitations in the experimental setup. The results and conclusions carry over to realistic settings with faster processors and more advanced quality control methods.

6.3.4 Notes on the Processing Time Overhead

In our experiments, we did not implement the estimator on the target platform, because we did not fully elaborate the encoding/decoding of the parameter values (recall that we only estimated the code sizes of the parameters using Shannon's entropy metric). Instead, we emulated the main part of the estimator algorithm at higher abstraction level. Thus, we could not measure the processing time overhead of this algorithm (in terms of processor cycles per VOP); and we can reason about this overhead only based on the algorithmic complexity of the calculations.

In the estimator implementation of this case study, the processing time overhead can be managed efficiently, because the amount of calculations does not depend much on the input data and the calculations mainly comprise the decoding of parameter values from the header and the calculation of a few linear formulas: the formulas of the actor delays, six rules for loop-level characterization, Equalities (5.7) and Equality (6.5). Intuitively, those calculations are negligible compared to decoding a VOP.

This is due to the fact that we could reduce the amount of run-time calculations by doing the loop-level characterization at design time. However, for the general case, we do not have a method for design-time loop-level characterization; therefore, when implementing an Estimator for other applications, it might be necessary to do the loop-level characterization at run time. Let us discuss this option briefly.

Among all the variables contributing to the complexity of loop-level characterization, the only concern is H – i.e., the number of iterations of the algorithmic rule to calculate lateness of an IPC graph – see Section 4.4.3. This variable is the only variable that is in the worst-case exponential in the representation of the IPC graph. We implemented this algorithm and saw that, in our case study, H could change by a factor of 2 due to a 0.00001% change of an actor delay, which could be explained by the fact that multiple bits are required to represent that change accurately, H being worst-case exponential in that number. Nevertheless, when we represent the actor delays with a reasonable accuracy of 0.1%, then H stays below 10 in our case study. To improve the robustness of our method, finding tight approximations of the graph lateness with polynomial algorithmic cost is an important future work topic.

6.4 Notes and Summary

In this chapter, we have evaluated the method proposed in this thesis for the performance analysis in the context of run-time adaptation of application quality to the variable computation workload. Hereby we have taken care to select proper ingredients for a demonstration of our approach.

Firstly, because our method is oriented to multimedia streaming applications with scalable audio/video quality, as a case study we selected a modern streaming application – the MPEG-4 arbitrary-shape decoder – enhanced with a practical quality adaptation manager, as described in Section 6.1.2. Secondly, because a major novel element being introduced by our performance analysis method is the support of network-on-chip multiprocessors, we mapped the application to two processors of such a platform, as described in Section 6.2.2. Thirdly, because modern dynamic streaming applications require support of multiprocessor resource budgeting and concurrent execution, we make use of these features in this case study and model them using our

elaborate timing model – the implementation-enhanced HSDF (in its final form – the IPC graph). The elements of this model for this case study are described in Sections 6.2.3 (the graph structure) and 6.2.4 (where the elements that model the processor and communication scheduling). Finally, because the complexity of performance analysis is a very important practical issue, in Sections 6.2.4, 6.2.5 and 6.2.7, we demonstrate the mathematical framework that we employ in this thesis to reduce the level of detail in the timing model while offering sufficient accuracy. In those sections we start from a IPC graph mode; endowed with detailed actor execution traces and end by a limited set of algebraic expressions that can be quickly evaluated at run-time to quickly and accurately predict the application performance with limited overhead.

In this evaluation, we have shown that our performance method yields accurate and conservative resource utilization predictions needed for run-time resource and quality management in low-power embedded multimedia systems. At the same time, we showed that the worst-case throughput analysis failed to yield good results. The latter is the only previous performance analysis technique that could be applied at run time for our implementation-enhanced HSDF timing model. In addition, our evaluation has shown acceptable overhead in terms of the processing time and the data size. We published this evaluation in [76].

The conclusions and future work directions are summarized in the next chapter.

7

7 Conclusions and Future Work

7.1 Thesis Summary

In Chapter 1, we have set up the goals of this thesis, and in this chapter we summarize to which extent and how these goals have been met.

This thesis focuses on computer systems that are important for consumer electronics, namely, on embedded systems for multimedia applications. We make a motivated choice to work on a very important class of such systems – the multiprocessor networks-on-chip.

Our main application area has been defined as dynamic *streaming* applications, also referred to as digital signal processing applications and coding/decoding audio/video applications. Within that context, our major goal is run-time performance analysis that aids the dynamic adaptation of the computer system to run-time changes in the processing workload. The analysis has to be accurate and conservative. To support the streaming applications in the best way, the analysis has to natively support arbitrarily long execution runs of applications on multiprocessors, which means that it has to reason in terms of application *throughput* rather than in terms of *response times*, like it is done primarily for control applications.

Whereas response-time methods focus on calculation of the worst-case execution paths, throughput analysis techniques focus on a certain state of equilibrium – a steady state – reached by the computational model that represents the analyzed system. However, the latter is not trivial

to do in the context of dynamic data-dependent system workload, because such a workload, in general, makes it hardly possible to define a single steady state of the system.

Therefore, we generalized the steady-state approach for analyzing throughput of streaming applications to multiple steady states – called *scenarios*. Application scenarios are a well-known concept for dealing with dynamic system workload [27], [99], but our work for the first time associates an application scenario with a distinct state of equilibrium of a model of computation and includes transient behavior. (A very recent publication of Marc Geilen – to be published soon – also does that [21]) This allows us to unify the throughput-oriented reasoning needed for streaming applications and the scenario-based approach needed for dynamic applications.

In the context of dynamically changing processing workload, the scenarios have unpredictable characteristics and unpredictable transitions between them. To deal with the uncertainty due to unpredictable workload, our performance analysis approach follows the common method for all scenario-based methods, namely, it exploits run-time characteristics of the scenarios that are known *a priori*.

To capture the concurrent execution of the application in a multiprocessor, a certain model of computation has to be chosen. We have chosen the HSDF (Homogeneous Synchronous Data Flow) model of computation and motivated that choice in Chapter 2. There, we establish a relationship between that abstract model of computation and a real application implemented on a multiprocessor. We give a practical context for our HSDF performance analysis by describing an implementation trajectory from the specification through a design flow to the implementation decisions taken at run time. The performance analysis acts as a tool for continual evaluation of decisions taken at each implementation step and for guiding these decisions towards optimal solutions. At design time, the performance analysis can make use of existing theoretical results on the steady-state throughput analysis of HSDF graphs. However, to guide the run-time implementation decisions, the throughput analysis has to support dynamic data-dependent execution delays in HSDF graphs. When the execution delays are data-dependent, reasoning about the HSDF throughput appears to be a practically intractable problem in general. In the end of Chapter 2, we briefly explain how the methodology proposed in this thesis can avoid that difficulty by exploiting the a-priori run-time characteristics of multiple scenarios.

In Chapter 3, we describe the enhancements we made to the HSDF model so that it can play a role of a performance-analysis model for data-dependent streaming applications running on modern multiprocessor systems-on-chip. The main novelty of this chapter is the treatment of the communication channels in the on-chip multiprocessor interconnection network, also known as *network-on-chip*. We build a framework for modeling the channels with guaranteed throughput and bounded FIFO buffers as subgraphs of the HSDF graphs. Hereby, we support *complex channels* that transfer the data tokens of different original simple channels in a fixed order. We presented this contribution in [75]. The only closely related work on this subject is the work of Arno Moonen *et al*. In [60], [63], they introduce network channel models that are similar to ours, but have important differences. On one hand they do not support complex channels. On the other hand, they propose more powerful and less pessimistic models for the TDMA scheduling of network packets. Note that their models reflect the events at the hardware-specific level of granularity, whereas our models work at the application-specific data granularity level. Therefore, if the application uses data samples consisting of a large number of network data words, then the performance analysis complexity for our models is much smaller. This is particularly favorable for the performance analysis that is done at run time.

Our framework for modeling the network-on-chip channels can be exploited in a design flow that maps streaming applications to a multiprocessor network-on-chip. This flow is responsible for a gradual transformation of our implementation-enhanced HSDF model from the specification to the final form. In Chapter 3, we sketch a hypothetical preferred mapping flow and the corresponding HSDF model transformations using a JPEG decoder application as a case study. The contribution of this thesis to multiprocessor mapping flows is focused on networks-on-chip communication aspects, whereas, for bus-based multiprocessors, mapping flows and implementation-enhanced HSDF models have been known already for some time, see [5], [83]. In particular, our contribution is that we enable bottleneck analysis of buffer capacity minimization for complex network-on-chip channels under application throughput constraints. The bottleneck analysis can be used in heuristic iterative improvement algorithms for this problem. Although we do not formulate a general algorithm, we demonstrate this idea using the JPEG decoding case study in Chapter 3 and in [75]. In the final section of Chapter 3, we discuss different related publications on buffer capacity minimization. From that discussion we can conclude that no FIFO buffer capacity minimization method known to us can directly address the complex-channel capacity minimization problem without making too restrictive assumptions, and, therefore, this problem remains open.

Of special importance for our methodology is the initial design-flow phase, which precedes the multiprocessor mapping. We refer to that phase as the *application preparation*. It is a generalization of conventional profiling and worst-case execution time analysis. Instead of assuming constant worst-case/average-case processor clock cycle counts for the HSDF graph actors, the application preparation characterizes the actor processor-cycle counts as functions on input data parameters, called actor-level parameters. On one hand, the parameter values are assumed to be constant when making the design-time decisions. On the other hand, at run time, we exploit the *a priori* information on the parameter value variations for accurate performance estimations at run time, as it becomes apparent from the following chapters of the thesis. In Chapter 3, we propose to use the confidence intervals of the linear regression method to make the coefficients of the actor-level parameters conservative. This is necessary to give performance guarantees, which is a major goal of this thesis.

Chapter 4 revisits the steady-state throughput analysis of HSDF graphs with static delays and introduces a new metric for the graphs, called lateness. This metric characterizes the transient phase of the timing behavior when the HSDF graph execution is underway to a steady state. The lateness is important for extension of the single-steady-state throughput analysis to multiple steady states, because it helps to evaluate the impact of transitions between different steady states. Unfortunately, the only known exact algorithm to calculate the lateness is exponential. Nevertheless, in practice, this has never resulted in long runtimes for us and there are ways to give up some accuracy while reducing the probability of large calculation overhead.

In Chapter 5, we realize the idea of using multiple steady states that are reached at different intervals of application execution run, in order to analyze the execution time and throughput in the case of dynamic data-dependent execution delays. We do that through quantization of actor execution delays. Different quantization levels correspond one-to-one to different steady states of the HSDF graph.

We define a *scenario* as a sub-space in the space of possible values of the vector of actor-level parameters of the given application. The scenario should be distinguished by the most

influential actor-level parameters, i.e., those that have a large impact on multiple actors of the HSDF graph.

An important contribution of Chapter 5 is a graph-path analysis algorithm to calculate the timing overlap between different scenarios. This timing overlap appears due to the inherent parallelism of HSDF graph execution, whereby some actors may start executing in a new scenario earlier than the other actors. We presented the timing overlap technique in [79], [78], [76]. As discussed in Section 1.5 and 5.4, the only relevant related work on this subject – although focusing on a different topic – is the work of Zhe Ma *et al* [55], [56]. Their work on energy-aware scheduling implies a performance analysis method that has much in common to the actor execution delay quantization as proposed in this thesis. However, the main focus of that work lies on a different subject – i.e., efficient multiprocessor scheduling, whereby they do not yet explicitly exploit the steady-state analysis. As we argued in Section 5.4, in order to apply their method for the same performance analysis problem as considered in this thesis, their approach would need to be essentially modified.

Our results on scenario-aware performance analysis give us the possibility to revisit the beginning of the design flow – the application preparation. Chapter 3 only provides characterization of performance at the level of actors, using functions on actor-level parameters. The results of Chapter 5 allow us to characterize the performance of the HSDF graph as a whole. The estimated duration of an execution run of the HSDF graph is a linear function of so-called loop-level parameters, which count the occurrence of different scenarios and scenario transitions in the given execution run.

Therefore, in the beginning of the design flow in our methodology, having defined the actor-level parameters, the application designer also has to identify the scenarios, which automatically leads to the definition of the loop-level parameters. The coefficients for the loop-level parameters are obtained from the graph analysis of the final HSDF graph after it has undergone all the transformations made by the design flow. Because the graph analysis needs to know the actor delays per scenario, it uses certain values of actor-level parameters in every scenario, referred to as characteristic values. Note that for the identification of a good scenario set of the given application, we do not propose any general approach, although we argue that the technique proposed by S. V. Gheorghita *et al* [28], [24] can be applied.

The values of loop-level parameters and the characteristic values of actor-level parameters per scenario form, in fact, the a priori information exploited by our performance-analysis approach at run time. From these data, our run-time algorithm for the estimation of the execution-time (and throughput) can give estimations that are both conservative (with high probability) and that can be made as accurate as necessary by providing a large enough number of scenarios (at the cost of a larger overhead). This finalizes the development of the methodology of this thesis.

Chapter 6 demonstrates our methodology using an application case study, an MPEG-4 arbitrary-shape video decoder with a simple quality-of-service manager. For this application, we go through the design flow, especially focusing on the aspects important for the performance analysis. At the end of the flow, we have all the necessary data for run-time prediction of the decoding times of video frames. We simulate the run-time predictions of decoding times and the visual quality adaptation algorithm guided by those predictions. The results show that our method has a reasonable overhead and that it yields conservative predictions with enough accuracy to achieve a visual quality that is close to the best achievable quality. To the best of our

knowledge, no other existing performance-analysis methods can give conservative execution time predictions for multiprocessor systems that can match this result.

7.2 Limitations and Future Work

In this section, we summarize the main limitations of the work presented in this thesis and identify the major topics for future work.

First of all, the chosen model of computation, i.e., HSDF, has only a limited support for the conditional execution of actors, see Section 2.2.1. Also, HSDF supports only a single-level nested loop at the graph level and no external inputs and outputs. Extending our performance analysis for a better support of conditional execution, for more complex loops, and for external inputs and outputs (with a fixed periodic pattern of data arrival and data required times) is a topic of future work. However, before extending the analysis to support more general models of computation, in future work, one also needs to handle a known issue for HSDF graphs, namely a possible lack of accuracy for graphs that have multiple actors that can be identified as so-called sinks of interest (see Section 5.2.4 for more details). In this respect, it would also be useful to try our performance-analysis approach on more HSDF benchmarks, first of all on random HSDF graphs with random sets of scenarios.

The functional usability of our analysis approach needs to be explored in more application case studies of run-time adaptation of quality/energy consumption and resource budgets. For the distributed multiprocessor environment like the one we assumed in this thesis, the latter two cases may involve the reconfiguration of hardware resources and task migration between different processing tiles, which did not get attention in this thesis. Another important issue that is taken in this thesis for granted is the requirement to produce conservative results, i.e., to provide guaranteed performance, whereas we still, for efficiency reasons, allow our results to be non-conservative. We have not considered yet the question on how ‘bad’ it would be for real-life multimedia streaming applications for consumer electronics to use a performance analysis that often gives too optimistic results and whether one could find a good ‘middle point’ between the resource use efficiency and the reliability of performance estimations.

The overhead of the performance analysis is a very important concern, especially if the analysis is performed at run time. In this respect, a weak point of our approach is that we use an exponential-complexity algorithm to calculate the lateness of an HSDF graph (see Section 4.4.2). This problem does not necessarily manifest itself in practice and we have discussed some ways to work around this problem if necessary (see Sections 4.4.3 and 6.3.4). In future work, developing a lateness calculation algorithm (possibly an approximation algorithm) with a polynomial complexity would be a valuable extension of this work.

Our performance analysis method is based on actor delay quantization levels, determined by the scenarios. The only automatic technique for scenario identification that we are aware of is the technique of S. V. Gheorghita [28], [24]. Although we argued that it can be adapted for our implementation trajectory, it was originally developed for sequential implementations, and thus can provide non-optimal solutions for multiprocessors. In our case study we have seen an example where the choice of a better set of scenarios is influenced by the structure of the dataflow graph. Therefore, an interesting future work topic is development of scenario identification techniques taking into account the parallelism of different actor executions.

Another help for the application designer in this context would be automation of the generation of appropriate coding schemes (such as Huffman trees) for encoding the complexity parameters in the application data headers to be used in run-time adaptation. This task is very similar to the generation of encoding schemes for new video standards, such as MPEG-4. For example, also in this case there is a certain trade-off between the data-size savings due to lossy coding and the quality of the output results. However, the definition of the encoding schemes in video standards is a more complex task, which probably will remain to be manual work, whereas the encoding of the complexity parameters is probably a more routine task that can be automated.

Recall that, apart from the run-time performance prediction, another goal of our performance analysis approach is to aid the design flow for networks-on-chip, especially the communication mapping phase. Our main contribution in this area is the bottleneck analysis for minimizing the capacities of complex buffers. Hereby, so far we have only considered only one application case study – the JPEG decoder. Therefore, also this contribution needs more evaluation on application benchmarks and random graph examples. Moreover, in our preferred mapping flow, other topics remain open and deserve future investigation. So far we are not aware of any algorithms to merge simple channels into complex channels, especially when we speak of the network (i.e., 'real') channels, and not local memory buffers. Another open question lies in the more global context of the mapping flow. In the JPEG decoding case study, we have observed that assuming average actor processing times during mapping may often lead to inaccurate 'optimistic' results, i.e., assuming that the throughput constraint is met, whereas in reality it is not met even for the sample input stream used to measure the average processing times in the graph. An open question remains on how to make the mapping flow better aware of the dynamic variations of the actor processing times, so that it produces mapping solutions that are more reliable in terms of throughput constraint satisfaction, while still not making too pessimistic assumptions on the actor processing times. Using scenarios in the context of the mapping flow might be a promising approach in that direction, because they are already used in certain mapping flows (see e.g. [55]) to ensure better energy efficiency of the multiprocessor mapping solutions. Hereby, unlike existing approaches, one needs to also take the scenario transitions into account using timing overlap calculation techniques (see Section 5.2.3).

Last but not the least, to improve the usefulness of our performance analysis approach for the state-of-the art embedded systems, we need to extend it with the support of shared memory hierarchies and with the of on-the-fly reconfigurations of the running applications.

7.3 Main Conclusions

In the context of embedded multimedia multiprocessor systems with dynamic streaming applications, our work brings some contributions into two major research areas: run-time adaptation and the design flow.

For run-time adaptation, we have proposed a methodology for run-time prediction of the processing throughput, or in other words, for run-time prediction of the execution times necessary for processing a certain amount of input data. The execution-time/throughput prediction techniques help the run-time adaptation methods to act proactively when trying to maintain the necessary throughput. When predicting too long execution times, the adaptation methods can proactively adjust either the processor clock frequency/voltage, or the application quality-of-service level, or adapt the resource usage in some other way. We propose a method,

the only one known to us so far, that can do execution-time/throughput estimations that are both accurate and conservative.

Giving guarantees on the performance is a key property of our contribution to the run-time adaptation area. This concept is important for ensuring smooth and reliable application execution runs without ever overloading the computation or the communication resources. In previous work, the only technique that could produce conservative throughput estimation was worst-case throughput analysis that assumes the worst-case execution delays of every application subroutine. The extension to execution-delay quantization and multiple scenarios, as explained earlier in this chapter, helps us to achieve good estimation accuracy in addition to being conservative. Our application case study – the MPEG-4 arbitrary-shape video decoder – demonstrates good accuracy and conservative predictions at a reasonable cost. Because this application is one of the most complex applications in the domain of video applications, we believe that these results are also promising for many other multimedia applications.

Our secondary contributions are in the area of the design flow for mapping streaming applications to multiprocessors based on packet-switched networks-on-chip. To map the communication of an application to the network communication channels, one needs to understand the impact of the mapping decisions on the throughput of the application. We were the first to propose models that capture the performance impact of the mapping decisions involving the capacity of the input and output FIFO, the bandwidth of the channels, and the merging of multiple application communication channels into larger complex network-on-chip connections. As we demonstrated using the JPEG decoder case study, the modeling techniques help to take correct mapping decisions throughout the whole mapping flow, and especially the communication mapping decisions.

References

[1] B. Ackland, A. Anesko, D. Brinthaupt, S. J. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C. J. Nicol, J. H. O'Neill, J. Othmer, E. Säckinger, K. J. Singh, J. Sweet, C. J. Terman, and J. Williams. A Single-Chip, 1.6-Billion, 16-b MAC/s Multiprocessor DSP. In *IEEE Journal of Solid-State Circuits*, vol. 35, no. 3, pages 412-424, IEEE, March 2000.

[2] M. Adé, R. Lauwereins, and J. A. Peperstraete. Data Memory Minimisation for Synchronous Data Flow Graphs Emulated on DSP-FPGA Targets. In *Proc. DAC-97, the 34th Conference on Design Automation*, pp. 64-49, ACM, 1997.

[3] www.arm.com

[4] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity*. New York: Wiley, 1992.

[5] N. Bambha, V. Kianzad, M. Khandelia, and S. S. Bhattacharrya. Intermediate Representations for Design Automation of Multiprocessor DSP Systems. In *Design Automation for Embedded Systems*, vol. 7, pages 307-323, Kluwer Academic Publishers, 2002.

[6] A. C. Bavier, A. B. Montz, and L. L. Peterson. Predicting MPEG Execution Times. In *Proc. of ACM SIGMETRICS 98, Joint international Conference on Measurement and Modeling of Computer Systems*, pages 131-140, ACM, 1998.

[7] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. van Meerbergen. Predictable Embedded Multiprocessor System Design. In *Lecture Notes in Computer Science*, LNCS 3199, pages 77-91, Springer, January 2004.

[8] M. Bekooij, R. J. H. Hoes, O. Moreira, P. Poplavko, B. Mesman, J. D. Mol, S. Stuijk, S. V. Gheorghita, and J. van Meerbergen. Chapter 15. Dataflow Analysis for Real-time Embedded Multiprocessor System Design. In *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, Philips Research Book Series, vol. 3, pages 81-108, Springer, 2005.

[9] M. Bereković, H.-J. Stolberg, and P. Pirsch. Multicore System-On-Chip Architecture for MPEG-4 Streaming Video. In *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 12, no. 8, pages 688-699, IEEE, 2002.

[10] G. Bontempi, and G. Lafruit. Enabling Multimedia QoS Control with Black-box Modeling. In *Soft-Ware 2002: Computing in an Imperfect World. Lecture Notes in Computer Science*, LNCS 2311, pages 46-59, Springer, 2002.

[11] J. Bormans, N. P. Ngoc, G. Deconinck, and G. Lafruit. Terminal QoS. Advanced Resource Management for Cost-Effective Multimedia Appliances in Dynamic Contexts. In *Ambient Intelligence: Impact on Embedded System Design*, pages 183-201, Kluwer Academic Publishers, 2003.

[12] N. Brady. MPEG-4 Standardized Methods for the Compression of Arbitrarily Shaped Video Objects. In *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 9, no. 8, pages 1170-1189, IEEE, 1999.

[13] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, K. K. Kulkarni, P. G. Kjeldsberg, T. van Achteren, T. Omnes. *Data Access and Storage Management for Embedded Programmable Processors*, Kluwer Academic, 2002.

[14] S. Chatterjee, and A.S. Hadi. Influential Observations, High Leverage Points, and Outliers in Linear Regression. In *Statistical Science*, vol. 1, no. 3, pages 379-416, Institute of Mathematical Statististics (IMS), 1986.

[15] M. Coenen, and S. Murali, A. Ruadulescu, K. Goossens, and G. De Micheli. A Buffer Sizing Algorithm for Networks on Chip Using TDMA and Credit-based End-to-end Flow Control. In *Proc. 4th International Conference on Hardware/Software Codesign and System Synthesis*, pages 130-135, ACM, 2006.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. Second Edition, the MIT Press, 2001.

[17] Cradle Technologies, Inc. *Multiprocessor DSPs: Next Stage in the Evolution of Media Processing DSPs*. White paper, available at: www.cradle.com.

[18] D.E. Culler, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999.

[19] A. Dasdan, and R. K. Gupta. Faster Maximum and Minimum Cycle Algorithms for System-Performance Analysis. In *IEEE Transations on CAD of Integrated Circuits and Systems*, vol. 17, no. 10, pages 889-899, IEEE, 1998.

[20] R. Garg, C. Y. Chung, D. Kim, and Y. Kim. Boundary Macroblock Padding in MPEG-4 Video Decoding. In *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 12, no. 8, pp. 719-723, 2002.

[21] M. Geilen, Synchronous Dataflow Scenarios. To appear in Proc. of *EMSOFT 2008, the 8th ACM & IEEE International Conference on Embedded Software*.

[22] A. H. Ghamarian, M. C. W. Geilen, T. Basten, and S. Stuijk. Parametric Throughput Analysis of Synchronous Data Flow Graphs. In *Proc. of DATE 2008, Design, Automation and Test in Europe*, pages 116-121, IEEE Computer Society Press, 2008.

[23] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij. Throughput Analysis of Synchronous Data Flow Graphs. In *Proc. of ACSD-2006, 6th International Conference, Application of Concurrency to System Design*, pages 25-34, IEEE, 2006.

[24] S. V. Gheorghita, T. Basten, and H. Corporaal. Profiling Driven Scenario Detection and Prediction for Multimedia Applications. In *Proc. of IC-SAMOS 2006, the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 63-70. IEEE Computer Society Press, 2006

[25] S. V. Gheorghita, T. Basten, and H. Corporaal. Intra-task Scenario-aware Voltage Scheduling. In *Proc. of CASES-2005, the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 177-184. ACM Press, 2005.

[26] S. V. Gheorghita, T. Basten, and H. Corporaal. Application Scenarios in Streaming-Oriented Embedded System Design. In *Proc. of SoC 2006, International Symposium on System-on-Chip*, pages 175-178. IEEE, 2006.

[27] S. V. Gheorghita, T. Basten, H. Corporaal. Application Scenarios in Streaming-Oriented Embedded System Design. To appear in *IEEE Design & Test of Computers*.

[28] S. V. Gheorghita, T. Basten, H. Corporaal. Scenario Selection and Prediction for DVS-Aware Scheduling of Multimedia Applications. In *Journal of Signal Processing Systems*, vol. 50, no. 2, pages 137-161, Springer, 2008.

[29] S. V. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, T. Basten, L. Eeckhout, H. Corporaal, F. Catthoor, F. Vandeputte, and K. De Bosschere. A System Scenario based Approach to Dynamic Embedded Systems. In *ACM Transactions on Design Automation of Electronic Systems*, to appear in 2009.

[30] K. Goossens, J. Dielissen, and A. Radulescu. The Aethereal Network on Chip: Concepts, Architectures, and Implementations. In *IEEE Design and Test of Computers*, vol. 22, no. 5, pages 414-421, IEEE, Sept-Oct 2005.

[31] K. Goossens, J. Dielissen, J. van Meerbergen, P. Poplavko, A. Radulescu, E. Rijpkema, E. Waterlander, and P. Wielage. Guaranteeing the Quality of Services in Networks on Chip. In *Networks on Chip*, ed. by A. Jantsch and H. Tenhunen, pages 61-82, Kluwer Academic Publishers, 2003.

[32] R. Govindarajan, and G. R. Gao. Rate-Optimal Schedule for Multi-Rate DSP Applications. In *Journal of VLSI Signal Processing*, vol. 9, no. 3, pages 211-232, Springer, 1995.

[33] R. Govindarajan, G.R. Gao, and P. Desai. Minimizing Buffer Requirements under Rate-Optimal Schedule in Regular Dataflow Networks. In *Journal of VLSI Signal Processing*, vol. 31, pages 207-229, Kluwer Academic Publishers, 2002.

[34] A. Hansson, K. Goossens, and A. Radulescu. A Unified Approach to Mapping and Routing on a Network on Chip for both Best-effort and Guaranteed Service Traffic. In *VLSI Design - Special Issue on Networks-on-Chip*, article ID 68432, 16 pages, Hindawi Publishing Corporation, 2007.

[35] A. Hansson, M. Coenen, and K. Goossens. Undisrupted Quality-of-service during Reconfiguration of Multiple Applications in Networks on Chip. In Proc. Of *DATE-2007, Design, Automation and Test in Europe, Conference and Exhibition*, pages 954-959, IEEE, 2007.

[36] P. Hoang, and J. Rabaey. Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput. In *IEEE Transactions on Signal Processing*, vol. 41, no. 6, pages 2225-2235, IEEE, June 1993.

[37] C. A. R. Hoare, *Communicating Sequential Processes* (Prentice-Hall International Series in Computer Science), Prentice Hall, Englewood Cliffs, N. J., 1985

[38] R. J. H. Hoes. *Predictable Dynamic Behaviour in NoC-based Multiprocessor Systems-on-Chip*, Master Thesis. Eindhoven University of Technology (TUE), Department of Electrical Engineering, 2004.

[39] Y. Huang, S. Chakraborty, and Y. Wang. Using Offline Bitstream Analysis for Power-aware Video Decoding in Portable Devices. In *Proc. of the 13th ACM International Conference on Multimedia*, pages 299-302, ACM, 2005.

[41] Y. Huang, A. V. Tran, Y. Wang. A Workload Prediction Model for Decoding MPEG Video and its Application to Workload-scalable Transcoding. In *Proc. of the 15th ACM International Conference on Multimedia*, pages 952-961, ACM, 2007.

[42] ISO/IEC 14496-2:1999/ Amd 1:2000, *Coding of Audio-Visual Objects – Part 2: Visual, Amendment 1: Visual Extensions*, Maui, December 1999.

[43] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. of IFIP 74, Information Processing*, pages 471-475, North-Holland Publishing Co., 1974.

[44] R. M. Karp, and J. B. Orlin. Parametric Shortest Path Algorithms with Application to Cyclic Staffing. In *Discrete Applied Mathematics* 3, pages 37-45, 1981.

[45] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunei, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers. YAPI: Application Modeling for Signal Processing Systems. In *Proc. of the 37th ACM/IEEE Conference on Design Automation*, pages 402-405, IEEE, 2000.

[46] E. A. de Kock, Practical Experiences: Multiprocessor Mapping of Process Networks: a JPEG Decoding Case Study. In *Proc. of the 15th International Symposium on System Synthesis*, pages 68-73, ACM, 2002.

[47] H. Kopetz, Software Engineering for Real-time: a Roadmap. In *Proc. of the Conference on The Future of Software Engineering*, pages 201-211, ACM, 2000.

[48] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and Y. Ha. Analyzing Composability of Applications on MPSoC Platforms. In *Journal of Systems Architecture*, vol. 54, no. 3-4, pages: 369-383, Elsevier B.V., 2008.

[49] R. Lauwereins, M. Engels, M. Ade, and J. A. Peperstraete. *Grape-II: A System-Level Prototyping Environment for DSP Applications*. In *IEEE Computer*, vol. 28, no. 2, pages 35-43, February 1995.

[50] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.

[51] E. A. Lee. Absolutely Positively on Time: What would it take? In *IEEE Computer*, vol. 38 no. 7, pages 85-87, IEEE, 2005.

[52] E. A. Lee, and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. In *IEEE Transactions on Computers*, vol. 36, no. 1, pages 24-35, IEEE, 1987.

[53] Y.-T. S. Li, and S. Malik, *Performance Analysis of Real-Time Embedded Software*, Kluwer Academic Publishers, 1999.

[54] C. L. Liu, and J. W. Layland. Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment. In *Journal of the ACM*, vol. 20, no. 1, pages 40-61, ACM, January 1973

[55] Z. Ma, C. Wong, P. Yang, J. Vounckx, and F. Catthoor. Mapping MPEG-4 Visual Texture Decoder Efficiently on a Heterogeneous Multi-processor Platform. In *IEEE Signal Processing Magazine*, vol. 22, no. 3 (Special Issue on Hardware/Software Co-design for DSP), pages 65-74, 2005.

[56] Z. Ma, and F. Catthoor. Scalable Performance-Energy Trade-off Exploration of Embedded Real-Time Systems on Multiprocessor Platforms. In *Proc. of DATE-2006, International Conference on Design Automation, and Test in Europe*, pages 1073-1078, ACM, 2006.

[57] Z. Ma, D. P. Scarpazza, and F. Catthoor. Run-time Task Overlapping on Multiprocessor Platforms. In *Proc. of ESTIMedia 2007, IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia*, pages: 47-52, IEEE, 2007.

[58] H. De Man. 'On Nanoscale Integration and Gigascale Complexity in the post .com World'. keynote speech at *DATE-2002, International Conference on Design Automation and Test in Europe*, available at: http://www.date-conference.com/conference/2002/

[59] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserve: Operating System Support for Multimedia Applications. In *Proc. of International Conference on Multimedia Computing and Systems (ICMCS)*, pages 90-99, IEEE, May 1994.

[60] A. Moonen. *Modeling and Simulation of Guaranteed Throughput Channels of a Hard Real-time Multiprocessor System*. Master Thesis, Eindhoven University of Technology, Department of Electrical Engineering, 2004.

[61] A. Moonen, M. Bekooij, R. van den Berg, and J. van Meerbergen. Decoupling of Computation and Communication with a Communication Assist. In *Proc. DSD-2007, the 10th Euromicro Conference on Digital System Design*, pages 63-68, IEEE, 2007.

[62] A. Moonen, M. Bekooij, R. van den Berg, and J. van Meerbergen. *Analyzing the Impact of a Communication Assist in a Multiprocessor System-on-chip*. Technical Report ESR-2007-05, Eindhoven University of Technology, 2007. http://www.es.ele.tue.nl/esreports/

[63] A. Moonen, M. Bekooij, and J. van Meerbergen. Timing Analysis Model for Network-based Multiprocessor Systems. In *Proc. of ProRISC-2004, the 15th Annual Workshop on Signal Processing, Integrated Systems and Circuits*, pages 91-99, STW Dutch Technology Foundation 2004. www.ics.ele.tue.nl/~epicurus

[64] O. Moreira, J. D. Mol, M. Bekooij, and J. van Meerbergen. Multiprocessor Resource Allocation for Hard Real-Time Streaming with a Dynamic Job Mix. In *Proc. RTAS 2005, the 11th Symposium on Real Time and Embedded Technology and Applications*, pages 332-341 IEEE, 2005.

[65] O. Moreira, J. D. Mol, and M. Bekooij. Online Resource Management in a Multiprocessor with a Network-on-Chip. In *Proc. of the 2007 ACM Symposium on Applied Computing*, pages 1557-1564, ACM, 2007.

[66] O. Moreira, F. Valente, and M. Bekooij. Scheduling Multiple Independent Hard-real-time Jobs on a Heterogeneous Multiprocessor. In *Proc. EMSOFT 2007, the 7th ACM&IEEE International Conference on Embedded Software*, pages 57-66, ACM, 2007.

[67] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli. A Methodology for Mapping Multiple Use-Cases on to Networks on Chip. In *Proc. of DATE-2006, Design, Automation and Test Conference in Europe*, pages 118-123, ACM, March 2006.

[68] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee. Joint Minimization of Code and Data for Synchronous Dataflow Programs. In *Journal of Formal Methods in System Design*, vol. 11, no. 1, pages 41-70, Springer, July, 1997.

[69] N. P. Ngoc. *QoS Management at End-systems for Advanced Multimedia Applications*, PhD Thesis, Katholiek Universiteit Leuven, 2004.

[70] NXP Semiconductors. *PNX4008 Nexperia Mobile Multimedia Application Processor*. Doc. nr: 9397 750 14167, www.nxp.com, November 2004.

[71] C. M. Otero Pérez, L. Steffens, P. van der Stok, S. van Loo, A. Alonso, J. F. Ruíz, R. J. Bril, M. García Valls. QoS-Based Resource Management for Ambient Intelligence. In *Ambient Intelligence: Impact on Embedded System Design*, pages 159-182, Kluwer Academic Publishers, 2003.

[72] M. Pastrnak, P. Poplavko, P. H. N. de With, and D. S. Farin. Data-flow Timing Models of Dynamic Multimedia Applications for Multiprocessor Systems. In *Proc. SoCRT-2004, the 4th IEEE International Workshop on System-on-Chip for Real-Time Applications*, pages 206-209, IEEE, 2004.

[73] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD Dissertation, EECS Department, University of California, 1995.

[74] A. Pearmain, J. Cosmas, A. Carvalho, and V. Typpi. The MoMuSys MPEG-4 Mobile Multimedia Terminal and Field Trials. In *Proc. of ACTS Mobile Communications Summit*, pages 741-746, June 1999.

[75] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman. Task-level Timing Models for Guaranteed Performance in Multiprocessor Networks-on-Chip. In *Proc. CASES-03, the International Conference on Compilers, Architecture and Synthesis for Embedded Systems* pages 63-72, ACM, 2003.

[76] P. Poplavko, T. Basten, and J. van Meerbergen. Execution-time Prediction for Dynamic Streaming Applications with Task-level Parallelism. In *Proc. DSD-2007, the 10th EUROMICRO Conference on Digital System Design*, pages 228-235, IEEE, 2007.

[77] P. Poplavko, M. Pastrnak, T. Basten, J. van Meerbergen, and P.H.N. de With. Mapping of an MPEG-4 Shape-Texture Decoder onto an On-chip Multiprocessor. In *Proc. PRORISC 2003, 14th Annual Workshop on Signal Processing, Integrated Systems and Circuits*, pages 26-27, STW Dutch Technology Foundation, 2003. www.ics.ele.tue.nl/~epicurus

[78] P. Poplavko, T. Basten, M. Pastmak, J. van Meerbergen, M. Bekooij, and P. de With. Extended Abstract: Estimation of Execution Times of On-chip Multiprocessor Stream-oriented

Applications. In *Proc. of MEMOCODE-2005, the 3rd ACM& IEEE International Conference on Formal Methods and Models for Codesign*, pages 251-252. IEEE, 2005.

[79] P. Poplavko, T. Basten, and J. van Meerbergen. *Run-time Prediction of Execution Times of Stream-oriented Applications in Multiprocessors On-chip*. Technical Report ESR-2005-06, Eindhoven University of Technology, 2005. http://www.es.ele.tue.nl/esreports/

[80] K. Richter, M. Jersak, and R. Ernst. A Formal Approach to MP-SoC Performance Verification. In *IEEE Computer*, vol. 36, no. 4, paegs 60-67, IEEE, 2003.

[81] E. Rijpkema, K. Goossens, and A. Radulescu. TradeOffs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip. In *Proc. of DATE-03, International Conference on Design Automation, and Test in Europe*, pages 350-355, ACM, 2003.

[82] M. Rudack, M. Redeker, J. Hilgenstock, and S. Moch. A Large-Area Integrated Multiprocessor System for Video Applications. In *IEEE Design & Test of Computers*, vol. 19, no. 1, pages 6-17, IEEE, 2002.

[83] S. Sriram, and S.S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2002.

[84] M. T. J. Strik, A. H. Timmer, J. L. van Meerbergen, and G.-J. van Rootselaar. Heterogeneous Multiprocessor for the Management of Real-time Video and Graphics Streams. In *IEEE Journal of Solid-State Circuits*, vol. 35, no. 11, pages 1722-1731, IEEE, 2000.

[85] S. Stuijk, T. Basten, B. Mesman and M. Geilen. Predictable Embedding of Large Data Structures in Multiprocessor Networks-on-chip. In Proc. of DSD-2005, the 8th Euromicro Conference on Digital System Design, pages 388-395, IEEE, 2005.

[86] S. Stuijk, M. C. W. Geilen, and T. Basten. Exploring Trade-offs in Buffer Requirements and Throughput Constraints for Synchronous Dataflow Graphs. In *Proc. of DAC-2006, the 43rd Design Automation Conference*, pages 899-904, ACM, 2006.

[87] S. Stuijk, M.C.W. Geilen, and T. Basten. Throughput-Buffering Trade-off Exploration for Cyclo-Static and Synchronous Dataflow Graphs. In *IEEE Transactions on Computers*, vol. 57, no. 10, pages 1331-1345, IEEE, 2008.

[88] S. Stuijk, T. Basten, M.C.W. Geilen, and H. Corporaal. Multiprocessor Resource Allocation for Throughput-Constrained Synchronous Dataflow Graphs. In *Proc. of DAC-2007, the 44th Design Automation Conference*, pages 777-782, ACM, 2007.

[89] S. Stuijk, T. Basten, M. C. W. Geilen, A. H. Ghamarian, and B. D. Theelen. Resource-Efficient Routing and Scheduling of Time-Constrained Streaming Communication on Networks-

on-Chip. In *Journal of Systems Architecture*, vol. 54, no. 3-4, pages 411-426, Elsevier, March-April 2008.

[90] S. Stuijk. *Predictable Mapping of Streaming Applications on Multiprocessors*. PhD Thesis, Eindhoven University of Technology, Eindhoven, the Netherlands, 2007.

[91] T. K. Tan, A. Raghunathan, G. Lakshminarayana, and N. K. Jha. High-Level Energy Macromodeling of Embedded Software. In *IEEE Transcations on Computer-aided Design of Integrated Circuits and Systems*, vol. 21, no. 9, pages 1037-1050, IEEE, 2002.

[92] M. B. Taylor, J. Kim, J. Miller, D. Wentzla, F. Ghodrat, B. Greenwald, H. Hoffman, J.-W. Lee, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. S. M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. In *IEEE Micro*, vol. 22, no. 2, IEEE, 2002.

[93] B. D. Theelen. *Performance Modeling for System-Level Design*. PhD. Thesis, Eindhoven Univ. of Technology, Eindhoven, the Netherlands, 2004.

[94] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk. A Scenario-Aware Data Flow Model for Combined Long-Run Average and Worst-Case Performance Analysis. In *Proc. of MEMOCODE-2006, the 4th ACM & IEEE Conference on Formal Methods and Models in CoDesign*, pages 185-194, IEEE, 2006.

[95] F. Thoen, and F. Catthoor. *Modeling, Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems*. Boston, MA: Kluwer, 1999.

[96] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile 1.28 TFLOPS Network-on-Chip in 65-nm CMOS. In *Proc. of ISSCC-07, IEEE International Solid-State Circuits Conference*, pages 98-589, IEEE, 2007.

[97] M. H. Wiggers, M. J. G Bekooij, and G. J. M. Smit. Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs. In *Proc. of DAC-2007, the 44th Annual Conference on Design Automation*, pages 658-663, ACM, 2007.

[98] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Modeling Run-time Arbitration by Latency-rate Servers in Dataflow Graphs. In *Proc. of SCOPES-2007, the 10th International Workshop on Software & Compilers for Embedded Systems*, pages 11-22, ACM, 2007.

[99] P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, and R. Lauwereins. Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems. In *Proc. ISSS-02, the 15th International Symposium on System Synthesis*, pages 112-119, ACM, 2002.

Samenvatting

Een Nauwkeurige Analyse voor Gegarandeerde Bewerkingssnelheid van Multiprocessor Multimedia Toepassingen

Reeds lange tijd zijn elektroniche apparaten beschikbaar voor vermaak, onderwijs of telecommunicatie doeleinden, die gebaseerd zijn op *multimedia toepassingen*, d.w.z., toepassingen die stromen van audio en video data bewerken in digitale vorm. Het is de verwachting dat in de toekomst de multimedia mogelijkheden in draagbare apparaten meer en meer alledaags zullen worden. Dit leidt tot uitdagingen met betrekking tot kostenefficiëntie en kwaliteit. Dit proefschrift draagt modellen en analysetechnieken bij om de kostenefficiëntie, en daarom ook de kwaliteit, van de multimedia apparaten te verbeteren.

Draagbare elektroniche apparaten moeten enerzijds flexibele functies aanbieden en anderzijds lage vermogensdissipatie vertonen. Deze twee eisen zijn in strijd met elkaar. Daarom concentreren wij ons op een klasse van hardware die een goed compromis tussen die twee eisen vertegenwoordigt, namelijk op applicatiedomein specifieke *multiprocessor systemen-op-een-chip (MP-SoC)*. Ons onderzoek levert een bijdrage tot *dynamische (d.w.z., run-time) optimalisatie* van MP-SoC systeemmetrieken. De centrale vraag daarbij is enerzijds hoe het systeem aan de eisen betreffende tijdaspecten kan voldoen en anderzijds hoe belangrijke systeemmetrieken zoals de waargenomen multimedia kwaliteit of de vermogensdissipatie geoptimaliseerd kunnen worden. In deze gevallen, praten we over vermogensdissipatie of quality-of-service (QoS) management.

In dit proefschrift streven wij het *apriori gegarandeerd* voldoen aan de eisen aan bewerkingssnelheid na, voornamelijk door middel van analytisch redeneren. Dat houdt in dat de analyse van de bewerkingssnelheid *conservatief* moet zijn, d.w.z. het moet pessimistisch zijn wat betreft de omstandigheiden die de bewerkingssnelheid van het systeem negatief kunnen beïnvloeden. In de ingebedde-systemen industrie is conservatief ontwerp het belangrijkste middel om een stabiele kwaliteit te bereiken. Daarom vormt deze benadering ook de basis voor dit onderzoek. Het hoofdonderwerp van dit proefschrift is dus de *analyse van de gegarandeerde bewerkingsnelheid* van multimedia toepassingen op multiprocessoren.

Onze analysemethode is voornamelijk bedoeld voor het aansturen van de dynamische optimalisatie van de systeemtoestand, dat typisch als volgt verloopt. Een manager van de beschikbare hardware en software middlen of een kwaliteitsmanager voorspelt de *uitvoeringstijd*, d.w.z., de tijd die het systeem nodig heeft om een bepaald aantal eenheden invoergegevens te bewerken. Uitvoeringstijden zijn afhankelijk van de bewerkte gegevens. Wanneer de uitvoeringstijden kleiner worden kan de manager de controleparameter voor de gewenste systeemmetriek zodanig instellen dat de metriek verbetert maar het systeem vertraagt. In het geval van de optimalisatie van de vermogensdissipatie wordt het systeem dan ingesteld op een regime met een lagere vermogensdissipatie. Wanneer de uitvoeringstijden groter worden, kan de manager de controleparameter instellen voor snellere bewerking van de invoergegevens om zo de uitvoergegevens op tijd te produceren. Bij QoS management zal de toepassing dan ingesteld worden op een kwaliteitregime met wat lagere kwaliteit. Aan de tijdeisen wordt op die manier altijd voldaan, terwijl de belangrijke systeemmetrieken zo zo goed mogelijk gehandhaafd worden.

Jammer genoeg is het handhaven van systeemmetrieken als vermogensdissipatie en kwaliteit op het optimale niveau tegenstrijdig met ons hoofdvereiste, d.w.z., gegarandeerde bewerkingssnelheid. Om een gegarandeerde bewerkingssnelheid te garanderen moet men af en toe wat kwaliteit of vemogensdissipatie opofferen. Daarom dient de bewerkingssnelheidsanalyse niet alleen conservatief te zijn, maar ook nauwkeurig, zodat de belangrijkste systeemmetrieken niet teveel lijden onder de conservativiteit. Dit idee is echter niet gemakkelijk te realiseren in de aanwezigheid van twee factoren, namelijk, parallelle uitvoering van de applicatie op een aantal processoren en de afhankelijkheid van de uitvoeringstijden van de inputgegevens. Niettemin bereiken we het doel van een conservatieve en nauwkeurige schatting van de bewerkingssnelheid voor een belangrijke klasse van multiprocessoren en multimedia toepassingen.

We beschouwen een algemeen MP-SoC platform dat een dynamische verzameling van toepassingen uitvoert, waarbij elke toepassing gebruik maakt van één of meer processoren. We

veronderstellen dat de toepassingen onafhankelijk zijn. Om tijdeisen te ondersteunen, vereisen we dat het platform gegarandeerde reken-, communicatie- en geheugenbudgetten aan toepassingen kan verstrekken. In overeenstemming met belangrijke trends in systemen-op-een-chip communicatie, onderstenen we zowel globale bussen als *netwerken-op-een-chip*.

Wij modelleren elke toepassing als een homogene synchrone dataflow (HSDF) graaf, waar de toepassingstaken als graaf knooppunten, 'actoren' genaamd, worden gemodelleerd. We ondersteunen dynamische gegevenafhankelijke bewerkingsvertragingen voor actoren. Dit maakt HSDF grafen zeer bruikbaar om moderne multimedia toepassingen te modelleren. Onze reden om HSDF grafen als basismodel te accepteren is verder dat zij een goede basis vormen voor analytische berekening van de bewerkingssnelheid.

In de geschetste context levert dit proefschrift drie belangrijke bijdragen:

1. Gegeven een toepassing die op een MP-SoC platform afgebeeld is, gegeven de snelheidsgaranties voor de processoren en het communicatie netwerk, en gegeven constante vertragingen van de actoren, berekenen we de bewerkingsnelheid van het systeem als geheel.

2. Gegeven een afgebeelde toepassing en snelheidsgaranties zoals in het vorige punt, breiden we onze benadering uit van constante actorvertragingen naar dynamische gegevensafhankelijke actorvertragingen.

3. We stellen een globaal implementatietraject voor dat met de toepassingsspecificatie begint en verder bestaat uit fasen die uitgevoerd worden gedurende het ontwerptraject en gedurende de operationele fase van het systeem. Het implementatietraject gebruikt een uitgebreide versie van het HSDF basismodel als middel om de ontwerpbesluiten die worden genomen weer te geven. We stellen het implementatietraject niet alleen voor om de eerste twee bijdragen in de juiste context te plaatsen, maar ook om onze visie op de verschillende delen van het implementatietraject te presenteren, wat een compleet beeld oplevert.

Onze eerste bijdrage is gebaseerd op zogenaamde IPC (inter-processor communicatie) grafen. Zo'n graaf is gebasserd op het idee om één enkel bewerkingsmodel (namelijk, HSDF) voor alle onderdelen van het systeem te gebruiken, namelijk de rekeneenheden, de communicatieonderdelen en de FIFO (*first-in-first-out*) geheugen modules die gebruikt worden als buffers tussen de rekeneenheden en de communicatie onderdelen. We hebben als eerste HSDF graafstructuren voorgesteld voor de modellering van zowel de in capaciteit beperkte FIFO buffers als ook on-chip netwerk verbindingen die bandbreedte garanties aanbieden. Op die manier maken onze HSDF modellen de graaf-theoretische formulering mogelijk van het minimalisatie probleem van de vereiste FIFO buffercapaciteiten onder tijdeisen. Dat geeft een middel om bij een gegeven kandidaatoplossing voor de buffergroottes de bottlenecks in bewerkingsnelheid op te sporen, om ze vervolgens te kunnen verwijderen. Om dit aan te tonen, gebruiken we de JPEG decoder case study toepassing. Ook, tonen we aan dat, met constante actorvertragingen die conservatief zijn voor een gegeven JPEG beeld, we uitvoeringtijden kunnen voorspellen van het JPEG decoderen op twee processoren met een nauwkeurigheid van 21%.

Onze tweede bijdrage is gebaseerd op een uitbreiding van de scenariobenadering. Deze benadering is gebaseerd op de observatie dat het dynamische gedrag van een toepassing typisch samengesteld is uit een beperkt aantal sub-gedragingen, d.w.z., scenario's, die gelijkaardige vereisten hebben in termen van benodigde hardware middelen, wat neerkomt op gelijkaardige actorvertragingen in de context van dit proefschrift. Het voorafgaande werk aangaande scenario's behandelt slechts toepassingen voor een enkele processor of multiprocessortoepassingen die niet alle flexibiliteit van het HSDF model benutten. Wij ontwikkelen nieuwe op scenario's gebaseerde technieken in de context van HSDF grafen, om de tijdsoverlap in de overgang tussen verschillende scenario's af te leiden, wat in het algemeen essentieel is om een goede nauwkeurigheid te bereiken voor het geval van een multimedia toepassing die draait op een multiprocessor. Wij realiseren dit idee in een case study toepassing – de MPEG-4 decoder van willekeurig-gevormde video objecten, en bereiken een voorspelling van de uitvoeringstijd met een gemiddelde nauwkeurigheid van 11%. Voor zover wij weten, kan, voor de beschreven context, geen andere bestaande prestatieanalysetechniek een vergelijkbare nauwkeurigheid bereiken en tegelijkertijd de te realiseren bewerkingssnelheid garanderen.

Acknowledgements

At the end of this thesis, I would like to express my gratitude to everyone who directly and indirectly helped and supported me in the research work and writing.

First of all, I am very thankful to Philips Research Laboratories (nowadays, NXP), Eindhoven, which was the major institution where – for most time – I conducted the research work for this PhD and which supported my PhD work financially, in 2001-2005. In this period, I worked part-time at the Eindhoven University of Technology and – for three months – at IMEC in Leuven, Belgium. I am very thankful for the excellent facilities and working conditions provided.

This work would not have been possible without the everyday guidance of Prof. Jef van Meerbergen, who throughout the whole period of my PhD contract has been acting as my supervisor. When I was a graduate student, Jef was my University instructor. He introduced me into the world of embedded systems and offered me an exciting research direction for my PhD project. This direction enjoyed much interest in the Research Labs, and thus I got an opportunity to work in a team of enthusiastic and talented researchers. Jef has planned every major step in my PhD research project, continually helping me to move in the direction to success. He also provided indispensable help in the early stages of thesis writing. For all this, I am deeply thankful to him.

I am also thankful to Prof. Ralph Otten for accepting to be promotor at my PhD defense and especially for the firm belief he expressed in my ability to successfully defend my thesis. I am thankful to all the members of the PhD graduation committee for providing valuable feedback.

Next, I would like to give many thanks to my supervisor at the university and copromotor Twan Basten. He is a kind of a supervisor I wish every PhD student could have. Due to his amazing productivity in providing detailed in-depth feedback and due to his creativity and enthusiasm in analyzing open research problems, I could improve the quality of my technical writing and overcome many technical difficulties. All this is despite the fact that for a significant fraction of time I worked at a different location (i.e. at the Research Labs).

I am also very thankful to other colleagues at Eindhoven University of Technology for technical discussions and feedback to my work. Unfortunately, I cannot mention everyone who helped me, but I would like to mention Sander Stuijk, Bart Mesman, Marc Geilen, Amir Hossein Ghamarian, Valentin Gheorghita and Bart Theelen. I owe special thanks to Rian and Marja, the secretaries of the ES group, for helping with the formal paper work. Thanks to all the people with whom I worked in the ES group at the university; I really felt that not only my supervisors cared about my work being successful.

Possibly, I would not have been able to find such a promising and exciting research subject, had it not been for Marco Bekooij and the HIJDRA research group at the Research Labs. Mainly due to joint research work with Marco, my work started to converge into the concrete topic of this thesis. I am also very thankful to the other colleagues at the Research Labs, especially to the employees and students of the HIJDRA and ÆTHEREAL projects. I cannot mention all names, but I would like to mention Kees Goossens, Andrei Radulescu, Edwin Rijpkema, and Orlando Moreira. I have learned a lot when working in the same team with them. I owe special thanks to Erwin de Kock for the source code of the JPEG application, which I used as a case study.

Milan Pastrnak was a PhD student with whom I worked together on mapping video applications to multiprocessors platforms. He has proven to be a very productive software

developer, making the MPEG-4 application case study possible. And most of all I appreciate him as a very good friend.

Also the three months I spent at the DESICS division of IMEC, the institute of microelectronics in Belgium, have helped me to expand my scope in the research area of this thesis. I remember my IMEC colleagues as brilliant researchers and really good friends.

Further on, I would like to thank my current colleagues, at Magma Design Automation. When my PhD contract finished in 2005, they offered me a very interesting research and development job at Magma. This job allowed me to stay in Eindhoven and to not give up with my thesis. Interestingly enough, working in a different research topic has helped me to get new ideas for my thesis writing. My thesis even owes some technical terms to Magma, for example, ‘preparation’, ‘characterization’, and ‘timing modes’, although with a different meaning.

My PhD project was preceded by a graduation program at the Eindhoven University in 1999-2001. I am very thankful to Prof. Jochen Jess, because my education and carrier in the Netherlands would be impossible without the effort he invested in organizing my graduation program. I am also very much grateful to Joep Beriére and Gerard Beenker, from Philips, for organizing the scholarship for my graduation program.

I think I owe a lot not only to colleagues, but also to my friends and family. I appreciate very much the friendship of Herman van Hooff, Vladimir Kabzar, Andrei Sazonov, Irina Sazonova, Andrei Yakunin, Manindra Sarkar, Milan Pastrnak, Nelly and Natasha Litvak, Srinath Naidu, Satnam Singh, and many others. I would like to express many thanks to ‘4U2’, a Christian organization that welcomes foreign students in Eindhoven, for hospitality and for a great experience of participating in their organization. I would like to also express my gratitude to Trinity Church Eindhoven and Orthodox Parish of St. Nektarios in Eindhoven for everything they have meant to me during the last years.

I am very thankful to my father Yuri and my mother Larisa, for bringing me up and encouraging me to become a technical scientist, just as themselves. They have provided a lot of care, encouragement and support to me and my wife, even living 2000 km away from me, during my long stay in the Netherlands. Without my father’s encouragement, I would never have had enough courage to study in a foreign country. I am also very thankful to other family members, especially to my sisters Mariya and Irina, my mother-in-law Valentina, my sister-in-law Mariya, and to all my friends in Ukraine, for keeping in touch and supporting me and my wife via e-mail/Skype communication and when we visit Ukraine.

In 2005 my family suffered a big loss, as my mother passed away, abruptly. I owe so much to her love, care and help. I dedicate this thesis to her.

Last by not the least, I wish I could express in words as many thanks to my dear wife Darya as she really deserves. She has shown a lot of patience and has given me a lot of care and support. Her love has provided me with comfort and inspiration to get this work finished.

About the Author

Peter – official name Petro Poplavko – was born on the 16th of June 1977 in Kiev, Ukraine. In June 1998, he got the Bachelor Degree in Computer Science from the Computer-Aided Design (CAD) group of the National Technical University of Ukraine.

In September 1999, Peter was offered a scholarship for a Master Program in Electronics from Philips Research Laboratories in Eindhoven, the Netherlands (nowadays NXP Semiconductors). Peter received the Master Degree in Information Technology from the Electronic Systems (ICS/ES) group of the Eindhoven University of Technology in June 2001. The topic of his graduation project was optimization of field-programmable gate arrays, and it was carried out at the embedded system architecture department at Philips Research Labs.

In July 2001, Peter was offered a PhD research assistant position at the Eindhoven University of Technology, again with the support of Philips Research Labs. In the first period of his PhD project (2001-2002), Peter worked in a network-on-chip project group at the digital circuit design department at Philips. There, he was involved in the development of networks-on-chip design methodology for multimedia systems-on-chip. During this period, Peter had a three-month internship at the IMEC institute of microelectronics in Belgium.

In the second period of his PhD project (2003-2004), Peter worked in the embedded multiprocessor group at Philips. There, he was involved in the development of novel performance analysis and optimization techniques for mapping the streaming applications on multiprocessor systems-on-chip.

Since October 2004, Peter is with Magma Design Automation (Magma DA), Eindhoven, as a research and development (R&D) employee. The current topic that Peter works on at Magma DA is physical synthesis and optimization of clock distribution networks in integrated digital circuits.

List of Publications

Book Chapters

- K. Goossens, J. Dielissen, J. van Meerbergen, P. Poplavko, A. Radulescu, E. Rijpkema, E. Waterlander, and P. Wielage, "Guaranteeing the Quality of Services in Networks on Chip". In Networks on Chip, ed. A. Jantsch and H. Tenhunen, pages 61-82, Kluwer Academic Publishers, 2003.
- M. Bekooij, R. Hoes, O. Moreira, P. Poplavko, M. Pastrnak, B. Mesman, J. D. Mol, S. Stuijk, S.V. Gheorghita, J. van Meerbergen, “Dataflow analysis for real-time embedded multiprocessor system design”, Chapter 15, Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices, ed. P. van der Stok, Philips Research Book Series, vol. 3, pages 81-108, Springer, 2005.

Conferences

- P. Poplavko, T. Basten, and J. van Meerbergen. “Execution-time Prediction for Dynamic Streaming Applications with Task-level Parallelism” In Proc. DSD-2007, the 10th EUROMICRO Conference on Digital System Design, pp: 228-235, IEEE, 2007.

- P. Poplavko, T. Basten, M. Pastrnak, J. van Meerbergen, P.H.N. de With, “Extended Abstract: Estimation of Execution Times of On-chip Multiprocessor Stream-oriented Applications”. Proc. International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp: 251-252, IEEE, 2005.
- M. Pastrnak, P. Poplavko, P.H.N. de With and D.S. Farin, “Data-flow Timing Models of Dynamic Multimedia Applications for Multiprocessor Systems”, Proc. System-on-Chip for Real-Time Applications, The 4th IEEE International Workshop, (IWSOC04), pp: 206-209, IEEE, 2004.
- M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. van Meerbergen “Predictable Embedded Multiprocessor System Design”, Proc. 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES), pp: 77-91, Springer, 2004.
- P. Poplavko, M. Pastrnak, T. Basten, J. van Meerbergen, P.H.N. de With, “Mapping of an MPEG-4 Shape-Texture Decoder onto an On-chip Multiprocessor”, 14th Workshop on Circuits, Systems and Signal Processing (PRORISC), pp: 139-147, Dutch Technology Foundation STW, 2003.
- P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman, “Task-level Timing Models for Guaranteed Performance in Multiprocessor Networks-on-Chip”, International Conference on Compilers, Architectures, and Synthesis of Embedded Systems (CASES), pp: 63-72, ACM, 2003.
- M. Pastrnak, P. Poplavko, P.H.N. de With, J. van Meerbergen, “On Resource Estimation for MPEG-4 Video Object Shape-Texture Decoding on Multiprocessor Network-on-Chip”. 4th Seminar on Embedded Systems (PROGRESS), Dutch Technology Foundation STW, 2003.
- P. Poplavko and M. Pastrnak, “Modeling Predictable Multiprocessor Performance for Video Decod-ing” Workshop On the Design of Multimedia Architectures: pp: 133-136, Technische Universiteit Eindhoven, 2003.
- P. Poplavko, K. Leijten-Nowak and J. van Meerbergen, “Placement Algorithms for Datapath-Oriented FPGAs”, 12th Workshop on Circuits, Systems and Signal Processing (PRORISC), pp: 561-567, Dutch Technology Foundation STW, 2001.
- P. Poplavko, C.A.J. van Eijk and T. Basten, “Constraint Analysis and Heuristic Scheduling Methods”, 11th Workshop on Circuits, Systems and Signal Processing (PRORISC), Dutch Technology Foundation STW, 2000.
- K. Shcherbin, S. Odoulov, P. Poplavko, “Photorefractivity of CdTe:Ge at 1.06 and 1.32 /spl mu/m”, Lasers and Electro-Optics Europe, IEEE, pp: 297-297, 1998.