Exploring resource/performance trade-offs for streaming applications on embedded multiprocessors

Citation for published version (APA):
Yang, Y. (2012). *Exploring resource/performance trade-offs for streaming applications on embedded multiprocessors*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Electrical Engineering]. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR733434

DOI:
10.6100/IR733434

Document status and date:
Published: 01/01/2012

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

Exploring Resource/Performance Trade-offs for Streaming Applications on Embedded Multiprocessors

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 2 juli 2012 om 16.00 uur

door

Yang Yang

geboren te Sichuan, China

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. T. Basten
en
prof.dr. H. Corporaal

Copromotor:
dr.ir. M.C.W. Geilen

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Yang, Yang

Exploring Resource/Performance Trade-offs
for Streaming Applications on Embedded Multiprocessors
/ by Yang Yang. - Eindhoven : Technische Universiteit Eindhoven, 2012.
A catalogue record is available from
the Eindhoven University of Technology Library
ISBN: 978-90-386-3170-7
NUR 959
Trefw.: multiprogrammeren / elektronica ; ontwerpen / multiprocessoren /
ingebedde systemen.
Subject headings: data flow graphs / electronic design automation /
multiprocessing systems / embedded systems.

Exploring Resource/Performance Trade-offs for Streaming Applications on Embedded Multiprocessors

Committee:

prof.dr.ir. T. Basten (promotor, TU Eindhoven)
prof.dr. H. Corporaal (promotor, TU Eindhoven)
dr.ir. M.C.W. Geilen (copromotor, TU Eindhoven)
prof.dr.ir. A.C.P.M Backx (chairman, TU Eindhoven)
prof.dr.ir. M.J.G. Bekooij (University of Twente, NXP Semiconductors)
prof.dr.ir. C.H. van Berkel (TU Eindhoven, ST Ericsson)
Prof.Dr. S. Chakraborty (TU München)

This work has been carried out as part of the Octopus project with Océ Technologies B.V. under the responsibility of the Embedded Systems Institute. This project was partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program.

Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 257.

Printing: Printservice Technische Universiteit Eindhoven

Abstract

Exploring Resource/Performance Trade-offs for Streaming Applications on Embedded Multiprocessors

Embedded system design is challenged by the gap between the ever-increasing customer demands and the limited resource budgets. The tough competition demands ever-shortening time-to-market and product lifecycles. To solve or, at least to alleviate, the aforementioned issues, designers and manufacturers need model-based quantitative analysis techniques for early design-space exploration to study trade-offs of different implementation candidates. Moreover, modern embedded applications, especially the streaming applications addressed in this thesis, face more and more dynamic input contents, and the platforms that they are running on are more flexible and allow runtime configuration. Quantitative analysis techniques for embedded system design have to be able to handle such dynamic adaptable systems.

This thesis has the following contributions:

- *A resource-aware extension to the Synchronous Dataflow (SDF) model of computation.*
- *Trade-off analysis techniques, both in the time-domain and in the iteration-domain (i.e., on an SDF iteration basis), with support for resource sharing.*
- *Bottleneck-driven design-space exploration techniques for resource-aware SDF.*
- *A game-theoretic approach to controller synthesis, guaranteeing performance under dynamic input.*

As a first contribution, we propose a new model, as an extension of static synchronous dataflow graphs (SDF) that allows the explicit modeling of resources with consistency checking. The model is called resource-aware SDF (RASDF). The extension enables us to investigate resource sharing and to explore different scheduling options (ways to allocate the resources to the different tasks) using state-space exploration techniques. Consistent SDF and RASDF graphs have the property that an execution occurs in so-called iterations. An iteration typically corresponds to the processing of a meaningful piece of data, and it returns the graph to its initial state. On multiprocessor

platforms, iterations may be executed in a pipelined fashion, which makes performance analysis challenging. As the second contribution, this thesis develops trade-off analysis techniques for RASDF, both in the time-domain and in the iteration-domain (i.e., on an SDF iteration basis), to dimension resources on platforms. The time-domain analysis allows interleaving of different iterations, but the size of the explored state space grows quickly. The iteration-based technique trades the potential of interleaving of iterations for a compact size of the iteration state space. An efficient bottleneck-driven design-space exploration technique for streaming applications, the third main contribution in this thesis, is derived from analysis of the critical cycle of the state space, to reveal bottleneck resources that are limiting the throughput. All techniques are based on state-based exploration. They enable system designers to tailor their platform to the required applications, based on their own specific performance requirements. Pruning techniques for efficient exploration of the state space have been developed. Pareto dominance in terms of performance and resource usage is used for exact pruning, and approximation techniques are used for heuristic pruning.

Finally, the thesis investigates dynamic scheduling techniques to respond to dynamic changes in input streams. The fourth contribution in this thesis is a game-theoretic approach to tackle controller synthesis to select the appropriate schedules in response to dynamic inputs from the environment. The approach transforms the explored iteration state space of a scenario- and resource-aware SDF (SARA SDF) graph to a bipartite game graph, and maps the controller synthesis problem to the problem of finding a winning positional strategy in a classical mean payoff game. A winning strategy of the game can be used to synthesize the controller of schedules for the system that is guaranteed to satisfy the throughput requirement given by the designer.

Contents

1 INTRODUCTION

"A journey of a thousand miles begins with a single step."

– Lao Tzu

Nowadays, the modern life of mankind is more or less defined by the devices someone uses in his or her daily life, such as a mobile phone, portable media player, e-book reader, and so on. Most of these devices are dedicated to a few functions and hide computer systems inside so that the computer systems are invisible to users. These hidden computers are called ***Embedded Systems*** and have become ubiquitous in our lives. The wide use of embedded systems digitizes our life and pushes us into the so called post-PC era.

1.1 The emergence of embedded streaming applications.

The design of many embedded systems requires domain-specific knowledge. In many of these embedded systems, one can find signal processing algorithms, ranging from data fusion of sensor nodes to radar imaging on satellites. A lot of core functions of consumer electronics, industrial products, and defense systems are based on knowledge from the signal processing domain. Since the data of these signal processing applications are continuously flowing through the systems like streams, these applications are also called streaming applications [154]. Example streaming applications include audio/video codecs in portable MP3/MPEG4 players, image processing systems such as cameras and printers, and communication systems such as mobile phones and base stations. Design of streaming applications on embedded systems is an important subdomain of embedded system design.

The software design of streaming applications is driven by the dataflow paradigm. When designing streaming applications on embedded systems, block diagrams are frequently used for application description. The algorithm designers are used to describe their signal processing algorithms as block diagrams on paper, in which data are ***flowing*** through and are processed by these blocks. Since these blocks and the connections among those blocks only capture high level information of the system, they are easy to understand and change relatively slowly when compared to their underlying implementations. So the block diagrams are frequently used as media for documenting and communicating algorithm designs among the designers. For example, Figure

1.1 shows a block diagram of an MPEG2 encoder from a design white paper provided by CoFluent® Design. The block diagram clearly shows the flow of image frame data and the processing steps of the MPEG2 encoder [26].

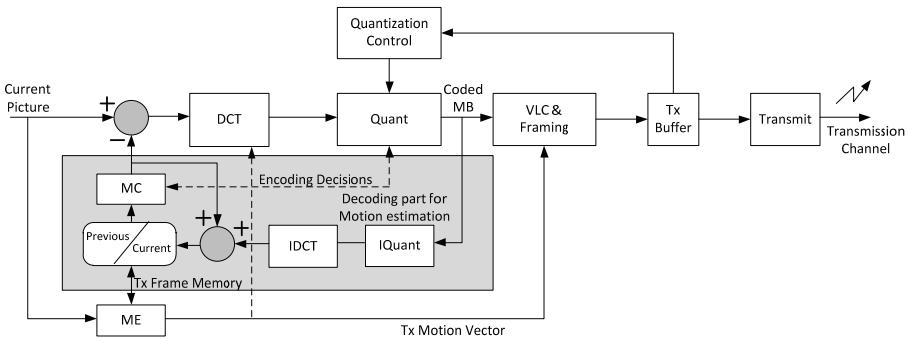


Figure 1.1 Block diagram of the MPEG2 Encoder [26]

The hardware design of streaming applications is driven by the progress of manufacturing technology and the utilization of the nature of streaming applications. The progress in hardware manufacturing technology so far manages to make the integration of digital circuits adhere to Moore's law. By scaling geometrically, we transit from the single core era to a multi/many-core era while keeping power consumption under physical limits. The innovation of hardware architecture turns the advance in the integration level to a gain in performance. Many of those architecture innovations, such as Application-Specific Instruction-set Processor (ASIP) [71], Very Long Instruction Word (VLIW) [50], Single Instruction, Multiple Data (SIMD) [78], Reconfigurable Computing [32] architectures, are driven by observations from the streaming application domain. Through heterogeneous integration, different functional units of an embedded system such as sensors, processing units and memories are put on the same chip, resulting in a System-on-Chip (SoC). From mobile phones to complex communication networks, SoCs are widely used in these final products for performance/power efficiency and cost reduction. Moreover, non-digital parts such as sensors, actuators and biochips are also starting to be integrated into the system; the system is then called a system-in-package (SiP). Integration-level is raised with both "more Moore" (increasing digital integration by scaling down logic gates) and "more-than-Moore" (increasing non-digital contents integration such as MEMS) [175].

Embedded streaming applications always need to satisfy tight and strict performance requirements. In many cases, the streaming data of these applications must be processed within a tight time budget to meet their throughput constraints to ensure customer satisfaction or good user experience. For example, depending on the profiles, the video bit rate of H.264/AVC ranges from 64 kbps to 240 Mbps [161]. Meanwhile, the resource limitations on many embedded systems constrain the options of design and implementation. System designers have to balance between performance requirements and resource constraints, and obtain the optimal solutions based on customer needs, which requires careful mapping between signal processing algorithm blocks and hardware components. The better the match between the mathematical operations of the software blocks and their underlying hardware implementations, the higher the probability that the given performance/power requirements of whole system are reached. A good design always implies a good ***mapping*** between an ***application*** and its ***architecture***.

Both industry and academic worlds show great interest in streaming applications in the last two decades. Commercial tools for developing streaming applications include Signal Processing Worksystem (SPW) [9, 123] from Cadence (later acquired by Coware, in turn acquired by Synopsis), COSSAP [129] (later merged into Cocentric System Studio, formerly called El Greco [22]) from Synopsis, ADS [122] (formerly EEsof from HP) and SystemVue [139] from Agilent, Labview [90] from National Instruments and System Canvas [111] from Angeles Design Systems. Researchers around the world also develop academic tools for modeling and analysis purposes, such as Ptolemy [23] from U.C. Berkeley, DIF [83] from University of Maryland, StreamIt [154] from MIT, PeaCE [149] from Seoul National University and SDF^3 [146] from Eindhoven University of Technology.

In a nutshell, a good embedded streaming application system design needs to consider application (software), architecture (hardware) and mapping (hardware/software codesign) aspects together. But none of the aspects is easy in embedded system design due to the increasing complexity. Both industry and academia put a lot of effort in investigating the problem, i.e. how to design and implement embedded streaming systems in a systematic and better way. We list below the challenges in designing streaming applications on embedded systems and the trends of solutions for the challenges that we think are most important.

1.2 The challenges in embedded system design.

The ever-increasing complexity and the ever-shortening time-to-market: The increasing capabilities of embedded systems, together with the perhaps even more dramatic increase in their application in everyone's daily life causes the design complexity of embedded system to grow sharply. Designers cannot simply consider different parts of a system separately and then try to put them together in a naive way. All those different parts now are connected to each other and influence each other. To make things even worse, the fast evolution of technology and fierce competition in the market also shorten the patience of customers. The ever-shortening time-to-market puts additional pressure on designers while they are handling design complexities that they have never seen before. The International Technology Roadmap for Semiconductor (ITRS) showed in its report of 2009 [86] the ever-enlarging design gap between HW/SW productivity and manufacturability.

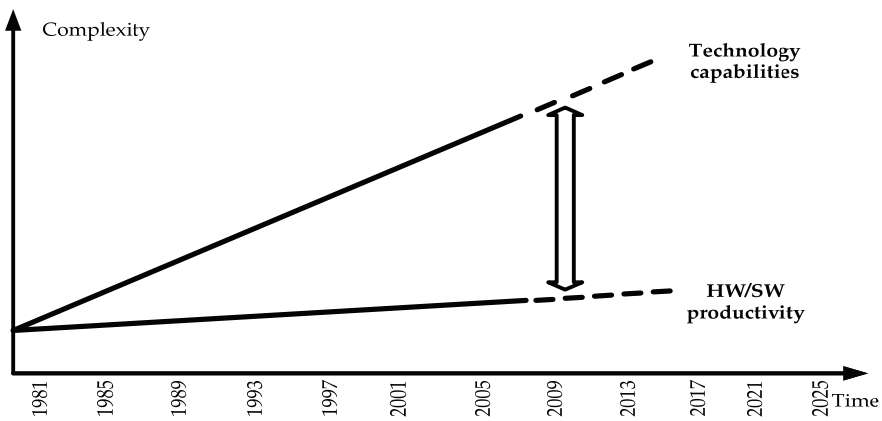


Figure 1.2 Hardware and Software design gap versus Time (based on [86]**)**

Figure 1.2 shows a design gap prediction graph based on the ITRS report. The complexity of the design of software/hardware is measured in a logarithmic scale. The technology capabilities such as manufacturing process advances or new materials are not necessarily turned into benefits for end producers because of the ever-enlarging design gap between the productivity of developers and the capabilities of technology. The increasing design cost might eventually stifle the whole industry if a sustainable product development cycle becomes economically impossible due to the gap.

The ever-increasing development cost: For consumer electronics, we already got used to the phenomenon that the prices of new products drop sharply a few month later to give way to even newer products that are more powerful and efficient. Moreover, the fine segmented market also limits the volume of a single product and narrows the profit margin of the product. At the same time, the Non-Recurring Engineering (NRE) cost of products, i.e. one time cost during the research, design, development, and testing phases just keeps soaring with the increasing complexities of products no matter the products are consumer electronics or professional and industrial products. The progress in manufacturing technology drops the price of hardware every year. However, the salary and productivity of programmers for both software/hardware keeps more or less stable or slowly grows. As a result, the share of "software" (including software and programmable hardware code) in the total cost of products keeps increasing. The designers have to increase their productivity to make a sustainable profit from their products.

The multi-objective and non-functional requirements: In the nature of embedded system design, there are many design constraints other than functional correctness. Due to the deployment environment, resource restrictions on implementations, constraints on performance related metrics, many non-functional requirements need to be taken into consideration when designing an embedded system, such as performance, power and reliability. Compared to well-developed methodologies that ensure functional correctness of systems, we still lack systematic ways of designing systems to satisfy non-functional requirements. Moreover, we frequently find that we have to design a product that has to satisfy multiple objectives while some objectives are in conflict with each other.

The multi/many-core era: In order to complete many different tasks or react to different types of input in real-time, embedded systems are running multiple applications or multiple tasks of one application on so-called Multi-Processor System-on-Chip (MPSoC) platforms at the same time. In order to enable parallel applications, the underlying architectures also have some kinds of parallel features, from multi/many cores at the system level to VLIW or SIMD at the instruction level. Keeping the parallelism intellectually manageable, i.e., ensuring that designers are capable of designing, debugging and deploying such kinds of systems, is very challenging [107].

The environment: Last but not least, the environment is also one of the most important design constraints for embedded systems. Embedded systems aim to embed into products and react to the environment in one way or the other. The environment influences the behavior of an embedded system by providing all kinds of inputs. In a streaming application, the environment will feed input as a data stream while the type and content of data might change depending on the situation. Taking the environment into consideration not only makes the design more reliable and predictable, but also makes the design more efficient. However, it also requires more effort from designers due to the consideration of this new dimension.

To conclude, the challenges faced by embedded system designers are caused by the gap of ever-increasing desires of customers and the relatively slow-growing design capability that can satisfy the desires. The success of past designs of embedded systems boosts the expectation for future embedded systems. The tension between the means and ends will continue as embedded systems already become an important part of our daily lives.

1.3 The trends in embedded system design.

In order to solve the aforementioned challenges, the research and practice on design methodologies for embedded systems also made progress in the last few decades. There are a few noticeable trends in current embedded system design that we discuss below.

Model-based Design: The computer industry advances by introducing models and raising the abstraction level. By doing so, we manage to understand and solve complex problems at different levels. Instead of designing a product directly with unnecessary details, a model-based approach provides a way to design products at a different abstraction level with only essential information. Depending on their abstraction levels, models approximate the behavior of final products in different ways and are used for different purposes. The models can be viewed as virtual prototypes of products and are used by designers for evaluation and estimation. Comparing to long and expensive procedures to develop physical prototypes for evaluation or very primitive spreadsheet calculations which are based on designers' experiences, the model-based approach is much faster, effective and accurate. The most aggressive model-based design approaches even support generating implementations directly from models. For streaming applications, different types of dataflow

models are proposed for performance and resource usage evaluations, such as Real-time Calculus [152], Event Model [69], and Synchronous Dataflow [101]. Many commercial products are already developed for functional simulation or fast prototyping purposes such as SymTA/S [76] and SPW [9].

Separation of Concerns: Due to the increasing complexity of embedded systems, designers can handle all aspects neither at the same time nor within the same person. Instead, approaches that try to separate concerns such an aspect-oriented [97, 132] or domain-specific approach [81, 108], or the Y-chart methodology [8, 98] are proposed. A complex embedded system design problem is divided into many different orthogonal or loosely coupled subproblems, which are solved separately and efficiently by aspect/domain experts with their domain-specific knowledge. Then system designers or a central design system are responsible for integrating all components together. On the one hand, an aspect-oriented approach maximizes the efficiency of designers; on the other hand, it allows modification of one aspect with as few as possible side-effects to other aspects. Aspect-oriented or domain-specific design is rooted in a problem solving skill of human beings with a long history: divide and conquer. Obviously, the loose coupling to other design aspects and efficient and holistic integration are key challenges for the approach. The Y-chart methodology is another good example of separating embedded system design concerns. It separates application functionality aspects (typically designed in software) from platform aspects (typically hardware) and the mapping of application functionality onto the platform. There are many commercial tools that provide domain-specific languages for specifying requirements or modeling behaviors [18, 35, 79]. The domain-specific languages allow designers to improve their productivity while utilizing domain-specific information for efficient optimization [4, 81].

Hardware/Software Co-design: While splitting loosely coupled concerns into different aspects, considered separately, we have to consider tightly coupled aspects all together as a whole. Since the line between hardware and software is blurred due to introduction of high-level synthesis tools, the Hardware/Software Co-design approaches [19–21] became popular in current embedded system design. Designers can choose to map a computation task on an acceleration unit (hardware implementation) or on a general purpose processing unit (software implementation), the only difference being the processing time and resource usage. By careful design, end users will not be aware of the difference. The Hardware/Software Co-design approach also

requires designers to use high-level specifications that abstract from implementation details since the final implementation can vary significantly. In order to support Hardware/Software Co-design, model-based or virtual platform based design methodologies are developed. Model-based exploration allows fast evaluation of implementation choices since both software and hardware components are modeled and can interact with each other through well-defined interfaces. Virtual platforms provide cycle-level models or faster transaction-level hardware models in the early stage of software design for hardware/software co-design.

System-Level Exploration and Early Design Space Exploration: Abstraction and a model-based design approach enable designers to perform design space exploration at system level. We can do early design space exploration as soon as we have system-level models. Even though the system-level models do not contain many details, we can still make some important and crucial decisions at the early stages of embedded system design. For example, [86] predicts that the contribution of ESL design to system power minimization will account for 80% in 2015. Early design space exploration [70] is very important in the whole design cycle since it helps designers to identify the important decisions at system-level so that they can figure out a way to properly handle concerns early on. By reducing the problems at the system-level at an early stage, the number of iterations for a design can decrease noticeably. In turn, it is possible to shorten the time-to-market.

Correct by construction: Traditional approaches verify the correctness of a system at the final stage, which might be too late for any meaningful remedies if a design error is found. On the contrary, correct-by-construction design [43, 44] that is proposed by Prof. E.W. Dijkstra stresses that the correctness of systems should be ensured by the way of constructing the system, i.e., through formal methods [45]. For example, Design-by-Contract [109] is a well-known technique for software engineering that, by enforcing formal and verifiable interface specifications for software modules, ensures the correctness of the system. The techniques for correct-by-constructions have their roots in formal specification and verification [7, 92].

Trade-off aware design: Since designing for multiple objectives [177] such as performance and energy efficiency becomes common in embedded system design and some of these objectives might be in conflict with each other, it is very useful for a design method to provide trade-offs for designers to select

one implementation among different design options. Even more, the trade-offs can be used for tailoring existing designs for different customer needs or configuring running applications for different environments.

Reusable, Adaptable and Adaptive design: In order to improve productivity, reusing existing designs or adapting them for new situations is good practice. Intellectual Property (IP) libraries and Platform-based design [106, 134] try to reuse at component level and platform level respectively. Though the requirements of a product are heavily dependent on time, locations, and individual needs, we can always find some requirements/functions that are the same or similar to existing designs. By reusing existing designs, we can construct the known part quickly and focus our efforts on new functional parts and their interface to reused parts. An ideal design can adapt to changes, no matter whether the changes that happen at design time (adaptable) or runtime (adaptive). The changes may even cross different generations of the product or different markets of the product. At design time, an adaptable design should be able to be adapted to changes in requirements or modifications of different aspects of the specification. At run time, an adaptive design should be able to adapt to different environment inputs and react with different strategies, such as different schedules, operating frequencies or voltages. Adaptable and adaptive design can alleviate the problem of soaring NRE cost by amortizing cost over similar products that target different customers and different generations of the same products. Moreover, an environment sensitive design can ensure the efficiency of products and achieve efficiency that is not achievable through a static approach [3, 48]. By tailoring platform resource configurations for different needs at design time [96] or by changing configuration at runtime to react to environment changes [17], the investment of reusable, adaptable and adaptive parts and the experience and knowledge of designers are preserved and utilized.

More design automation: One limiting factor that causes the productivity gap is the lack of enough manpower for design. There are two reasons: the first one is the lack of skilled designers to manage the ever-increasing complexity; the second one is the huge amount of complex design tasks generated from the ever-increasing demands on embedded systems. While improving our education system can solve the first problem, more design automation is the remedy for the second one. By shifting well-defined and automatable design tasks to computers, the hands of designers can be liberated for those less well-defined and highly creative design tasks. The prosperity of the last few

decades of semiconductor industry is largely due to the progress that was made in design automation. The future of the industry needs the continuation of the success.

Tackling emerging needs: The advances of embedded systems lead to new design techniques to tackle emerging problems such as thermal aspects [85, 153], security [160, 173], reliability [159], resource management [17, 72], and concurrency [135, 168]. The variety of design issues raised in embedded system design reflects the influence of embedded systems in our life. The need to solve these emerging issues leads the trends in the design automation research and development.

We list many design trends for embedded systems above. Some of them have already become practice in industry and may still be improved while others need further research to reach industrial strength for real applications. Moreover, all these trends are not independent from each other, but closely related. The progress of one will also improve the other. How to turn those "nice to have" research tools to some really "reducing design pains" tools is one of the most important motivations behind design automation research.

In this thesis, we are motivated by the challenges faced by embedded system design and are inspired by the design trends shown in the past decades. We are investigating the problem of designing streaming applications on embedded platforms. Given the wide use of streaming applications and complexity of the design problem, this thesis is only a small step of this long journey, but we believe it is a constructive one.

1.4 Problem Statement

To solve the challenges facing embedded system design is not an easy journey. Here, we focus in particular on a specific subdomain of embedded system design: streaming applications.

1.4.1 The performance versus resource cost trade-off

A key issue for developing streaming applications is the analysis of performance and resource usage of a streaming application. The two aspects, performance and resource usage, are tightly related to each other. Normally, higher performance requires more resources. Embedded system designers face the natural dilemma to increase resources for more performance or to decrease

resources for a more economic implementation. This dilemma, the so-called trade-off problem (or opportunity-cost problem in economics), is widely seen in many engineering fields, i.e., there typically is no single optimal option, but instead there are a lot of different options, each with its own advantages and disadvantages. The trade-offs are Pareto-optimal options [56] in their corresponding design space. The goal of trade-off analysis is to find these Pareto-optimal options in the design space for which there is no alternative option that is strictly better in all aspects.

1.4.2 Approaches for performance and resource usage analysis

There is a lot of work that studies performance and resource usage for streaming application systems. Models of Computation (MoCs) are models that are used to study this type of problems related to computation. We discuss a few MoCs below.

Network Calculus and Real-Time Calculus: Network Calculus (NC) [19, 33, 34] is developed as a deterministic queuing theory for computer networks. It applies min-plus algebra to networking systems where the addition is replaced by computation of minimum and the multiplication is replaced by addition. It can be used to reason about timing properties of *event streams* in queue networks. The envelope of event streams can be captured by arrival curves [33, 34]. Since streaming application systems can be viewed as a network of processing blocks, NC can be used for analyzing end-to-end delays, event rates and buffer requirements (backlog in NC terminology) between processing blocks. In NC, latency-rate ($\mathcal{LR}$) server [143] is introduced to model and analyze traffic scheduling in communication networks. The behavior of a $\mathcal{LR}$-server is determined by two parameters, the latency (Θ) and the allocated service rate (ρ) for the input traffic stream. Several scheduling algorithms such as Round-Robin scheduling can be classified as $\mathcal{LR}$-servers. It can be used to analyze performance of SoC architectures [155] in which tasks are captured as a set of traffic flows with associated latency constraints.

Real-Time Calculus (RTC) [152] is an extension of NC. A detailed comparison between NC and RTC can be found in [20]. In contrast to NC, RTC uses so-called *interval bound functions* to characterize both event streams and available resources. By applying interval analysis on event streams, i.e., sliding an interval window over event streams and counting the maximum (upper arrival curve) and minimum (lower arrival curve) number of events appearing

during the interval Δt. Resources are indirectly captured by service curves which denote numbers of input events have been served to a so-called greedy processing unit. RTC is used to study the schedulability and buffer requirements of real-time system.

Modular Performance Analysis (MPA) [27, 156] is an analysis framework developed on top of RTC. In this framework, a system is composed by a number of building blocks of which the inputs and outputs are characterized by pairs of upper and lower arrival curves and service curves. The building blocks can connect their own inputs and outputs, for events and services, with other blocks' outputs and inputs respectively. The framework allows interface-based design and provides modular analysis. A case study is given in [157] to show how to apply MPA for embedded system performance analysis.

Event Models: An Event Model (EM) [69, 76, 127] is a set of event streams that share common qualitative properties such as a period T, jitter J and deadline D etc. A system is viewed as a set of processing blocks that can be independently scheduled and be analyzed locally. These blocks communicate or interact via *event streams*. Event streams represent the interface between blocks and sub-systems. Two category functions are defined for event streams to capture the properties of a stream. One is $\eta(\Delta t)$ that returns the number of possible event occurrences within a time interval of size Δt and the other is $\delta(n)$ that returns the time distance Δt between n successive events in the stream. Actually, the event stream function $\eta(\Delta t)$ can be converted to the upper bound and lower bound arrival curves for the event stream [127]. The inputs and outputs of processing blocks are captured by the EM. The Event Model Interface (EMI) is developed for connecting different event models. In [128, 133], techniques are developed for hierarchical scheduling analysis of hierarchical event stream system. A EM based commercial tool SymTA/S is developed for performance and scheduling analysis [76].

Dataflow approach: For streaming application design, the Synchronous Dataflow (SDF) graph model is one of the most popular models both in academic and industrial communities. Among the reasons for its popularity, the two most important ones are its expressiveness, i.e., the ability to capture system behaviors accurately, and analyzability, i.e., the ability to reason about system properties efficiently. A given streaming application can be converted to an SDF graph with moderate abstraction and modeling efforts. Task concurrency and data parallelism in parallel application specifications on

multi/many-core systems can be naturally captured by SDF graphs. A system is viewed as a set of *actors*, i.e. processing blocks, that communicate with each other through *channels*. Fixed amounts of data (called rates) are consumed and produced by each actor for every execution. With fixed data rates, we can define an *iteration* of a graph to contain a number of actor firings that returns the graph to its initial condition. An iteration typically corresponds to processing a meaningful unit of data in the physical world such as an image frame in an image processing application or an audio frame in music players. With the definition of an iteration, we can define the metric *throughput* as the number of iterations completed per time unit to measure the performance of streaming applications modeled by SDF graphs. The constraint on data rates of actors enables powerful analysis of SDF graphs. For single processor platforms, it is quite easy to construct a schedule with fixed execution orders and bounded buffer requirements [101]. For multi-processor platforms, state-based and max-plus based analysis techniques are developed for performance analysis and trade-off analysis [64, 144]. SDF graphs normally only model distributed and non-shared resources such as FIFOs. In order to capture resource sharing among actors, specific techniques are required. For instance, runtime scheduling of actors on shared resources whose starvation free arbiters (e.g. Round-Robin arbiter) can be modeled as $\mathcal{LR}$-servers. [11, 164, 165] model each task as a component that consists of two dataflow actors that capture latency and resource allocation rate based on the $\mathcal{LR}$ model of the corresponding resource arbiter.

Comparison of the three approaches: All approaches are able to model and analyze the streaming behavior of an embedded system while each has its own strengths and weaknesses. NC/RTC/MPA and EM/SymTA/S are trace-driven models, i.e., they need knowledge about the timing properties of inputs collected from traces to reason about the whole system. EM targets real-time scheduling and translates periodical event logs with jitters and deadlines into stream event functions such as $\eta(\Delta t)$ and $\delta(n)$. After conversion of the event stream to event functions, EM and NC/RTC/MPA can use the same analysis techniques since they both use arrival and service curves based on min-plus and max-plus algebras. Both NC/RTC/MPA and EM do performance analysis by local component analysis and global iterative fixed-point analysis. They are able to provide closed form analysis. Recently they also are able to analyze cyclic dependencies in applications [91]. In contrast to NC/RTC/MPA and EM, SDF is not dependent on trace information and can easily handle cyclic

dependencies. The topological structure of SDF is very close to the application itself. It does have to know the processing tasks to fix the data rates of each input and output. The constraint of fixed rates for input and output is a double-edged sword. On the one hand, it allows fast and accurate analysis. On the other hand, it lacks the flexibility of the arrival curves of NC/RTC/MPA and EM to handle dynamism. There is some recent research work that tries to extend classical SDF with the ability to handle dynamism [147, 163]. In NC/RTC/MPA, the resource availability is captured by service curves. A greedy processing component (GPC) takes the resource service curve as input and outputs a remaining resource service curve as output. The outputted remaining resource service curve then feeds into the next GPC that needs the same resource. This models priority based arbitration and the order of connection of GPCs decides the priority of tasks sharing the same resource. In dataflow modeling, the resource sharing is handled by static-order schedules fixed at design time [37] or models such as $\mathcal{LR}$-servers for starvation free runtime schedulers [164]. The static order can be modeled by additional edges in the dataflow graph while $\mathcal{LR}$-servers can be modeled as a dataflow component [11, 37, 140, 164, 165]. Static-order scheduling uses at design-time the known data dependencies within one application to generate more efficient schedules compared to run-time schedulers. The static-order schedulers can typically give tighter performance bounds than static analysis of dynamically scheduled behavior with $\mathcal{LR}$-server models. An embedded system often runs multiple streaming applications, the data dependencies only exist within an application itself. The resource sharing among multiple applications can only be scheduled at runtime. One solution is to do static-order scheduling within one application while using the $\mathcal{LR}$-server model to analyze multiple independent applications [110]. A goal of this thesis is developing more efficient static-order schedules for a single application in the case of shared resources.

Last but not least, we want to point out that some analysis techniques for the three models share the same mathematical structure of min-plus and max-plus algebra. The similarities between the analysis techniques hint that the MoCs can be unified in a general framework. There is already some work on interface theory that explores such possibilities [60, 138].

We use the SDF graph model as our tool for analyzing streaming application systems since it naturally represents streaming applications.

Moreover, we can analyze it both in the state-based approach or in the max-plus algebra approach. It gives us flexibility for exploring our trade-off analysis problem.

1.4.3 The problems faced in dataflow modeling

Our research questions are derived from the two aspects of SDF: expressiveness and analyzability. For expressiveness, we noticed that almost all existing work on SDF handle resources in the following two ways. In the first way, it makes implicit resource assumptions. For example it assumes that resources are unlimited so that they do not have to be taken into consideration. In the second way, it modifies dataflow graphs to embedded scheduling decisions on shared resources. For example, it assumes that resources are used in a fixed order so that the resources can be modeled as data tokens that are communicated between actors [37]. Or it uses dataflow components to replace one task actor to capture starvation-free schedulers in the $\mathcal{LR}$ server class [164]. If we change the underlying assumptions and make the amount of resources that we are interested in limited and the order in which they are used more flexible, can we still model it. For this, we need a new MoC to explicitly handle resource concerns. If we express our resource concerns in a new MoC, how should we analyze it? The extension of expressiveness will impact the analyzability of the new model and requires reconsidering existing techniques for the well-known SDF MoC.

For example, a common trade-off for streaming applications is throughput vs. buffer resources. The SDF model has efficient analysis algorithms for throughput and distributed buffer size [144, 148]. However, for more generic resources that are expressed by the new MoC, analysis methods for the trade-offs between throughput and resource usage have to be explored. Are there efficient algorithms for the analysis of trade-offs? In general, trade-off analysis is an important research question for such a new model.

Last but not least, if we add a new design dimension, input of a system, into the modeling process besides its resources dimension, how should we handle it? Can we utilize the knowledge on inputs to predict the performance and resource usage of streaming applications more precisely? For streaming applications, we can sometimes model the inputs and the transitions among inputs with a Markov Chain or Finite State Machine (FSM). For example, the MPEG-2 decoder has three *types* of different input, I, B, P frames. The

processing steps and execution times for these frames are different. The transition from input with one frame type to another follows some rules. In [116, 141], a Markov Chain is used to model the traffic in MPEG-2 Video. In [62, 151] performance prediction of streaming applications is improved with input models that are Markov Chains and FSMs. Is it possible for us to react to the inputs at runtime for more efficient resource usage or higher performance? For example, we can allocate the same resource to different tasks based on the context, or change operating voltage or frequency based on the predication of input. So the dynamic environment is another challenge for our research.

To summarize, the thesis focuses on the following three major questions:

1. How to extend the SDF model with a (shared) resource aspect for streaming applications on an embedded platform while still maintaining the possibility of efficient analysis techniques?
2. How to efficiently explore trade-offs in mapping of a streaming application on its platform with the new model?
3. How to model the environmental aspect (i.e., input) and find an efficient way to adapt to environmental changes at runtime?

The above research questions have motivated our work and our thesis to answer them. Through investigation of the above three questions, we made contributions, which we introduce in the next section, to the existing work.

1.5 Contributions

By addressing the questions posed in the previous section, we made the following contributions:

Resource-Aware Synchronous Dataflow (RASDF): We propose the Resource-Aware Synchronous Dataflow model as an extension to the well-known SDF model. It explicitly models the shared resource aspects of a streaming application. An SDF graph is consistent if its production and consumption rates are such that the SDF graph can execute indifferently without deadlock and within bounded memory. The concept of consistency of SDF is extended to include consistency of resource usage. The resource consistency of a given RASDF is necessary for the existence of a schedule for the RASDF without deadlock and with bounded resource requirements. The explicit modeling of resources without specifying their order of use enables a more flexible analysis. Since resources can be shared by multiple actors, the resource allocation order

will impact the execution order of actors of an application, i.e., the schedule of the application, and therefore impact the performance of the application. Better trade-offs are possible when compared to SDF analysis. This work has been published in [170].

Trade-off analysis techniques: We developed trade-off analysis techniques for RASDF. We use state-space exploration techniques, both in the time-domain [169, 170] and on an iteration-basis [171]. These techniques make use of the concept of Pareto dominance for pruning redundant explorations. The time-domain analysis allows better schedules due to interleaving of iterations, but the size of the explored state space grows quickly. The iteration-based technique trades the interleaving of iterations for a more compact size of the state space. The work has been published in [171].

Bottleneck-driven DSE: We developed automatic bottleneck-driven design space exploration techniques for efficient dimensioning of the required resources. By exploring the state space of an RASDF graph and analyzing the dependencies between actor firings, bottleneck resources that are limiting the throughput are identified through analysis of dependencies involving limited resources. With this bottleneck information, the design space can be more efficiently explored. This work is published in [169] and [171].

Game-theoretic controller synthesis: We exploit an analogy of the controller synthesis problem for a streaming application to a game between a controller player and the application environment player. The controller player decides the runtime schedule corresponding to the input decided by the environment player and context at the decision time. By mapping the synthesis problem to the well-known mean-payoff game, synthesizing a controller can be reduced to finding a winning strategy for the controller player. We consider the role of the environment into our modeling efforts and combine a parameterized RASDF with a specification of scenarios of input behavior, in a combined model called Scenario- and Resource-Aware SDF (SARA SDF). The resulting analysis approach draws from the studies of automata (such as SDF, RASDF, SARA SDF), max-plus algebra, and decision and game theory, which hints at new research directions for streaming application modeling research. The work is published in [172].

1.6 Thesis Overview

This thesis is organized as follows. Figure 1.3 shows the structure of the thesis. Chapter 2 introduces the new MoCs: RASDF and SARA SDF that are used throughout the thesis with their related parent models. Chapter 3 introduces the metrics that we use for measuring the performance and resource usage of streaming applications, the concept of Pareto optimality and dominance, and the trade-offs that we are interested in. Chapter 4 develops the analysis techniques for RASDF in the time domain. Pruning techniques and heuristics are discussed for avoiding an explosion of the state space. It also develops a bottleneck-driven design space exploration technique for efficient exploration of the design space. Chapter 5 explores the design space through an alternative view, i.e., on an iteration basis. Techniques for handling resources and pruning are introduced. Chapter 6 studies the controller synthesis problem for streaming systems from a game-theoretic view. Chapter 7 concludes the thesis and points out directions for future work. The appendix includes some dataflow graphs used in this thesis.

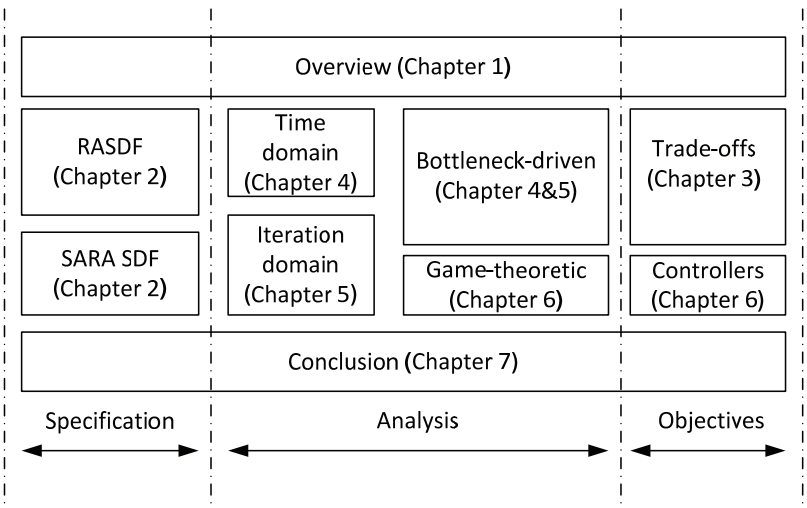


Figure 1.3 Overview of the structure of the thesis

2 DATAFLOW MODELS

"The whole of science is nothing more than a refinement of everyday thinking"

- Albert Einstein

In this chapter, two new MoCs, i.e., RASDF and SARA SDF are discussed. In Section. 2.1 we give an overview of the relations between these two MoCs and other existing dataflow models. From Section 2.2 to Section 2.5, we introduced SDF, SADF, RASDF and SARA SDF respectively. In Section 2.6, we discuss related work. Section 2.7 concludes with a summary of the chapter.

2.1 Overview

Models of Computation (MoCs) [49, 89, 136] are tools that we use to study aspects of the physical objects that perform computations, such as the functionality, the communication, synchronization, and timing behavior of the computations. MoCs are formal abstractions of the computation objects that get rid of all non-relevant characteristics and only keep essential ones for a given purpose like performance predication or property verification. The formal description of a MoC allows us to rigorously study the computation performed by the physical objects.

Compared to general models of computation such as Turing Machines that are used to study computability and complexity, some specialized models of computation are used for specific purposes. *Finite state machines* (FSMs), for instance, are used for circuit synthesis, *synchronous languages* for designing safety-critical controller systems, *timed automata* for verification of real-time systems, or *Petri nets* for description and analysis of distributed systems. The specializations of MoCs often allow efficient analysis of problems in specific target domains.

Streaming applications are widely used in our daily life. And there are many open problems for design and implementation of such applications. In order to satisfy the performance requirements and resource constraints of streaming applications on embedded systems, developers have to exploit different levels of parallelism and engage in concurrent programming (both software and hardware) for implementation while keeping resource usage within budget. However, traditional MoCs can neither capture such concurrent

activities easily (e.g. Turning Machines or FSMs), nor concisely capture the activities for efficient analysis (e.g. Petri nets). This leads to difficulties in analysis and implementation. New MoCs are needed to solve these problems for streaming applications.

Researchers develop different types of MoCs [89, 142, 174] to study the problems faced by streaming applications and to investigate how to analyze them efficiently. One important category of MoCs are **dataflow** models of computation. Depending on the expressiveness and analyzability, different variants of dataflow models are proposed [147]. By using different types of dataflow models, we are able to analyze the performance (how much time does it take?) and the resource usage (how much processing, storage, communication resources does it use?) of streaming applications and we can explore the options to implement an application efficiently.

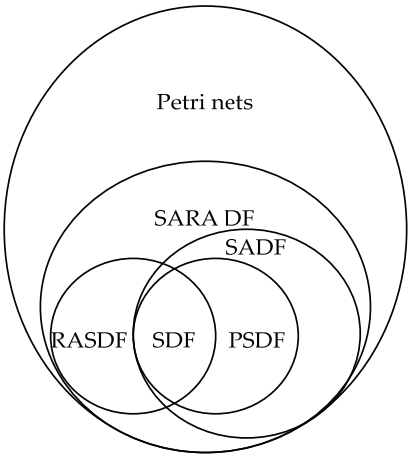


Figure 2.1 Venn diagram of dataflow models of computation

Figure 2.1 shows an overview of the dataflow models that will be discussed in detail in this chapter (SDF, PSDF, SADF, RASDF and SARA SDF) and their relations with each other. All these models can be viewed as subclasses of the Petri nets MoC [118, 119]. Synchronous Dataflow (SDF) [101] is the basic model that allows to capture static, multi-rate signal processing systems. PSDF [13, 63] allows data-dependent, dynamic digital signal processing (DSP) systems to be modeled with parametric rates and execution times. Scenario-Aware Dataflow (SADF) [59, 151] uses an FSM to capture dynamic changes among different inputs and allows to analyze performance for data-dependent applications.

Resource-Aware Synchronous Dataflow (RASDF) (developed in this thesis) specifies dataflow systems with explicit resource-awareness and models shared resources to allow more flexible scheduling. Scenario- and Resource-Aware Synchronous Dataflow (SARA SDF) (developed in this thesis) combines the resource aspect of RASDF and the data-dependent aspect of SADF together to study the interplay between an application and its environment. The thesis is developed around the last two models, RASDF and SARA SDF, and their analysis techniques.

2.2 Synchronous Dataflow Graphs

The behavior of certain streaming applications can be captured and analyzed by Synchronous Dataflow (SDF) graphs. SDF graphs are rooted in Computation Graphs (CG) [95] and it is equivalent to a subclass of Petri nets, so-called Weighted Marked Graphs (WMG) [31, 150]. The reason of the popularity of SDF is its specific combination of expressiveness and analyzability. It is very easy to capture certain static streaming applications with SDF graph, to decide whether a schedule with desired performance exists or not and to determine the size of FIFO buffers to avoid deadlocks.

Figure 2.2 shows an example of an **untimed** SDF graph. The circular nodes are *actors* and represent computations. Actor computations are atomic and performed repeatedly. Actors transfer information to each other through FIFO *channels* (solid directed edges) via data items called *tokens* (black dots). Each actor *firing* represents one computation and consumes a fixed number of input tokens from each connected input channel and produces a fixed number of output tokens to each connected output channel. These numbers of consumed and produced tokens are called *rates*. The fixed rate is an essential property of SDF graph. When the rates of all actors of an SDF graph are equal, we call it a homogeneous SDF (HSDF) graph. It is a specialization of normal SDF graph.

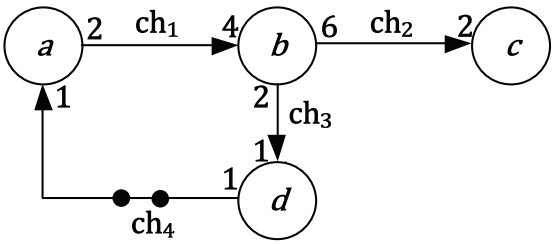


Figure 2.2 An example of a Synchronous Dataflow (SDF) graph

SDF graph is very important since many dataflow models are derived from it. There are many extensions to SDF graph, such as CSDF [117], BDF [24], VRDF [162], SADF [62, 151], to capture more complex behavior of some streaming applications. Our RASDF and SARA SDF are extensions of the basic SDF model with different trade-offs of expressiveness and analyzability.

We can now give a formal definition of untimed SDF graph that is similar to the definition in [65, 145].

Definition 2.1: Untimed Synchronous Dataflow Graph (Untimed SDF)

An untimed SDF graph is a tuple (A, C, wr, rd) *that consists of a set* A *of actors, a set* $C \subseteq A \times A$ *of directed channels, a read function* $rd: C \mapsto \mathbb{N}^+$ *that annotates (the sink of) each channel with a positive integer that denotes the rate of consuming input from the channel, and a write function* $wr: C \mapsto \mathbb{N}^+$ *that annotates (the source of) each channel with a positive integer that denotes the rate of producing output to the channel.*

Note that we do not allow multiple channels between the same pair of actors since the channels can be replaced by one equivalent channel of which input (output) rates are equal to the sum of input (output) rates attached to multiple channels.

We use a *channel quantity* to capture the numbers of tokens on each channel of an SDF graph.

Definition 2.2: Channel Quantity

A channel quantity $\delta \in \mathbb{N}^{|C|}$ *associates with each channel* $c \in C$ *an amount of tokens.*

For a weighted and directed graph, we can use a *topology matrix* [101] to represent its topology or structure, in which matrix row entries correspond to channels and column entries correspond to the actors. The topology matrix shows the input and output relations among actors and channels. We can represent the input and output relations in two matrices Γ_{rd} and Γ_{wr} respectively, and derive the topology matrix from the two matrices. We use a function $src: C \mapsto A$, to denote the source actor of a channel $c \in C$, and a function $sink: C \mapsto A$, to denote the sink actor of a channel $c \in C$. The elements of the two matrices are defined in the following two equations.

$$\Gamma_{wr}(c, a) = \begin{cases} wr(c) & if\ a = src(c) \\ 0 & otherwise \end{cases}$$

and

$$\Gamma_{rd}(c,a) = \begin{cases} rd(c) & if\ a = sink(c) \\ 0 & otherwise \end{cases}$$

Then the topology matrix Γ is defined as:

$$\Gamma = \Gamma_{wr} - \Gamma_{rd}$$

So the elements of the topology matrix of SDF graph are given by the following equations:

$$\Gamma(c,a) = \begin{cases} wr(c) & if\ a = src(c)\ \ and\ src(c) \neq sink(c) \\ -rd(c) & if\ a = sink(c)\ and\ src(c) \neq sink(c) \\ wr(c) - rd(c) & if\ src(c) = sink(c) \\ 0 & otherwise \end{cases}$$

When a channel $c \in C$ of an SDF graph has the same actor $a \in A$ as both its source actor and its sink actor, i.e., $a = src(c) = sink(c)$, then the channel c is called a *self-edge* of the actor a.

Consider the example of Figure 2.2. Let the columns of the matrix correspond to the actors a, b, c, d and the rows of the matrix to the channels ch_1, ch_2, ch_3, ch_4 in ; then Γ_{wr} and Γ_{rd} are given below:

$$\Gamma_{wr} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \Gamma_{rd} = \begin{bmatrix} 0 & 4 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

For example, the element in the first row of Γ_{wr} (that corresponds to channel ch_1) and the first column of Γ_{wr} (that corresponds to actor a) is 2. It denotes that actor a writes 2 tokens to the channel ch_1 after each firing.

The topology matrix of the example SDF graph is as follows:

$$\Gamma = \Gamma_{wr} - \Gamma_{rd} = \begin{bmatrix} 2 & -4 & 0 & 0 \\ 0 & 6 & -2 & 0 \\ 0 & 2 & 0 & -1 \\ -1 & 0 & 0 & 1 \end{bmatrix}$$

Given the topology matrix, we can define many useful properties and make use of these properties in our analysis. More importantly, the extensions of SDF discussed in this thesis can be derived from different variations of the topology matrix as we will see in the following discussions.

When an actor in an SDF graph fires, it consumes tokens from its input channels and produces tokens to its output channels. If a channel c is a self-edge of an actor a in an SDF graph, $wr(c)$ has to be the same as $rd(c)$. Otherwise, the number of tokens in the self-edge channel will during system execution eventually be either 0 (leading to deadlock in the system) or grow towards infinity (which implicates that a buffer overflow will eventually happen in the system). More generally, we can check whether a system modeled by an SDF graph is *consistent* or not by solving the **balance** equation:

$$\Gamma q = 0$$

The equation is called the balance equation since it means that after a certain number of actor firings, given by q, the number of tokens that are read from each channel equals the number of tokens that are written to each channel. Consistency is a necessary condition to avoid deadlocks and buffer overflows.

If a non-trivial integer solution vector q exists, i.e., the elements in q are positive and coprime, the SDF graph is said to be *consistent* and the vector q is defined to be the *repetition vector*. Note that the consistency of an SDF graph does not imply it is deadlock-free. Deadlock can still happen if the number of initial tokens is not enough on some channels. The existence of repetition vector is only a necessary condition for deadlock-free. The number of initial tokens needs to be sufficient to avoid deadlock. The repetition vector of the example SDF graph is $q = [2 \quad 1 \quad 3 \quad 2]^T$. It corresponds to 2 firings of a, 1 firing of b, 3 firings of c and 2 firings of d. With enough initial tokens, the application will return to the initial situation after 2 firings of a, 1 firing of b, 3 firings of c and 2 firings of d. For a given deadlock-free SDF graph, after a number of firings that corresponds to its repetition vector, the graph returns to its initial situation. This allows us to construct a periodic schedule that can repeat infinitely often.

An execution of an SDF graph is defined as a sequence of actor firings. A sequence of actor firings is called an *iteration* if it contains a number of firings of each actor that equals the corresponding element in the repetition vector of the SDF graph.

We can convert an SDF graph to a homogeneous SDF (HSDF) graph with conversion algorithms from [58, 101, 142]. The HSDF graph preserves actor firings (under a straight forward homomorphism). The advantage of HSDF graph is its regular structure, i.e., all read and write rates are equal. The length

of an iteration, i.e., the total number of firings in an iteration equals the number of actors in the HSDF graph. The transformation of an SDF graph to an HSDF graph is sometimes needed for analysis purpose. Figure 2.3 shows the homogeneous SDF (HSDF) graph of the example SDF graph.

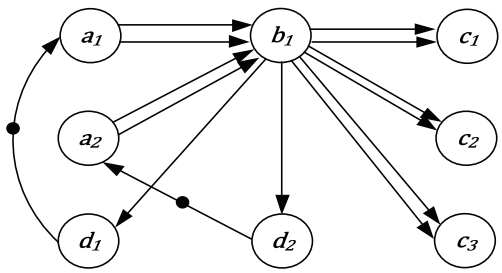


Figure 2.3 HSDF graph of the example in Figure 2.2

Many performance properties of SDF graphs, such as throughput [63, 64], latency, and order-sensitive resource usage, cannot be inferred without timing information. Throughput itself is defined with time while exact resource usage is determined by the order of start and end of actor firings, which depends on time. (A detailed discussion on these metrics follows in Chapter 3.) So actors are annotated with execution times for the purpose of analysis. The execution time of an actor is defined as the constant duration between the start of actor firing and the end of actor firing and is annotated inside the circle that denotes the actor. With the execution times of actors, we can analyze the timing of an execution of SDF.

Definition 2.3: Timed Synchronous Dataflow Graph (Timed SDF)

A timed SDF is a tuple (A, C, wr, rd, τ) *that consists of an untimed SDF* (A, C, wr, rd) *and an execution time function* $\tau: A \mapsto \mathbb{R}^+$ *that assigns to every actor* a *non-negative real number that denotes its execution time.*

Figure 2.4 shows an example of timed SDF graph. For example, the execution time of actor b is 2 time units. Figure 2.5 shows the behavior of one firing of actor b. Before time point t_1, actor b is waiting for input tokens on channel ch_1 (grey area). Once the number of tokens on ch_1 is sufficient for one firing at t_1, the actor b starts its firing. After 2 time units of execution, actor b writes 6 and 2 output tokens to channels ch_2 and ch_3 respectively. The performance of a timed SDF graph, i.e., its throughput, can be represented by the average number of actor firings during a fixed period of time.

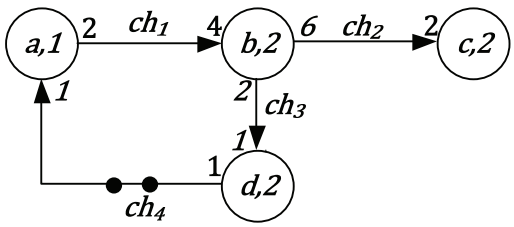


Figure 2.4 An example timed SDF graph

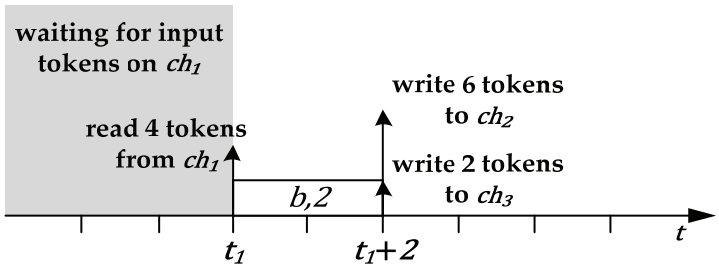


Figure 2.5 Execution behavior of one firing of *b*

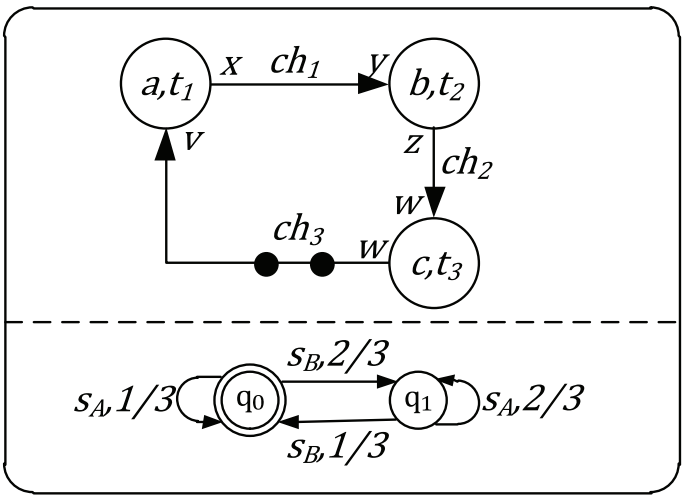
Performance metrics of SDF graph and other dataflow models are discussed in Chapter 3. Operational semantics of SDF and its extensions are discussed in the time domain (Chapter 4), the iteration domain (Chapter 5) and for input sensitive situations (Chapter 6).

*2.3 S*cenario-aware Dataflow Graphs

In the SDF model, the execution times of actors are usually based on worst-case assumptions, i.e., they are so-called worst-case execution times (WCETs). This assumption may lead to very conservative performance estimation. Many streaming applications show data-dependent behavior, meaning that actor execution times may vary with the data being processed. The concept of a scenario [66] is derived from the observation that the behavior of streaming applications is stable while processing the same type of data units, but vary among different types of data units. The type of data may decide the execution paths of programs and results in different execution times on different paths. By clustering and classifying those data-dependent behaviors into different scenarios, we can have a timed SDF for each scenario.

Scenario-aware SDF (SADF) [59, 151] extends SDF with a *scenario* FSM that captures input/environment changes and provides worst-case execution times

for different input types. It enables us to improve the quality of performance and resource usage analysis by refining the analysis for data-dependent behavior. Figure 2.6 shows an example of an SADF. It is composed by three parts: a parametric SDF (as shown in the top part of Figure 2.6), a scenario FSM (as shown in the bottom part of Figure 2.6), and a parameter table of the parametric SDF for different scenario state.



Scenarios	Rates					Execution times			Status
	x	y	z	w	v	t_1	t_2	t_3	c
s_A	3	2	0	0	0	2	3	0	disabled
s_B	2	1	1	1	2	1	2	1	enabled

Figure 2.6 An example SADF graph

A scenario FSM captures all scenarios and transitions among scenarios in a streaming application by a Finite State Machine. Figure 2.7 shows an example scenario FSM. It has two states (q_0 and q_1) and two scenarios (A and B). q_0 is the initial state of the scenario FSM. The directed edges between states are possible scenario transitions and each transition edge is labeled with the probability of the transition and its corresponding scenario. For example, from the initial state q_0, the system has probability 2/3 to receive input B and to transit to state q_1 with scenario S_B; it has probability 1/3 to receive input A and stay at the same state q_0 with scenario S_A.

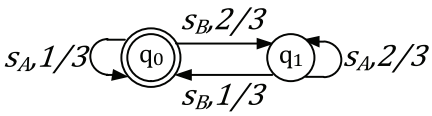


Figure 2.7 An example scenario FSM

Definition 2.4: Scenario FSM

A scenario FSM is a tuple $S_{FSM} = (Q, q_0, \delta, \Sigma)$ *that consists of a regular FSM, i.e., finite set* Q *of states, an initial state* $q_0 \in Q$*, and a transition relation* $\delta \in Q \times Q$ *together with a scenario labeling* $\Sigma: \delta \mapsto [0,1] \times S_T$ *where* $[0,1]$ *is the set of probabilities of the transitions in* δ *and* S_T *is a set of scenarios with its subscript* T *being a set of input types. Each scenario corresponds to a specific timed SDF graph.*

Note that a scenario FSM can also be viewed as a finite state Markov chain plus scenario labeling. In order to capture all timed SDF graphs in one model, we use a single SDF graph template to capture timed SDF graph instances of all possible scenarios; the SDF graph template is called a parametric SDF graph. Our definition of a parametric timed SDF graph is a combination of the definitions in [13, 63] by parameterizing both execution times and rates.

Definition 2.5: Parametric Synchronous Dataflow Graph (PSDF)

A parametric timed SDF graph is a tuple (A, C, TF) *that consists of a finite set* A *of actors, a finite set* C *of channels, and a template function* $TF: S_T \mapsto \mathbb{R}^{|A|} \times \mathbb{R}^{2|C|} \times \{enabled, disabled\}^{|A|}$ *that maps each scenario state in the scenario FSM to a set of SDF parameters, in which* $\mathbb{R}^{|A|}$ *contains the execution times of all actors, the element from* $\mathbb{R}^{2|C|}$ *contains the read/write rates for all channels, and the element from* $\{enabled, disabled\}^{|A|}$ *contains the active status of actors in the given scenario .*

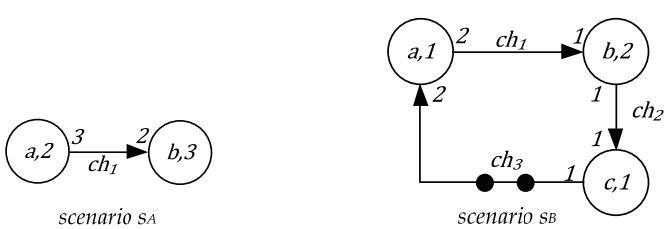


Figure 2.8 Timed SDF graphs for two scenarios s_A and s_B

Figure 2.8 shows two timed SDF graphs for two different scenarios s_A and s_B. Figure 2.8 (a) is a timed SDF graph for input A while Figure 2.8 (b) is a timed SDF graph for input B.

We use a parametric SDF graph to capture the two graphs, the template is shown in Figure 2.9. The template function TF of the parametric SDF graph is given in the parameter table of Figure 2.6. For simplicity, we only show parameters that vary. Using the parameter table, we can instantiate a timed SDF graph from a parametric SDF graph for a given scenario. For example, for scenario s_A, we disable actor c and remove actor c and all channels linked to it. Then we replace the parameters in Figure 2.9 with its values in the table.

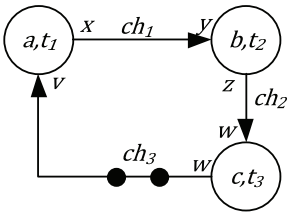


Figure 2.9 Parametric SDF graph for scenarios s_A and s_B

Analogous to normal SDF graph, parametric SDF graph also has parameterized output rate and input rate matrices Γ_{wr} and Γ_{rd}. For the example in Figure 2.9, we have

$$\Gamma_{\mathrm{wr}} = \begin{bmatrix} x & 0 & 0 \\ 0 & z & 0 \\ 0 & 0 & w \end{bmatrix} \text{ and } \Gamma_{\mathrm{rd}} = \begin{bmatrix} 0 & y & 0 \\ 0 & 0 & w \\ v & 0 & 0 \end{bmatrix}$$

From the two matrices, we can derive the parameterized topology matrix:

$$\Gamma = \begin{bmatrix} x & -y & 0 \\ 0 & z & -w \\ -v & 0 & w \end{bmatrix}$$

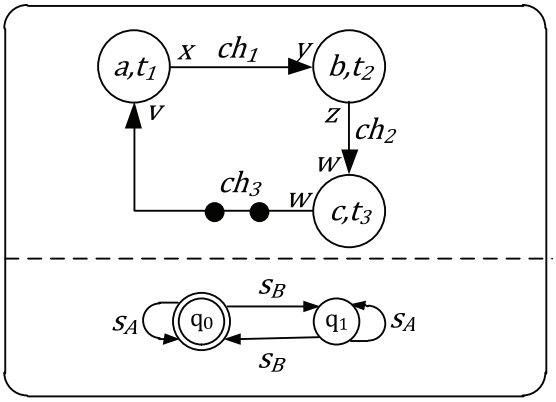
We define a topology matrix function $\Gamma_F: S_T \mapsto S_\Gamma$ that maps each possible scenario to a topology matrix, in which S_Γ is the set of topology matrices for all timed SDF graph instances. For the example in Figure 2.8, we have

$$\Gamma_F(s_A) = \begin{bmatrix} 3 & -2 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \text{ and } \Gamma_F(s_B) = \begin{bmatrix} 2 & -1 & 0 \\ 0 & 1 & -1 \\ -2 & 0 & 1 \end{bmatrix}$$

Obviously, the topology matrix of each scenario is an instantiation of the parameterized topology matrix of the parametric SDF graph. A parametric SDF graph is consistent if and only if all its instances that are instantiated from the scenarios in its template function table are consistent, i.e., every topology matrix has a non-trivial repetition vector. For instance, the example parametric SDF graph has repetition vector $q_{s_A} = [2 \quad 3]^T$ for scenario s_A and repetition vector $q_{s_B} = [1 \quad 2 \quad 2]^T$ for scenario s_B. The parametric SDF graph can capture the behavior for each scenario. However, it does not capture the dynamism introduced by input changes, i.e. the transition between different scenarios. By integrating scenario FSM, SADF graph can capture such dynamism.

Definition 2.6: Scenario-Aware Synchronous Dataflow Graph

A Scenario-Aware SDF (SADF) graph is a tuple (G_{PSDF}, S_{FSM}) *that includes a parametric SDF graph* G_{PSDF} *and a scenario FSM* S_{FSM}.



Scenarios	Rates					Execution times			Status
	x	y	z	w	v	t_1	t_2	t_3	c
s_A	3	2	0	0	0	2	3	0	disabled
s_B	2	1	1	1	2	1	2	1	enabled

Figure 2.10 An example FSM-based SADF graph

While the probability labels of transitions of the scenario FSM are relevant for long-run average performance analysis [62, 151]. For worst-case throughput analysis, we can simplify the scenario FSM by ignoring the probabilities, i.e. the *scenario labeling* $\Sigma: \delta \mapsto S_T$ only maps transitions to scenarios. This specialization

is called *FSM*-SADF graph [59, 62, 147]. Figure. 2.10 shows an FSM-based SADF graph simplified from the example in Figure. 2.6.

FSM-SADF graph is very close to the HDF model that is proposed in [67] with added execution times. This addition comes with efficient algorithms [59, 62] for analyzing the worst-case performance of an FSM-SADF graph.

2.4 Resource-Aware Synchronous Dataflow Graphs

One reason that SDF is popular is its efficient analysis algorithms such as performance analysis [63, 64] and buffer sizing [148]. However, the possibilities for resource modeling are limited and implicit. Only recently, there is work that encodes resource scheduling in SDF [37, 164]. In [11, 164, 165], a dataflow component that consists of two actors is used to model resources scheduled by $\mathcal{LR}$ servers. In [37], it only targets a specific type of static order schedule: Periodic Static-Order Schedules (PSOS).

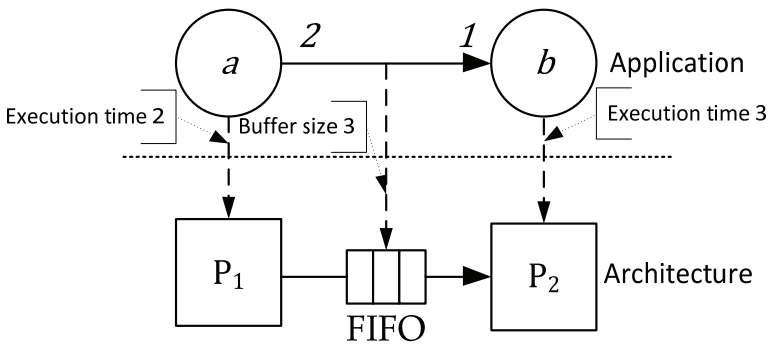


Figure 2.11 An example system with fixed mapping

We need to extend SDF to handle different ways of using resources. For example, Figure 2.11 shows a system with an application that consists of two actors a, b and an architecture that consists of two processors P_1, P_2 and one 3-token-sized communication FIFO. In the application, a is the producer and b is the consumer. Actor a claims two tokens space at the start of firing and produces two tokens at the end of firing while the consumer b consumes one token each firing and releases the one token space at the end of firing. The communication buffer is a FIFO buffer that can store 3 tokens. The mapping assigns actor a to processor P_1, actor b to processor P_2 and the communication

edge to the FIFO. The execution times of actors and the buffer size constraint on communication channels are annotated with the mapping edges.

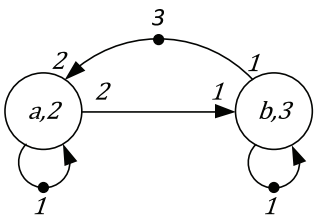
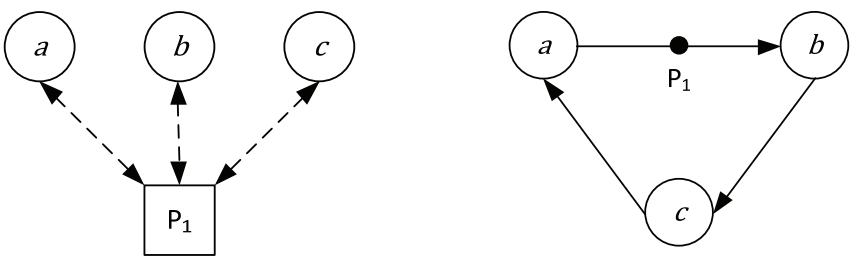


Figure 2.12 Timed SDF model of the example in Figure 2.11

Figure 2.12 shows the timed SDF for the example system in which all resources in the system are modeled implicitly. The processors are modeled as self-edges of actors with one initial token (enforcing that multiple firings of the same actor cannot execute in parallel, i.e., auto-concurrency is one). The size constraint of 3 on the FIFO is modeled as an edge in the opposite direction with three initial tokens. Thus, all resources are modeled with extra channels, where the amount of resources is captured by the number of initial tokens in these channels.

However, when it comes to modeling resources shared among many actors, SDF graphs lack expressiveness for such a situation. For example, the way to model a processor shared by three actors *a*, *b* and *c* of a system like that in Figure 2.13 (a), is depicted in Figure 2.13 (b). The processor is modeled as a resource token that is handed over among 3 actors in a *fixed order* (i.e., first a, then b, and then c) through a ring that is composed of three channels. The disadvantage is that you have to specify and fix the order explicitly.

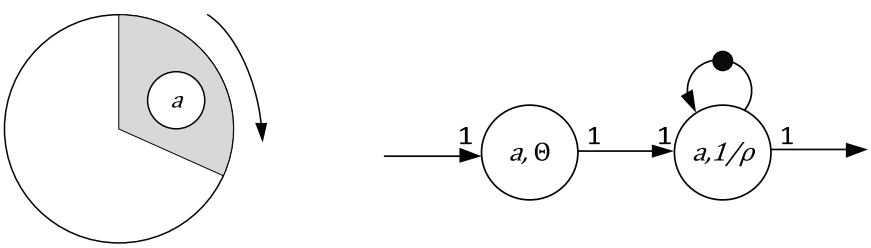


(a) 3 actors share 1 processor (b) Corresponding SDF graph

Figure 2.13 Modeling resource sharing with SDF

This way of modeling encodes fixed firing orders of actors into the SDF graph by introducing resource dependencies through channels. Thus it is limited for expressing flexibilities such as a change of firing order of actors across different iterations. Moreover, when an actor has multiple firings in one iteration, then the resource scheduling has to be specified explicitly at the HSDF graph level by complex resource token routing among those actors. For the given examples, we can see the weakness of SDF graphs for modeling resources: it over-specifies the way that resources are used, i.e., resources can only be handed over between two actors and the order to use resources is fixed. If we want to model some resources beyond FIFOs between two actors, it is almost impossible to use SDF graph to capture the system.

Another way to model the resource sharing in Figure 2.13 (a), is to assume that processor P_1 is scheduled by a starvation free scheduler modeled as a $\mathcal{LR}$-server, e.g., a time-division multiplexing (TDM) scheduler. Figure 2.14 shows the time slice allocation of actor a in a time wheel and its corresponding $\mathcal{LR}$-server dataflow model that consists of two actors [164]. Note that, a more tight dataflow model consisting of more actors can also be used instead of $\mathcal{LR}$-server, such as the model introduced in [140] to model a bi-rate $\mathcal{LR}$-server.



(a) Time wheel for TDM (b) Latency-rate server dataflow model

Figure 2.14 Latency-rate server dataflow model for TDM schedule [164]

The latency Θ and resource allocation rate ρ in the dataflow model depends on the type of scheduler. In Figure 2.14, we assume the period of the time wheel is P, the size of the time slice allocated to actor a is S_a, and the worst-case execution time of actor a is $WCET(a) = \tau_a$. From [164], we know that $\Theta = (P - S_a) \cdot (\lceil \frac{\tau_a}{S_a} \rceil - \frac{\tau_a}{S_a})$ and $1/\rho = P \cdot \tau_a / S_a$ gives a conservative model. By using the $\mathcal{LR}$-server dataflow model, the performance of applications that use starvation free schedulers can be approximated. It cannot handle schedulers

that do not belong to the $\mathcal{LR}$-server class (e.g. schedulers without guaranteed resource budget). The data dependencies of application that we know at design time can be used for generating static-order schedule with better performance.

In order to analyze complex resource scheduling in streaming applications, we have to overcome the above limitations and improve the expressiveness by extending the model. The goal is to generate efficient static-order schedule for a single application with shared resources at design time. The resource sharing between multiple independent applications at run-time can be done by $\mathcal{LR}$-servers. The two methods together, can provide a solution to design streaming applications with shared resources.

The Resource-Aware SDF (RASDF) model is proposed for solving the problem. First, we have to separate design concerns, i.e., we have to model resources explicitly and divide the whole system specification into three aspects: *application*, *architecture*, and *mapping*. By mapping the application to the architecture, we obtain the system specification. It follows the Y-Chart methodology proposed in [8, 98].

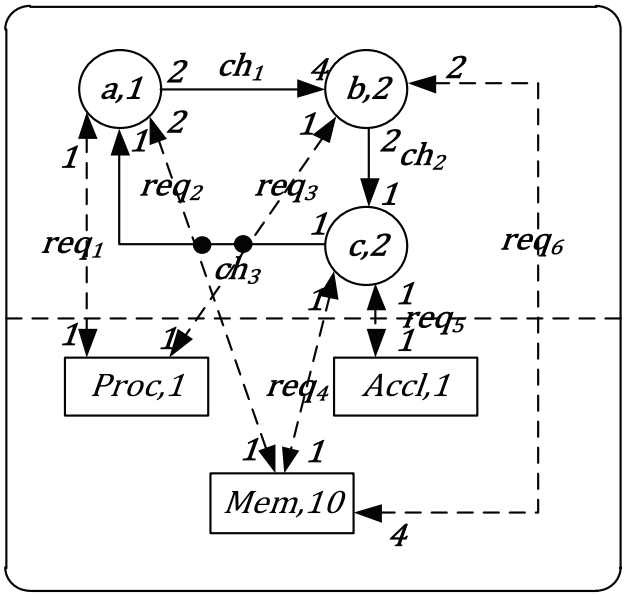


Figure 2.15 An example resource-aware SDF graph

Figure 2.15 shows an example RASDF graph. Real-life resources are abstracted as a set of integers that represent the amount of resources and are

denoted in the graph as a set of rectangles that are annotated with their corresponding resource names and resource amounts. The amounts of resources that are claimed or released by users (actors) are viewed as *requests* and are denoted by dashed bi-directional edges between actors and resources. For example, there is a memory resource with quantity 10 and actor b claims 2 units of memory tokens when it starts firing and releases 4 units (2 units claimed by actor a and 2 units claimed by actor b at their start) when it ends firing. There are two important assumptions here: first, actors always claim resources when they start and release resources when they end; second, the same resources can be claimed and released by different actors.

Definition 2.7: Resource-Aware SDF

A Resource-Aware SDF (RASDF) graph is a tuple $G_{RASDF} = (G_{SDF}, R, q, \tau, req)$ *that consists of three aspects of a system, i.e., the application graph* G_{SDF}*, the architecture* (R, q) *and the mapping* (τ, req)*. In* (R, q)*,* R *is a set of resources and* $q: R \mapsto \mathbb{N}$ *is the resource function that specifies the initial amount for each resource. In the mapping tuple* (τ, req)*,* $\tau: A \mapsto \mathbb{N}$ *is the execution time function, the same as in timed SDF, and* $req: A \times R \mapsto \mathbb{N}^2$ *is a mapping that maps each actor-resource pair to a pair of integers* (n, m) *that denotes the resource claim,* n*, at the start of actor firing and the resource release,* m*, at the end of actor firing. We define two functions* $clm: A \times R \mapsto \mathbb{N}$ *and* $rel: A \times R \mapsto \mathbb{N}$ *to map each actor-resource pair, i.e., a request edge, to its corresponding claim and release amounts respectively.*

An RASDF graph combines application (SDF graph), architecture (resources) and mapping (requests) aspects of a system into one graph.

Similarly to the channel quantity we defined for SDF graph, we can define resource quantity to represent the used resource tokens for an RASDF graph.

Definition 2.8: Resource Quantity

A resource quantity $\eta \in \mathbb{N}^{|R|}$ *that associates with each resource* $r \in R$ *an amount of resource tokens.*

Similarly to the two token *topology matrices* Γ_{wr} and Γ_{rd} in SDF graph, we can define the resource claim and release topology matrices Γ_{clm} and Γ_{rel}, which have a row for each resource and a column for each actor.

$$\Gamma_{clm}(a, r) = \begin{cases} clm(a, r) & if\ (a, r)\ is\ a\ request\ edge \\ 0 & otherwise \end{cases}$$

$$\Gamma_{rel}(a,r) = \begin{cases} rel(a,r) & if\ (a,r)\ is\ a\ request\ edge \\ 0 & otherwise \end{cases}$$

Note that the definition of the clm and rel functions is different from the definition of the rd and wr functions in SDF since resources can be shared by multiple actors while the channels only have one reader and one writer. For example, if we assume that resource $Proc$ is the first row, resource $Accl$ is the second row, and resource Mem is the last row, while actor a corresponds to the first column, actor b corresponds to the second column and actor c to the last column. Then the Γ_{clm} and Γ_{rel} for the example SDF is given as:

$$\Gamma_{clm} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 2 & 2 & 1 \end{bmatrix} \text{ and } \Gamma_{rel} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 4 & 1 \end{bmatrix}$$

The element in the first row and the first column of matrix Γ_{clm} is 1, which denotes that actor a (the first column) claims 1 unit from resource $Proc$ (the first row). Similarly, the element in the third row and the second column is 4, which denotes that actor b (the second column) releases 4 units from resource Mem (the third row).

The mapping topology matrix defines the net effect of an actor firing on the resource quantities. It is defined as $\Gamma_{mapping} = \Gamma_{rel} - \Gamma_{clm}$. So the mapping topology matrix of the example is

$$\Gamma_{mapping} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 2 & 0 \end{bmatrix}$$

Note that the mapping topology matrix is underdetermined since $rank(\Gamma_{mapping}) = 1 < 3$, which means that there exist two *linearly independent* possible firing vectors q that satisfy $\Gamma_{mapping} q = 0$. However, when we take the application topology matrix of the example RASDF, i.e., the topology matrix of the SDF graph part into consideration, the number of linearly independent firing vectors is one. In the example RASDF, the application topology matrix is

$$\Gamma_{app} = \begin{bmatrix} 2 & -4 & 0 \\ 0 & 2 & -1 \\ -1 & 0 & 1 \end{bmatrix}$$

We define a combined topology matrix of the whole system specification of the example as follows:

$$\Gamma_{sys} = \begin{bmatrix} \Gamma_{app} \\ \Gamma_{mapping} \end{bmatrix} = \left[\begin{array}{ccc} 2 & -4 & 0 \\ 0 & 2 & -1 \\ -1 & 0 & 1 \\ \hline 0 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 2 & 0 \end{array}\right]$$

Now we can define resource consistency similarly to data consistency in an SDF graph, i.e., the amount of available resources should remain the same after one iteration of actor firings. So if there exists a non-trivial repetition vector q such that it solves the balance equation $\Gamma_{sys}q = 0$, then we say the RASDF graph is consistent. Resource consistency is necessary to construct a periodic schedule that is deadlock free and has bounded resource usage. Inconsistent RASDFs will run out of resources eventually or it is impossible to realize because more resources are released than claimed.

Consistency is a very important property of RASDF graph. Many analysis algorithms need to check consistency upfront. In order to check it efficiently, we first check the consistency for the SDF graph part of the RASDF graph, then verify the resource consistency; since the SDF part of a consistent RASDF graph has to be consistent. The consistency of an SDF graph can be efficiently checked by using recursion [100]. If the number of linearly independent firing vectors of the topology matrix of the SDF graph part is more than one (i.e., Γ_{app} is underdetermined, which implies that it has multiple strongly connected component), we can then take the mapping topology matrix $\Gamma_{mapping}$ into consideration to further reduce the number of linearly independent firing vectors that solve the balance equation. If there does not exist such a firing vector, then the given RASDF is not consistent. The resource dependencies between the strongly connected components may then make the Γ_{sys} determined. If there is exactly one solution, then the SDF graph is data consistent and we can simply test the resulting vector to see the graph is also resource consistent.

Figure 2.16 shows the interpretation of RASDF graph in terms of the Y-chart methodology. The application is modeled as an SDF graph. The architecture that the application is running on is viewed as a provider of a set of resources. Through mapping, the execution times and resource requirements of actors are specified. The whole system is specified as an RASDF graph that includes all necessary information for analysis. Then we can analyze the metrics of the specification (introduced in Chapter 3) with methods that are introduced in

Chapter 4 and Chapter 5. Based on the analysis results, we can change application, architecture, mapping or all of them to meet the design requirements.

The argument here is to extend traditional SDF with explicit resource modeling to enable versatile analysis possibilities. The Y-chart methodology shows we can feed back the analysis results for tuning different aspects of the system. By modeling resources explicitly, we can analyze the resource usage (cost of streaming applications) in a more flexible way.

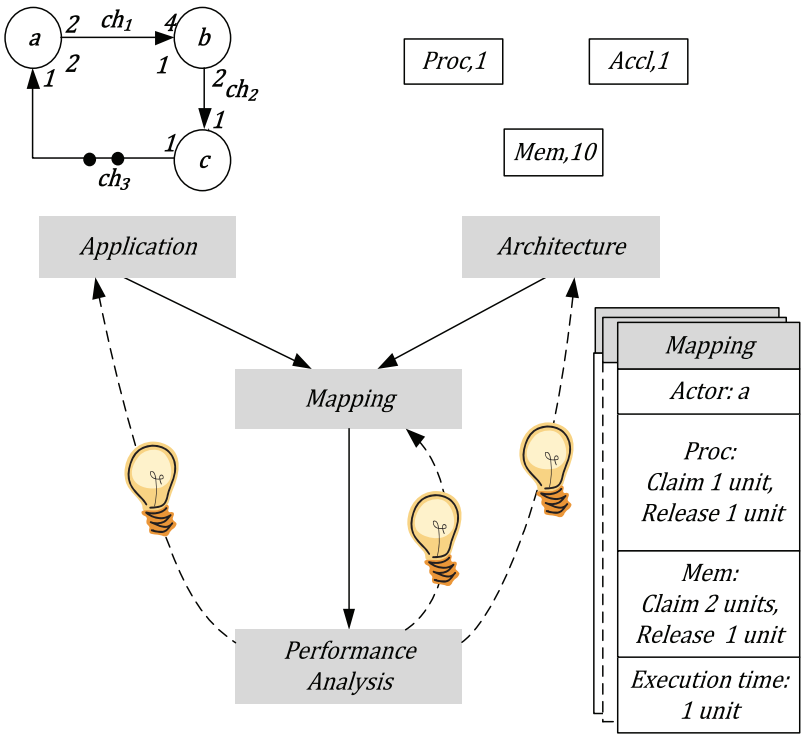


Figure 2.16 Decomposition of RASDF in the Y-chart methodology

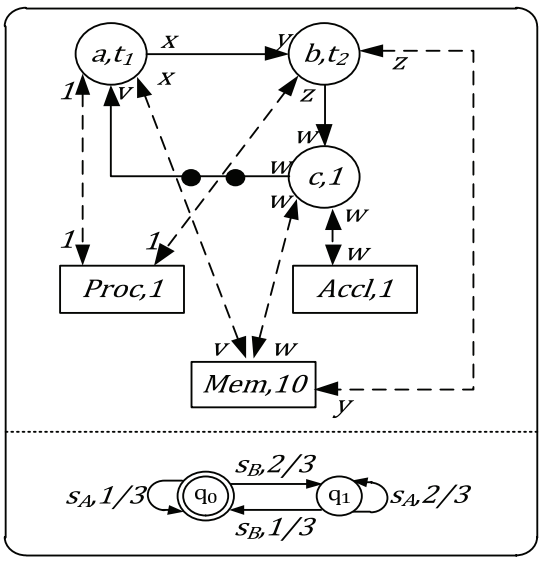
2.5 Scenario- and Resource-Aware Synchronous Dataflow Graphs

When we consider the changes in the environment of the system (input variation) and the usage of resources (resource requests) together, a natural extension to the existing models is to combine SADF and RASDF into *Scenario-*

and Resource-Aware Synchronous Dataflow (SARA SDF) graphs. In this new extension, the resource-aware SDF is replaced by its parametric version, a parametric resource-aware SDF that has to keep resource consistency as well as data consistency in each scenario and a scenario FSM that captures the transitions between scenarios.

Definition 2.9: Scenario- and Resource-Aware SDF (SARA SDF)

A Scenario- and Resource-Aware SDF (SARA SDF) graph is a tuple $G_{SARA} = (G_{PRASDF}, S_{FSM})$ *that contains a parametric resource-aware SDF graph and its corresponding scenario FSM.*



Scenarios	Rates					Execution times		Status
	v	w	x	y	z	t_1	t_2	c
s_A	1	1	2	4	2	2	1	disabled
s_B	0	0	3	2	0	2	2	enabled

Figure 2.17 An example SARA SDF graph

Figure 2.17 shows an example SARA SDF graph. Its parameters for different scenarios are given in the table below the graph. The scenario changes in scenario FSM influence not only the execution times and rates of actors, but also the amount of resources claimed and released by actors.

A consistent SARA SDF graph should keep the instances of parametric RASDF consistent in all scenarios. If a SARA SDF is consistent, its available resources will stay the same over an iteration of a scenario and after every possible scenario transitions (i.e., in every scenario, the corresponding RASDF is consistent), for instance, there is no memory leak between scenario transitions. Similar to SADF, we can also abstract from the probabilities of scenario transitions in SARA SDF graph and obtain FSM-SARA SDF graph.

When we take input changes into consideration, we are able to figure out strategies for the controller of a system to optimize performance or resource usage. It can be used to analyze the worst-case input sequence and synthesize performance guaranteed controller (in Chapter 6).

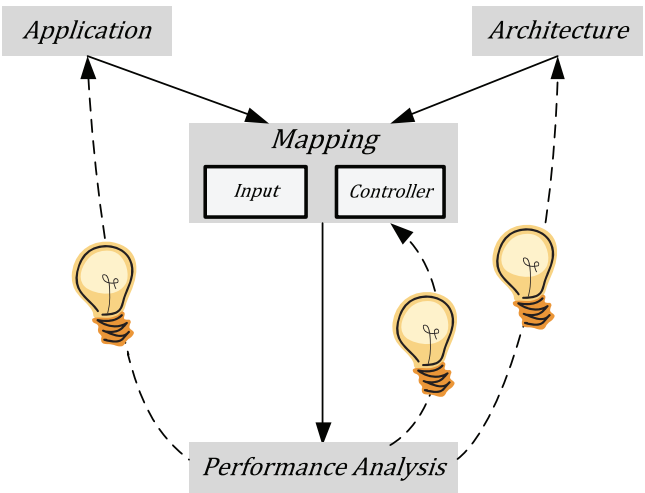


Figure 2.18 The role of input and controller in Y-chart methodology

Figure 2.18 shows the roles of input and controller in the Y-chart methodology in which they influence the mapping (execution times and resource requirements of actors) and further impact the performance and resource usage of streaming application. Based on the performance analysis of the specification, a designer can tune the actors of the application, the resources in the architecture, the mapping pairs between actors and resources in the mapping, the controller strategy to select different schedules or resource allocations.

2.6 Reflections and Related Work

In this chapter, we have introduced five existing and new MoCs: SDF, PSDF, SADF, RASDF and SARA SDF. The reason for introducing new models is to improve the combined expressiveness and analyzability of models for new design concerns such as resource and environment aspects of embedded systems. When compared to the traditional SDF model, the improved expressiveness is necessarily at the expense of losing some analyzability. For example, modeling resources and resource sharing explicitly results in many alternative actor scheduling options. The introduction of a scenario FSM for SARA SDF makes the performance and resource usage analysis more complex than the analysis for a single scenario case. In a nutshell, the introduction of new models means trading off between expressiveness and analyzability.

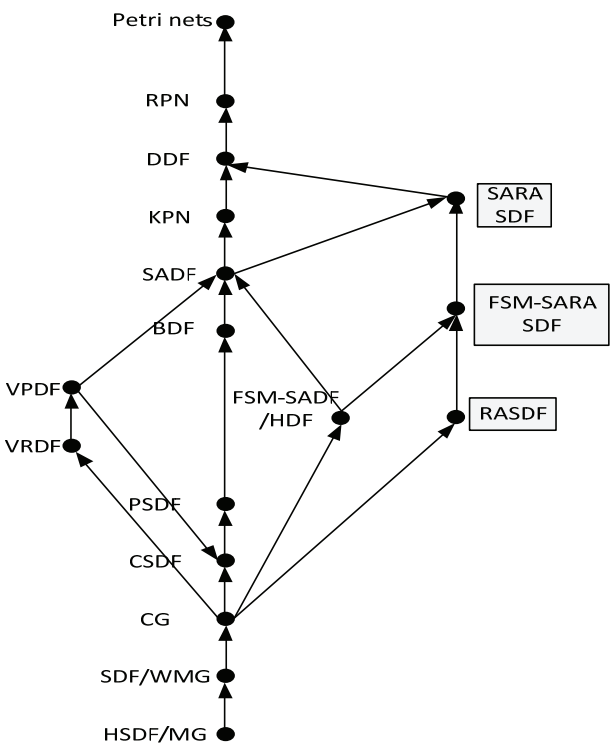


Figure 2.19 Expressiveness of Dataflow models (modified from [147]**)**

In [147], the expressiveness, implementation efficiency and analyzability of different types of dataflow MoCs are discussed in detail. Figure 2.19 shows the expressiveness of RASDF and SARA SDF models compared to existing dataflow models. Because of the introduction of the resource aspect, the new resource-aware models are arranged on a new vertical line. RASDF is more expressive than Computation Graphs (CGs) while FSM-SARA SDF and SARA SDF are more expressive than their counterparts FSM-SADF and SADF without resource consideration. [147] gives a detailed discussion comparing dataflow models. Generally speaking, these models are different from each other in rates and execution times. HSDF or Marked Graphs (MGs) has a single and fixed rate for all actors while SDF or Weighted Marked Graphs (WMGs) allows fixed rates, possibly different per actor and channel. Computation Graphs (CGs) allow that the number of tokens to enable a firing is different from the number of tokens to be consumed. RASDF allows selection when multiple actors are enabled but cannot fire at the same time due to lack of resources. CSDF allows rate changes in fixed patterns for a sequence of actor firings. Parametric SDF [13, 63] has parameterized rates and execution times so that symbolic analysis is possible. Variable Rate Dataflow (VRDF) [162] and Variable Phased Dataflow (VPDF) [163] allow actor rates to vary for each firing while keeping strong consistency. FSM-SADF, BDF, SADF, FSM-SARA SDF and SARA SDF all allow data-dependent behavior and are thus capable of capturing dynamism that is caused by input data. The rates of these models are determined by the input data. However, by defining the way rates change according to the input data, we generate different types of dataflow models with different levels of expressiveness and analyzability. KPN [93], DDF [24], RPN [61] and Petri nets can model system behaviors that change at runtime. They are more expressive than their counterparts but more difficult to analyze. It is easy to verify that SDF, FSM-SADF, SADF are special cases of RASDF, FSM-SARA SDF and SARA SDF, respectively. The introduction of explicit resource modeling allows choices in resource allocation and makes models harder to analyze than their counterparts. However, the improved expressiveness allows us to explore larger design spaces and further optimize our implementations.

The dataflow models are nothing more than a series of abstractions of streaming applications that we use every day. However, by abstraction, we keep the essence of the streaming applications that allows us to analyze the behaviors of applications and in turn improve the performance or reduce resource usage of streaming applications. The selection of different models is a

trade-off among expressiveness and analyzability. The introduction of resources into the modeling of dataflow allows us to explore in a larger design space.

2.7 Summary

In this chapter, we introduced the two dataflow MoCs that we analyze in the coming chapters. While keeping the core concept of consistency of the dataflow model, we extend the basic SDF model with resource-awareness and environment-awareness. The new MoCs are capable of modeling generic resources and dynamic input changes. Their relation with existing models such as SADF is shown through links between their topology matrices. We gave a brief overview of different kinds of dataflow models. In the coming chapters, we introduce the analysis techniques for the new possibilities that are introduced by these new models.

3 METRICS AND TRADE-OFFS

"Have no fear of perfection, you'll never reach it."

– Salvador Dali

3.1 Overview

Design activity itself is an art of compromising. Just like any design activities, designing streaming applications on embedded systems also has to trade among many conflicting but significant design parameters, such as performance, cost and power consumption.

In the early design stages of streaming applications, even the requirements themselves are obscured and can be flexible within certain ranges. It is important to identify the trade-offs in the design space and tune the design parameters later when uncertainties on requirements are narrowed.

Moreover, the awareness of trade-offs helps us to tailor a system to adapt to customers' needs more specifically, both at design time and at runtime. At design time, the identified trade-offs can help us tune the system in the cost-performance perspective to satisfy different requirements. At runtime, trade-offs can help us to select operating points to adapt to the preferences of different users or to different environments.

In this chapter, we first discuss the important design parameters of streaming applications, and then the trade-offs for streaming applications on embedded systems.

3.2 Throughput

An intuitive way to evaluate the performance of streaming applications is to measure the amount of data processed per time unit. However, since the processing time of one unit of data depends on the content of the data and the context of the system at runtime, such a performance number may vary. Instead of measuring instantaneous numbers, we measure the average number of data units processed by a system over time as the ***throughput*** of the system. Since guaranteed performance is needed for applications to meet timing constraints and user expectations, system designers have to know the

guaranteed throughput of their models. We define the exact meaning of *guaranteed throughput* in mathematical form in the following discussions.

Dataflow models may have multiple output actors with different output rates; or they may have different scenarios with different output rates. If we define throughput as the average amount of data produced by a specific actor per time unit, we obtain different throughput numbers for different actors. In order to define the throughput for the dataflow models as a whole, we first need an abstraction of the general unit of data that does not associate with any specific actor and can be used for different kinds of streaming applications. The concept of *an iteration* is an ideal abstraction for this. There are two reasons for using iteration as the abstraction. First, an iteration normally corresponds to a unit of data (such as a video frame or image to be printed) processed by streaming applications in a physical system and the length of a repetitive schedule for the system is a whole number of iterations. Second, it is a generic definition that can be applied to all introduced dataflow models.

For a given RASDF graph G there can be multiple, different, executions (for example, due to different scheduling rules). We use σ to denote one of its executions. Different executions may exhibit different throughputs, so we have to distinguish the throughput of one execution, i.e., $Th(\sigma)$, from the throughput of the graph, i.e., $Th(G)$. We first define the throughput of an execution.

Definition 3.1: Throughput of a dataflow graph execution

The throughput of a dataflow graph execution σ is defined as the eventually lower bound of the average number of iterations that is completed per time unit during the execution of the application, i.e., $Th(\sigma) = \liminf_{t\to\infty} \frac{n(\sigma,t)}{t}$ in which $n(\sigma,t)$ is the number of iterations that are finished before time t of the execution σ.

Recall that we want to find *guaranteed* throughput. We define the throughput of an execution as a property obtained eventually (i.e., when time approaches infinity). However, that limit may not exist. We then use the so-called eventually lower bound, i.e., $liminf$, to compute the throughput. ($liminf$ equals the limit if it exists.) The throughput definition uses the lower bound of the average number of iterations, so it is a conservative meaning of throughput.

For example, for an RASDF graph shown in Figure 3.1, a possible execution σ_1 that repeats a fixed pattern ($abcd$ and $acbd$) after the first iteration (after

time unit 10) and the number of iterations completed up to time t in the execution are shown in Figure 3.2.

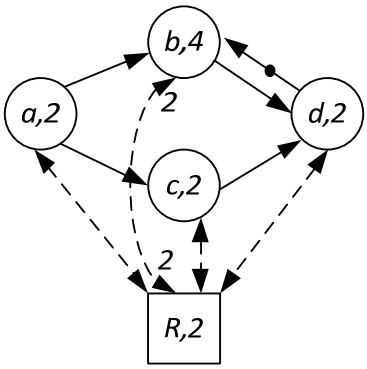
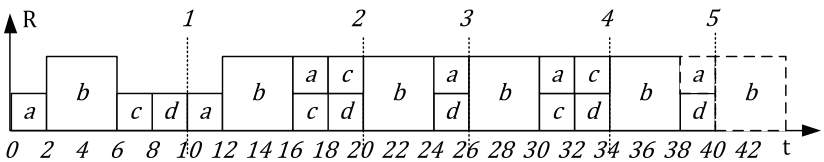
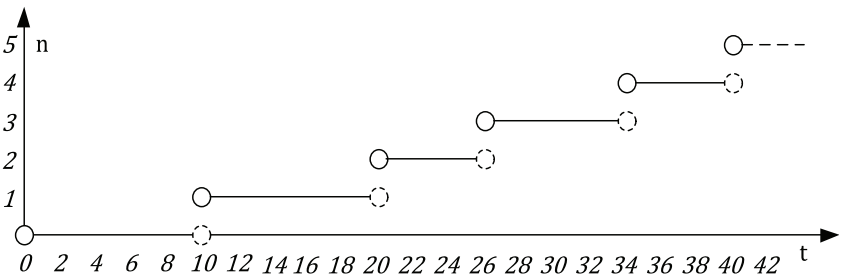


Figure 3.1 A running example RASDF



(a) An example execution that repeats $abcd$ and $acbd$ after 10



(b) Number of iterations completed up to time

Figure 3.2 An execution of the running example in Figure 3.1

In Figure 3.2 (a), we can see that the completion time of each iteration depends on its resource allocation (e.g. actor c uses a different amount of resource units of R) and schedule (e.g. the order of actor b and c). In the given

execution σ_1, the graph follows a fixed schedule pattern ($abcd$ and $acbd$) after the first iteration, the completion time increases by 6 and 8 alternatingly after every iteration beyond the second one (as shown in Figure 3.2 (b)). So we can describe the iteration completion time T_c of σ_1 for iteration n as follows:

$$T_c(\sigma_1, n) = \begin{cases} 10 & n = 1 \\ 6 + 7n & n \geq 2, n\ even \\ 5 + 7n & n \geq 2, n\ odd \end{cases}$$

So the throughput of the example execution σ_1 in Figure 3.2 (a) is

$$Th(\sigma_1) = \liminf_{t \to \infty} \frac{n(\sigma_1, t)}{t} = \liminf_{n \to \infty} \frac{n}{T_c(\sigma_1, n)} = \frac{1}{7}$$

A graph can have different executions with different throughput. For the same graph in Figure 3.1, we can also construct an execution σ_2 that always repeats the schedule $abcd$ such that actor a uses a different resource unit after the first iteration (as shown in Figure 3.3).

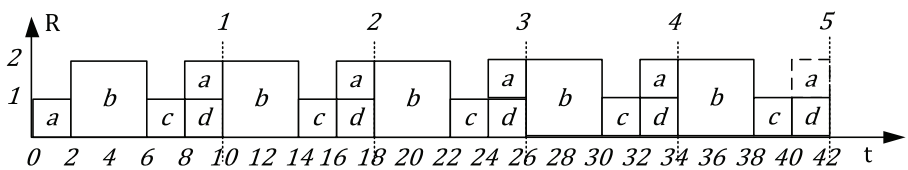


Figure 3.3 An execution of the running example repeats the schedule *abcd*

The iteration completion time T_c of σ_2 for iteration n as follows:

$$T_c(\sigma_2, n) = \begin{cases} 10 & n = 1 \\ 2 + 8n & n \geq 2 \end{cases}$$

So the throughput of the execution σ_2 is

$$Th(\sigma_2) = \liminf_{t \to \infty} \frac{n(\sigma_2, t)}{t} = \liminf_{n \to \infty} \frac{n}{T_c(\sigma_2, n)} = \lim_{n \to \infty} \frac{n}{2 + 8n} = \frac{1}{8}$$

If an execution of the example graph randomly selects its iteration schedules from the two patterns $abcd$ and $acbd$ and lets the completion times increase by 6 or 8 accordingly, the throughput of execution will be a value between 1/7 and 1/8. Since a dataflow graph can have multiple executions with different throughput, we also have to define the throughput of a dataflow graph.

Definition 3.2: Throughput of a dataflow graph

The throughput of a dataflow graph G is defined as the highest throughput among all possible executions of G, i.e.,

$$Th(G) = \sup_{\sigma} \liminf_{t \to \infty} \frac{n(\sigma, t)}{t}$$

So the throughput of a model is the best throughput that any of its executions can reach. The throughputs of executions in Figure 3.1 can be $1/7$, $1/8$, or anything in between, or even below 1/8 (by inserting idle time slots to delay actor firings). The throughput of the graph is the best attainable throughput, $1/7$.

3.3 Resource Usage

For designing a system, we are not only interested in the performance of the system, but we also care about how much we have to pay for achieving the performance. An important indicator of the system cost is its resource usage. Normally, the more resources that are used, the more expensive (such as memory cost) or less efficient (such as power consumption) the system is. Thus, designers are interested in minimization of resource usage.

In concrete systems, there are different kinds of resources, such as memories, processors, and buses. In order to perform computations, actors need to claim certain amounts of resources before computations start and release them or hand them over to other actors after computations end. In Chapter 2, we introduced the abstraction of resources as a set of numerical quantities in the RASDF model. Each resource is abstracted as an integer quantity that represents the amount of available resource. For real implementations, the resource allocation and de-allocation can happen at any time during an actor firing. For simplicity, we conservatively assume that the resource claims happen at the start of actor firings and the releases happen at the end of actor firings so that the amount of resource used and the duration of resource usage of the model are not shorter than the amount and duration in real implementation.

For a resource R, the resource usage increases m units after a claim request that claims m units of the resource and decreases n units after a release request that releases n units of the resource. For a claim request that happens at time t_1

and claims m units of resource R, we use $mU(t-t_1)$ to denote the resource usage change where

$$U(t-t_1)=\begin{cases}0 & t<t_1\\ 1 & t\geq t_1\end{cases}$$

Similarly, the resource usage change after a release request that happens at time t_2 and releases n units resource R is $-nU(t-t_2)$. For any resource, the resource usage can be defined as the combination of the basic functions.

Definition 3.3: Resource Usage Function

For a given execution, we sort its resource claim and release requests in time order. If claim and release requests happen at the same time, we assume the release requests are issued first in any order, followed by the claim requests in any order. Assume the requests occur at times t_i, where i is the index ranging over the sorted list of requests. The resource usage function of a resource R for a given execution σ is defined as: $R_\sigma(t)=\sum_{i=0}^{\infty} r_i U(t-t_i)$ in which r_i is positive if request type is claim, otherwise r_i is negative if the request type is release and where $|r_i|$ is the amount of resource claimed or released by the ith request.

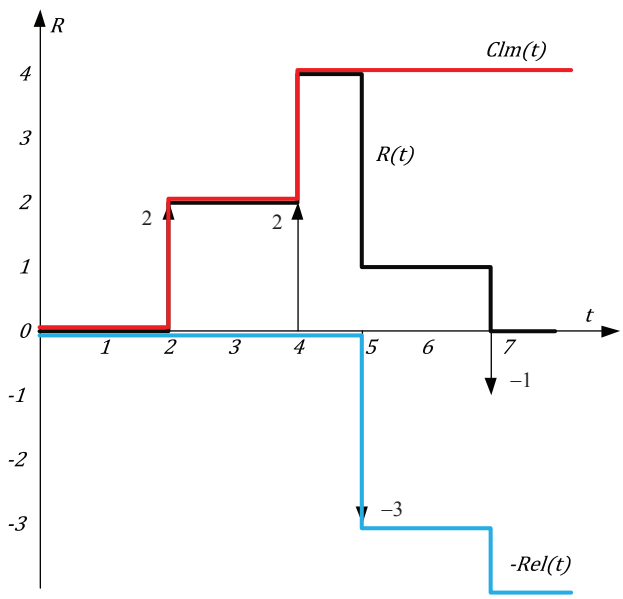


Figure 3.4 Decomposition of the resource usage function of an example.

Figure 3.4 illustrates the resource usage function of an example execution which claims 2 units of resource R at time 2 and time 4 respectively and releases 3 units of R at time 5 and 1 unit of R at time 7 respectively. A resource request at time t is denoted by an arrow at t whose length denotes the amount requested and its direction denotes the type of request (up for claim and down for release.) The resource claimed and released is denoted by two functions: a resource claim function (red curve) and a resource release function (blue curve). The resource usage function is denoted by summing up the release and claim changes.

Let us use m_i to denote the amount of resource claimed at the ith claim request that happens at time t_i, and use n_j to denote the amount of resource released at the jth release request that happens at time t_j. Then we can compute the resource usage function $R_\sigma(t)$ of an execution σ as follows:

$$R_\sigma(t) = \sum_{i=0}^{\infty} r_i U(t - t_i) = \sum_{i=0}^{\infty} m_i U(t - t_i) - \sum_{j=0}^{\infty} n_j U(t - t_j) = Clm(t) - Rel(t)$$

$$= 2U(t-2) + 2U(t-4) - 3U(t-5) - U(t-7)$$

For an execution, we are normally interested in the maximal value of the resource usage function since it is the least amount of resource that we have to allocate for the execution.

Definition 3.4: Resource Usage of a Resource

The resource usage of resource R in an execution σ is the maximal value of its resource usage function: $Ru(\sigma) = \max_t R_\sigma(t)$.

Note that, for a finite number of claim and release requests, the maximal resource usage in the above equation is only determined by the order of addition and subtraction. The order of subsequent subtractions does not change the max value. The maximal resource usage is not determined by the exact time of the requests. For example, the maximal resource usage of execution in Figure 3.4 is 4, which happens from time 4 to 5. The maximal resource usage will not change as long as the two release events happen after the two claim events and the order of the two release events does not change the maximal resource usage. If release events happen earlier than claim events, the maximal resource usage will be less than 4 but never more than 4.

As mentioned, the usage of resource *R* in an execution only depends on the order of claim and release requests. Since we assume the execution times of actors in the model to be the worst-case execution times in a realization, the release events may happen earlier. However, as long as the order of claim and release requests is preserved, the execution will have the same resource usage. If released events happen earlier, the resource usage might be less but never more than the analyzed result. Moreover, the order among release events (or claim events) has no impact on the maximal resource usage as the order between claims and releases does not change. For example, we change the order of release events in Figure 3.4 while keeping their orders with the claim events; the maximal resource does not change and the resource usage is shown in Figure 3.5.

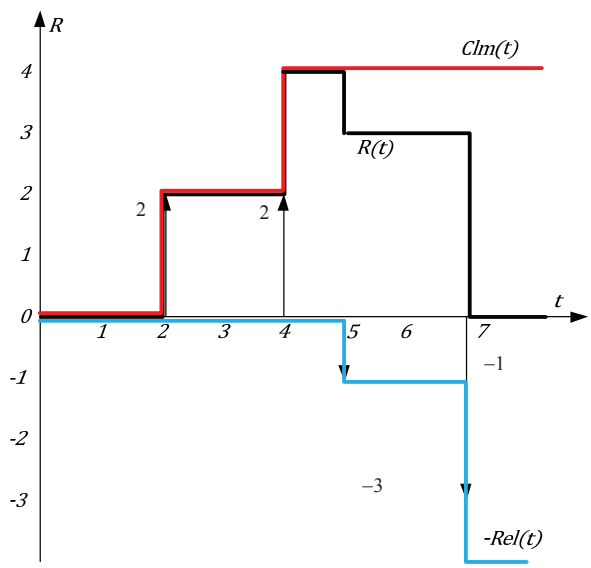


Figure 3.5 Resource usage with different firing order

3.4 Generalization of Metrics

In the previous sections, we discussed important metrics such as throughput and resource usage for a given execution. In this section, we generalize these metrics to cover other interesting metrics such as *power consumption* and *latency*.

An observation about throughput and resource usage is that they can be expressed as average and max functions over its execution. Throughput for example, is the average number of iterations per time unit of an execution.

We can define for some cost (or benefit or reward), a cumulative cost function $C(t)$ that, for a given execution σ, denotes the cumulative cost of the execution σ until time t. Since $C(t)$ diverges to infinity when time increases, we are often interested in average cost over time.

Definition 3.5: AVG Quantity

An AVG Quantity is defined as the average cost per time unit until time t is $C_{avg}(t) = \frac{C(t)}{t}$.

For example, the average number of iterations per time unit until time t is an Average Quantity. We can also define for some cost, e.g. resource usage, a non-cumulative cost function $NC(t)$, which denotes the non-cumulative cost of the execution σ at time t. In such cases we are typically interested in the maximum and minimum value of the costs.

Definition 3.6: MAX/ MIN Quantity

A MAX/MIN Quantity is defined as the max/min cost until time t, i.e., $C_{max}(t) = \max_{u \le t} NC(u)$ and $C_{min}(t) = \min_{u \le t} NC(u)$.

Note that both $C_{max}(t)$ and $C_{min}(t)$ are monotone functions.

For cumulative cost functions, we are normally interested in the long term stable behavior, i.e., the limit of these quantities or the eventually lower bound or eventually upper bound of the quantities.

Definition 3.7: Eventually Lower/Upper Bound of an Average Quantity

We denote the eventually lower bound of an average quantity as $C_{avg,l} = liminf_{t \to \infty} \frac{C(t)}{t}$, *and the eventually upper bound of average quantity as* $C_{avg,u} = limsup_{t \to \infty} \frac{C(t)}{t}$.

For MAX/MIN Quantities, since the quantity functions are monotonous, the limits always exist (here we accept both ∞ and $-\infty$ as the possible limits.) So we can directly use the limit and do not need $liminf$ or $limsup$.

$$C_{max} = \lim_{t\to\infty} C_{max}(t) \text{ and } C_{min} = \lim_{t\to\infty} C_{min}(t)$$

After we defined the generalized metrics, we can replace the general cost function with specialized cost functions to obtain the normal metrics that we are interested in.

	Name of Metric	Cost Function	Metric Definition
Average Metrics	Throughput	$n(t)$: Iterations finished before t	$\mathrm{Th}_{\mathrm{avg,l}} = \liminf_{t\to\infty} \frac{n(t)}{t}$
	Power Consumption	$P(t)$: Power consumption up to t	$\mathrm{P}_{\mathrm{avg,u}} = \limsup_{t\to\infty} \frac{P(t)}{t}$
Max/Min Metrics	Resource Usage	$Ru(t)$: Resource usage up to t	$\mathrm{R}_{\mathrm{max}} = \lim_{t\to\infty} Ru(t)$
	Latency	$L(t)$: Max length of one iteration before t	$\mathrm{Latency} = \lim_{t\to\infty} L(t)$

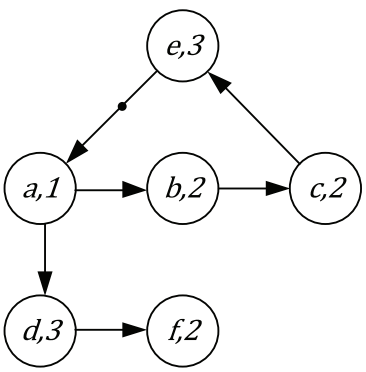
Table 3.1 Some metrics in general form.

In Chapter 4, we investigate the properties that we deduced from generalized metrics and apply them directly to two specialized instances, i.e., throughput and resource usage. Table 3.1 shows some metrics that can be classified into the two categories. For example, we can annotate the energy consumed by each actor firing to our model; then, we can derive the average power consumption of the application. We can also observe the latency, i.e., length of every iteration, and use the maximal latency to determine the latency of the execution. We will discuss how to compute the throughput and resource usage metrics in Chapter 4, but the techniques can be similarly used to other average or min/max properties.

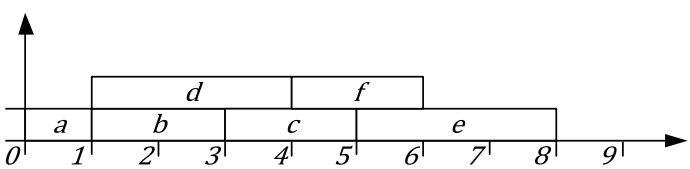
3.5 Conversion from executions to strong static-order schedules

In the next three chapters we discuss how to explore the schedules of RASDF and SARA SDF graphs. The results obtained from explorations cannot be directly used as schedules. We have to convert executions to schedules when implementing them in a system. We can reconstruct the start times of actor firings. This can be directly translated into a static timed schedule in which all actors are fired at exactly the time indicated by the schedule, even when they are enabled earlier. We here discuss how to relax a static timed schedule for possible better performance.

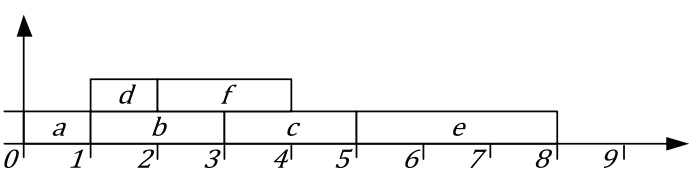
A static timed schedule puts tight constraints on actor firings. The worst-case execution time assumption underlying these static timed schedules might lead to pessimistic performance results since the execution times of actors may be less than the worst case. An earlier end of one firing might lead to an earlier start of another firing, which is not allowed in static timed schedule. An earlier firing start might lead to a higher performance.



(a) An example SDF graph with worst-case execution times



(b) An execution with actor *d*'s execution time equals to 3

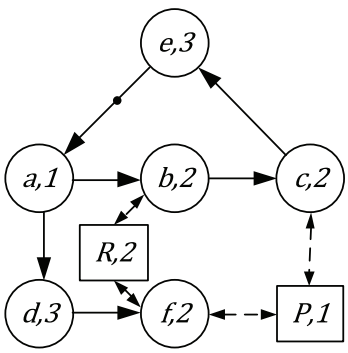


(c) An execution with actor *d*'s execution time equals to 1

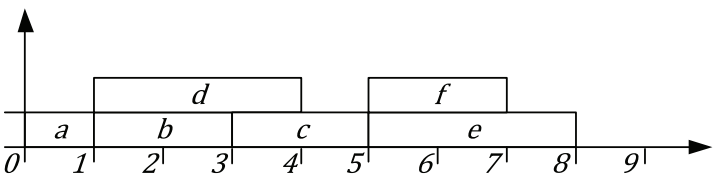
Figure 3.6 Two executions of an SDF graph with different execution times

Without shared resources, SDF graph adhere to the so called monotonicity property for its executions, i.e., decreasing execution times of actors results in non-increasing start times of actors. Or in other words, an earlier actor start time cannot lead to worst performance. So we can use a static order schedule to replace a static timed schedule for plain SDF graphs. As long as the order of

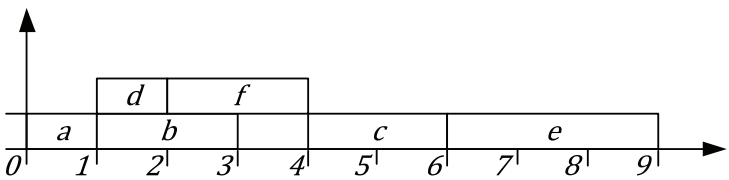
actor firing in an SDF graph is preserved, actors can start as early as possible since no actor will start late. For example, Figure 3.6 gives an example of an SDF graph and lists two executions in which the actor *d* has different execution times. The execution time of actor *d* in Figure 3.6 (b) equals the worst-case execution time specified in Figure 3.6 (a) while it is less than the worst-case execution time in Figure 3.6 (c). The decreasing of the execution time of actor *d* does not cause an increasing firing time of any actor in the graph.



(a) An example RASDF graph with worst-case execution times



(b) An execution with actor *d*'s execution time equals to 3



(c) An execution with actor *d*'s execution time equals to 1

Figure 3.7 Two executions of an RASDF graph with different execution times

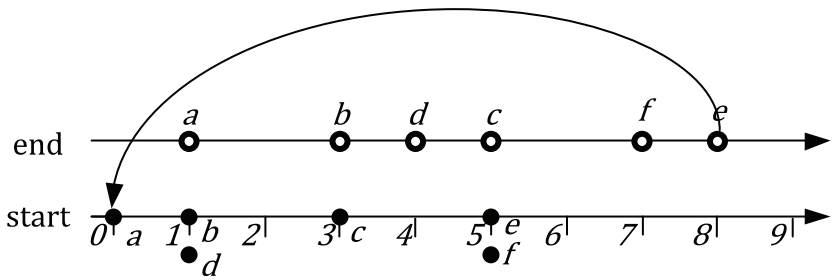
Due to resource sharing, the monotonicity of SDF graphs does not hold anymore. If the firing of one actor may occupy resources allocated to a firing of another actor early, then it may delay the firing of the other actor. The delay

might degrade performance. Figure 3.6 gives two executions of the example RASDF graph with different execution times of actor d. The decreasing execution time of actor d leads to the earlier firing of actor f. Since actor f and actor c share resource P, at any one time only one actor instance is able to run. The earlier firing of actor f occupies the resource and delays the firing of actor c (so we do not have monotonicity anymore). The delay of actor c in turn downgrades the performance since the periodic execution takes longer time in Figure 3.7 (c) than periodic execution in Figure 3.7 (b). For resource usage, the earlier actor firing can also lead to additional resource usage. For instance, if actor b and actor f both require a resource R for execution, the earlier firing of actor f causes a resource usage of resource R of 2 in the execution in Figure 3.7 (c) rather than 1 in the execution in Figure 3.7 (b).

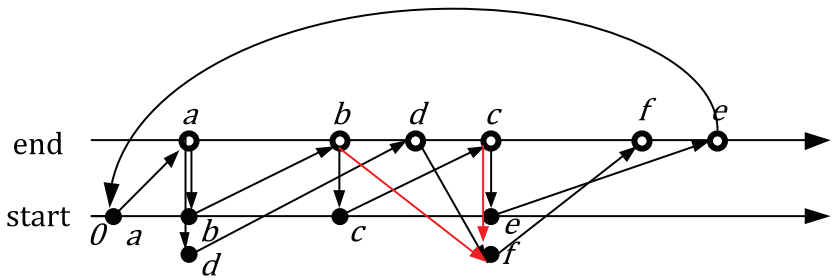
To make performance and resource usage predictable for RASDF and SARA SDF graphs without a static timed schedule, we have to keep the order of claim and release events of resources to ensure that the maximal resource usage of relaxed schedules at any point in time cannot exceed the resource usage of a static timed schedule. We call this schedule type a strong static-order schedule. From the analysis of resource usage patterns in Section 3.3, we derive the following rules when we relax the static timed order schedule to a strong static-order schedule. First, a claim event of a resource cannot happen earlier than the release events of the same resource before it in the static timed schedule. Any actor firing with an earlier claim event than the release events found in the static timed schedule will lead to more resource usage. Second, a release event of an actor firing can happen as early as possible, i.e., the execution time of an actor is less than its worst-case execution time, since it brings no side-effect due to the first rule.

For instance, the static timed schedule corresponding to the execution in Figure 3.7 can be noted down as a sequence of actor firings with start times, i.e., $(a,0)\cdot(b,1)\cdot(d,1)\cdot(c,3)\cdot(e,5)\cdot(f,5)$. Since we assume claim events to happen at the beginning of actor firings while release events happen at the end of actor firings, the static timed schedule can be translated to an ordered schedule of claim and release events with time information as shown in Figure 3.8 (a). Here we use filled dots to denote the actor start firings, respectively, and circles to denote the actor end firings, respectively. We can relax the static timed schedule to a partial order description of claim and release events of actor firings. In Figure 3.8 (b), we see that we add resource dependencies at time points 5 between release events and claim events of resources P and R to

avoid the claim event at this point to start earlier than the release events of these resources before time point 5. Then we can remove the time information and obtain the strong static-order schedule. The execution of the example RASDF graph with actor *d*'s execution time 1 is shown in Figure 3.9 with the same performance and resource usage as the execution in Figure 3.7 (b).



(a) Static timed schedule



(b) Strong static order schedule

Figure 3.8 Static timed schedule and Strong static order schedule

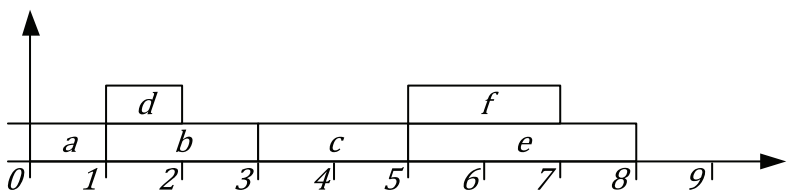


Figure 3.9 An execution of example that follows strong static-order schedule

In implementation, we have to implement event monitors to ensure the order specified by the strong static-order schedule. The resulting throughput of

the system under a strong static-order schedule is at least as good as the throughput under the static timed schedule.

3.6 Trade-offs and Pareto Optimization

When we have multiple design objectives, our design space also becomes multi-dimensional and our design problem becomes a multi-objective optimization problem. The multi-objective nature of designing a streaming application on an embedded system introduces trade-offs, i.e., a number of design alternatives exist and usually none of them is strictly better than the others on all aspects. To explore the trade-offs and to tailor systems for specific needs are challenging tasks for system designers.

Pareto optimality is a qualitative measure of the efficiency used in many domains with multi-objective optimization problems, such as economics, system design, and game theory. We can use Pareto optimality to evaluate design alternatives of streaming applications in a multi-dimensional design space. A design alternative is Pareto optimal if no other alternative is strictly better than it in any of its design metrics.

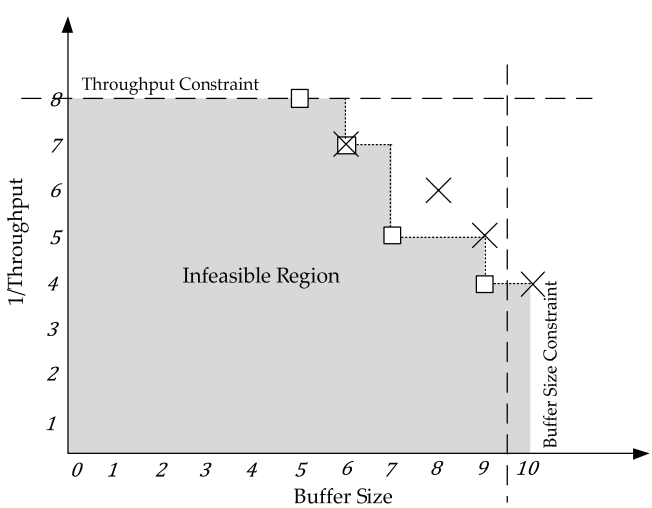


Figure 3.10 Trade-offs of two different architectures

For example, assume we have two different architectures (denoted with A and B) for a streaming application with buffer size still tunable, the design

points with different buffer sizes and throughputs are given in Figure. 3.10, in which the cross points are for architecture A and the square points are for architecture B. For each architecture, there are sets of trade-off points, i.e., different compromises between throughput and buffer size. Note that we use the reciprocal of throughput on the vertical axis to make sure that the lower the number is, the better the solution is on both horizontal and vertical axes. With the given throughput and buffer size constraints (denoted with dashed lines), we can select only design points within the lower left corner area, i.e., the white space below the throughput constraint line and on the left side of the buffer size constraint line. Apparently, the design points of architecture B (squares) dominate the design points of architecture A (crosses) since no design point of A is better than points of B. The Pareto optimal solutions, here the design points of B, are called the *Pareto front* of the design space. The grey area below the Pareto front is called the *infeasible region*, as there are no (known) design solutions in this space. The trade-off points of architecture B that are within the constraints can be further pruned for the final decision when more constraints such as user preferences or product budget are given. The trade-offs keep flexibility in the early design stage, which is crucial for the later design activities.

We want to find the metrics that we are interested in in the early design stages, and form the Pareto-front of the design space of a given specification (an RASDF model). In order to keep the trade-off discussion simple, we limit our metrics to throughput and resource usage in discussions in Chapter 4 and Chapter 5. However, it is not difficult to generalize the conclusion to other metrics as long as we can classify them into average quantities (throughput) or max/min quantities (resource usage).

In Figure 3.6, the trade-offs of architecture A are fully dominated by the trade-offs of architecture B. However, in many cases, the trade-offs from two candidate architectures do not fully dominate each other. In order to compare the quality of the trade-offs provided by two different architectures, we have to evaluate the quality of the trade-offs quantitatively. Quality evaluation for Pareto points found by multi-objective exploration is lacking a single standard. A few methods are proposed. Among different evaluation methods, the Average Distance to Reference Set (ADRS) [36] method and the ϵ-indicator method [177] are the most common choices in literature. We use these two methods for evaluation of our experimental results in the next two chapters. Generally speaking, ADRS is often used to evaluate the quality of a set of

points to approximate a known reference set of Pareto-optimal points while the ϵ-indicator is used to compare two different point sets.

Here we use the two sets of points in Figure 3.5 to illustrate the approach. We use PA to denote the set of 4 square points, i.e., $\{(5,8),(6,7),(7,5),(9,4)\}$; and we use PB to denote the set of 4 cross points, i.e., $\{(6,7),(8,6),(9,5),(10,4)\}$.

In the ADRS approach, the average distance of a set of Pareto points to the reference set of Pareto points is measured. The distance function does not necessarily have to be the geometric distance function and it can be customized for an application. Here, we define the distance between two points as the maximal ratio of value change among all objective dimensions. The distance of point a of a set PB to another set PA is the minimal distance of a to all points in the set PA. For example, we use PA as reference set since it is optimal. The distance of point (9,5) in PB to PA is:

$$\min(\max(4,3),\max(3,2),\max(2,0),\max(0,1))=1.$$

The ADRS of PB to reference set PA is $\frac{0+1+1+1}{4} = 0.75$.

In the ϵ-indicator approach, we use $I_\epsilon(PA,PB)$ to define by how much the points in PB need to be scaled so that they are all dominated by points in PA. Here, we define the ratio between two points as the maximal ratio of value change among all objective dimensions. The ratio of a point of a set PB to another set PA is the minimal ratio of a to all points in the set PA. The ϵ-indicator $I_\epsilon(PA,PB)$ is the maximal ratio of all points in PB. Loosely speaking, the set PB is better than the set PA when $I_\epsilon(PA,PB)$ is larger than $I_\epsilon(PB,PA)$. If $I_\epsilon(PA,PB)$ is larger than 1, then PB contains new Pareto points compared to PA. In Figure 3.6, since PA fully dominates PB and does not have to be scaled, we have $I_\epsilon(PA,PB) = 1$. And $I_\epsilon(PB,PA) = 1.2$ since the scaled set PA' is $\{(6,9.6),(7.2,8.4),(8.4,6).(10.8,4.8)\}$ is fully dominated by PB.

3.7 Summary

In this chapter, we discussed the metrics of interest for streaming applications. Further we generalized the two metrics throughput and resource usage to the more generic form of average and maximal quantities. We introduced the concept of trade-offs and Pareto optimality in design space exploration and its usage in early design stages. In the coming chapters, we discuss how to use the

models introduced in Chapter 2 to explore the metrics and trade-offs that have been introduced in this chapter.

4 TIME-DOMAIN ANALYSIS

"But there is no point in making mistakes unless thereafter we are able to learn from them. "

–E.W.Dijkstra

4.1 Overview

Dataflow models are used to help us to analyze streaming applications. With dataflow models introduced in Chapter 2, we can specify the structure of streaming applications in an intuitive way. The models help engineers to have an overview of the applications and ease the communication with each other. However, with the structural information of a dataflow graph, we only know static information of a model, such as which actors communicate to each other or how much data is communicated. In order to obtain metrics introduced in Chapter 3, such as throughput and resource usage, we need to know the operational semantics of a dataflow model and infer its behavioral information based on the semantics. With both structural and behavioral information of the model, engineers can communicate with each other without ambiguity and can utilize all kinds of analysis techniques to determine the metrics of interest.

RASDF is different from the traditional SDF since it explicitly takes resources into consideration. The behavior of a single actor is quite simple as illustrated in Figure 4.1. The firing of an actor is enabled when both the numbers of input *and* resource tokens are sufficient. At the start of actor firing, the actor reads input tokens and claims resources. After a finite amount of execution time, the firing ends, it writes the output tokens at the end of firing and releases resources.

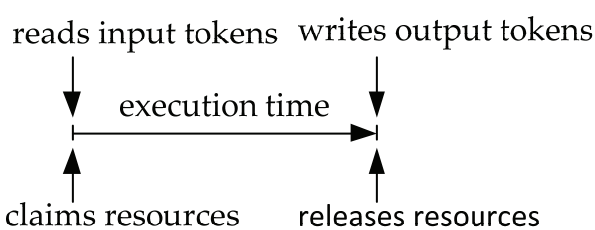


Figure 4.1 Operational semantics of an actor firing in RASDF

This extension of SDF with resources, i.e., RASDF, has both positive and negative impacts on analysis. The positive side is that the explicit modeling of resources enables analysis of resource sharing and may lead to more resource-efficient results. For instance, different FIFO channels can share the same physical memory location at different times to reduce the total memory usage. The negative side is that the sharing of resources may lead to potential conflicts among actors, increasing the complexity of analysis. Due to resource contention, multiple actors are enabled but they cannot fire at the same time, multiple firing options are open for exploration, and their firing order may impact performance, which results in a larger state space and longer exploration time. The advantage and disadvantage of RASDF drive us to develop techniques to make the analysis of RASDF more efficient.

In this chapter, we introduce the analysis of RASDF in the time-domain and, in the next chapter, the analysis of RASDF in the iteration-domain. In Section 4.2, the operational semantic of RASDF is introduced for analyzing it formally. In Section 4.3, it is explained how to explore the state space of an RASDF model. In Section 4.4, a heuristic approach is introduced to explore the state space more efficiently. In Section 4.5, a bottleneck-driven approach is built on top of the heuristic approach. In Section 4.6, we do case studies on a few RASDF graphs. Section 4.7 discusses related work and Section 4.8 concludes this chapter.

4.2 Operational Semantics of RASDF

In order to analyze an RASDF graph, we introduce some terminology and definitions to formally describe the execution of an RASDF graph.

Definition 4.1: **State**

Given an RASDF graph $(G_{SDF}, R, q, \tau, req)$ *with* $G_{SDF} = (A, C, wr, rd)$, *a state of the graph is a triple* (δ, η, υ) *that consists of a channel quantity* δ *that denotes the amount of data tokens in the channels of the graph at that state, a resource quantity* η *that denotes the amount of used resource tokens in that state, and a function* $\upsilon: A \mapsto \mathbb{N}^{\mathbb{N}}$ *that associates with each actor* $a \in A$ *a multiset of numbers representing the remaining times of different active firings of* a.

We assume that the initial state of an RASDF is given by an initial channel quantity δ_0, i.e., some initial token distribution, an initial resource usage η_0 (not necessarily zero since there may exist some resource allocation for initial data.)

and no actor firings. So we can use as initial state: $(\delta_0, \eta_0, \{(a, \{\})|a \in A\})$, with {} denoting the empty mutliset. The use of a multiset of numbers to keep track of actor progress allows multiple simultaneous firings of the same actor (auto-concurrency). The auto-concurrency is limited by the amount of available resources.

Recall that we define functions rd and wr for each channel in Chapter 2 (see Definition 2.1) as well as clm and rel functions for each resource (see Definition 2.7). For an actor operation on channel quantity δ and η, we define the following functions:

- a function $Rd: A \mapsto \mathbb{N}^{|C|}$ that maps each actor to a channel quantity, i.e., the amount of tokens read from channels when a firing starts;
- a function $Wr: A \mapsto \mathbb{N}^{|C|}$ that maps each actor to a channel quantity i.e., the amount of tokens written to channels when a firing ends;
- a function $Clm: A \mapsto \mathbb{N}^{|R|}$ that maps each actor to a resource quantity, i.e., the amount of tokens claimed from that resource when a firing starts;
- a function $Rel: A \mapsto \mathbb{N}^{|R|}$ that maps each actor to a resource quantity i.e., the amount of tokens released from that resource when a firing ends.

Note that the four functions are vector form of the rd, wr, clm and rel functions.

The effect of an actor firing on a state of an RASDF graph can be denoted by addition and subtraction of the corresponding quantities of the state. To check whether an actor is enabled or not, we have to compare channel and resource quantities. Here, we define the dominance relation between two quantities.

Definition 4.2: Comparison between two quantities

Given two channel (resource) quantities q_1 and q_2, if the number of tokens for any channel c (resource r) of q_1 is the same or less than the number of tokens for the corresponding channel (resource) of q_2, we denoted the relation by $q_1 \preccurlyeq q_2$,

The dynamic behavior of an RASDF during execution is described by transitions. There are three different types of transitions in the time domain: start of an actor firing, end of an actor firing, and time progress through discrete clock ticks.

Definition 4.3: Transition

A transition of an RASDF $(G_{SDF}, R, q, \tau, req)$ *from a state* $(\delta_1, \eta_1, \upsilon_1)$ *to another state* $(\delta_2, \eta_2, \upsilon_2)$ *is denoted by* $(\delta_1, \eta_1, \upsilon_1) \xrightarrow{\beta} (\delta_2, \eta_2, \upsilon_2)$ *where label* $\beta \in \{A \times \{start, end\}\} \cup \{clk\}$ *denotes the type of transition.*

There are three types of transitions:

- $start$ transition: Label $\beta = (a, start)$ corresponds to the firing start of actor $a \in A$. This transition results in $\delta_2 = \delta_1 - Rd(a)$, $\eta_2 = \eta_1 + Clm(a)$ and $\upsilon_2 = \upsilon_1[a \mapsto \upsilon_1(a) \uplus \{\tau(a)\}]$ (where $\uplus$ denotes multiset union). It may occur only if $Rd(a) \preccurlyeq \delta_1$ and $Clm(a) + \eta_1 \preccurlyeq q$, where q is the amount of available resource in the RASDF graph; and when no end transition is enabled.
- end transition: Label $\beta = (a, end)$ corresponds to the firing end of actor $a \in A$. This transition results in $\delta_2(a) = \delta_1(a) + Wr(a)$, $\eta_2 = \eta_1 - Rel(a)$ and $\upsilon_2 = \upsilon_1[a \mapsto \upsilon_1(a) \setminus \{0\}]$ (where $\setminus$ denotes multiset difference). It is enabled if $0 \in \upsilon_1(a)$.
- clk transition: Label $\beta = clk$ denotes a clock transition which is enabled if no end transition is enabled. This transition results in $\delta_2 = \delta_1$, $\eta_1 = \eta_2$ and $\upsilon_2 = \{(a, \upsilon_1(a) \ominus 1) | a \in A\}$ where $\upsilon_1(a) \ominus 1$ denotes a multiset of natural numbers containing the elements of $\upsilon_1(a)$, which are all strictly positive, reduced by one.

Due to resource constrains, not all start transitions with sufficient input tokens may actually be able to start simultaneously, i.e., before the next clk transition. There may exist multiple combinations of start transitions of actors with sufficient input tokens that can start at the same time and keep resource usage within resource constraints for the resulting starts. Note that *end* transitions are not constrained by resources and are always enabled and executed eagerly. In Chapter 2, we define an execution as a sequence of actor firings. With the definitions of state and transition, we can formally define the execution as below.

Definition 4.4: Execution

An execution σ *of an RASDF graph is a finite or infinite alternating sequence of states and transitions, i.e.,* $\sigma = s_0 \xrightarrow{\beta_0} s_1 \xrightarrow{\beta_1} s_2 \cdots$. *We use* $t(\sigma)$ *to denote the completion time of* σ. *We use* $\sigma(i)$ *to denote the ith state of an execution* σ.

An execution does not necessarily start with the initial state of the RASDF. It can represent an execution that starts from any state. When the labels are not relevant, we can also write it as $\sigma = s_0 s_1 s_2 \cdots$. We use $|\sigma|$ to denote the length of the execution, i.e., the number of *transitions*. $|\sigma| = \infty$ if the execution is infinite. We use $t(\sigma)$ to denote the number of clk transitions in σ. We use $\sigma = \sigma_a \cdot \sigma_b$ to denote an execution that consists two executions σ_a and σ_b, where $\cdot$ stands for concatenation.

end transitions only release resources, and the data produced by end transitions will be available to enable new start firings of actors. Therefore, we give end transitions priority over other transitions. If a number of subsequent $start$ transitions are taken at the same time, the order in which they are taken has no impact on the resource usage or resulting state. When no more $start$ transitions are selected, a clk transition occurs, possibly leading to new end transitions and so on.

We call the process that controls the execution of an RASDF graph its *execution engine*. The execution engine of an RASDF can be illustrated as an infinite execution flow chart in Figure 4.2.

The execution engine of an RASDF graph starts from an initial state s_0. Since the initial state can be any state during an execution of the RASDF graph. The RASDF graph first executes all enabled end transitions until all end transitions are finished. Then, the execution engine go to check whether there are started transitions, it will select **some** enabled actors to fire, such that the resource usage of these actors are within the amount of available resources. The selection is based on a cost function on the numbers of completed firings that will be discussed later. It may be beneficial to start fewer actors than resources allow to achieve higher throughput. An example is given in Figure 4.4 (b). After firing the selected actors, the execution engine will perform a clk transition and then return to the beginning to check whether end transitions are enabled and repeat the steps again. Since we assume streaming applications have unlimited amount of input, the execution engine also execute infinitely. In next section, we discuss how to explore the state space of an RASDF graph.

Compared to self-timed execution discussed in [64, 144], the major difference between a self-timed execution of an SDF graph and an execution of an RASDF graph is that a self-timed execution will fire **all** enabled actors when

actors have enough input tokens while the execution of an RASDF graph selects **some** enabled actors due to resource constraints. This difference leads to multiple possible paths for RASDF graph execution, where an execution of an SDF graph is determinate.

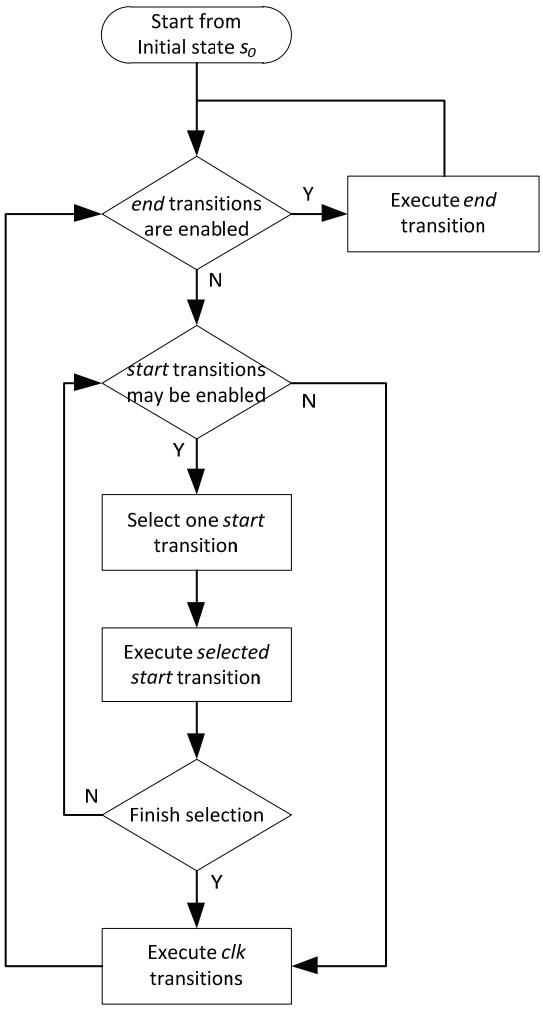


Figure 4.2 Flow chart of the execution engine of RASDF graph

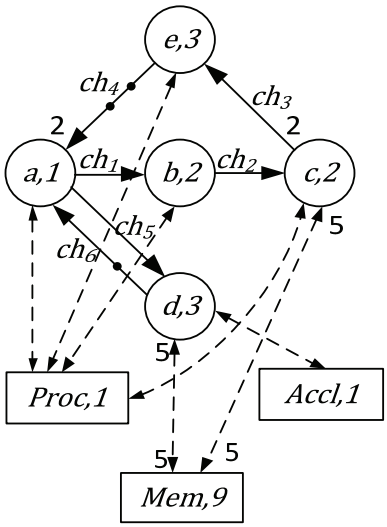
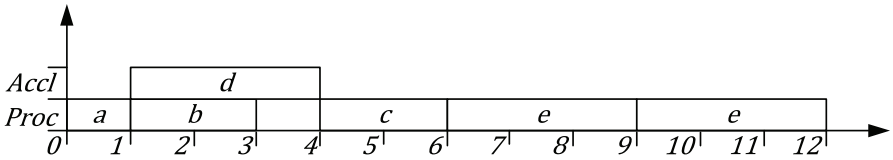
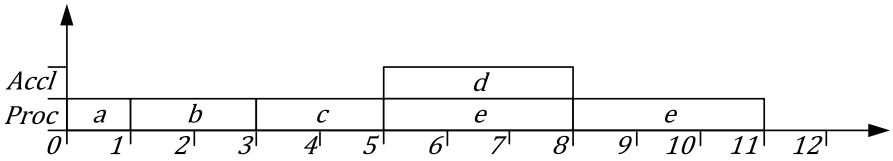


Figure 4.3 An example RASDF graph with a resource conflict

Figure 4.3 shows an example RASDF graph (rates equal to one are omitted for simplicity) with resource conflicts between actors c and d, i.e. actors c and d cannot execute at the same time since they both require 5 units of memory for execution.



(a) Execution that fire actor as soon as possible



(b) Execution with delaying actor firing

Figure 4.4 Two possible executions of the example in Figure 4.3

Figure 4.4 shows two possible executions for the example RASDF in Figure 4.3. Once firing of actor a has ended, both actor b and actor d are enabled for firing. If we fire both actors, actor c will be delayed due to lack of units of memory. In the end, the throughput of the system is only $1/12$ due to the delay in critical path, i.e., the gap between actors b and c due to execution of actor d (see Figure 4.4(a)). However, if we select a different scheduling strategy, and delay the firing of actor d until the firing of c is ended, we can have a better throughput $1/11$ (see Figure 4.4(b)). From this example, we can see that resource constraints can influence the order of actor execution and further influence the throughput of an application. In order to find Pareto-optimal executions in the design space, we have to explore multiple possible executions.

4.3 Exploring the state-space of RASDF

In the previous section, we saw there can be multiple executions for an RASDF graph. This will influence the state space exploration techniques that we use. From the definition of state of an RASDF graph, we can easily see that the size of the state space is finite if the amount of data and resource tokens is bounded, i.e., $\boldsymbol{\delta}$ and $\boldsymbol{\eta}$ are bounded. The bounded resources limit the degree of auto-concurrency of actors, thus limit the size of $\boldsymbol{v}$. The execution times of actors are also limited. So the number of different combinations of remaining execution times $\boldsymbol{v}$ is also finite. Since $\boldsymbol{\delta}, \boldsymbol{\eta}, \boldsymbol{v}$ are all finite, the number of different states $\boldsymbol{s}$ is also finite.

For a finite state space, an important property of an infinite execution is that the execution **revisits** at least one state infinitely often. For example, Figure 4.5 shows the state space of the example RASDF graph shown in Figure 4.3. The initial state $s_0 = (< 0,0,0,2,0,1 >, < 0,0,0 >, \{(a, \{\}), (b, \{\}), (c, \{\}), (d, \{\}), (e, \{\})\})$. When actor a starts firing, it consumes the initial tokens on ch_4 and ch_6, claims processor $Proc$ and its state goes from the initial state s_0 to state $s_1 = (< 0,0,0,0,0,0 >, < 1,0,0 >, \{(a, \{1\}), (b, \{\}), (c, \{\}), (d, \{\}), (e, \{\})\})$. Then the time advances for the minimal remaining execution time of any of the active actors. This is one time unit in this example. Actor a then ends firing, produces 1 token on both ch_1 and ch_5, and releases $Proc$. Resource conflicts (e.g., c and d compete for Mem) and scheduling freedom (e.g., the firing order of b and d) for the system allow multiple possible transitions and lead to different executions of the system. The state s_1 has 3 possible transitions that lead to state s_2, s_6 and s_7 respectively. In general, during a state transition from one state to another state, some actors end firing, produce tokens on channels and release occupied

resources while some other actors start firing, consume tokens and claim required resources.

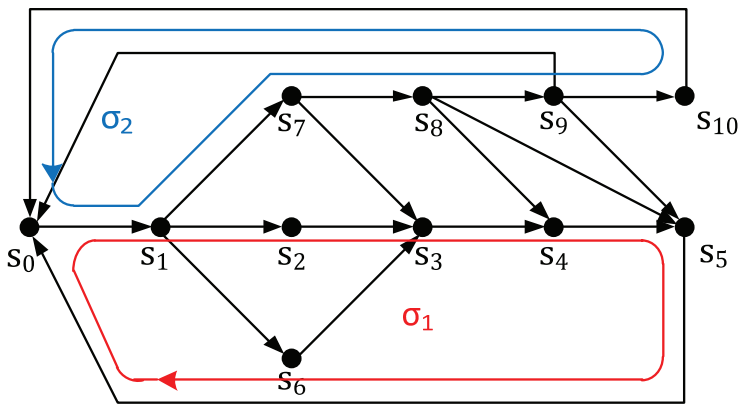


Figure 4.5 State space of example RASDF in Figure 4.3

We recall the MAX quantity and AVG quantities introduced in Chapter 3 (see Definitions 3.5 and 3.6). We can define the basic quantity q as a function of state, i.e., $q(s_i)$. Then the MAX quantity for q for a given execution $\sigma = s_0 s_1 s_2 \cdots$ is $q_{max}(\sigma) = max\{q(s_i) | \, 0 \leq i < |\sigma|\}$. An important property of this type of quantity is monotonicity. For example, given an execution $\sigma_a = \sigma_b \cdot \sigma_c$, we always have $q_{max}(\sigma_a) \geq q_{max}(\sigma_b)$. The resource usage of an execution σ of an RASDF graph, i.e., $Ru(\sigma) = max(\eta_0, \eta_1, \eta_2, \cdots)$ is a MAX quantity.

Similarly, an AVG quantity of q of an execution can also be defined as a function of states, i.e., $q_{avg} = liminf_{N \to |\sigma|} \frac{\sum_{i=0}^{N} q(s_i)}{t(\sigma_N)}$ where $t(\sigma_N)$ is the completion time for the execution σ_N . The throughput of an execution can be viewed as an AVG quantity.

Definition 4.5: Simple Execution

A simple execution $\sigma = \sigma_{pre} \cdot \sigma_{per}^{\omega}$ is an infinite execution starting from the initial state of an RASDF graph that is composed of two parts: a finite length prefix execution σ_{pre}, not containing any duplicate states, and an infinite periodic repetition of execution σ_{per} that is a cycle that starts and ends in the same state and has no duplicate states either. There are no shared states between σ_{pre} and σ_{per}.

For example, Figure 4.5 shows multiple simple executions, such as $\sigma_1 = (s_0 s_1 s_2 s_3 s_4 s_5)^\omega$ and $\sigma_2 = (s_0 s_1 s_7 s_8 s_9 s_{10})^\omega$ both with empty prefix. A simple execution generates a finite schedule consisting of a prefix schedule and a periodic schedule.

An important property of a simple execution $\sigma = \sigma_{pre} \cdot \sigma_{per}^\omega$ is that the value of AVG quantities such as throughput are fully determined by the periodic part σ_{per}, i.e., $Th(\sigma) = Th(\sigma_{per})$. For a consistent RASDF graph, the value of MAX quantities is determined by the prefix part σ_{pre} and periodic part σ_{per} together, i.e., $(\sigma) = Ru(\sigma_{pre} \cdot \sigma_{per})$.

Definition 4.6: Pareto-Optimal Execution

An execution σ_1 dominates another execution σ_2 if and only if σ_1 is not worse than σ_2 in any of the metrics of interest. An execution σ is Pareto optimal if and only if it is not dominated by any other execution.

When we explore the design space of an RASDF graph, we want to find good executions that can satisfy our design constraints. All possible executions are the target design space that we want to explore. A Pareto-optimal execution with its metric values is called a Pareto point in the design space. In order to find Pareto points in the design space without redundant exploration, we have to use some rules to prune our exploration paths. We discuss pruning techniques for efficient exploration of the state space below. Due to the monotonicity of MAX quantities, it is easy to prove the following proposition. It is illustrated in Figure 4.6 (a).

Proposition 4.1:

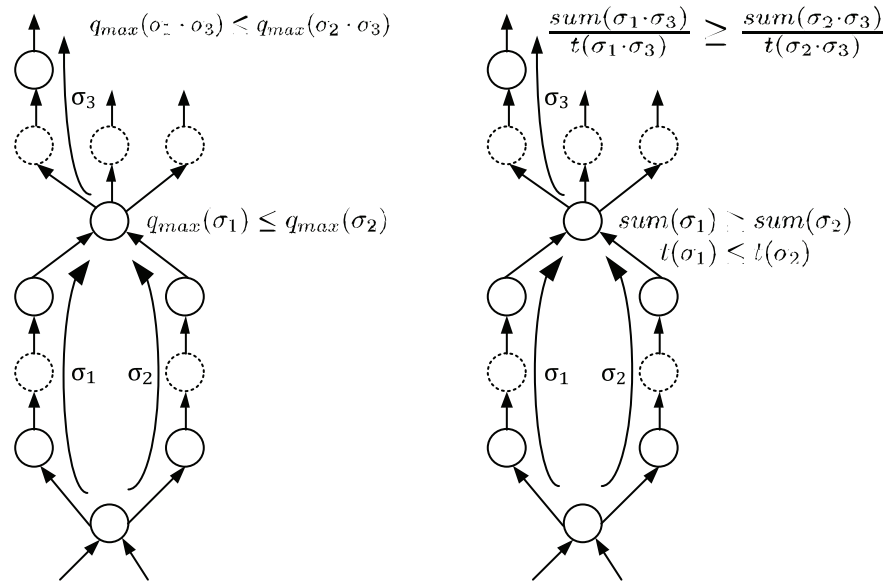
Given two finite (partial) executions σ_1 and σ_2 that start from the same state s_1 and end in the same state s_2, if $q_{max}(\sigma_1) \leq q_{max}(\sigma_2)$ for a MAX quantity q_{max}, then for any execution $\sigma_b = \sigma_2 \cdot \sigma_3$ execution $\sigma_a = \sigma_1 \cdot \sigma_3$ has $q_{max}(\sigma_a) \leq q_{max}(\sigma_b)$.

Proof: Since $q_{max}(\sigma_1 \cdot \sigma_3) = \max\big(q_{max}(\sigma_1), q_{max}(\sigma_3)\big) \leq$

$$\max\big(q_{max}(\sigma_2), q_{max}(\sigma_3)\big) = q_{max}(\sigma_2 \cdot \sigma_3),$$

it follows that $q(\sigma_a) \leq q(\sigma_b)$ AVG quantities do not have the monotonicity property of MAX quantities. So which execution ending in the same state is better may depend on the future execution sequence. In order to decide locally

at any given state whether an execution is guaranteed to be better, we have to use more strict conditions than those for MAX quantities. The corresponding proposition is illustrated in Figure 4.6 (b).



(a) MAX Quantity (b) AVG Quantity

Figure 4.6 Pruning redundant executions in a state space.

Proposition 4.2:

Given two finite (partial) executions σ_1 and σ_2 that start in the same state s_1 and end in the same state s_2. If $\sum_{i=0}^{|\sigma_1|} q(\sigma_1(i)) \geq \sum_{i=0}^{|\sigma_2|} q(\sigma_2(i))$ and $t(\sigma_1) \leq t(\sigma_2)$ for an AVG quantity q, then for any execution $\sigma_b = \sigma_2 \cdot \sigma_3$, execution $\sigma_a = \sigma_1 \cdot \sigma_3$ has $q(\sigma_a) \geq q(\sigma_b)$.

Proof: Let $m_1 = \sum_{i=0}^{|\sigma_1|} q(\sigma_1(i))$ and $m_2 = \sum_{i=0}^{|\sigma_2|} q(\sigma_2(i))$.

Then $q(\sigma_a) = \lim_{N \to |\sigma_3|} \frac{m_1 + \sum_{i=0}^{N} q(\sigma_3(i))}{t(\sigma_1) + t(\sigma_3^N)} \geq \lim_{N \to |\sigma_3|} \frac{m_2 + \sum_{i=0}^{N} q(\sigma_3(i))}{t(\sigma_2) + t(\sigma_3^N)} = q(\sigma_b)$. ■

The above two propositions can be used for pruning redundant executions.

When considering MAX and AVG quantities simultaneously for searching Pareto optimal executions, the conditions given by Prop. 4.1 and Prop. 4.2 have

to be satisfied at the same time in order to discontinue the exploration of partial execution σ_2.

We show that exploring simple executions is enough to obtain Pareto-optimal trade-offs with one AVG property and multiple MAX properties in a design space since the trade-offs obtained by non-simple executions are always dominated by simple executions. If there is more than one AVG property, the simple executions cannot cover all the trade-offs, since alternating execution of two simple executions affects the averages and thus may lead to extra Pareto-optimal points. Note that with multiple AVG quantities, trade-offs between these quantities can be achieved if two different schedules with different values for AVG quantities can be alternatively applied in arbitrary ratios.

Proposition 4.3:

Given an arbitrary infinite execution σ of an RASDF graph, $\sigma = s_0 s_1 s_2 \cdots$, and a metric space consisting of one AVG quantity q and an arbitrary number of MAX quantities $p_1, p_2, \cdots, p_m$. Then there exists a simple execution σ_s that dominates σ in this metric space.

Proof: The states in σ can be divided into two sets: S_T (transient set) and S_R (revisit set), such that states in S_T are only visited finitely often while the states in S_R are visited infinitely often. As the number of states in S_T is finite, after a finite length of execution, σ_{pre}, new state transitions only happen in S_R. The infinite path through states of S_R essentially consists of (possibly nested) repeated visits of simple cycles in the state space. As the number of the states in S_R is also finite, the number of different simple cycles is also finite. Assume we find M_N simple cycles for $\sigma_N = s_0 s_1 s_2 \cdots s_N$ after N transitions. From the property of AVG, we have:

$$q(\sigma) = lim_{N \to \infty} \frac{M_{N,1} \cdot q_{avg}(\sigma_{c_1}) + M_{N,2} \cdot q_{avg}(\sigma_{c_2}) + \cdots + M_{N,k} \cdot q_{avg}(\sigma_{c_k})}{M_{N,1} + M_{N,2} + \cdots + M_{N,k}}$$

where $\sum_{i=1}^{k} M_{N,i} = M_N$ and $M_{N,i}$ is the number of complete visits of simple cycle σ_{c_i} after N states.

So $q(\sigma) \leq q_{max}(\sigma)$, where $q_{max}(\sigma) = max\left(q_{avg}(\sigma_{pre}), q_{avg}(\sigma_{c_1}), \cdots, q_{avg}(\sigma_{c_k})\right)$. Let $\sigma_s = \sigma_{pre1} \cdot \sigma_{cmax}^{\omega}$ be an execution such that σ_{pre1} is a prefix of σ, which eventually visits simple cycle c_{max} infinitely often, where c_{max} is the cycle with

maximum property value $q_{max}(\sigma)$. Then $q_{max}(\sigma_{c_{max}}) = q_{max}(\sigma_s)$ and $q_{max}(\sigma_s) \geq q_{max}(\sigma)$.

Furthermore, $\mathrm{p_i}(\sigma_s) = \max\left(p_i(\sigma_{pre1}), p_i(\sigma_{c_{max}})\right) \leq p_i(\sigma)$, where $p_i(\sigma) = max\ \left(p_i(\sigma_{pre}), p_i(\sigma_{c_1}), \cdots, p_i(\sigma_{c_n})\right)$. So, σ_s dominates σ.■

From Proposition 4.3, we know that we only have to consider simple executions, because for arbitrary executions, we can always find a simple execution that dominates it. So we can use a Depth First Search (DFS) based algorithm to find all simple cycles and use conditions from Propositions 4.1 and 4.2 to prune the search space during exploration.

If we encounter a state which is already on the DFS stack, we have closed a simple cycle and we can analyze the cycle for its AVG quantity, store the result (if not dominated), and back-track. Moreover, if we encounter a state which is not on the DFS stack, but which we have visited before, then we check Pareto dominance of any of the previous visits of the state over the current visit. If it is dominated, we back-track; otherwise, we have to revisit the state. It is easy to see the approach terminates because there is only a finite number of states, thus the number of simple executions are finite.

4.4 Heuristic Search

In this section, we discuss the heuristics that we made for the implementation of our exploration algorithms. It is based on the exploration method proposed in the previous section, but it implements several features to facilitate the exploration of large state spaces. The price to be paid for using these heuristics is (potential) loss of optimality, but the exploration times are reduced.

We use a hash table as the data structure to store the visited states for quick checking. However, to allow exploration of large state spaces efficiently, we cannot only depend on a good data structure. We have to limit the size of the explored state space to keep our tool fast while ensuring that the exploration can find enough interesting design points.

The heuristic options for exploration are listed in Table 4.1. The options are divided into 3 groups: scheduling constraints, resource constraints and exploration algorithm constraints.

Heuristic Category	Option
Scheduling constraints	Limit of stack size
	Limit of number of iterations
	Rules for branch selection
	Limit on number of branches
	Priorities for actors
	Channel quantities bounds
Resource constraints	Resource quantities bounds
Exploration algorithm constraints	Back-track step
	Number of found simple cycles
	Exploration time

Table 4.1 Heuristic options for exploration

In the scheduling constraints category, the length of the resulting schedules is often an important design constraint for embedded systems. The constraint on stack size will limit the depth of the DFS algorithm and thereby constrains the length of schedules. Similarly, for an RASDF graph with repetition vector γ, by limiting the number of iterations to n, the schedule length, i.e., the number of actor firings, is no longer than $n\sum_{a\in A}\gamma(a)$.

The rules for branch selection and the maximal number of explored branches limit the number of branches explored during DFS. We implemented a rule to guide the exploration. We use a rule to guide the algorithm to select branches based on a cost function which is computed from the repetition vector γ of the RASDF graph. The cost of a decision d is $C(d) = \sum_{a\in A} d_a c_a$, with d_a the number of firing starts of actor a in decision d and $c_a = f_a/\gamma(a)$ the cost of one firing of actor a, where f_a is the accumulated firing count of actor a since the start of execution. The larger the cost of one actor, the more firings of one actor have been executed and the higher the selection cost of the actor for the next firing start when there are resource conflicts. The fairness rule can avoid that the algorithm selects actors greedily based on the order in the data structure when exploring the state space partially. By selecting the minimal cost firing, we ensure fairness in actor firings, i.e., the percentage of firings of every actor in one iteration should be balanced during execution. Another scheduling constraint that maybe defined is a priority ordering among actors. When actors compete for the same resource, they will be assigned according to their priority. The search algorithm will only explore options that satisfy the priority order when resource conflicts happen. For actors with the same priority, the selection of firing is decided by the fairness rule. Bounds on

channel quantities constrain the buffer size for each communication channel and can be specified with back edges. This can provide another constraint on the explored executions. By limiting the maximum number of tokens that can be stored in each channel, the number of enabled actor firings is smaller, thus the available schedules will also be limited.

The constraints on resource amounts influence the scheduling of an RASDF graph and results in different throughputs. By setting resources to configurations we are interested in, we can limit the search space and explore the interesting resource region.

The search algorithm itself can also be configured. Since the size of the state space may remain large, we use the back-track step to avoid that the search becomes trapped into some local regions. By doing this, we only explore executions with a specific execution prefix a few times rather than explore all possible executions all with the same specific prefix execution. For example, we can set the back-track step to 5, and the search algorithms will back-track at least 5 states before it starts forward exploration again.

Each simple execution corresponds to a design point in the design space. A time limit for one exploration with a fixed resource constraint is a natural termination condition for designers to control the time budget of the exploration. The advantage of a time limit is that it is easy to estimate the total exploration time for a given number of explorations while the exploration time for other constraints are graph dependent.

In experiments, it turns out that the exploration results are sensitive to the selected searching options and resource bounds. To ensure we cover the interesting range of the design space, we use a *grid search* strategy to explore the combinations of these heuristic search options and resource bounds, i.e., so-called configurations. Each configuration is treated as one grid point in the design space, and the design space exploration algorithm iterates every grid point. We assign each configuration some exploration time T_p and explore all possible N_p configurations on a multiprocessor system with N_{proc} processors since each search is independent from each other search and can be executed in parallel. This strategy has two advantages. First, it can be distributed on a parallel system and thus allows parallel search. Secondly, the total exploration time $T_t = \frac{(T_p+T_{ov})N_p}{N_{proc}}$ where T_{ov} is the overhead for each configuration, which is very useful for designers to estimate the total exploration time needed. For

large state spaces, the overhead T_{ov} can be omitted as the exploration time per configuration dominates.

We show results in the experimental evaluation of Section 4.6. In the next section, we discuss how to use the knowledge that we find during exploration to accelerate the exploration process.

4.5 Bottleneck-driven Design Space Exploration

In the grid search strategy, the search algorithm iterates over all exploration configurations exhaustively, including all different resource configurations. However, we observe that only few resource configurations are really necessary to be explored for obtaining all trade-offs in the design space. Bottleneck-driven design space exploration tries to explore the design space more efficiently by identifying resource bottlenecks and by guiding the exploration in specific directions, i.e., by only increasing the amount of bottleneck resources rather than increasing the amount of all resources one by one. Compared to exhaustive searching, the bottleneck-driven DSE can reduce the number of configurations considerably.

Although the identification of bottlenecks depends on the problem at hand, the design flow of many bottleneck-driven DSEs can be seen as a specialization of the well-known Y-chart method [98]. In general, given a system design problem, design alternatives of the application and the architecture are represented via a set of parameters, and the metrics of interest are defined. Next the metrics are evaluated and the bottleneck parameters are identified. With this information, the system parameters are tuned. By iterating these steps, optimal parameter settings and metrics can be found in the design space.

For a streaming system, application, architecture or mapping are sometimes fixed beforehand and only the amount of resources can be tuned for performance. Developers have to dimension the amounts of resources for efficient resource usage while meeting performance constraints.

Figure 4.7 shows an overview of the flow for DSE of systems modeled by RASDF. The flow can be divided into 4 steps. First, we capture the system with an RASDF graph (1. **specification**). Then we execute the model and evaluate its throughput (performance metric) from its state space (2. **performance evaluation**). Through the analysis of the state space, we construct the data and resource dependencies between actors (to be explained later) and identify the

potential bottlenecks of the system (3. **bottleneck identification**). Then we increase the amount of identified bottleneck resources and iterate the above steps to obtain the profile of the design space of the given system (4. **dimensioning and tuning**). For example, for resource configuration < 1,1,9 >, we deduce that the actors c and d depend on the availability of Mem in the $critical\ path$. Given that Mem is a potential bottleneck, we increase the amount of Mem to generate a new configuration < 1,1,10 > which potentially has a better performance. We apply the new configuration to the example and repeat the above procedure automatically until the design space of the example RASDF graph has been explored. We explain the details of the techniques in the following text.

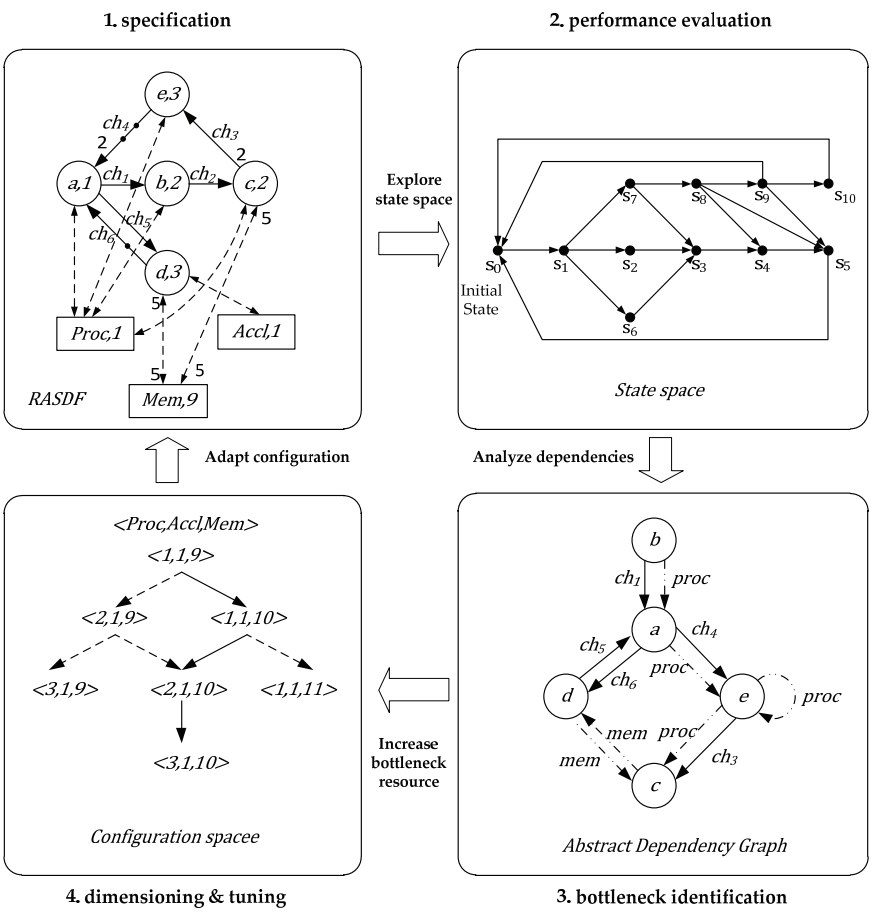


Figure 4.7 Bottleneck-driven DSE for RASDF

The maximal throughput of a system may be limited by the amount of available resources (e.g. the number of processors, the size of memory, the bandwidth of a bus). In [144, 148], techniques based on a dependency graph are introduced to capture the dependencies on channel capacities between actor firings of an SDF graph. The dependencies are used to analyze the bottleneck in channel capacities. In this thesis, we adapt those techniques to find bottleneck resources in RASDF graphs. In an execution, for example, one or more actors may not be able to start firing while waiting for the other running actors to end firing and release resources, so that they can claim the released resources. Increasing the amount of resources may enable the waiting actors to start firing earlier and possibly increase the throughput. We would like to detect such situations as indications of a potential bottleneck. The dependency of the start of firing of an actor on a resource released or tokens produced by the end of firing of another actor is called a *causal dependency*.

Definition 4.7: Causal Dependency

A firing of actor a causally depends on a firing of actor b if and only if the firing of a claims resources or consumes tokens that are released respectively produced by the firing end of b without any time progress between the firing start of a and the firing end of b.

The causal dependencies can be classified into two types, *data dependencies* and *resource dependencies*.

Definition 4.8: Data Dependency

A causal dependency caused by producing and consuming tokens on channel $c \in C$ is a data dependency, denoted by δ_c, or $\delta_c(x, y)$ to make the involved firings of actor x and y explicit.

Definition 4.9: Resource Dependency

A causal dependency caused by releasing and claiming resource $r \in R$ is a resource dependency, denoted by δ_r or $\delta_r(x, y)$ to make the involved firings of actors x and y explicit.

A data dependency only exists between two consecutive actor firings of actors that are connected to the same channel, and can thus be easily detected by checking $start$ and end transitions of the pair of actors during execution. If

the tokens produced by the source actor are immediately consumed by the destination actor, there is a data dependency between the two actors.

Compared to data-dependency detection, the detection of a resource dependency depends on how the resource is shared. If it is only shared between two actors, it can be detected very easily. For a resource shared by more than two actors, the resource dependencies cannot be easily detected as the released resource tokens have no information about the actors that release them. Instead, we assume that dependencies exist among all the actors that produce tokens and the actors that consume tokens.

Given a resource $r \in R$ that is shared by more than two actors and a time instant t, the producer set A_p contains firings that end and produce r at t and the consumer set A_c contains firings that start and consume r at t. Without losing any resource dependencies, we assume that the resource dependency exists between every firing in A_p and every firing in A_c if and only if the available amount of resource r, i.e. $R_{avail}(r)$, at the time t before release is less than the total amount of r that is claimed, i.e., $\delta_r(p,c)$ exists for every pair $(p,c) \in A_p \times A_c$ if and only if $R_{avail}(r) < \sum_{c \in A_c} Clm(c,r)$. This assumption simplifies the detection of resource dependencies, but it also introduces false positive dependencies for actor firings that only use already available resources, as discussed later in detail.

Definition 4.10: Causal Dependency Graph

Given a simple execution $\sigma = \sigma_{pre} \cdot \sigma_{per}^{\omega}$ of an RASDF graph with repetition vector q and the periodic execution part σ_{per} that contains n iterations of the graph, the causal dependency graph $G = (D, E)$ contains a node a_i $(0 < i \leq n \cdot q(a))$, for the i-th firing a_i of the actor $a \in A$ in an iteration of σ_{per}; the set of dependency edges E contains an edge from a_i $(0 < i \leq n \cdot q(a))$ to b_j $(0 < j \leq n \cdot q(b))$ denoted by $\delta_c(a_i, b_j)$ or $\delta_r(a_i, b_j)$ if and only if there exists a causal dependency for channel c or resource r between firings a_k and b_l in σ_{per}^{ω} such that $i = k \bmod n \cdot q(a))$ and $j = l \bmod n \cdot q(b))$.

Figure 4.8 shows the process of building the dependency graph for an execution $\sigma = (s_0 s_1 s_2 s_3 s_4 s_5)^{\omega}$ of the RASDF graph in Figure 4.3. It shows the resource allocations of the execution σ in the top part of the figure while the dependency between firings is shown in the bottom part of the figure. At state s_1, actor a finishes its first firing a_1, outputs tokens to channels ch_1 and ch_5 and

releases resource $Proc$. Actors b and d start their first firings b_1, d_1 and consume tokens from ch_1 and ch_5 and b claims resource $proc$ when they start firing at state s_1. So we have dependency relations among actor firings a_1, b_1, d_1 which are shown below state s_1 in Figure 4.8 as part of the causal dependency graph. Similarly, at state s_2, actor b outputs tokens and releases $proc$; but no actors need the tokens and resource $proc$ immediately, so no dependency exists at state s_2. We repeat checking dependencies every state until we reach recurrent state s_0. We can build the causal dependency graph for the execution σ by collecting the dependencies from Figure 4.8 as shown in Figure 4.9 (a).

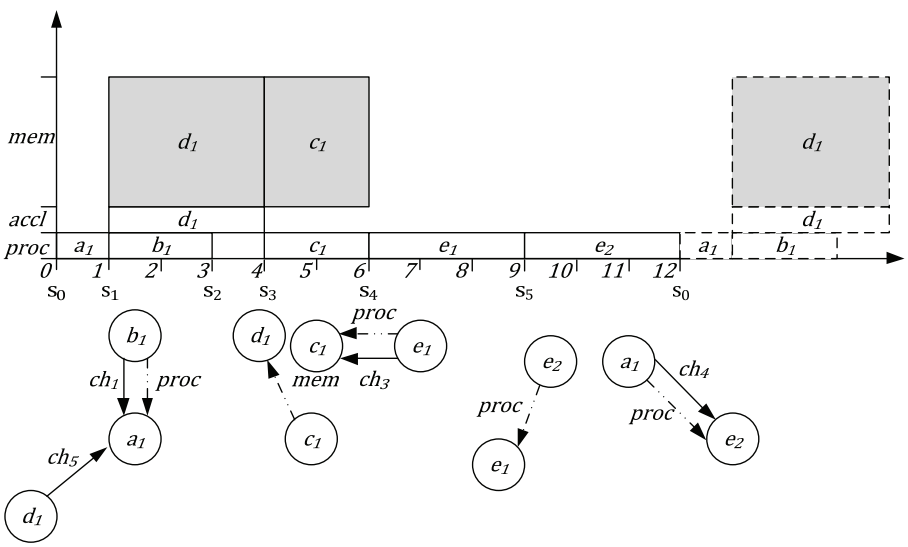
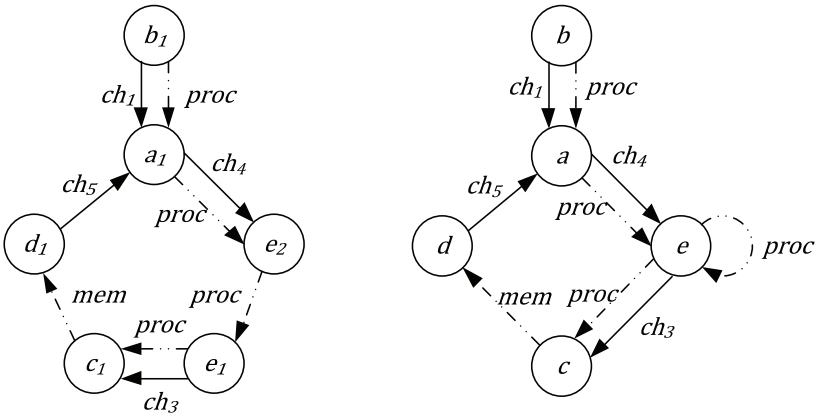


Figure 4.8 Dependency detection for an execution

The size of the dependency graph can be very large. For example, the minimal number of actor firings in one period of the sample rate graph of [16] is 612. Therefore, we use an abstraction of the dependency graph to capture all dependencies between the firings in σ_{per}. Figure 4.9 (b) shows the abstract dependency graph of Figure 4.9 (a). The transformation from dependency graph to abstract dependency graph maps actor firings to actors and maps the dependencies between firings to dependencies between actors. Since the abstract dependency graph does not discern firings, the nodes of the same actor in the dependency graph merge into one node in the abstract dependency graph while keeping their dependency relations with other nodes. The dependencies between firings of the same actor are kept by self edges. For

example, the dependency between e_1 and e_2 in Figure 4.9 (a) turns into the self-edge in Figure 4.9 (b).



(a) Dependency Graph (b) Abstract Dependency Graph

Figure 4.9 Causal Dependency Graph and Abstract Dependency Graph

Definition 4.11: Abstract Dependency Graph

Given a causal dependency graph (D, E), the abstract dependency graph (D_a, E_a) contains a node $a \in D_a$ for each actor $a \in A$ and a dependency edge $\delta(a, b) \in E_a$ between actor a and b for each dependency edge $\delta(a_k, b_l) \in E$.

Dependencies between actor firings in the periodic part σ_{per} of a simple execution σ can form cyclic dependencies, called a *causal dependency cycles;* data dependencies cannot be changed. In this example, the cycle indicates that both mem and $proc$ are potential bottlenecks. By increasing these resources, the dependency cycle may disappear.

Definition 4.12: Causal Dependency Cycle

A causal dependency cycle is a simple cycle in the dependency graph.

The throughput of a simple execution is limited by some of those cycles in the dependency graph. For example, the throughput of the execution in Figure 4.8 is limited by the dependency cycle, i.e., $\delta_{proc}(a_1, e_2) \cdot \delta_{proc}(e_2, e_1) \cdot \delta_{proc}(e_1, c_1) \cdot \delta_{mem}(c_1, d_1) \cdot \delta_{ch_5}(d_1, a_1)$ in Figure 4.9(a), that contains $\delta_{mem}(c_1, d_1)$. If a resource dependency δ_r appears in a causal dependency cycle, for some $r \in R$, the throughput may increase if we can remove the dependency

by increasing the amount of resource r, e.g. the increase of mem allows d and c to fire at the same time. Resource dependencies that do not occur in cycles are not critical to the performance. For example, $\delta_{proc}(b_1, a_1)$ is not on a dependency cycle in Figure 4.9 (a) and b_1 is not critical for the execution. If we delay b_1 one time unit, the throughput remains the same.

Definition 4.13: Bottleneck

A resource $r \in R$ in an RASDF graph is a bottleneck of execution σ under resource configuration ρ if and only if increasing r in resource configuration ρ is needed for an increase of the throughput of the graph.

By construction, any dependency cycle in the dependency graph gives a dependency cycle in the abstract graph. Note however, that a simple cycle need not remain simple. For example, the simple cycle in Figure 4.9 (a), i.e., $a_1 e_2 e_1 c_1 d_1$ corresponds to two cycles in Figure 4.9 (b), i.e., $aecd$ and e due to the merge of the two firings of actor e. Dependency edges related to bottleneck resources thus appear at least once in a dependency cycle of the abstract dependency graph. Therefore, the abstract dependency graph contains sufficient information to identify potential bottlenecks (step 3 in Figure 4.7).

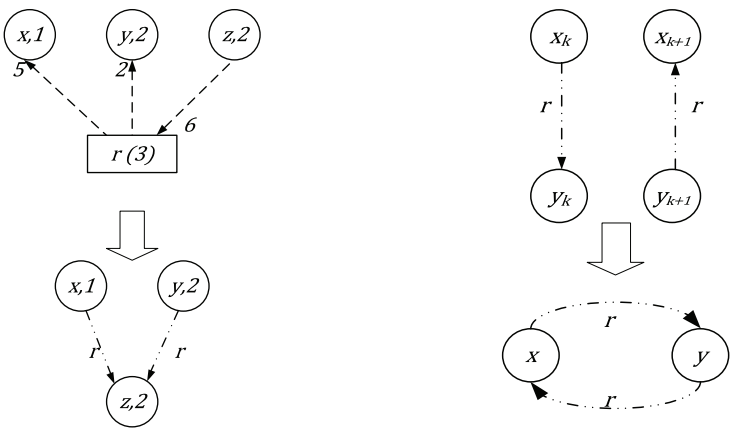
Causal dependencies are defined based on firings. However, an execution may enter into a deadlock state, where no actor is able to fire. We need to refine the causal dependency concept to handle the deadlock case, because lack of resources in the chosen resource configuration may be the cause of the deadlock. With the adapted definition, an abstract dependency graph can be derived as before.

Definition 4.14: Causal Dependency In Deadlock

In a deadlock state, a firing of actor a causally depends on a firing of actor b if an only if the firing of a needs tokens that may be produced by or resources that may be released by a firing of actor b.

The dependency assumption in the shared resource case and the use of the abstract dependency graph instead of the dependency graph can lead to false positive dependencies detected during exploration. Figure 4.10 shows two examples of false dependencies. In Figure 4.10 (a), the available amount of resource r is 3 units in a specific state while actor z is running and ready to end its firing. The end of firing z releases 6 units; at the same time, the next two

firings of x and y claim 5 and 2 units of resource r respectively. If firing y uses the available amount of r and x uses the amount of r released by z, then only x depends on z and y can start earlier before z ends. However, since we do not identify the producer of tokens, i.e., whether the tokens come from firing z or are just available resource, we can only assume there is also a dependency between y and z. In Figure 4.10 (b), there is no cycle in the dependency graph, but a false dependency cycle exists in the abstract dependency graph. The false dependencies in the abstract dependency graph can cause some non-bottleneck resources to be detected as bottleneck resources and may lead to redundant exploration (but not to wrong results).



(a) False resource dependency (b) False dependency cycle

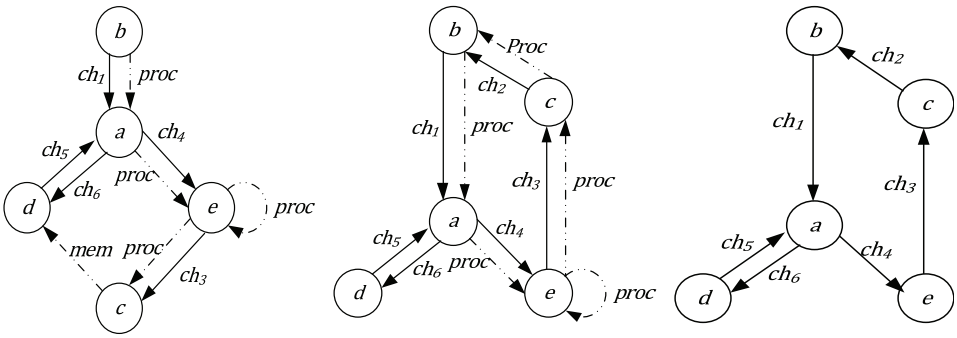
Figure 4.10 False dependencies

Normally, there are multiple simple executions in the state space of an RASDF graph, and each execution may have an abstract dependency graph of its own. Often, bottleneck resources are the same. For efficiency, we build only a single dependency graph for the entire state-space exploration that merges all dependency graphs into one graph by adding dependencies whenever cycles are encountered anywhere in the statespace. This may again lead to false dependency cycles. Again, real bottleneck resource dependencies are always detected. By tolerating those false detections, we keep the exploration to not lose real resource bottlenecks while making it more efficient.

Proposition 4.4:

If a resource r is a bottleneck, then the abstract dependency graph has a dependency cycle containing a resource dependency for r.

Proof: If resource r is a bottleneck as defined in Definition 4.12, there is at least one resource dependency edge between two firings in a dependency cycle. If not, then throughput is fully determined by data cycles and cannot be improved anymore. We assume the two firings in the dependency cycle are a_k and b_l. So there exist two paths from a_k to b_l and from b_l to a_k. Obviously, in the abstract dependency graph, there exist two paths from a to b and from b to a. So a resource dependency on r also exists in a dependency cycle of the abstract dependency graph. Since a cycle in the dependency graph is preserved in the abstract dependency graph, the resource dependency edge of dependency graph is also preserved in the abstract dependency graph.



(a) $\rho =< 1,1,9 >$ (b) $\rho =< 1,1,10 >$ (c) $\rho =< 3,1,10 >$

Figure 4.11 Bottleneck identification

The bottleneck information for an RASDF graph is embedded in its abstract dependency graph. We can compute the graph's maximal achievable throughput without resource limitations efficiently from its strongly connected components through state space analysis using techniques of [64]. In [144, 148], channel-buffer capacity bottlenecks of an SDF graph are detected through an abstract dependency graph and used to guide the exploration of buffer configurations to find optimal buffer sizes. Similarly, we also use the inferred bottleneck information to increase the amount of relevant resources until the graph reaches the maximal throughput or the maximal resource configuration that we are willing to explore. For example, in the abstract dependency graph of Figure 4.11 (a), Mem and $Proc$ are identified as potential bottlenecks. We choose to increase Mem and raise its amount from 9 to 10. The Mem dependency edge disappears and only $Proc$ dependency edges still exist in the new abstract dependency graph shown in Figure 4.11 (b). When increasing $Proc$ from 1 to 3, all resource dependencies disappear and the maximal

throughput is reached. The corresponding abstract dependency graph is shown in Figure 4.11 (c).

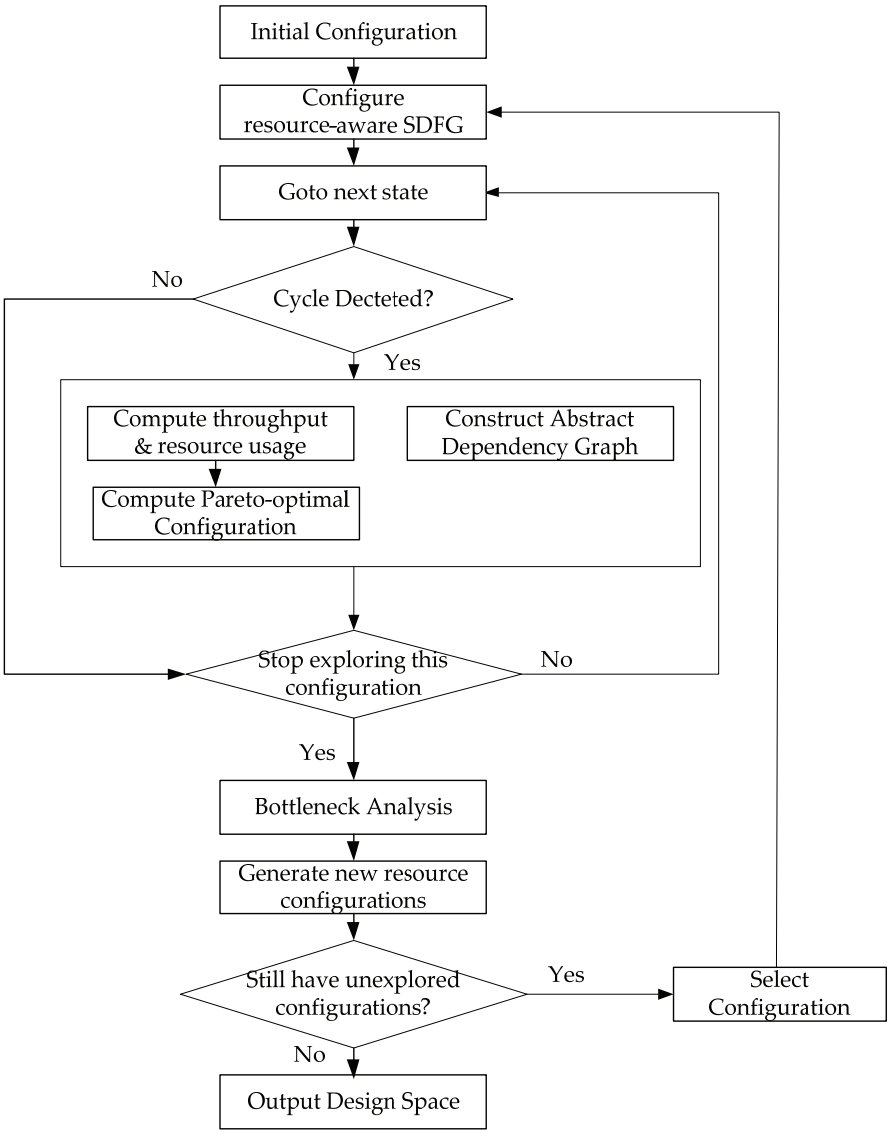


Figure 4.12 Bottleneck-driven DSE flow

Figure 4.12 shows the DSE algorithm. It uses an initial resource configuration to configure an RASDF graph and explores the state space of the configured graph. Upon detecting a cycle in the state space, throughput and resource usage of the execution are computed. Pareto-optimal design points among all explorations are kept as the output of the DSE. When detecting a cycle, causal dependencies are added to the abstract dependency graph. As multiple cycles can exist in the state space, the abstract dependency graph is complete only after the state-space exploration stops. Bottleneck analysis is then performed by identifying resource dependencies in strongly connected components of the abstract dependency graph. Identified bottleneck resources are each increased by a minimal step, specified by the user. The reason to set the step size by user is because a minimal amount of resource increase may be very small and may lead to a large exploration time since the bottleneck resource has to be increased multiple times. Thus, a new set of unexplored configurations is generated and pushed into the configuration queue. Breadth-first search is used to search the configuration space. Dynamic programming is used to avoid redundant explorations of configurations that have already been explored. The algorithm terminates if the configuration queue is empty.

4.6 Case Studies

To evaluate the analysis techniques that we introduced in this chapter, we experiment with a set of DSP, multimedia and printer datapath models on an Intel 2.2 GHz Core™ 2 with 4GB RAM Desktop PC. We first compare our trade-off analysis techniques to existing techniques that cannot handle resource sharing among multiple actors. The purpose of the experiment is to show that our analysis allows efficient use of memory and enables better trade-off points. We also include a case study about analyzing multiple use cases on a given printer architecture with shared resources. Then the trade-offs of the printer architecture for a specific use case are shown. We do experiments to compare the exploration times for analysis without and with bottleneck-aware techniques. Finally, a comparison between genetic algorithm based DSE and bottleneck-driven DSE is discussed. The experiment graphs are divided into two categories: the graphs of the first category (see the appendix) are from the literature and include an example graph from [144], an artificial bipartite graph from [16], a modem [16], a satellite receiver [131], a sample rate converter [16], an MP3 decoder [144] and an H.263 decoder [144]; the graphs of the second category are from an industrial case study provided by Océ (www.oce.com)

where the design space of the digital datapath of a professional printer is explored.

For each graph, we explore the trade-offs between throughput and buffer size requirement. The search parameters of our algorithm are set as follows. For most streaming applications, we want to have a short schedule, i.e., cycles in the state space, only after a few iterations. So we set the range of the iteration number from 1 to 3. For DFS, the number of branches and the backtrack step directly impact the exploration time. The large number of branches makes exploration only capable of exploring a small part of the space. We therefore set the number of branches from 2 to 3 and the backtrack step range from 1 to 2. The fairness rule is used to avoid firing the same actor greedily. The memory scan range is from the lower bound of [57] (lowest possible memory for an execution under a shared memory assumption) to the upper bound of [144] (highest possible memory for maximal throughput under distributed memory assumption) and is uniformly divided into a few steps (we use 10) for our grid search. The time budget for each exploration is 1 second for the first part of the experiment and 60 seconds for the second part. So the total exploration time for an RASDF graph is controlled to less than an hour (the worst possible exploration time is $2 \times 2 \times 10 \times 1min = 40min$). Though we cannot compare the results with the optimal results with shared memory, as they are not known, we compare our results with the experimental results when exploring longer (60s) for each configuration.

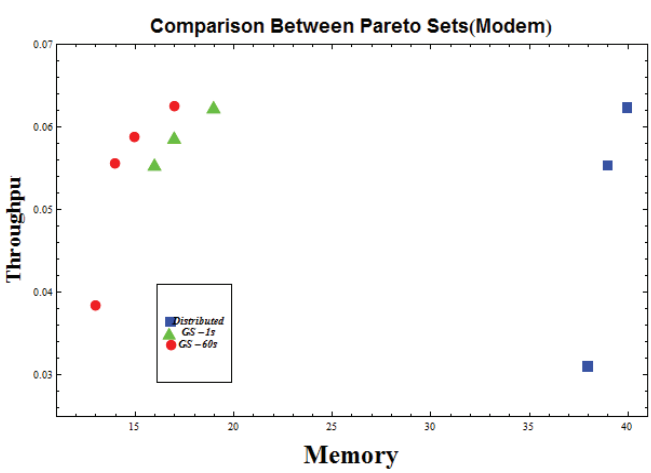


Figure 4.13 Pareto points of Modem

For example, Figure 4.13 shows the Pareto points of the Modem graph. The blue squares are Pareto points found by [144] for distributed buffers. The green and red points are Pareto points found by our algorithm with different time limits, 1 second and 60 seconds respectively. The comparison of the results shows that the results can be improved for some graphs by using a longer exploration time. However, the total time spent on the exploration is increasing very quickly.

In order to quantity the difference between two results, we define the average memory reduction MR_{avg} as a metric to compare a Pareto set S_{new} (our result) with a reference Pareto set S_{ref} (the optimal Pareto points for the distributed memory model found by the algorithm in [144]). The memory reduction for each reference point $r \in S_{ref}$ is the maximal memory reduction of its counterpart $a \in S_{new}$ which has throughput $Th(a)$ not less than throughput $Th(r)$.

$$MR_{avg} = \frac{1}{|S_{ref}|} \sum_{r \in S_{ref}} \max_{a \in S_{new}} d(r, a)$$

where $|S_{ref}|$ is the number of points in the set S_{ref} and

$$d(r, a) = \begin{cases} (mem(r) - mem(a))/mem(r), & Th(r) \leq Th(a) \\ 0 \quad , & Th(r) > Th(a) \end{cases}$$

For example, in Figure 4.13, the set of trade-off points found by grid search (time limit 1 second) is $\{a_1 = (16,0.0556), a_2 = (17,0.0588), a_3 = (19,0.0625)\}$ while the reference trade-off points set for distributed case are $\{r_1 = (38,0.0313), r_2 = (39, 0.0556), r_3 = (40,0.0625)\}$.

$$d(r_1, a_1) = \frac{38-16}{38} = 57.9\%, d(r_1, a_2) = \frac{38-17}{38} = 55.3\%, d(r_1, a_3) = \frac{38-19}{38} = 50\%$$

$$d(r_2, a_1) = \frac{39-16}{23} = 59\%, d(r_2, a_2) = \frac{39-17}{39} = 56.4\%, d(r_2, a_3) = \frac{39-19}{39} = 51.2\%$$

$$d(r_3, a_1) = 0, d(r_3, a_2) = 0, d(r_3, a_3) = \frac{40-19}{40} = 52.5\%$$

We use the average of the maximal reduction of each reference point to compute the average memory reduction $MR_{avg} = \frac{57.9\%+59\%+52.5\%}{3} = 56.5\%$.

To investigate the impact of sharing resources, we compare our results with [144] which is known to be optimal when memory cannot be shared. The

results of the experiment are summarized in Table 4.2. It shows the number of actors and channels in each graph (2nd row), the minimal buffer space (4th row) for the smallest positive throughput (3rd row) that can be achieved, i.e., buffer sizes that make the graph deadlock free). It also shows the minimal buffer space (6th row) for the highest possible throughput (5th row). It also shows the number of Pareto points (7th row), the execution time of the tools (8th row) and the max (9th row), min (10th row), average (11th row) and standard deviation (12th row) of memory reductions achieved by sharing memory. The 60 seconds results are shown in parentheses if they are different from the 1 second results. The results in Table 4.2 show that by sharing memory among actors, the required memory can be reduced by 3% to 50%. The average 3% memory reduction is obtained for the H263(QCIF) case, shown in Figure 4.14. Since the shared memory case only has a low memory reduction when compared to a lot of points of the distributed memory case around the upper right corner of Figure 4.14, it leads to low average memory reduction. The fact that minimally obtained resource reduction is positive in all cases shows that for the experiment we can always achieve the same throughput as the throughput found by [144]. Although 60 seconds results are sometimes better than 1 second results, a 60 seconds budget needs much longer overall analysis times.

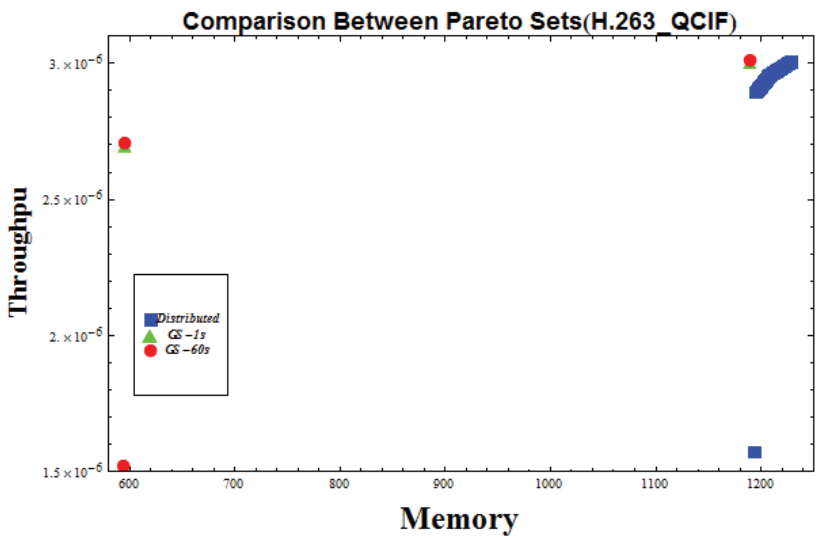


Figure 4.14 Pareto points of H.263(QCIF)

	Example	Bipartite	Sample Rate	Modem	Satellite	MP3	H.26(QCIF)
Actors/channels	3/2	4/4	6/5	16/19	22/26	13/12	4/3
Min. Thr	1.25×10^{-1}	3.09×10^{-3}	$1.00(1.02)\times10^{-3}$	5.56×10^{-2}	7.60×10^{-4}	1.90×10^{-7}	1.52×10^{-6}
Min. Buffer Size	4	26	23	16(13)	962	11	595
Max. Thr	2.50×10^{-1}	3.97×10^{-3}	1.04×10^{-3}	6.25×10^{-2}	9.47×10^{-4}	2.68×10^{-7}	3.01×10^{-6}
Min. Buffer Size	7	32	31	19(17)	1220	14	1190
Pareto Points	4	7	7(5)	3(4)	3	3	3
Exec. Time(min)	0.45(0.45)	1.17(2.55)	5.52(147)	2.34(24.7)	5.56(147)	5.18(95.5)	2.22(18.6)
Max. Mem Reduction	22.2%	13.3%	30.3%	59%(65.8%)	37.7%	50.0%	50.1%
Min. Mem Reduction	10.0%	7.1%	8.8%(28.2%)	52.5%(57.5%)	21.0%	46.2%	0.5%
Avg. Mem Reduction	14.9%	10.7%	22.4%(29.2%)	56.5%(61.6%)	29.4%	48.1%	3.2%
Std. Deviation	0.05	0.02	0.10(0.01)	0.02(0.03)	0.08	0.02	0.08

Table 4.2 Experimental results of graphs from literature

Use Case No	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Actor/channels	5/6	12/15	9/9	3/3	3/3	8/8	9/9	11/13	14/19	14/19	5/6	8/8
Execution time(s)	0.876	0.846	54.994	0.365	0.297	0.254	0.257	0.304	6.587	0.306	0.385	0.244
State count	2204	1888	12651	854	422	10	11	383	3470	346	946	10

Table 4.3 Analysis of printer use cases

The substantial memory reduction and obtained throughput together with the analysis efficiency indicate that our technique performs well. In Table 4.2, we also notice that the H.263 decoder has low average reduction. The reason for this is due to the fact that the large number of Pareto points found by the algorithm of [144] are dominated by the nearby Pareto point found by our tool by a small margin while the single, lower throughput point is improved quite a lot. The average memory reduction metric defined above does not capture this situation very well. The execution time of our method is reasonable. The Pareto points found by our tool are more resource efficient compared to the Pareto points found by [144] which does not allow sharing memory among channels.

In the printer case study, the processing units of the data path share memory and the memory bus. Twelve use cases such as print, scan and copy which are frequently seen in daily printer use are investigated. The models cannot be disclosed in detail due to confidentiality reasons. In [87], some simplified printer data path use cases are analyzed. An example modeled as an SDF graph is shown in Figure 4.15. The use case captures a loop to print a document from a data store. It performs two image processing steps (IP1 and IP2) and several USB and control actions. The printer architecture model is shown in Figure 4.17. We model the twelve considered use cases as RASDF graphs and analyze them with our tool.

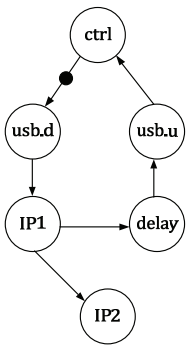


Figure 4.15 ProcessFromStore use case from [87]

The first set of experiments considers single use-case analysis for one specific architecture configuration, so in this particular case, we are not looking for trade-offs, but only to evaluate metrics in a particular design solution. The metrics we are interested in are the peak and average usage of resource in the data path and the throughput of the datapath for those use cases. From the

previous section, we know that the throughput and average resource usage can be easily computed from the prefix and periodic part of the execution. Table 4.3 shows the execution time of our algorithm and the number of states of the execution. For most of the use cases the execution time of the algorithm is less than 1 second. The two exceptions are use cases with actors that have large but slightly different execution times. For these two cases, for a number of iterations, the states are different slightly. It causes a periodic execution phase with a large number of states and a longer execution time for analysis.

The second set of experiments concerns the design-space exploration of printer architectures. We study the trade-offs among peak memory and bandwidth usage with performance (throughput), obtained by different schedules. By using our grid search method, we can get a profile of the design space of a specific architecture, which can help a system designer make decisions on questions like how much memory and how much bandwidth are needed for some specific performance requirements. Figure 4.16 shows the normalized 3-dimensional Pareto space in the design space of a particular use case for some platform configuration, the trade-off points are throughputs under two configurable resources: Bandwidth and Memory. The figure clearly shows how the change of resources impacts the performance of the printer. The results of this case study show that our tool is sufficiently flexible to support design space exploration: it allows us to explore the trade-offs between several objectives.

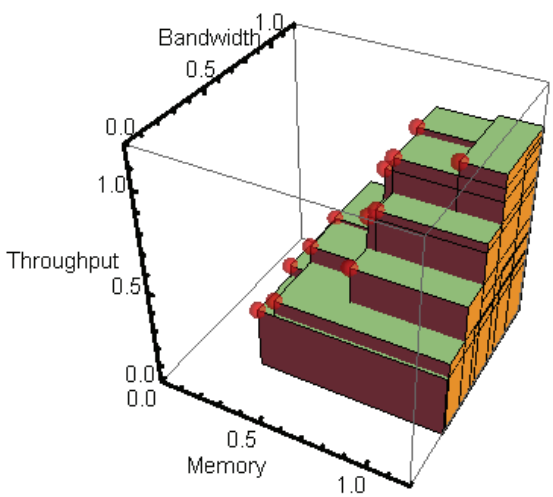


Figure 4.16 Normalized design space of a printer use case

To evaluate the efficiency of bottleneck-driven DSE, we compare our bottleneck-driven implementation with the regular grid search implementation without bottleneck information. Through bottleneck analysis, we expect that many grid points do not need to be searched by the bottleneck-driven DSE, and the exploration times are reduced.

In the first experiment, we compare the execution times of the approaches on the six RASDF graphs in the first category. For each graph, we explore the trade-offs between throughput and the size of the shared memory. As the state spaces of RASDF graphs are often large, we also enforce some resource limits to confine the search. The exploration options are the same as the grid search without bottleneck-driven guidance. The column for each graph in Table 4.4 gives execution times and the number of explored configurations for both the bottleneck-driven search (BD) and full grid search (Non-BD).

The results show that bottleneck-driven DSE has two effects on the execution time. On the one hand, it avoids the exploration of some unnecessary resource configurations. One the other hand, the bottleneck analysis brings some overhead. For *Bipartite* graph, the two approaches explore the same configurations and the number of simple executions for each configuration is very large, so the bottleneck-driven approach is actually worse. For Modem, many unnecessary configurations can be avoided by bottleneck-driven DSE and the analysis overhead is more than compensated by the reduction in configurations. The other graphs give results in between those two extremes.

To test the bottleneck analysis for multiple resources and large configuration spaces, we did experiments with distributed memory. For MP3, for example, we divided the channels to three different buffers. The right column for each graph in Table 4.4 gives the results. Substantial reductions are obtained in all cases. As expected, the performance of bottleneck-driven DSE improves with an increasing number of resources.

Then we do the printer case study provided by Océ. We aim to dimension the memory and bus usage of printer architectures as shown in Figure 4.17. There are three options: one reference architecture, one with faster processing units, and one with additional processing units (denoted by dash lined blocks).

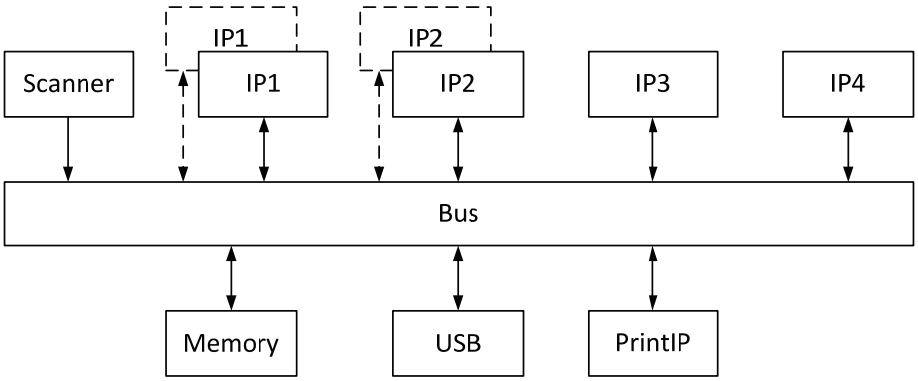
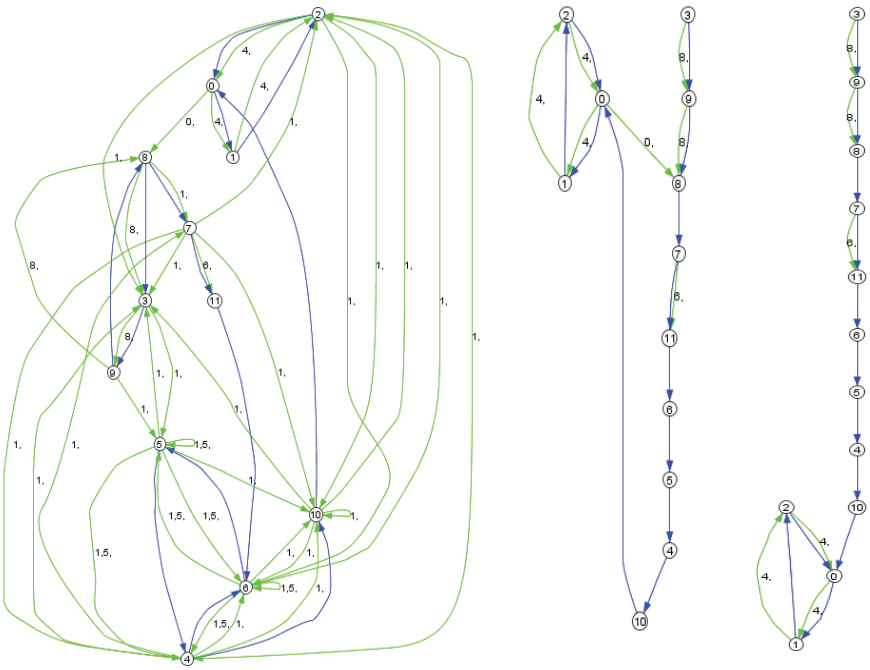


Figure 4.17 Printer architecture for exploration



(a) (b) (c)

Figure 4.18 Abstract dependency graphs during printer architecture exploration

Figure 4.18 shows three abstract dependency graphs visualized by dot (www.graphviz.com). The tasks of the targeted application are denoted by circles annotated with the corresponding task id. The green edges denote resource dependencies and are annotated with the corresponding resource ids while the blue edges denote data dependencies. The figure shows different

bottlenecks that exist at different exploration stages for the printer application on one of the given printer architecture configurations in Figure 4.17. In Figure 4.18 (a), both memory size and bus bandwidth are bottlenecks. When bandwidth is increased, the memory size is the remaining bottleneck resource in Figure 4.18 (b). When memory size and bus bandwidth both are large enough, the scanner of the printer is the new bottleneck of the system.

Table 4.5 shows that bottleneck analysis reduces the number of explored configurations, and even if the overhead for bottleneck analysis is substantial (Arch 3), the overall execution time reduction is still good.

The configuration space of grid search with and without bottleneck analysis for the first printer architecture is shown in Figure 4.19. The (blue) squares are configurations explored without bottleneck analysis. The (green) triangles are configurations explored with bottleneck analysis. Thanks to the bottleneck identification, exploration stops increasing specific resources if they are no longer a potential bottleneck of the system. The (red) circles show the resource usage of the Pareto points found. They do not coincide with grid points because the optimal resource usage may be off the configured grid.

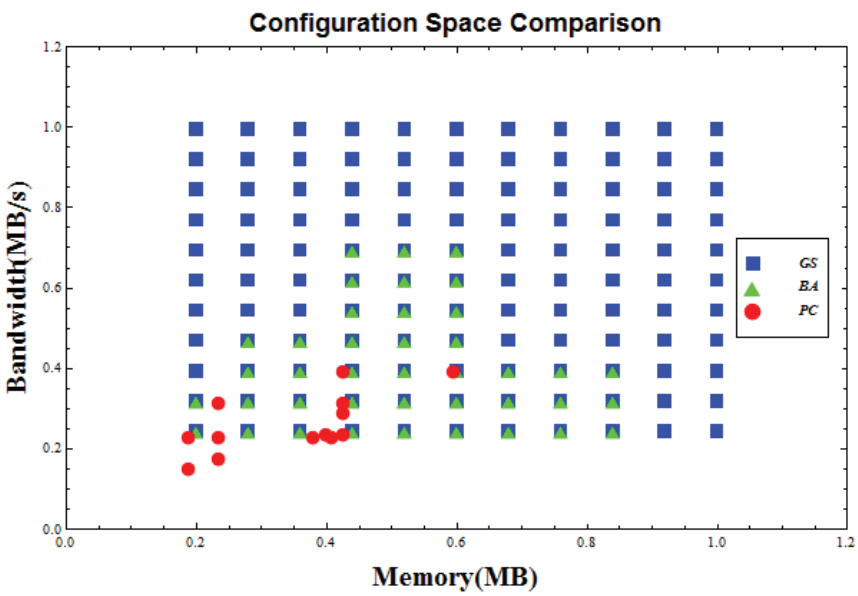


Figure 4.19 Comparison between Configuration Spaces

	Bipartite		Samplerate		Modem		Satellite		MP3		H.263 (QCIF)	
No. of Shared buffers	1	2	1	3	1	2	1	2	1	3	1	2
Conf No. Non-BD	168	286	288	1176	336	125	264	245	408	360	264	45
Exec Time Non-BD(s)	134.3	92.4	523.7	404.8	359.2	83.5	562.2	585.4	577.9	237.8	252.5	85.7
Conf No. BD	168	168	216	181	89	19	90	27	118	89	264	14
Exec Time BD (s)	181.9	86.1	496.5	89.3	116.8	16.0	207.9	62.9	235.3	145.6	259.3	19.5
Conf Reduction	0%	41%	25%	85%	73%	85%	66%	89%	71%	75%	0%	69%
Exec Time Reduction	-35%	6%	5%	78%	67%	80%	63%	89%	59%	39%	-3%	77%

Table 4.4 Exploration algorithm comparison for graphs from literature

	Arch 1	Arch 2	Arch 3
Conf No. Non-BD	110	110	110
Exec Time Non-BD(s)	71.9	128.2	120.0
Conf No. BD	37	44	58
Exec Time BD (s)	40.2	78.3	100.0
Conf Reduction	66%	60%	47%
Exec Time Reduction	44%	39%	20%

Table 4.5 Exploration algorithm comparison for 3 Printer architectures

Finally, in [102], a comparison between our bottleneck-driven approach and DSE using evolutionary algorithms (EA) are discussed. The comparison uses the six multimedia graphs that we used in the first experiment. It uses the Strength Pareto Evolutionary Algorithm [178] (SPEA2) and the Non-dominated Sorting Genetic Algorithm [41] (NSGA-II) tools for tuning exploration parameters while using the same state-based analysis tool for RASDF analysis. It uses a binary string to encode both state-space analysis algorithm parameters (branching number, iteration number etc.) and the resource configuration of the RASDF graph. The DSE is performed through mutation of the encoded information. The comparison is made for the twelve cases also discussed in Table 4.4. In terms of the quality of results measured by the ϵ-indicator (see Section 3.6), in 6 out of 12 cases the bottleneck-driven approach is better than the EA approach, in 5 out of 12 cases the bottleneck-driven and EA approaches give equal or incomparable results, and in one case (Samplerate with a single buffer) the EA approach is better. [102] also makes a comparison on execution times between the bottleneck-driven approach and both SPEA2 and NSGA-II. The bottleneck-driven approach is faster than SPEA2 and NSGA-II approach in many cases. Generally speaking, using the bottleneck-driven approach is thus preferred over the EA approach.

4.7 Related Work

There are many works on analysis of SDF graphs and optimization of their schedules subject to one or more criteria [14, 15, 57, 68, 75, 112, 113, 130, 167]. In order to minimize code size for single processor, Single Appearance Scheduling (SAS) is proposed in [7]. In an SAS, every actor appears only once within a looped schedule. In [14, 15], buffer size minimization techniques are developed while preserving the minimal code size property of SAS. In [112, 113], more efficient buffer usage for SAS are obtained by sharing memory through lifetime analysis of memory tokens. However, SASes are not necessarily optimal when other objectives than code size are to be optimized, such as performance or energy consumption. The single appearance constraint on schedules is relaxed in [167] by exploiting some trade-offs through code-sharing and memory efficiency and code-size efficiency to obtain more buffer size reduction. In [75], buffer lifetime analysis and layout selection are used to select schedules (not limited to SASes) for memory usage reduction. We are

motivated by the result that sharing resources can significantly reduce resource usage and introduce resource sharing explicitly into our RASDF model.

For multi-processor platforms, where the schedule length does not necessarily lead to extra code size, non-SAS schedules can be better than SAS in performance since actors can firing concurrently In [130], the context-switch cost is minimized instead of schedule length while the vectorization is maximized, and the generated schedule is so-called Single Appearance Minimized Activation Scheduling (SAMAS). By allowing consume and produce multiple times of rates data tokens, the number of activation actor code is reduced. However, this leads to larger data memory usage. In [68], linear constraints on the firing time of actors and on buffer requirements are formulated and linear programming is used to solve the "Minimizing Buffer requirements under Rate Optimal schedules" problem. In [57], an exact method for exploring arbitrary schedules and generating minimum memory requirements for an SDF graph is given by using model-checking tool SPIN to explore the possible state space.

Throughput analysis has been studied extensively in [39, 40, 64]. [39, 40] use Maximum Cycle Mean (MCM) analysis to compute throughput. This can only be used for HSDF graph. Conversion from an SDF graph to an HSDF graph is possible, but frequently leads to a sharp increase of the graph size making the algorithms of [39, 40] fail to be efficient. In [64], the costly conversion is avoided by analyzing the state space of the self-timed execution of SDF graph. We focus on throughput and extend [64] by relaxing the self-timed scheduling constraint and are capable of analyzing resource sharing cases.

Trade-off analysis for SDF graph is mostly limited to single processor platforms [16, 176]. [16] explores the trade-off between code size and data memory. [176] gives a CD2DAT example to show the trade-off between code, data memory and execution time for SAS, based on an evolutionary algorithm. Only recently, trade-offs for SDF graphs on multiprocessor platforms are investigated [144, 148]. [144] gives an exact method to explore the trade-offs between total buffer size and throughput for multiprocessor platforms based on techniques taken from [57, 64]. In [148], it is extended to include CSDF graphs and provides a fast approximation algorithm to tackle graphs with many similar Pareto points. Our work generalizes [144, 148] with respect to SDF graph analysis to considering trade-offs between throughput and shared resources.

A design-space exploration (DSE) framework for multiprocessor systems-on-chip based on SDF specifications is proposed in [99]. The framework focuses on a single objective, i.e, make-span of an SDF graph, and SDF models are limited to HSDF graphs without cyclic dependencies. The Y-chart methodology [8, 98] is widely used to analyze embedded systems and as the basis for DSE [10, 105, 121]. However, [105, 121] formulate the DSE problem as an integer programming problem and solve it with evolutionary algorithms. These approaches do not work for problems that cannot be formulated as an integer programming problem (e.g. throughput analysis for SDF graphs). Bottleneck analysis is an important aspect of system performance analysis [88]. In [8, 98], the bottleneck information obtained from the performance analysis stage is used to guide the adaption of application, architecture and mapping aspects. Hardware optimization [51, 84] identify bottlenecks in hardware to guide architecture improvement. In [51], an event counter model that counts cache miss, branch misprediction, and resource contention events is used to monitor system behavior and identifity bottlenecks in different phases. The larger the counter at a time, the higher possibility that a resource is the bottleneck of the system. In [2], bottlenecks for parallelization of program are identified from the program trace. Each instruction of a program is represented by a set of nodes: dispatch nodes, execution nodes, commit nodes. Dependencies between nodes are identified and a program dependency graph is built. The program dependency graphs are used for analysis of the impact of bottlenecks. Throughput and buffer trade-off analysis [144, 148] extract the dependency graph from the state space of an SDF graph. By analyzing the critical cycles in the dependency graph, bottleneck buffers and trade-offs between buffer size and throughput are found. However, they only allow distributed resources and only deterministic self-timed execution is possible. In [170], we propose RASDF graph as a generic resource-aware dataflow model for trade-off analysis and propose a bottleneck-driven DSE technique for RASDF graphs for automatic system dimensioning [169]. This material is the basis of the current chapter.

We use state-based exploration techniques for trade-off analysis. The model properties, such as throughput and resource usage are derived from its state space. This technique has its root in model checking [7, 92]. Model checking is widely used in system verification such as hardware verification and protocol verification. It is also used for scheduling and scheduling related problems [1, 5, 57, 179]. In [57], model checking is used to explore the schedule space of untimed SDF graph to find schedules with minimized buffers. In [1], the job-

shop scheduling problem is modeled as a timed automaton, and finding optimal path becomes finding the shortest path problem in the timed automaton. In [5], the schedule synthesis problem is modeled with Petri nets and finding a schedule is finding a path in its state space that satisfies a specific property. In [179], model checking is used to optimize the length of static tasks and bus accesses schedules. Multi-objective model checking is only studied recently such as via qualitative property verification [47] for stochastic models and for the routing problem with multiple constraints [124]. In [137], a SAT solver is incorporated with evolutionary algorithm for an SDF task-graph model and it uses list scheduling to find a feasible schedule. We leverage the knowledge from dataflow models and multi-objective optimization to solve the trade-off analysis problem as a multi-objective model checking problem. The goal is to find multiple execution paths that are Pareto optimal in throughput (derived from cycles) and resource usage (derived from the whole execution path).

4.8 Summary

In this chapter, we consider the trade-off analysis problem for RASDF graphs and use state-based analysis to obtain Pareto-optimal points in their design space. Pareto dominance and SDF specific information are used to prune the search space. We implemented an algorithm with many configuration options that enable users to customize for their own needs. Our tool allows analyzing the throughput-memory trade-off when memory can be shared among channels. Case studies show that our tool can explore the design space very quickly while providing a good characterization of the available trade-offs. We further developed a bottleneck-driven DSE approach to explore the design space. The approach guides the search by information, the dependency graph, collected during the evaluation of metrics of interest. Experimental results show that, for systems with multiple resources and large configuration spaces, the bottleneck-driven approach saves up to 89% in analysis time compared to a brute-force approach.

5 Iteration-Domain Analysis

"Every truth has two sides; it is as well to look at both,
before we commit ourselves to either"

–Aesop

In this chapter, we investigate trade-off analysis techniques for RASDF graph from a different perspective than the time domain introduced in Chapter 4, i.e., using an iteration-by-iteration approach to analyze an RASDF graph.

5.1 Introduction

In a streaming application, input is a stream of data that is organized with its own logical structure. For example, the input stream of an MPEG-4 decoder is encoded in frames while the input of an image processing pipeline in a printer is processed in pages. In the single-core era, applications can only process one unit of data at a time and the start of processing a new unit has to wait for the finish of processing of the previous unit. In the multi-core/processor era, pipelined processing and data-parallel processing become common practice. Now the processing of new units no longer needs to wait for the finish of processing of a previous unit. However, logically, the execution of an application can still be conveniently partitioned into separate, but pipelined or paralleled units, *iteration*s.

In Chapter 2, we already saw that a sound RASDF graph requires consistency and has a non-trivial repetition vector q if and only if it is consistent. For any sequence of actor firings that conforms to q, i.e. an ***iteration***, the number of data tokens in the channels as well as the number of resource tokens in the resources after the sequence of firings are equal to their numbers before the sequence.

The time-domain analysis of RASDF graph (Chapter 4) does not exploit the fact that the data of streaming applications are processed unit-by-unit, i.e., that the system execution occurs in iterations corresponding to the logical structure of the application, such as frames or pages. In this chapter, we introduce an iteration-based trade-off analysis technique for RASDF graph, aiming for an improved quality of the results and improved efficiency of the analysis to complement the techniques in the time-domain. The approach is rooted in

max-plus algebra [6, 25, 77] to capture the iteration-based execution of RASDF graph and is developed to model the resource-sharing situation in RASDF graph.

5.2 Max-plus Algebra and its relation to RASDF

In this section, we introduce the basic concepts and definitions for *max-plus algebra* and its relation with our dataflow models. In the iteration domain, the progress of an execution is measured in firings rather than clock ticks. The effect of a firing in an RASDF graph is the change of time stamps and locations of tokens, which is naturally captured in max-plus algebra.

Max-plus algebra is an algebra on the real numbers extended with $-\infty$, i.e., $\mathbb{R}_{max} = \mathbb{R} \cup \{-\infty\}$. It can easily express the concept of synchronization and is commonly used for analysis of discrete event models. In the following sections, we use the notations in max-plus algebra to capture the executions of RASDF graphs and derive the way to compute or approximate its throughput.

In max-plus algebra, for elements $a, b \in \mathbb{R}_{max}$, the operations $\oplus$ and $\otimes$ are defined as

$$a \oplus b \stackrel{\text{def}}{=} \max(a, b) \qquad \text{and} \qquad a \otimes b \stackrel{\text{def}}{=} a + b$$

The zero element (also known as absorbing element) of addition is $-\infty$ (often written as ϵ in max-plus literature. The identity element of addition is $e \stackrel{\text{def}}{=} 0$ (also known as neutral element). In addition to that, ϵ is also the neutral element of the max operation. One defines $\max(a, -\infty) = \max(-\infty, a) \stackrel{\text{def}}{=} a$ and $a + (-\infty) = -\infty + a \stackrel{\text{def}}{=} -\infty$ for any element $a \in \mathbb{R}_{max}$, which using max-plus notation are written as:

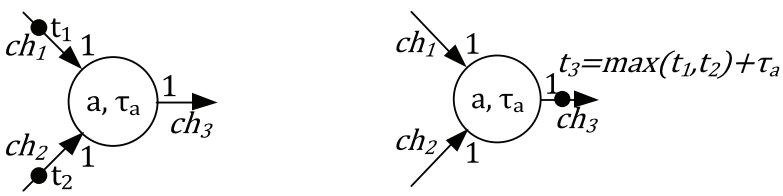
$$a \oplus \epsilon = \epsilon \oplus a = a \qquad \text{and} \qquad a \otimes \epsilon = \epsilon \otimes a = \epsilon$$

Similarly, $a + 0 = 0 + a = a$, which is written as $a \otimes e = e \otimes a = a$.

We can find that the roles of ϵ and e for the $\oplus$ and $\otimes$ operators are similar to 0 and 1 for the conventional operations + (addition) and × (multiplication) respectively on the set $\mathbb{R}$. The set $\mathbb{R}_{max}$ together with the operations $\oplus$ (maximization operator) and $\otimes$ (addition operator) is called *max-plus algebra* and is denoted by $\mathcal{R}_{max} = (\mathbb{R}_{max}, \oplus, \otimes, \epsilon, e)$. It is an example of a so-called ***semiring***.

The algebra on $\mathbb{R}_{max}$ is further extended to an algebra on matrices and vectors. For a vector $a = [a_1, \cdots, a_n] \in \mathbb{R}^n_{max}$, a vector norm is defined as follows. $\|a\| = \max(a_1, \cdots, a_n)$. A vector $\vec{a}$ is called *normalized* if $\|a\| = e$. For a non-normalized vector a, with a norm larger than ϵ, we can obtain its normalized vector as follows. $a^{norm} = a - \|a\| = [a_1 - \|a\|, a_2 - \|a\|, \cdots, a_n - \|a\|]$ in which the largest element is 0.

For more details on max-plus algebra, we refer interested readers to [6, 25, 77].



(a) Before firing (b) After firing

Figure 5.1 Max-plus semantics for an actor firing

Figure 5.1 shows an SDF actor a and its token status before actor firing and after actor firing. The firing consumes two data tokens (black dots) from two separated input channels and outputs one token to an output channel. In this chapter, we annotate tokens with their production times, with so-called *time stamps*. The time stamps for the input tokens have some values, say t_1 and t_2 (in Figure 5.1 (a)). Because the actor executes in a self-timed manner, it starts firing as soon as both tokens are available, i.e., at time $\max(t_1, t_2)$ and thus it completes its firing after executing for τ_a time units and produces a new token at time t_3 with time stamp $t_3 = \max(t_1, t_2) + \tau_a$ (Figure 5.1 (b)). Hence the process of the SDF actor firing and the time stamp values of the output token can be captured by the following max-plus algebra equation:

$$t_3 = (t_1 \oplus t_2) \otimes \tau_a$$

Note that data tokens in SDF graph are always consumed in a first come first served way. When it comes to resource tokens in RASDF graph, this is not necessarily the case. The time stamps of output tokens depend on the tokens selected, which may be decided by some resource allocation technique. Figure 5.2 shows the status of tokens before an actor firing in RASDF graph. There are two data tokens, one with time stamp t_1 at channel ch_1 and one with t_2 at

channel ch_2, and two resource tokens with time stamps t_3 and t_4 at resource R. The actor a only consumes one resource token for one firing, so it can select one of the two resource tokens. However, the selection may lead to different time stamps for the output tokens.

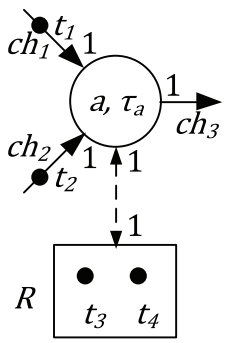
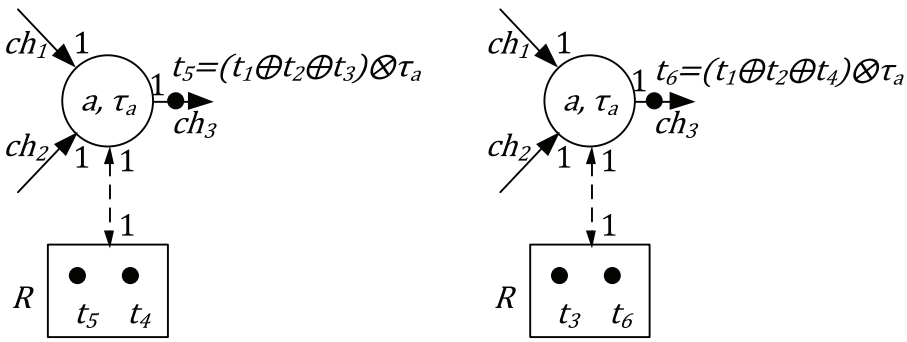


Figure 5.2 Token status before an actor firing in a RASDF

Figure 5.3 shows the possible token status after actor firing if the actor selects different resource tokens. We use a multiset of time stamps to denote the token status. Before the actor a firing, the status is $\{\{t_1\}_{ch_1}, \{t_2\}_{ch_2}, \{t_3, t_4\}_R\}$ which denotes the time stamps on channels ch_1, ch_2 and on resource R respectively. The different selections lead to two different statuses after the actor firing: $\{\{t_5\}_{ch_3}, \{t_5, t_4\}_R\}$ in which $t_5 = (t_1 \oplus t_2 \oplus t_3) \otimes \tau_a$ and $\{\{t_6\}_{ch_3}, \{t_3, t_6\}_R\}$ in which $t_6 = (t_1 \oplus t_2 \oplus t_4 \otimes \tau_a)$.



(a) after selecting t_3 in Figure 5.2 (b) after selecting t_4 in Figure 5.2

Figure 5.3 Tokens status for an actor selected different resource token

In the iteration domain, we define an execution of a consistent RASDF graph as a sequence of actor firings compared to a sequence of transitions, i.e., $start$, end, and clk, in the time domain. In an execution, every actor firing can be captured by two operations on a set of time stamps: first a $\oplus$ (maximization) operation on time stamps of data tokens consumed and time stamps of *selected* resource tokens and then a $\otimes$ (addition) with the actor execution time.

Definition 5.1: Firing State

For an RASDF graph, we can denote its firing state by a multiset of locations and time stamps of all its tokens, including both data and resource tokens. The firing operates on the set of time stamps and produces a multiset of new tokens to new locations with new time stamps. The multiset of new time stamps of produced tokens together with the time stamps of unconsumed tokens constitute a new state for the next actor firing.

Note that the firing state does not capture the state of execution at a specific instant in time. Instead, it captures the state of an execution after some number of firings and contains information from multiple different instants in time.

Definition 5.2: Execution in the Iteration Domain

An execution σ in the iteration domain is denoted as a sequence of actor firings, including the firing actor, the time the firing starts and a multiset of selected resource tokens.

For example, the firing in Figure 5.3 (a) is denoted as $(a, t_1 \oplus t_2 \oplus t_3, \{t_3\}_R)$ while the firing in Figure 5.3 (b) is denoted as $(a, t_1 \oplus t_2 \oplus t_4, \{t_4\}_R)$. The execution of an RASDF graph consists of a sequence of actor firings denoted in this form such that σ also contains resource allocation information.

In order to compare the progress of different executions that start from the same state in the iteration domain, we can only compare their states after the same number of actor firings so that the numbers of tokens in each location for different executions are the same. We use a firing count vector to count the number of actor firings.

Definition 5.3: Firing Count Vector

A firing count vector $\gamma(\sigma) = [\gamma_a(\sigma) \mid a \in A]$ with A the set of actors denotes the total number of firings $\gamma_a(\sigma)$ of each actor a in the execution σ.

The *comparable* states of different executions have the same number of tokens at the same locations, channels and resources. Hence, they also have the same number of time stamps. We order the time stamps of a state into a vector a by firstly ordering them to follow the order of the types, i.e., channels and resources, then sorting them according to the value of the time stamps (only needed for resource tokens, channels tokens are already sorted).

5.3 Iteration-based Execution

In this section, we use a running example to illustrate the semantics of iteration-based execution. Assume we have a streaming application that consists of 3 tasks a, b and c. We map the application to a platform with multiple processors and a shared memory for which neither the size of the memory (x) nor the number of cores (y) are decided. Figure 5.4 shows its system model as an RASDF graph. In this running example, one iteration consists of three firings of a, two of b, and one of c, i.e. its repetition vector is $q = [3,2,1]$.

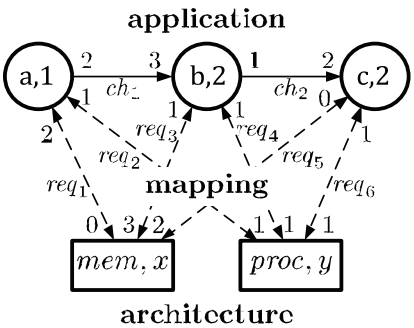


Figure 5.4 A running Example for dimensioning the resources

Figure 5.5 shows one of the executions of the example graph with memory size $x = 5$ and $y = 2$ processors, in which the horizontal axis is time and the vertical axis shows resources and actors. We separate different resources (*mem*, *proc*) on the vertical axis of the chart, into their individual tokens. Above the horizontal dashed line, it shows a Gantt chart with the individual actor firings. Below the horizontal dashed line, it shows the occupation times of resource tokens with round-cornered rectangles. The acquisition happens at the left side of the rectangle and the release happens at the right side. We use different shades to distinguish different iterations (the first iteration is darker grey and the second iteration is lighter grey). The small circles with enclosed numbers

denote the beginning and end of the iterations of the graph and indicate when that resource token is ultimately released for the execution of the iteration; the number inside the circle is the iteration count. For example, the two processor tokens are released at time 7 and 9 respectively after the first iteration finished. After the second iteration, we observe that the resource release times are identical to the release times after the first iteration, except that they are all shifted forward by 8 time units. Thus, we can use this periodical part as a valid periodic schedule, reproducing this behavior forever with period 8.

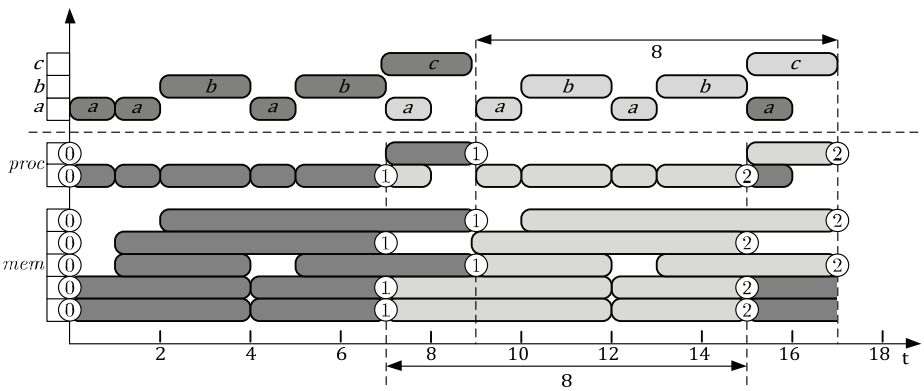


Figure 5.5 Execution chart of the running example

The execution in Figure 5.5 is written as a sequence of actor firings, i.e., $\sigma = (a, 0, \{\epsilon\}_{proc}, \{\epsilon, \epsilon\}_{mem})(a, 1, \{1\}_{proc}, \{\epsilon, \epsilon\}_{mem})(b, 2, \{2\}_{proc}, \{\epsilon, \epsilon\}_{mem}) \cdots$. In iteration-based analysis, we maintain time stamps of tokens indicating their first moment of availability. The execution of a consistent RASDF graph is captured by the evolution of time stamps of tokens, i.e., so-called firing state (the location and timestamp of tokens) after every firing. Note that the state describes the overlapping of concurrent firings instead of information at some given point of time. For instance, this representation can express the state of the graph after the first iteration in Figure 5.5, where in the timed view, this is not possible.

Although the number and locations of tokens may vary with the firings within an iteration, they return to their original number and places at the end of the iteration (recall the definition of consistency). Then only the time stamps have changed. We use the notation $\psi_c(\sigma) = \{(m_1, \tau_1), (m_2, \tau_2), \cdots, (m_k, \tau_k)\}$ to denote the time stamps of the tokens in the channel c after σ in a compact way, in which (m_k, τ_k) means that there are m_i tokens with the same time stamp τ_k.

For a resource r, we define $\psi_r(\sigma)$ in the same way. The state of an RASDF graph after a finite execution σ in iteration view is defined by the combination of the state of the channels C and the state of the resources R, as

$$\Psi(\sigma) = \begin{bmatrix} \psi_c(\sigma) & c \in C \\ \psi_r(\sigma) & r \in R \end{bmatrix}$$

For example, in Figure 5.5, the initial state of the running example with an empty execution (denoted by σ_0) is

$$\Psi(\sigma_0) = \begin{bmatrix} \{\}_{ch1} \\ \{\}_{ch2} \\ \{(5,\epsilon)\}_{mem} \\ \{(2,\epsilon)\}_{proc} \end{bmatrix}$$

Its state after a finite execution $= (a,0)(a,1)(b,2)$, the first three actor firings in Figure 5.5, is:

$$\Psi(\sigma) = \begin{bmatrix} \{(1,2)\}_{ch1} \\ \{(1,4)\}_{ch2} \\ \{(3,4)\}_{mem} \\ \{(1,\epsilon),(1,4)\}_{proc} \end{bmatrix}$$

At this state, there is one token in channel ch_1 with time stamp 2, one in channel ch_2 with time stamp 4, 3 memory resource tokens with time stamps 4, one still unused processor with time stamp ϵ, and one processor resource token with time stamp 4. In Figure 5.5, the state includes information at time point 4 for all memory tokens and 1 processor token, and 1 processor token at the beginning.

With a fixed ordering of the channels and representing individual tokens, we can alternatively represent states in vector form for simplicity. For example, the above state can be rewritten as the following vector using the order $ch_1, ch_2, mem, proc$:

$$\zeta(\sigma) = [2\ 4\ 4\ 4\ 4\ \epsilon\ 4]$$

The first entry in the vector ζ(σ) corresponds to the channel token element $(1,2)_{ch1}$ from the state Ψ(σ). The third, fourth and fifth elements correspond to the memory tokens $(3,4)_{mem}$. In general, each entry (m_i, τ_i) in the state $\Psi(\sigma)$ gets expanded into m_i entries in the vector ζ(σ) each with value τ_i. We

use $\Psi_{\sigma,i}$ to denote the ith state (the state after the first i actor firings) of an execution σ and $\zeta_{\sigma,i}$ to denote its corresponding time stamp vector.

From the definition of the norm in maxplus algebra, we know that we can use the norm of a timestamps vector to denote the maximal time stamp in the timestamps vector. Since the maximal time stamp denotes the finish time of the last firing, we can use it to denote the completion time of an execution.

Definition 5.4: Completion time of an execution

The completion time of an execution is denoted by the norm of the time stamp vector of an executionσ, i.e., $\|\zeta(\sigma)\|$, i.e., the maximal element in the time stamp vector, also the finish time of the last firing.

For example, the completion time of the execution $\sigma = (a, 0, \{\epsilon\}_{proc}, \{\epsilon, \epsilon\}_{mem})(a, 1, \{1\}_{proc}, \{\epsilon, \epsilon\}_{mem})(b, 2, \{2\}_{proc}, \{\epsilon, \epsilon\}_{mem})$ is $\|\zeta(\sigma)\| = \|[2\ 4\ 4\ 4\ 4\ \epsilon\ 4]\| = \max(2,4,4,4,4,\epsilon,4) = 4$. We can see in Figure 5.5 that the last firing of σ, i.e., the firing of actor b completes and the last tokens are returned at time 4.

When two time stamp vectors t_a and t_b have the same length n and for all $1 \leq i \leq n$, it holds that $t_{a,i} \leq t_{b,i}$, then we say that t_a *dominates* t_b and we use $t_a \preccurlyeq t_b$ to represent this. So, if t_a and t_b are vectors of time stamps from comparable states, we use $t_a \preccurlyeq t_b$ to denote that the time stamps of state a are all no later than the time stamps of the corresponding tokens in state b. For two executions that contain the same number of actor firings, we can compare their time stamp vectors to check whether an execution completes all its actor firings earlier than the other.

During an execution of an RASDF graph, every firing generates a new state, i.e., a time stamp vector. The collection of all reachable time stamp vectors consists of the state space of the RASDF graph in the iteration domain. Similar to the performance analysis in the time domain, we find recurrent states in an execution, to find periodic parts of executions, to determine the throughput of the execution. Since the time stamps in the subsequent states of an execution keep growing as the execution continues and time progresses, we normalize the time stamps of the state to check for recurrence in the visited states by only comparing the relative differences of the time stamps of the states since the recurrent states have the same relative differences. We therefore store the max-plus normalization of the vector ζ(σ) in memory during state-space exploration.

For a consistent RASDF graph, the tokens return to their original location after every iteration only with different time stamps. And if there exists a recurrent state, the actor firings between the recurrences of that state must be an integer number of iterations. This implies that we only need to check for recurrence of a state after each whole iteration to determine the throughput. For an execution σ, we use σ_i to denote the execution σ up to i iterations. Assume the execution first visits its recurrent state after the n_1th iteration and revisits it after the n_2th iteration, then the execution between the n_1th and n_2th iteration forms a cycle in the state space. Assume after k iteations of execution, this cycle has been repeated n times, so that $k = n_1 + n \cdot (n_2 - n_1)$. The completion time of the ith iteration is the norm of $\zeta(\sigma_i)$, i.e. $\|\zeta(\sigma_i)\|$. We can compute its throughput (the average number of iterations per time unit) with the following equation:

$$\begin{aligned}\mathrm{Thr}(\sigma) &= \lim_{k\to\infty} \frac{k}{\zeta(\sigma_k)} \\ &= \lim_{n\to\infty} \frac{n_1 + n \cdot (n_2 - n_1)}{\|\zeta(\sigma_{n_1})\| + n \cdot (\|\zeta(\sigma_{n_2})\| - \|\zeta(\sigma_{n_1})\|)} \\ &= \frac{n_2 - n_1}{\|\zeta(\sigma_{n_2})\| - \|\zeta(\sigma_{n_1})\|}\end{aligned}$$

For example, in Figure 5.5, the time stamp vector for the first iteration is $\zeta(\sigma_1) = [7\ 7\ 7\ 9\ 9\ 7\ 9\ ]$ (the first 5 elements for mem tokens and the last 2 elements for $proc$ tokens), the recurrent state, i.e., the time stamp vector for the second iteration is $\zeta(\sigma_2) = [15\ 15\ 15\ 17\ 17\ 15\ 17]$ and $\zeta(\sigma_1)^{norm} = \zeta(\sigma_2)^{norm} = [-2, -2, -2,\ 0,\ 0, -2,\ 0]$, $\zeta(\sigma_1) = \zeta(\sigma_1)^{norm} + 9$ and $\zeta(\sigma_2) = \zeta(\sigma_1)^{norm} + 17$ so the throughput of the execution is

$$\begin{aligned}\mathrm{Thr}(\sigma) &= \lim_{n\to\infty} \frac{1 + n \cdot (2 - 1)}{\|\zeta(\sigma_1)\| + n \cdot (\|\zeta(\sigma_2)\| - \|\zeta(\sigma_1)\|)} \\ &= \frac{2 - 1}{\|\zeta(\sigma_2)\| - \|\zeta(\sigma_1)\|} = \frac{2 - 1}{17 - 9} = \frac{1}{8}\end{aligned}$$

In the iteration-based view, the state vector contains the time stamps of all available resource tokens. All resource token time stamps are initialized to ϵ. The time stamp of a resource token that was ever used has become larger than ϵ. Hence, the number of initial tokens minus the number of ϵ time stamp tokens at a state $\Psi_{\sigma,i}$ for resource r is the amount of used resource at the state $\Psi_{\sigma,i}$ and denoted by $Ru_r(\Psi_{\sigma,i})$. So the resource usage of resource r in an

execution σ is $Ru_r(\sigma) = max\{Ru_r(\Psi_{\sigma,i})\}$, for all states $\Psi_{\sigma,i}$ in σ. We use $Ru(\sigma)$ to denote the vector $[Ru_r(\sigma) \mid r \in R\,]$. In Figure 5.5, the execution up to the first iteration is σ_1 contains 6 actor firings, $a, a, b, a, b, c,$ and the firings generate 6 subsequent states as follows:

$$\Psi_{\sigma,1} = \begin{bmatrix} \{(2,1)\}_{ch_1} \\ \{\}_{ch_2} \\ \{(3,\epsilon)\}_{mem} \\ \{(1,\epsilon),(1,1)\}_{proc} \end{bmatrix}, \Psi_{\sigma,2} = \begin{bmatrix} \{(2,1),(2,2)\}_{ch_1} \\ \{\}_{ch_2} \\ \{(1,\epsilon)\}_{mem} \\ \{(1,\epsilon),(1,2)\}_{proc} \end{bmatrix}, \Psi_{\sigma,3} = \begin{bmatrix} \{(1,2)\}_{ch_1} \\ \{(1,4)\}_{ch_2} \\ \{(3,4)\}_{mem} \\ \{(1,\epsilon),(1,4)\}_{proc} \end{bmatrix}$$

$$\Psi_{\sigma,4} = \begin{bmatrix} \{(1,2),(2,5)\}_{ch_1} \\ \{(1,4)\}_{ch_2} \\ \{(1,4)\}_{mem} \\ \{(1,\epsilon),(1,5)\}_{proc} \end{bmatrix}, \Psi_{\sigma,5} = \begin{bmatrix} \{\}_{ch_1} \\ \{(1,4),(1,7)\}_{ch_2} \\ \{(3,7)\}_{mem} \\ \{(1,\epsilon),(1,7)\}_{proc} \end{bmatrix}, \Psi_{\sigma,6} = \begin{bmatrix} \{\}_{ch_1} \\ \{\}_{ch_2} \\ \{(3,7),(2,9)\}_{mem} \\ \{(1,7),(1,9)\}_{proc} \end{bmatrix}$$

So the resource usage of $mem, proc$ are

$$Ru(\sigma_1) = \begin{bmatrix} Ru_{mem}(\sigma_1) \\ Ru_{proc}(\sigma_1) \end{bmatrix} = \begin{bmatrix} \max(2,4,5,5,5,5) \\ \max(1,1,1,1,1,2) \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \end{bmatrix}$$

Similarly, we have $Ru(\sigma_2) = \begin{bmatrix} 5 \\ 2 \end{bmatrix}$, since the execution repeats the periodic schedule between σ_1 and σ_2, so the resource usage of the execution σ is $Ru(\sigma) = \begin{bmatrix} 5 \\ 2 \end{bmatrix}$.

In the next section, we discuss about the techniques for trade-off exploration in the iteration domain.

5.4 Exploration Techniques

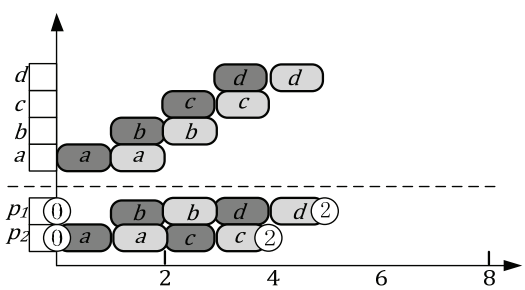
Different executions of an RASDF graph may lead to different cycles in the state space of the graph and have different throughput and resource usage properties. To explore the design space of a given RASDF graph, we need to explore all different executions and compute their corresponding metrics: throughput and resource usage. Though the number of different executions can be very large, we can explore the design space as efficiently as possible by exploiting some properties of executions.

Recall that the recurrent states of a state in the iteration domain can only happen after a whole number of iterations. Since the numbers and locations of data and resource tokens do not change after any number of complete

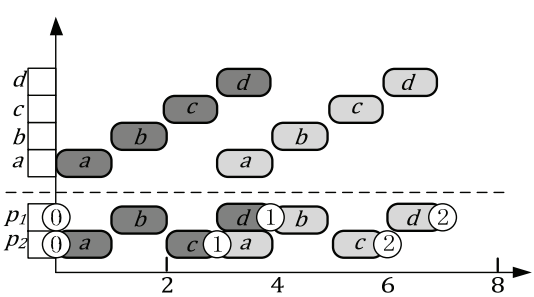
iterations (due to consistency), to detect recurrent states, we only need to compare the time stamps of data and resource tokens.

We use a nested exploration strategy that first explores the scheduling possibilities inside a single iteration (intra-iteration exploration) and only then constructs a state-space of *iteration states* (inter-iteration exploration), i.e., the iteration state space. We evaluate throughput and resource usage of an execution in the iteration state space during exploration, when recurrent states, cycles are found in the iteration statespace.

Note that we add a constraint for exploration in the nested exploration strategy, i.e., we can only explore one iteration each time during intra-iteration exploration. This constraint allows us to only overlap different iterations while forbidding interleaving them. It is a double-edged sword. It simplifies exploration and boosts the speed of exploration on the one hand, while potentially sacrificing performance gain and efficient resource usage by interleaving multiple iterations on the other hand.



(a) Interleaving firings of two iterations for resource allocation



(b) Overlapping firings of two iterations for resource allocation

Figure 5.6 Interleaving and overlapping firings for resource allocation

Figure 5.6 shows two different executions of an RASDF graph which contain two iterations. The interleaving of firings between two iterations in Figure 5.6 (a) leads to more efficient resource usage and higher throughput compared to the overlapping of firings of two iterations in Figure 5.6 (b) without interleaving them. However, the execution in Figure 5.6 (b) keeps the boundary between two iterations which allows us to use a nested exploration strategy, i.e., first exploring inside every iteration, then exploring directly on iteration states.

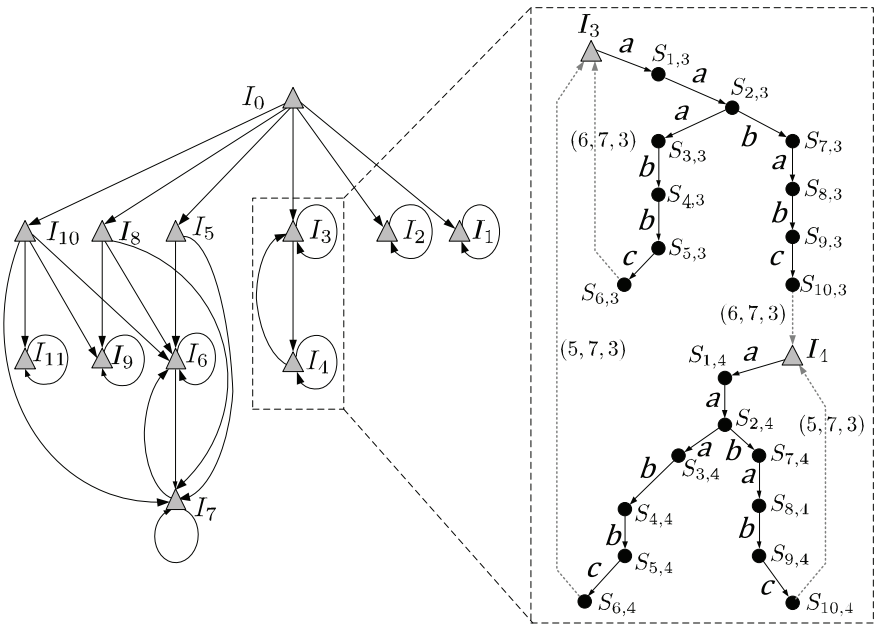


Figure 5.7 State space of the running example using iteration-based exploration

Figure 5.7 shows the state space of the running example using the nested exploration strategy with maximal memory size $x = 9$ and processor number $y = 3$. On the left side is the iteration state space of the example. We use I_k to identify different iteration states (triangular nodes in Figure 5.7) in the iteration state space and edges with arrows denote single iterations according to specific schedules. The right side of the figure zooms into the iterations starting from states I_3 and I_4 and illustrates the nested exploration within an iteration. Note that it shows two, separate nested explorations, one starting from I_3 and one starting from I_4. The zoomed figure shows the intra-iteration states labeled $S_{i,j}$

(ith state in jth iteration explored) and arrows labeled with actor names (individual actor firings).

Dashed edges from black circles to triangles denote the transitions between states to their *normalized* iteration states whose time stamp vectors are the normalization of the original states. The transitions can be occurrences of recurrent states in the state space (for instance, the revisited iteration state I_3 the normalized version of state $S_{6,3}$) or a transition from one iteration into the next new iteration (for instance, the newfound iteration state I_4 is the normalized state of $S_{10,3}$). The iteration period is the (average) time taken for one iteration and is thus inversely proportional to the throughput. For example, an execution σ may reach state I_3 after its first iteration (denoted as σ_1). At the end of the 3rd iteration of σ, i.e. after σ_3, it may reach $S_{6,4}$, which is *scaled* from I_3 for 11 time units ($\|\zeta(\sigma_3)\| = \|\zeta(\sigma_1)\| + 11$. The cycle $(I_3 \cdot S_{1,3} \cdot S_{2,3} \cdot S_{7,3} \cdot S_{8,3} \cdot S_{9,3} \cdot S_{10,3} \cdot I_4 \cdot S_{1.4} \cdot S_{2,4} \cdot S_{3,4} \cdot S_{4,4} \cdot S_{5,4} \cdot S_{6,4})^{\omega}$ contains two iterations with time durations, i.e. the norm of end states, of 6 and 5 respectively. The cycle can be repeated forever. For this cycle, the average number of time units per iteration is $\frac{6+5}{2} = 5.5$, the resource usage is 7 for the memory and 3 for the processors. For this example, we use a 3-value point (a, b, c) to denote the found Pareto points. The first value is the average time per iteration (i.e., the inverse of throughput), the second value is the memory usage, and the third is the processor usage. The found Pareto point is $(5.5, 7, 3)$.

By applying Pareto minimization on the metric points that we obtain from the exploration of the state space such as $(5.5, 7,3)$, we can find the different trade-off points between throughput and resource usage, i.e., the Pareto points in the metric space. In the exploration in Figure 5.7, we can find 4 different executions with different Pareto points and different memory usage and throughput: $(8,5,3), (7,6,3), (5,7,3), (4,9,3)$. The point $(5.5, 7, 3)$ is not Pareto optimal since it is dominated by (5,7,3). All these points use the same number of processors (3), but reach different throughputs with different memory usages.

Iteration-based exploration can save the space for storing intra-iteration states during exploration. This property sharply reduces checks for recurrent states and the size of the state space stored compared to the traditional state-space exploration approach in the time-domain (see Chapter 4). For example, only 123 iteration states need to be stored for full exploration of the example when using the iteration-based exploration, in comparison to 220 states when

using time-domain exploration to explore the state space to a depth of just 4 iterations.

So far, we know that different firing orders and resource allocations for actors may generate different states. The number of choices can grow exponentially. The success of this method depends on efficiently pruning the state space. We discuss our efficient pruning techniques of the state space of an RASDF graph from the two aspects.

Pruning Based on Actor Firing Order:

Under the same resource constraints, two executions may have the same number of firings for each actor, and only differ in the order of actor firings leading to different token time stamps at the end of the two executions. If all time stamps of an execution σ_y are larger than or equal to some other execution σ_x, while σ_x and σ_y have the same actor firings and resource usage, then σ_x is slower than σ_y. Therefore σ_y is redundant in a search for optimal executions and can be pruned.

Proposition 5.1:

Given two executions σ_x and σ_y such that the actor firing count vectors are equal, $\gamma(\sigma_x) = \gamma(\sigma_y)$ and $\zeta(\sigma_x) \preccurlyeq \zeta(\sigma_y)$, then for any infinite execution $\sigma_b = \sigma_y\sigma$, infinite execution $\sigma_a = \sigma_x\sigma$ can reach the same throughput with the same or lower resource usage.

Proof: Since $\gamma(\sigma_x) = \gamma(\sigma_y)$, the numbers and locations of data and resource tokens are identical between σ_x and σ_y. They only differ in time stamp values, which in $\zeta(\sigma_x)$ are no later than in $\zeta(\sigma_y)$(since $\zeta(\sigma_x) \preccurlyeq \zeta(\sigma_y)$). So any actor firing that is possible in σ_b *after* σ_y is also enabled in σ_a after σ_x. So for any execution after σ_b, we can have the same execution for σ_a. Once execution σ_b visits a recurrent state and enters into a periodical part, we can guarantee to construct an execution σ_a with the same firing sequence but every state dominates the corresponding state in the periodical part of σ_b. So the throughput of σ_a, i.e., $Thr(\sigma_a) \geq Thr(\sigma_b)$. Since $\zeta(\sigma_x) \preccurlyeq \zeta(\sigma_y)$, $Ru(\sigma_x) \preccurlyeq Ru(\sigma_y)$, we have $Ru(\sigma_a) = Ru(\sigma_x\sigma) \preccurlyeq Ru(\sigma_y\sigma) = Ru(\sigma_b)$. So we only need to explore execution with prefix σ_x. ■

From Proposition 5.1 we know that, if we find that the time stamp vector of a state is dominated by a time stamp vector already existing in the state space

with the same firing counting vector and resource usage, the exploration can backtrack since further exploration cannot lead to a better result. This pruning rule is used in both inter-iteration exploration and intra-iteration exploration.

Pruning Based on Resource Allocation:

Proposition 5.1 is used to prune executions with different firing orders. Among executions with the same firing order, the resource allocations can be different. We use an example in Figure 5.8 to illustrate the impact of different resource allocations on executions.

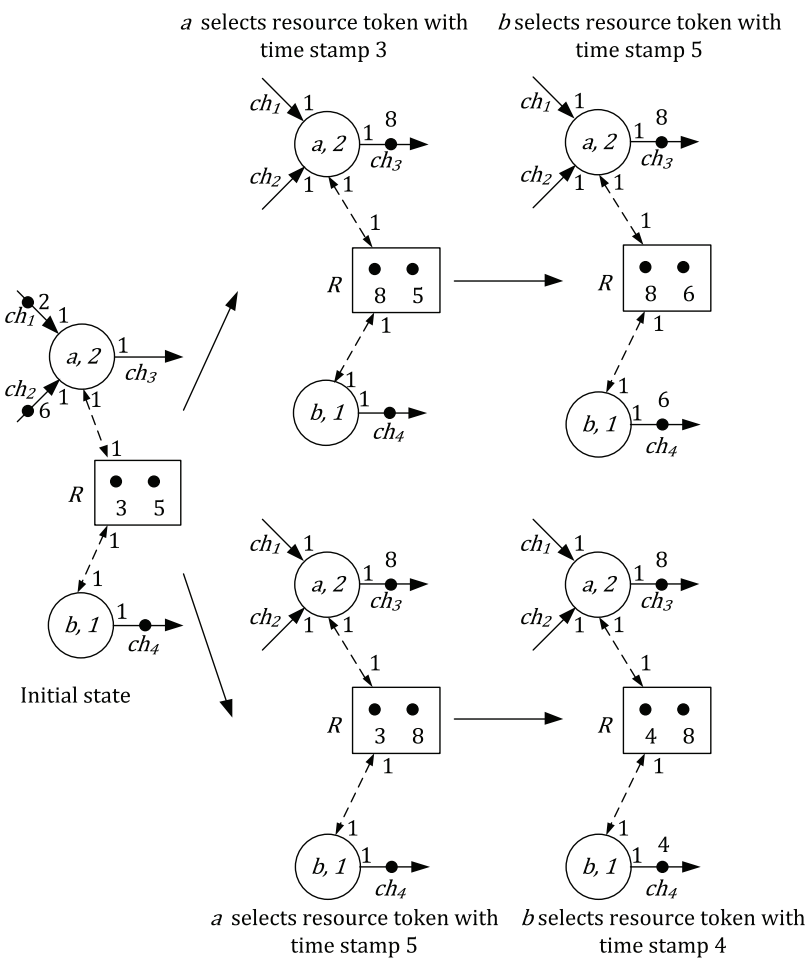


Figure 5.8 Different resource allocations

Figure 5.8 shows the same firing sequence ab of a part of an RASDF graph that starts from the same initial state with different resource allocations. For the firing of actor a, it can select either the resource token with time stamp 3 or with 5. It is enabled at time 6 (determined by the data token on ch_2). Depending on its resource selection, the execution can have two different execution paths shown in Figure 5.8. Then we schedule the firing of actor b to select the left resource tokens (5 or 3 depending on the selection of actor a). If it selects 5, it chooses the top execution path in Figure 5.8; otherwise it selects 3 and chooses the bottom execution path. From the proposition 5.1, we know that since the state $\{\{8\}_{ch_3}, \{5,8\}_R\}$ (top part) is dominated by the state $\{\{8\}_{ch_3}, \{4,8\}_R\}$ (bottom part) after firing a; the bottom execution path is better.

We can extract the resource allocation rule from the example of Figure 5.9 as the ***A**s **N**ew **A**s **P**ossible (**ANAP**)* policy. Let us consider a general case as shown in Figure 5.9: an actor is ready to be fired (i.e. enabled) at time t_{en}, i.e., the maximal time stamp of data and resource tokens, which denotes the moment when all data and resource token requirements are satisfied. The resource tokens are sorted in order of time stamps and shown in Figure 5.9 as a list of tokens annotated with their time stamps from small to large. Let t_{en} be the earliest time at which the actor is enabled. The k resource tokens it needs are available at t_k. We know that the enable time of the actor $t_{en} \geq t_k$. We always select the k tokens with the largest time stamps that are equal to or less than t_{en}. Since the selected order of firing in state-space exploration is not necessarily the time order of firing. *ANAP* leaves the earlier resource tokens to be used by other actor firings which may thus be able to fire earlier and so improve performance (e.g. actor b in Figure 5.8 can fire earlier). Since the time stamps are the newest tokens that are available at t_{en}, we call the resource token selection policy *ANAP*:

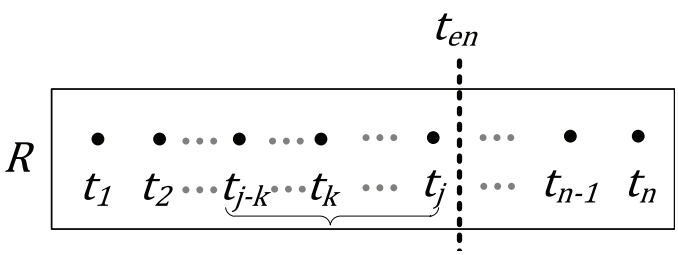


Figure 5.9 Optimal resource allocation policy

Proposition 5.2:

The resource allocation policy ANAP is throughput-optimal.

Proof: Given an execution at state s, and an actor a with execution time τ_a that is selected for firing at state s. Assume the actor enabling time is t_{en}. Then the output data and released resource tokens of the actor are time-stamped $t_{en} \otimes \tau_a$ since the firing starts at t_{en}. The time stamps for output data tokens and released resource tokens are the same for all resource allocation policies. We only need to compare the time stamps of unused tokens. The data tokens are consumed in *FIFO* order, so no matter how the resource tokens are selected the remaining data token time stamps are the same for any resource allocation policy. By using the $ANAP$ policy, the time stamps of unused resource tokens are the smallest since *ANAP* always chooses the time stamps closest to t_{en} with the highest values. So, the time vector generated by $ANAP$ always dominates time vectors generated by another resource allocation policy. From Proposition *5.1*, it then follows that $ANAP$ is the optimal resource allocation policy. ■

Proposition 5.2 shows that the ANAP resource allocation policy generates a time stamp vector that dominates other resource allocation policies and in turn can provide an execution that at least has the same throughput as executions from other resource allocation policies. It thus prunes other resource allocation choices for executions with the same firing sequence.

Throughput approximation:

In practice, we store the found schedule as a list with a prefix firing sequence and a periodic part repeated forever. The total length of prefix plus periodic part is limited due to storage constraints. Even though the state space of an RASDF graph is finite, the execution can take a number of iterations to reach the periodic part and the resulting schedule may exceed the constraint on the length of the firing sequence. In order to compute throughput for an execution that has reached the length limit and still has not reached the periodic part during exploration, we need to compute a conservative approximation for the throughput of the explored partial execution .

Figure 5.10 shows how to compute such a conservative approximation. Assume the execution length limit of an exploration is n iterations. We have an execution that after its nth iteration reaches a state with time stamp vector ζ_n (denoted by the grey dots in Figure 5.10), which has not been visited before

(not a recurrent state). Since we have to compute throughput from a periodic part, we need to construct a periodic schedule.

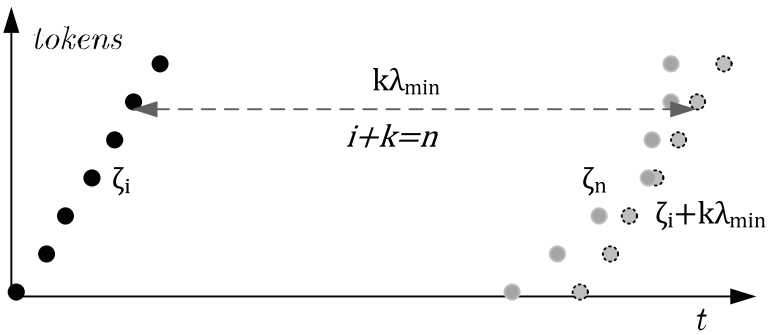


Figure 5.10 Throughput Approximation

We can construct a periodic schedule by repeating the schedule from an intermediate iteration state ζ_i ($0 < i < n$, denoted by black dots) to the last iteration ζ_n (denoted by grey dots) by computing the offset $k\lambda_{min}$ such that $\zeta_n \preccurlyeq \zeta_i + k\lambda_{min}$ in which $n = i + k$ and λ_{min} is the minimal value that satisfies $\zeta_n \preccurlyeq \zeta_i + k\lambda_{min}$. This means that we can delay the firings of a new iteration and use the tokens at times that are specified by the time stamp vector $\zeta_i + k\lambda_{min}$ (denoted by the grey dots with dashed edge) so that the execution enters into a periodic phase. Then we can always achieve a throughput $Thr = k/(k\lambda_{min}) = 1/\lambda_{min}$. We can iterate the intermediate iteration state in the execution path and estimate the approximated throughput for all of them. So for all stored iteration states from ζ_0 to ζ_{n-1}, we estimate the throughput based on that state and since we know they are all lower bounds on throughput, we keep the largest one as the throughput estimate.

Proposition 5.3:

The truncated throughput estimation is conservative, i.e., the schedule found with the above approximation, leads to a throughput which is no smaller than the approximated throughput.

Proof: Assume that the highest approximated throughput $Thr = 1/\lambda_{min}$ is obtained from the periodic part of an execution σ by repeating the part σ_k that contains k iterations between the ith iteration (timestamps vector ζ_i) and the $(i + k)th$ iteration (timestamps vector ζ_{i+k}) and adding firing delays for the

new iteration at the end of the $(i+k)th$ iteration to satisfy the time stamp constraints of tokens for repeating from ζ_i ($\zeta'_{i+k} = \zeta_i + k\lambda_{min}$). The constructed schedule is simply repeat the firing sequence of σ_k without delaying firings, i.e., the $(i+k)th$ iteration timestamps vector is ζ_{i+k}. Since the constructed schedule never generates later timestamps vector, its throughput is no smaller than $1/\lambda_{min}$. ■

5.5 Case Studies

We implemented the iteration-based algorithm and tested it on two sets of RASDF graphs, the same as the graphs used in Chapter 4, to evaluate the iteration-based approach, one set from literature and one from Océ. The experimental setup is the same as the setup in Chapter 4. We compare the results of the iteration-based approach with the time-based approach developed in Chapter 4.

For iteration based exploration, we can also apply the bottleneck analysis technique developed in Chapter 4. During exploration we annotate the resource tokens with the actors that released them. If the enable time of an actor a is equal to the time stamps of data tokens outputted by or resource tokens released by a previous firing of actor b, then a is dependent on b. We can build a dependency graph and do dependency analysis the same way as in Chapter 4.

We use a grid search approach as in Chapter 4 to search all the configurations with different iteration length and branching number. Each configuration has a fixed time budget (here we use 1 second so that estimated total exploration time is around a few minutes). We compare the number of explored configurations and the Pareto points obtained. We use two approaches proposed in [36, 177] to evaluate the quality of results. Both approaches are explained in Section 3.6. The quality of the Pareto points obtained from both approaches is compared using ϵ-Indicator [177] and using Average Distance to Reference Set (ADRS) [36]. The first one is typically used to compare two different point sets while the second one is often used to evaluate the quality of a method to approximate a known Pareto-optimal front.

Figure 5.11 illustrates the ϵ-Indicator for the MP3 case. In this case, we have to scale a factor of 1.09 to ensure that the scaled time-based results (red triangles) are dominated by the iteration-based results (blue stars). On the other hand, we have to scale by a factor of 1.156 to get the scaled iteration-

based results (blue plus) dominated by the time-based results (red squares). We can conclude that none of the results are strictly better than the other (both scaling factors are larger than 1), but if we have to make a choice based on the ϵ-Indicator, we would prefer the iteration-based set for this case.

As explained in Chapter 3, in ADRS, the average distance (distance function is user defined) of Pareto points to each Pareto point in the reference set is measured. In our experiment, the combined results of the two approaches are used as the reference set. For example, the three green circles in Figure 5.11 are the combined results of the two approaches and are used as points of the reference set. For every point in the reference set, we compute the minimal distance to the points in the selected set. The distance between two points P_1 and P_2 is defined as the maximal positive ratio of distance change compared to P_2 in the reference set among all objective dimensions, i.e., $d(P_1, P_2) = \max(\frac{(P_{1,x}-P_{2,x})}{P_{2,x}}, \frac{(P_{1,y}-P_{2,y})}{P_{2,y}}, 0)$. For example, we can compute the ADRS of the iteration-based approach as follows. For two reference points, which overlap with the points in the iteration-based approach, the minimal distance is 0. For the left upper green circle, the minimal distance of a point to it is 0.0929. So the ADRS is (0.0929+0+0)/3=0.031.

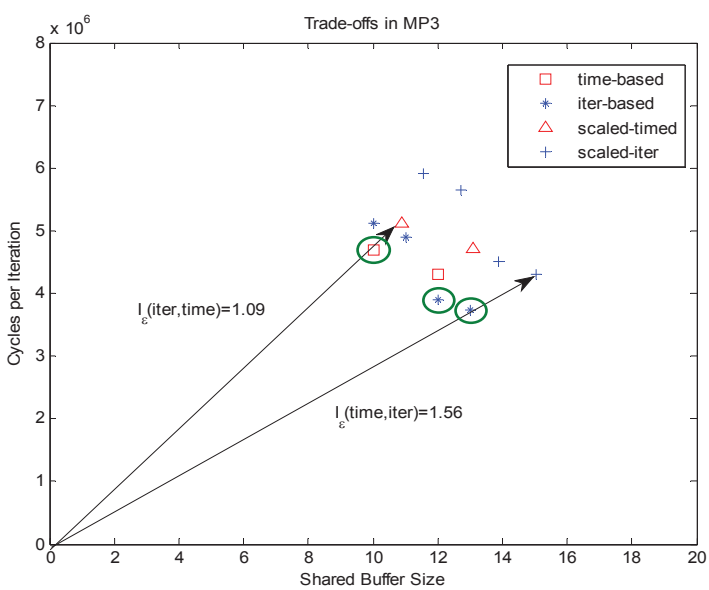


Figure 5.11 Comparison of trade-offs of MP3 with one shared buffer

Since both approaches do not fully explore the design space of large examples, they may miss optimal points. The iteration-based approach cannot exploit the interleaving of iterations (see Figure 5.6), i.e., firings in earlier iterations must use the resources before the corresponding firings in later iterations. The time-based approach can in principle exploit the opportunity of interleaving iterations. However, in practice, the size of the state space that can be explored is limited. As the size of the state space grows rapidly with the length of an iteration and slows down the exploration, the iteration-based approach typically completes faster than the time-based approach. Tables 5.1 and 5.2 show the comparison between iteration and time-based approaches. Table 5.1 shows the trade-off analysis results for the printer architecture case study provided by Océ. Table 5.2 includes a set of dataflow graphs from literature with different buffer sharing configurations.

	Arch 1	**Arch 2**	**Arch 3**
No. of Pareto Points (time based)	5	9	9
Conf No. (time based)	147	66	137
Exec Time (time based) (s)	144.6	65.4	134.8
No. of Pareto Points (iter based)	8	5	13
Conf No. (iter based)	60	6	57
Exec Time (time based) (s)	9.3	1.0	15.4
I_ϵ(time,iter)/(iter,time)	3.0/1.092	2.0/1.811	2.137/1/075
Exec Time Reduction	94%	98%	86%

Table 5.1 Iteration-based approach vs. time-based approach for printer architecture

In both tables, we compare the number of Pareto points found and the number of configurations explored by both approaches. The execution time and quality of results are compared. Since there is no reference set of Pareto points, in this first comparison, we only use the ϵ-Indicator as the quality indicator.

	Bipartite		Modem		Sample Rate		MP3		Satellite		H.263 (QCIF)	
No. of Shared buffers	1	2	1	3	1	3	1	3	1	3	1	2
No. of Pareto Points (time based)	7	8	4	1	6	2	2	4	1	1	2	3
No. of Conf (time based)	10	69	30	34	15	250	9	104	11	27	65	43
Exec Time (time based) (s)	7.8	1.7	29.7	6.7	15.3	47.9	9.3	57.2	11.4	27.4	60.9	30.9
No. of Pareto Points (iter based)	7	8	4	1	3	2	4	2	2	3	1	1
No. of Conf (iter based)	10	91	11	34	15	250	9	106	9	9	3	4
Exec Time (iter based) (s)	6.7	2.2	7.2	1.7	8.9	37.0	9.5	11.3	21.4	64.2	2.7	0.7
I_ϵ(time,iter)/(iter,time)	1/1	1/1	1/1.0625	1/1	1.024/1.035	1/1	1.156/1.09	1.156/1	1.019/1	1.019/1	1/1.074	1/2
Exec Time Reduction	14%	-30%	75%	75%	60%	23%	-2%	80%	-87%	-134%	95%	98%

Table 5.2 Iteration-based approach vs. time-based approach for graphs in literature

We see in our experiments that a considerable exploration time reduction is obtained in 11 out of 15 cases (3 printer architecture cases plus 12 multimedia related dataflow graphs) and new Pareto points are found with the iteration-based approach in 8 cases. Also in 8 cases, however, the time-based approach yields Pareto points not found by the iteration-based approach. For 4 use cases (Bipartite with two shared buffers, MP3 with one shared buffers, and two Satellite cases), the exploration time reduction is negative. This is due to the exploration in iteration domain at least having to explore two iterations to identify recurrent states if approximation is not used while the exploration in time-based domain may find recurrent states earlier before two iterations are finished.

There does not seem to be a systematic way to predict which of the approaches performs better in specific cases. The two approaches do strengthen each other. Running both analyses yields the best results, as illustrated for example in Figure 5.12.

Table 5.3 compares the combined approach with both individual, iteration-based and time-based, approaches. We can use the combined results as reference set and evaluate the ADRS quality indicator. The combined approach obviously dominates the two individual approaches quality wise, with acceptable execution times ranging from a few seconds to just over 2.5 minutes. It suggests that in trade-offs analysis, we can combine both methods to obtain better results.

5.6 Summary

In this chapter, we investigate the trade-off analysis in RASDF schedules from a different angle when compared to the time-based analysis in Chapter 4: the iteration-based statespace exploration approach. We exploit the consistency of RASDF graph and dominances of time stamp vectors of the iteration-based execution to explore the state space on an iteration-by-iteration basis. The experimental results on our RASDF graphs show that the new iteration-based approach and the traditional analysis in the time domain complement each other. The new approach finds new trade-off configurations not found by the traditional approach in 8 of 15 cases and it is often faster. Combining the two approaches is feasible and yields the highest quality results. The iteration-based approach allows for easy modeling of dynamic execution time changes between iterations, as in SADF graph. In the next chapter, we discuss the

adaption of our iteration-based analysis for input-dependent RASDF graph, i.e. SARA SDF graph.

	Bipartite		Modem		Sample Rate		MP3		Satellite		H.263 (QCIF)		Arch 1	Arch 2	Arch 3
No. of Shared buffers	1	2	1	3	1	3	1	3	1	3	1	2			
Trade-offs No. (combined)	7	8	4	1	5	2	3	2	2	3	2	3	10	14	18
Exec Time (combined) (s)	14.5	3.9	36.9	8.4	24.2	84.9	18.8	68.5	32.8	91.6	63.6	31.6	153.3	66.4	150.2
I_ϵ(iter,comb)/ (comb,iter)	1/1	1/1	1.0625/ 1	1/1	1.035/ 1	1.04 /1	1.09/1	1/1	1/1	1/1	1.08/ 1	2/1	1.092/ 1	1.81/1	1.05/1
I_ϵ(time,comb)/ (comb,time)	1/1	1/1	1/1	1/1	1.024/ 1	1/1	1.16/1	1.16 /1	1.02 /1	1.02/ 1	1/1	1/1	3.0/1	2.0/1	2.14/1
ADRS(time,iter) /(iter,time)	0/0	0/0	0/ 0.156	0/0	0.006 /0.019	0/ 0.04	0.09/ 0.031	0.08 /0	0.01 /0	0.013/ 0	0/ 0.037	0/ 0.036	0.39/ 0.016	0.265/ 0.310	0.30/ 0.01

Table 5.3 Combined approach vs. time-based and iteration-based approaches for graphs in literature

6 PLAYING GAMES WITH SARA SDF

"…beautiful mathematics eventually tends to be useful,
and useful mathematics eventually tends to be beautiful."

-Carl D. Meyer

6.1 Introduction

Emerging streaming applications have to adapt to environmental changes for implementation efficiency. Environmental awareness enables systems to achieve higher performance and lower resource usage in comparison to implementations without such environmental awareness. By knowing information such as the bits allocation needs of macro blocks in MPEG4 [3], or the network status in a wireless sensor network [48], or the environment and internal state of a cognitive radio system [104], system designers can achieve better designs such as achieving better perceptual quality with stable buffer levels, or an optimal flow rate for the network, or a flexible and autonomous next generation communication network. Many streaming applications show data-dependent behaviors, i.e., their execution times and resource usages are highly dependent on the properties of the input, which implies that we can utilize these properties for better implementation.

At the same time, advances in computer engineering allow both software and hardware to adapt their own behaviors at runtime in response to changes in the environment. For example, software can change its scheduling policy and grant resources to tasks in different orders while hardware can adapt its processors' voltages and frequencies, i.e., so-called Dynamic Voltage and Frequency Scaling (DVFS) [38, 120]. Progress in programmable hardware allows hardware implementation changes at runtime, with some overhead. For example, by downloading different bit streams at runtime, reconfigurable computing platforms can dynamically configure their FPGAs with different functionalities optimized for their corresponding input data [32].

To summarize, an adaptive system has the following two features: detecting changes of the environment (**cognitive ability**) and adapting its software/hardware aspects accordingly in time for efficiency (**reconfigurability**). As a result, the design of an adaptive system needs to answer the following two questions: "How to **model** the environment change?"

and "How to reconfigure software/hardware to **guarantee** design objectives in response to the given environment change?"

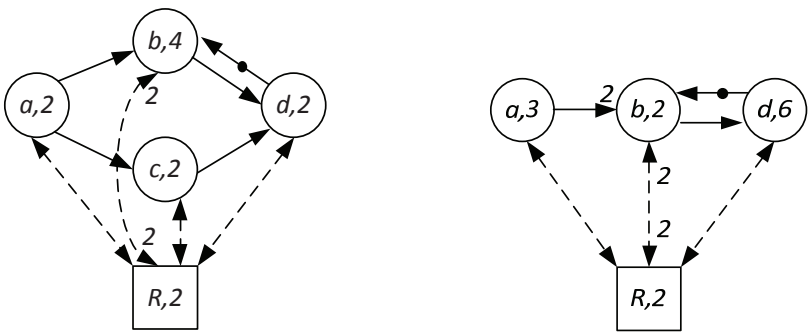
To answer the first question, we can use the fact that the properties of input data can be known beforehand and that these properties are embedded into the input data as metadata. Hence, we can model the environment changes with Markov Chains [151] or FSMs [59]. For example, in the MPEG-2 standard, the input image frames of a decoder can be identified as I frames, P frames or B frames respectively. For raw input without embedded information that helps identification, we can use techniques such as machine learning to classify them. Classification and detection of input data is studied extensively in literature [66, 103, 158]. We assume that we can capture (if necessary, over-approximate) the environment changes with a finite state machine (FSM). Here we mainly focus on the second question using a throughput target as our design objective.

We use the model introduced in Chapter 2, i.e., SARA SDF graphs, to model the system with dynamic input. The input types are represented by scenarios and captured by the scenario FSM in a SARA SDF graph. Input sequences and the state changes of a system are captured by the scenario sequences that are encoded by the scenario FSM. For different input scenarios, the controller of a system fires different actors and allocates different amounts of resources, such as processors and buffers. The reaction of the controller to different scenarios may lead to different processing times for different inputs. The interaction between the system and its environment is modeled as a formal, so-called mean-payoff game. In such a game, a good controller corresponds to a player in the game with a winning strategy, i.e., a way to react to different inputs and achieve a given throughput constraint. The strategy can be captured by an FSM whose transitions correspond to a reaction of the controller. On the one hand, finding a winning strategy becomes finding an FSM, i.e., synthesizing a controller, that satisfies the throughput constraint. This is a controller synthesis problem. On the other hand, the winning strategy of an environment is the worst-case input of the system. Based on this model, we solve both the controller synthesis problem under resource and throughput constraints as well as the worst-case environment identification problem.

To give some intuition for the question that we are trying to answer, i.e., how to synthesize a controller that satisfies a given throughput constraint, it is useful to walk through an example.

6.2 An Illustrative Example

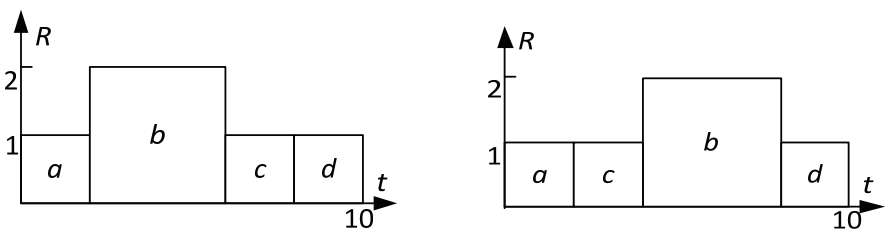
Assume an application with four actors a, b, c, d and data dependencies among those actors. The parameters of actors, i.e., execution times and data/resource rates, are dependent on the type of application input and are represented as scenarios: A, B. The four actors use the same resource R with b using two units at a time. Figure 6.1 shows the RASDF models for two possible types of input respectively, while the execution times of the actors, the amount of resources and the data/resource rates are annotated with the models. (Rates equal to 1 are omitted for clarity.)



(a) Input A (b) Input B

Figure 6.1 RASDF model under two different types of inputs

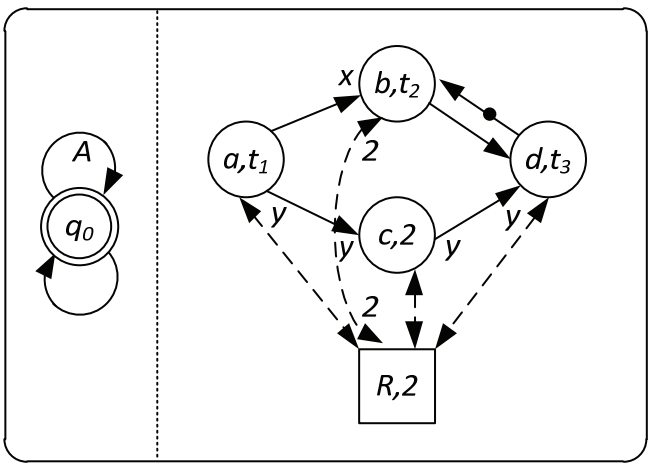
It is not very hard to see that, in (a), actors b and c cannot be executed in parallel since the resource constraint of R does not allow this to happen. So the schedule of the RASDF model under input A (i.e., the execution order of actors b and c) can be reconfigured based on the context (the two options are shown in Figure 6.2). In this example, reconfiguration means a change of the execution order of conflicting actors, i.e., changing the schedule.



(a) Schedule S_1 (b) Schedule S_2

Figure 6.2 Different schedules of the example application under input *A*

In general, a system can reconfigure many properties to adapt its behavior. Besides adapting its schedule to different scenarios, it may for example change the frequency and voltage, or resource allocation. We assume that the changes of behaviors after reconfiguration can be modeled as changed execution times and data/resource rate changes in RASDF. We call the entity that determines when to execute changes of behavior the ***controller.*** We model the behavior of RASDF graphs under different input types together as a SARA SDF graph (see Section 2.5) with parameterized execution times and data/resource rates. Figure 6.3 shows the SARA SDF graph of the example application of Figure 6.1 that captures system behaviors with both input A and input B.



Scenarios	Rates		Execution Times			Status
	x	y	t_1	t_2	t_3	c
S_A	1	1	2	4	2	enabled
S_B	2	0	3	2	6	disabled

Figure 6.3 SARA SDF graph of the running example

The design objective, i.e., throughput of the example application, is determined by the combination of inputs and the schedules of the application together. To determine the obtainable throughput of the system, we have to consider the roles of both inputs and schedules. The question that we have to answer contains the following three sub-questions.

- What is the highest throughput that we can obtain no matter what input sequence is encountered?

- What is the best strategy of the controller to reconfigure the system to obtain the highest throughput?
- What is the worst possible input sequence of the environment that we can have no matter what policy we use to configure system?

In this chapter, we approach the answers to these questions from a **game theoretic** point of view. Figure 6.4 illustrates the approach. The problem is viewed as a game played by two players: **environment** and **controller**. The environment player decides on the input sequence that feeds into the system while the controller player decides the schedule for every input. The worst-case situation refers to when the input sequence leads to the lowest throughput no matter how the controller reacts to it. The goal of controller player is to maximize the throughput while the 'goal' of the environment player is to minimize it (for worst-case analysis). Since we cannot really decide inputs for environment, we put 'goal' in quotes.

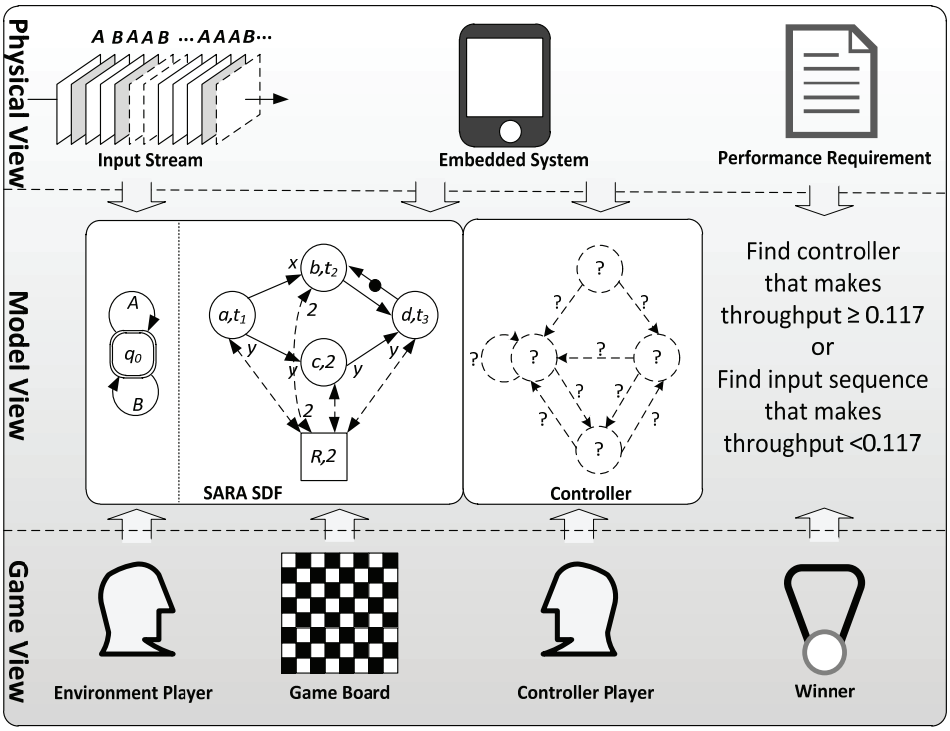


Figure 6.4 Game theoretic view interpretation of embedded system design

Figure 6.4 shows the design problem in three different views:

In the ***physical view***, a designer has to design an embedded system that processes a sequence of different types of input data. The performance requirement gives a throughput constraint that must be satisfied. Since the concrete sequence of types of input data is not predictable at design time, a designer has to design a controller that, for reason of efficiency, can control or reconfigure the system parameters such as scheduling or resource allocations based on the type of input data encountered.

In the ***model view***, the system itself is specified as an operational model (a parameterized RASDF graph) and the types of data (scenarios) that it supports, i.e., as a SARA SDF graph. The goal of the designer is to synthesize a controller that reconfigures the system based on the encountered input, while satisfying the throughput constraint. The possible input sequences are captured by the scenario FSM in the SARA SDF. The controller is also modeled as an, initially unknown, to be determined, FSM that gives schedules or configurations based on the current input and history of inputs.

Intuitively, we can interpret the interaction between two FSMs: scenario FSM and controller FSM as a ***game*** played between the environment player and the controller player. We call this the ***game view***, and use knowledge of game theory to quantitatively analyze the SARA SDF graph and synthesize a controller that is guaranteed to meet the throughput constraint.

To summarize, the problem is to find a winning strategy for the controller player (a controller FSM) to satisfy the throughput constraint of the embedded system no matter what its environment player (a scenario FSM) does, i.e., no matter what sequence of input data types are encountered.

In the following section, we briefly introduce some basic concepts of game theory, required for our approach.

6.3 Preliminaries of Game Theory

Game theory was originally developed as a mathematical tool to analyze games and economic behavior. Since then, it finds wide application in economics, communication, control and many other scientific disciplines where the interaction among multiple parts plays a central role in system behaviors. For embedded systems, by definition, the environment plays an important role.

With the advance of reconfigurability in embedded systems, game theory also became increasingly relevant in design and analysis of the interaction between environment and components of the platform [3, 48]. We briefly introduce some basics of game theory in this section that relate to the problem that we are interested in. For more mathematical backgrounds on game theory, we refer to [125].

A ***game*** [115] is a decision process that involves multiple decision-makers, so-called ***players***. The ***outcome*** of the game is determined by the ***actions*** of the players. The ***strategy*** of a player is a procedure or function that decides his response to the actions of other players during the game. Normally, we attach some rewards to the actions of players, and call it the **payoff** of the action. A game is called non-cooperative [114] if players make decisions independently. A zero-sum game is a, typical non-cooperative, game in which the sum of the payoff of players in the game is zero. The combination of strategies of all players in the game is called a joint strategy of the game. For a multiple-player game, if no player can achieve higher payoff by *unilaterally* switching to another strategy, the joint strategy of the game is called a **Nash equilibrium**.

A turn-based game assumes players make moves by taking turns one at a time. In our context, we assume the environment and controller play a turn-based game and the environment is the first player to make a move.

The task of synthesizing a controller for a system with desired throughput can be viewed as a turn-based game of infinite duration between two players: the ***environment player*** that provides different types of input and ***the controller player*** that configures the system with different schedules or other parameters in response to input observed. The winning condition of the game is given by the system specification. The goal is to find a strategy for the controller such that all sequences of inputs and configurations that can be produced according to the strategy satisfy the specification. The strategy itself can be seen as an implementation of the controller, i.e., the controller implementation always follows the actions decided by the winning strategy. For our kind of embedded systems, since environment and controller take their actions independently, the game is classified as a non-cooperative game.

Use of games as models for analysis and synthesis problems first occurs in [21], in which a specification is translated into a deterministic automaton and the circuit synthesis problem to the computation of a strategy on a finite game

graph. Here, we use a very similar approach, i.e., we translate the controller synthesis problem to finding a winning strategy for a well-known game, a so-called mean-payoff game on a bipartite game graph.

Definition 6.1: Bipartite game graph

A bipartite game graph is a tuple $G_g = (V_E, V_C, E_A, v_0, U)$ that consists of a set V_E of environment vertices, a set V_C of controller vertices, a set of action edges $E_A \subseteq E_E \cup E_C$ where $E_E = V_E \times V_C$, i.e., a set of edges from environment vertices to controller vertices, and $E_C = V_C \times V_E$, i.e., a set of edges from controller vertices to environment vertices, a starting vertex v_0 and a payoff function $U: E_A \mapsto \mathbb{R}$. The sets V_E and V_C are disjoint. We use $V = V_E \cup V_C$ to denote the set of all vertices.

A play on bipartite game graph G_g is a sequence $\alpha = v_0 v_1 \cdots$ of vertices, such that $(v_i, v_{i+1}) \in E_A$ for all $i \geq 0$ and a finite play with length n is denoted by $\alpha_n = v_0 v_1 \cdots v_n$.

In a bipartite game graph, we define the payoff function of player p on the ith edge $E_i = (v_i, v_{i+1})$ as follows:

$$U_p(E_i) = \begin{cases} U(E_i) & v_i \in V_p \\ 0 & v_i \notin V_p \end{cases}$$

where p is environment player E or controller player C.

We use $U_p(\alpha_n) = \sum_{i=1}^{n} U_p(E_i)$ to denote the total payoff of a player p in the play α. We are interested in plays of infinite duration (for streaming applications, we assume the input is infinite). Therefore, we assume that each vertex has at least one successor. For an infinite play sequence α, the total payoff of the play need not converge and may be growing forever for increasingly long prefix of α. Therefore, instead of total payoff, we are interested in the **mean payoff**

Definition 6.2: Mean Payoff Game

A *mean payoff game* is a game that is played by two players on a bipartite game graph and both players try to maximize the eventually lower bound of their mean payoffs in the game.

A strategy for the environment player in bipartite game graph G_g is a function $S_E: V^* V_E \mapsto V_C$ (V^* is the set of all finite prefixes of plays) and such that $S_E(xv) = v'$ in which x is the history of visited vertices during the play and v is the current vertex. The strategy for the controller can be defined symmetrically by swapping V_E and V_C. In many cases, a simple strategy that only depends on the current position, i.e., $S_E: V_E \mapsto V_C$ can already be optimal. Such a strategy is called a positional strategy and sometimes also called a memory-less strategy.

We have introduced the required basic concepts of game theory and mean payoff games. In the next section, we show the link between the game and the controller synthesis problem that we want to solve for SARA SDF.

6.4 Translation to a mean-payoff game

In Figure 6.5, there are two scenarios, s_A and s_B. For scenarios s_A and s_B, the repetition vectors are $q_{s_A} = [1,1,1,1]$ and $q_{s_B} = [2,1,0,1]$ respectively. Figure 6.5 shows an execution σ of the running example for scenario sequence $s_B s_A s_A s_B \cdots$, in which the x axis represents time, and the y axis is for resource tokens, T_i denotes the completion time of the ith iteration and L_i denotes the latency between two consecutive iteration completion times. We use a state-based execution model from the iteration view (see Chapter 5) to capture the execution of a SARA SDF. Every channel or resource token has a time-stamp which represents the time instant it was produced or released. The time-stamps of all tokens at the end of an iteration are collectively captured by a time-stamp vector representing the state. For example, the time-stamp vector for each iteration in the example execution in Figure 6.5 is denoted by circles annotated with their iteration numbers. This gives us time-stamp vectors γ (similar to dater functions in timed Petri-nets and the time-stamp vectors of SADF).

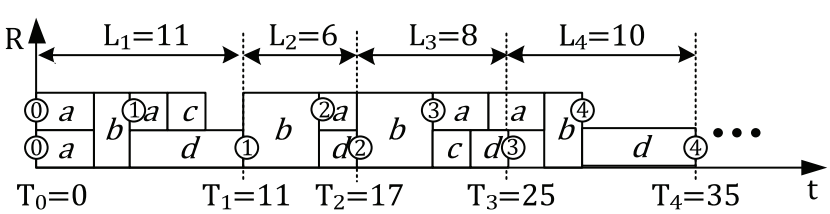


Figure 6.5 An Execution of Example SARA SDF

The time stamp vector together with the scenario FSM state q, are used as the iteration state of a SARA SDF graph. The time stamp vector includes the

available times of all tokens while the scenario FSM state q specifies all admissible inputs. An iteration state is thus denoted by a pair (γ, q). The time stamps of all tokens in the initial state are zero.

The exploration of an iteration state space of a SARA SDF graph can be listed as the following three steps.

Scenario selection and initialization: Before every new iteration of a SARA SDF graph, one of the transition edges in the scenario FSM is selected (scenario selection) and its corresponding scenario parameters are used to instantiate the RASDF graph (initialization).

Intra-iteration exploration: For each instantiated SARA SDF graph, i.e., an RASDF graph, we can compute the repetition vector of the selected scenario and the scheduling of actor firings in the iteration following the rules given in the RASDF graph for iteration-based exploration in Chapter 5. For a given execution according to a specific scenario sequence, we use $\gamma_i \in \mathbb{R}^n$ to denote the time-stamp vector after the ith iteration. We use $M_i: \mathbb{R}^n \mapsto \mathbb{R}^n$ to denote the schedule applied by the controller during the ith iteration by means of the effect it has on the state, in the form of an operator M_i such that $\gamma_{i+1} = M_i(\gamma_i)$. From initial state (γ_0, q_0) of a SARA SDF graph, we have to anticipate every possible scenario that q_0 accepts (for which it has a transition) and explore different schedules inside one iteration for each scenario. This generates different new iteration states.

Figure 6.6 shows the part of the iteration state space corresponding to the example execution in Figure 6.5. For example, at the initial state, all tokens are available at time 0 and it is in state q_0 of the scenario FSM which accepts both input types A and B. The environment player gives input type B and the controller decides to use schedule M_1 and the completion time compared to the start is 11.

Recall the maxplus algebra notations that we used in Chapter 5. $\|\gamma_i\|$ denotes the maximal time stamp in γ_i, which captures the completion time of the schedule for the ith input. The latency of the ith iteration is the completion time difference between γ_i and γ_{i-1}, i.e., $L_i = \|\gamma_i\| - \|\gamma_{i-1}\|$. In the example execution, the normalized vector of γ_1 is the same as the normalized vector of γ_4, i.e., $\gamma_1^{norm} = \gamma_4^{norm} = [-6,0]$ and they have the same input state q_0. Therefore the schedule M_4 results in a back edge from (γ_3^{norm}, q_0) to (γ_1^{norm}, q_0). The cycle in the state space allows for a periodic execution $\sigma_{per} = ((\gamma_1, q_0) \cdot$

$(\gamma_2, q_0) \cdot (\gamma_3, q_0))^{\omega}$. We will see that best and worst case performance is found on such cycles in the state space.

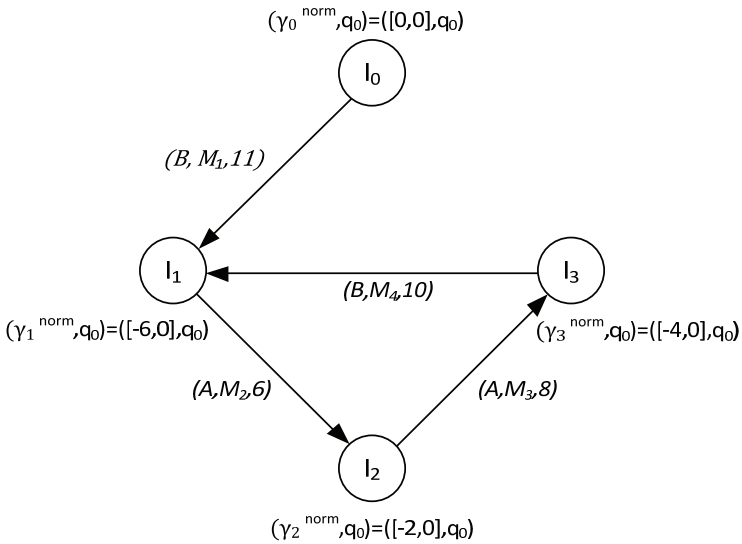


Figure 6.6 Part of the iteration state space for the example execution

Checking recurrent states and approximation: For every new generated iteration state, we will check it with the previously generated iteration states to avoid redundant exploration. If the current iteration state after normalization equals the normalization states of some already existing state, there will no further exploration on this visited state. Then the exploration backtracks to the previous state and tries to explore other possible input scenarios until all scenarios that that previous state accepts are tried. If a predefined iteration limit is reached and no recurrent states are found, a similar approximation to that in Chapter 5 is used to approximate some visited iteration state. The only difference here is that the approximated state needs to have the same scenario state as the current state.

By this process, we can construct the iteration state space of a given SARA SDF graph. Figure 6.7 shows the whole iteration state space of the running example. Due to different schedule possibilities, there are multiple different execution paths from the same iteration state, e.g., I_3 can select either schedule M_9 or schedule M_{10} for input type A.

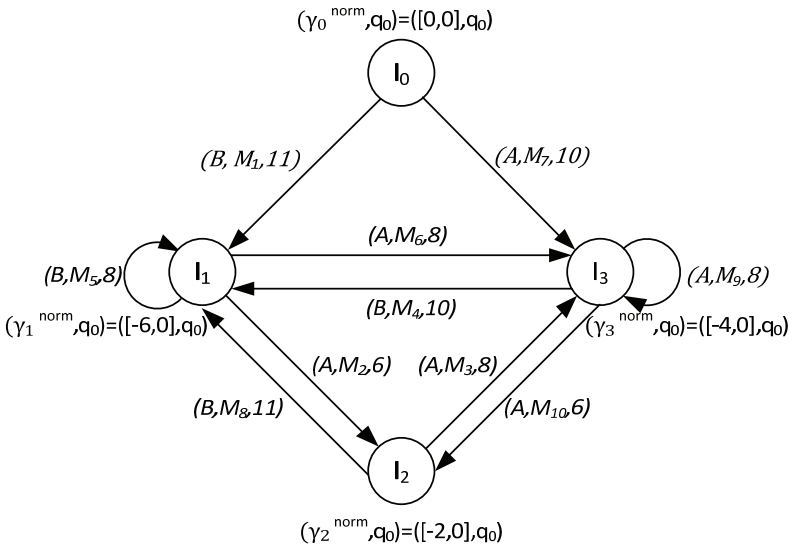


Figure 6.7 Iteration state space of the running example.

Like the throughput computation in Chapter 5, we can also compute the throughput of an execution of a SARA SDF graph from its state space. Given a SARA SDF graph and some execution $\sigma = (\gamma_0, q_0) \cdot (\gamma_1, q_1) \cdots (\gamma_n, q_n) \cdots$, the throughput of σ is, as usual, defined as the infimum limit of the number of iterations completed, divided by their completion time, or equivalently, as the reciprocal of the average latency, i.e.,

$$Th(\sigma) = \liminf_{n\to\infty} \frac{n}{\max(\gamma_n)} = \liminf_{n\to\infty} \frac{n}{\sum_{i=1}^{n} L_i}$$

The throughput of the execution with periodic input scenario sequence $s_B(s_A s_A s_B)^\omega$ in Figure 6.5 is $\frac{3}{L_1+L_2+L_3} = \frac{3}{24} = \frac{1}{8}$.

The throughput of a system depends on the choices of the controller. Let $\mathbb{C}$ denote all possible controllers of a given SARA SDF graph, and let $C \in \mathbb{C}$ be a particular controller. The throughput of controller C is defined as the infimum (worst case) of throughputs of all executions that are generated from arbitrary input scenario sequences, denoted by Σ_C.

$$Th(C) = \inf_{\sigma\in\Sigma_C} Th(\sigma)$$

Then, in turn, we can define the throughput of a SARA SDF graph G_{SARA} as the best possible throughput any controller can achieve, the supremum of the throughputs of all possible controllers of G_{SARA}.

$$Th(G_{SARA}) = \sup_{C \in \mathbb{C}} Th(C)$$

Note that, although the use of supremum suggests that a controller achieving the throughput need not exist, we will see in the following part that it always does.

Input scenarios and controller decisions thus form opposing forces that in their interaction determine the actual throughput. In the following, we explicitly model this in order to synthesize an optimal throughput controller that responds to different scenarios and to check whether it can satisfy the given performance constraints using the shared resources.

We use the latency of an iteration, i.e., L_i, as the payoff of the environment move (while $-L_i$ is the corresponding payoff of the controller move), then the average latency of an execution (the reciprocal of the throughput) is the mean of the payoffs of all iterations in a play of the game.

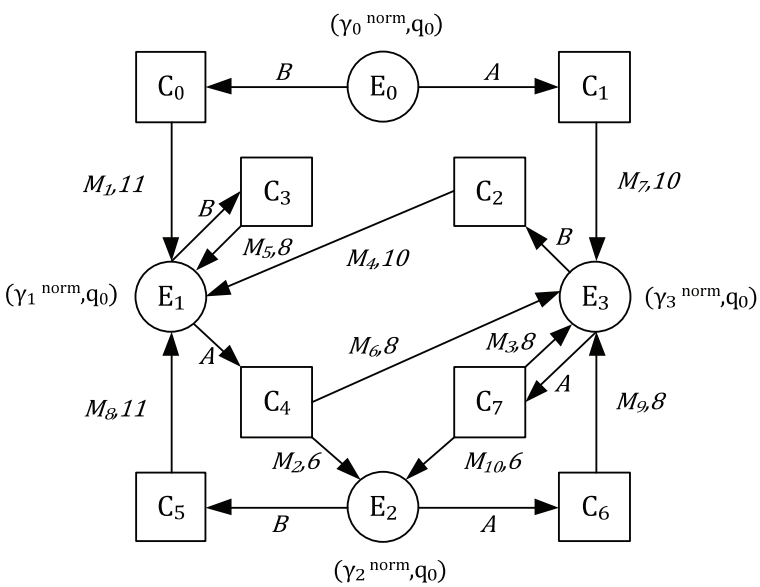


Figure 6.8 Bipartite game graph

Figure 6.8 shows the bipartite game graph constructed from the iteration state space of the running example (we explain below how it is constructed). The operation of a system controller on a given (infinite) input sequence can be viewed as a play of the game with infinite duration between two players on the graph: one iteration in an execution is one round of the game that includes an environment turn and a controller turn. The environment player (circular nodes) provides types (scenarios) of input (outgoing edges annotated with scenarios) and the controller player (square nodes) configures the system with different settings, schedules or other parameters (edges annotated with the corresponding schedules and with resulting latencies).During the state-space exploration, we create a new environment node for every new iteration state that we encounter and label the new node with the iteration state (normalized time-stamp vector and state of the scenario FSM). For every environment node, and each possible next input scenario from that state, a new controller node is created with an edge between the environment node and the new controller node, annotated with the selected input type. Next, we perform an intra-iteration exploration in which we explore different scheduling possibilities. For every new schedule found, we compute the normalized time stamp vector at the end of the iteration and check recurrence with existing environment nodes. An edge is created between the controller node and a newly created or revisited node. The edge is labeled with its schedule and the latency of the iteration. Note that it is not necessarily optimal to simply select the schedule with the smallest latency as the resulting end-state may have a negative effect on future behavior.

6.5 Solving the mean-payoff game

Once the bipartite game graph is built, we can analyze it. For a mean-payoff game, the adversarial environment player wants a strategy to maximize the average payoff per move, i.e., maximize average latency, *no matter how the controller player reacts,* i.e., no matter which schedule policy is chosen, the average latency is the largest. (Recall that this player's 'desire' to win and play an optimal strategy captures worst-case environment behavior. We do not assume that the environment providing input data has any real intention to lower throughput.) At the same time, the controller player wants to minimize the average loss per move, i.e., minimize average latency, *no matter how the environment player reacts,* i.e., no matter what type of input is given, the average latency is minimal. In a nutshell, both environment player and controller

player do not want to change their policies unless both sides change together, i.e., they reach a Nash equilibrium.

For controller synthesis, we prefer to find a strategy that only depends on the current state and not on the history of previous states, i.e., a positional strategy on the bipartite game graph. It is shown in [46] that there exist optimal positional strategies for both environment and controller to obtain a Nash equilibrium point with some mean payoff v [114]. And [180] shows the time complexity to find such an optimal positional strategy. So the optimal strategy of the controller to obtain maximal throughput, i.e., the reciprocal of the mean latency, is found by solving the mean payoff (i.e., mean latency) game. We use the algorithm from [42] to synthesize such an optimal positional strategy. The algorithm works by means of a process called *policy iteration* that has been first used in finding optimal policies for Markov Decision Processes (MDP). The steps of the policy-iteration algorithm are illustrated in Figure 6.10 for the example bipartite game graph in Figure 6.8 and are as follows.

Step 1 (Initialization): First, we initialize a randomly selected environment strategy $S_{E,0}$ and a randomly selected controller strategy $S_{C,0}$ on the bipartite game graph, i.e., each node of the environment and controller players randomly select exactly one outgoing edge as their next move for a given location. . The red edges are the moves of the environment player while the green edges are the schedules of the controller player (black edges are non-preferable actions of players. (see Figure 6.10 (a)).

Step 2 (Performance evaluation): After initialization, we traverse the game graph by following the initialized input types and schedules in depth-first search (DFS) manner. The DFS is started from all the nodes without any incoming strategy edges. Since the number of nodes in the game graph is finite (Recall that the iteration state space of an RASDF graph is finite) and every node has a unique successor node when the strategies are being followed, we always end up in a cycle using DFS from which we can compute the cycle mean, denoted by λ, i.e., the mean payoff (1/throughput) of the chosen environment-controller joint strategy. The larger λ is, the lower the throughput is. The goal of the controller player is to obtain the maximal throughput, i.e., the minimal λ while the goal of environment player is to obtain the minimal throughput, i.e., the maximal λ. We annotate the first found revisited node during DFS with a pair of numbers (τ, λ) in which τ is the offset to reaching the cycle mean from the annotated note and λ is the cycle mean of the cycle

reached. Since the revisited node is the first node to be annotated, the offset τ to the cycle mean is zero.

Step 3 (Value Propagation): We propagate the computed cycle mean (average latency) of a cycle to all nodes that will reach the cycle, in a reversed order, i.e., start from the revisited node and visit parent nodes in a DFS manner. Nodes obtain the same throughput if they can reach the same cycle, since they have the same cycle mean. The delay τ_C for controller node v_C is computed by $\tau_C = \tau_E + L - \lambda$ in which τ_E is the offset of v_E, i.e., the successor node of v_c and L is the latency attached to the edge between v_C and v_E. The offset τ_E for environment node v_E is equal to the delay of its successor controller node since it only decides on input. Figure 6.9 shows an execution given by Figure 6.10 (c) that starts from controller node C_1 and repeats between C_6 and C_7 for input sequence A^ω. It is not hard to see that the offset to the cycle mean of each controller node is the vertical distance between controller node and the dashed line where the slope of the dashed line is the cycle mean. The propagation of the cycle mean and delay ends when all nodes are updated.

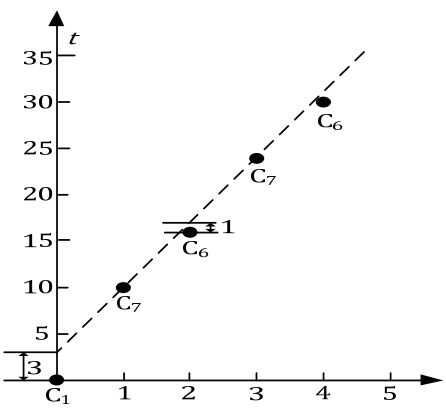
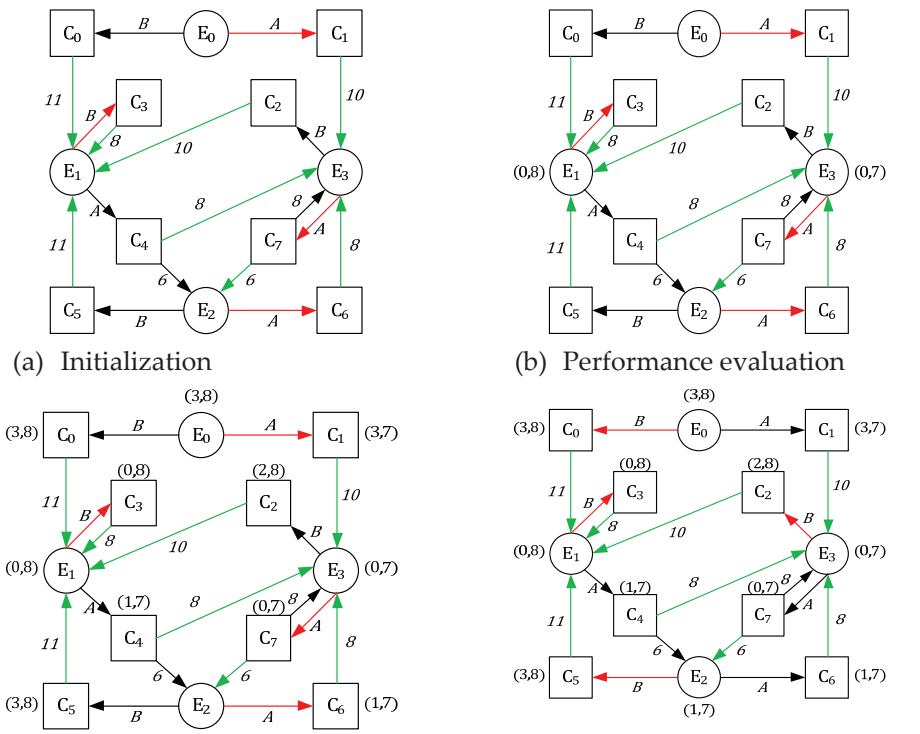


Figure 6.9 An execution given by bipartite game graph

Step 4 (Environment policy iteration): After value propagation, we start to iterate the environment policy. If the environment node can obtain a better payoff (larger cycle mean or equal cycle mean but with larger offset which leads to larger latency of one iteration), by changing input type, it will change its strategy, i.e., it will select the outgoing edge that leads to the larger cycle mean or higher delay. Note that, the larger cycle mean has higher priority than offset. Only when the cycle means to possible successors are equal which

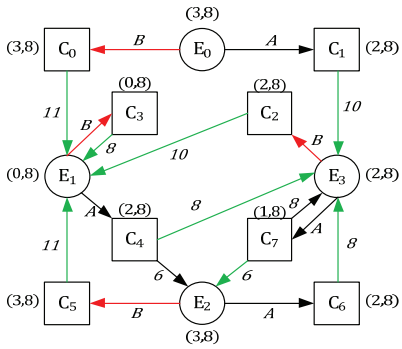
means they can reach the same throughput, then the offsets start to play a role in strategy choosing (see Figure 6.10 (d)).

Step 5 (Controller policy iteration): When the environment strategy does not change anymore, and is optimal for the given controller strategy, we start to evaluate the strategy of the controller. If a controller node can obtain better payoff (i.e., smaller cycle mean, then less delay) by changing its scheduling strategy, it changes it policy and chooses the schedule explored during iteration state-space exploration that can reach less cycle mean or equal cycle mean but with less delay. When no controller nodes can improve their decision anymore, we find the best controller strategy for the current environment strategy (see Figure 6.10 (f)). We repeat the steps 2 to 5 until neither controller nor environment player can improve their own strategies (see Figure 6.10 (g) to 6.10 (i)). This must eventually happen and we reach a Nash equilibrium of joint strategies for both environment and controller players (see Figure 6.10 (j)).

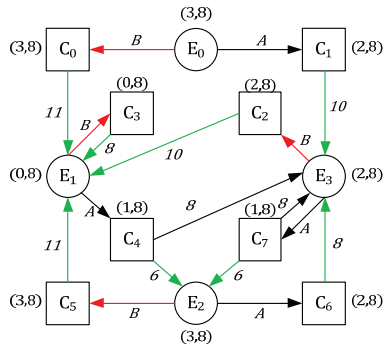


(a) Initialization (b) Performance evaluation

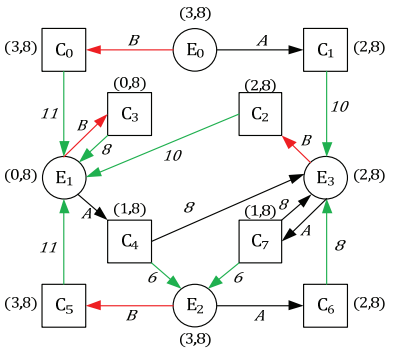
(c) Value propagation (d) Environment policy iteration



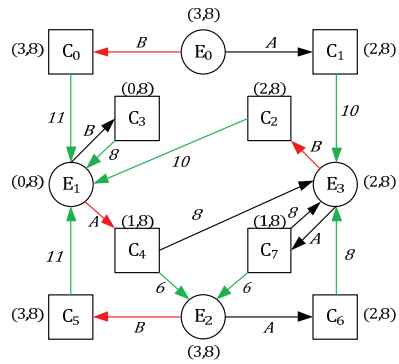
(e) Performance evaluation & Value propagation



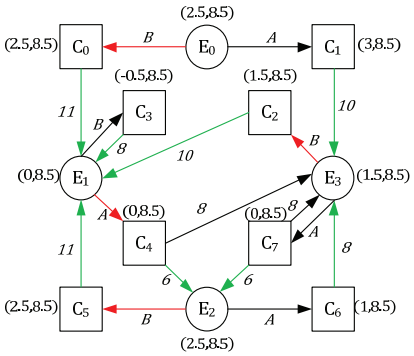
(f) Controller policy iteration



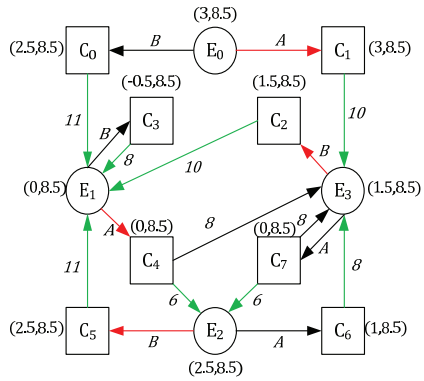
(g) Performance evaluation & value propagation



(h) Environment policy iteration



(i) Performance evaluation & value propagation



(j) Nash equilibrium

Figure 6.10 Policy iteration on a bipartite game graph

The controller obtained for the example graph is given in Figure 6.11. Figure 6.12 gives all the schedules executed by the controller in response to input from the environment. The grey blocks in figure denote the state before a new input arrives. For example, when the controller received input with type A at state C_4, it would choose schedule M_2 that starts firing c before firing b instead schedule M_6 that starts firing b before firing c. If both environment and controller follow their optimal strategies, they enter into a cycle, i.e., $(E_1C_4E_2C_5)^\omega$. The guaranteed throughput of this controller is 1/8.5 (the cycle mean of the cycle) under the worst-case input sequence $(AB)^\omega$. Note that the controller strategy also specifies schedules in response to non-worst-case input scenarios.

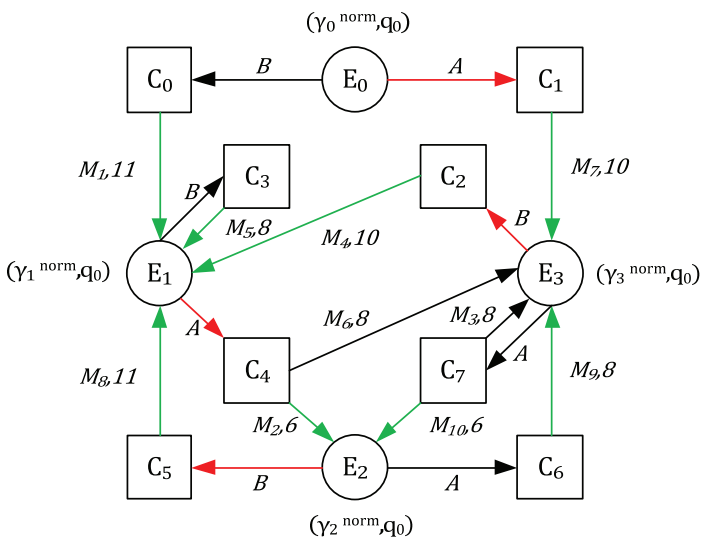
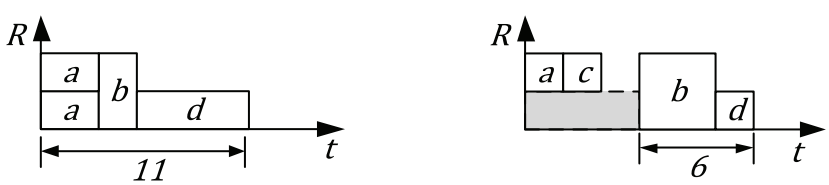
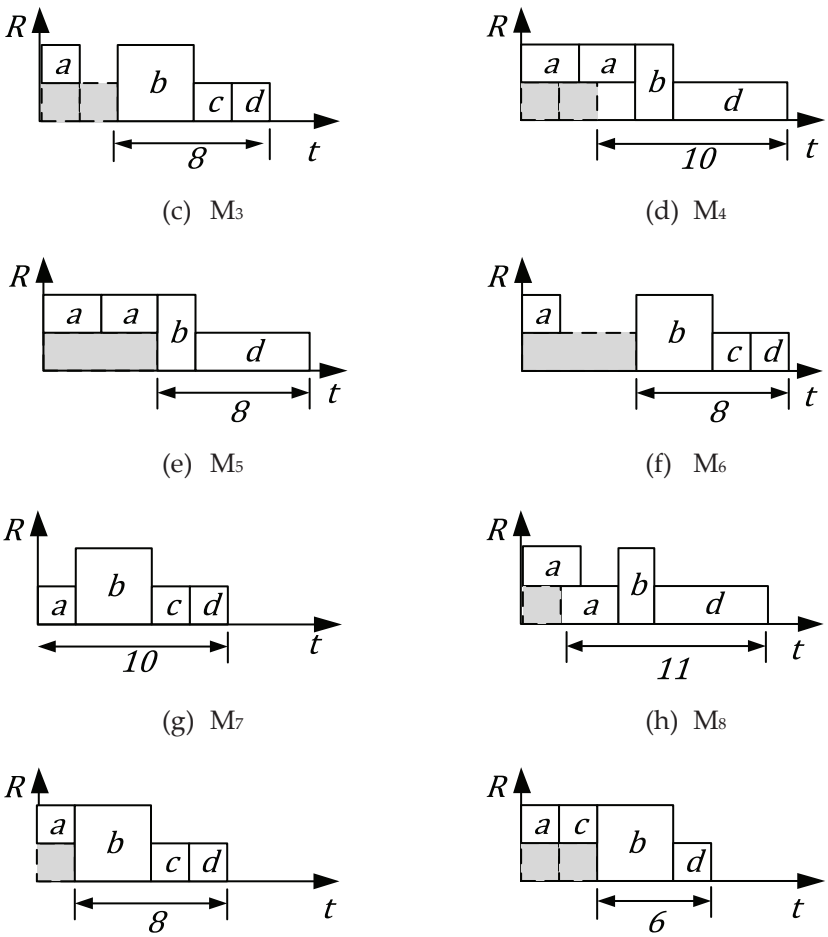


Figure 6.11 Synthesized controller of the running example



(a) M1 (b) M2



(c) M3 (d) M4

(e) M5 (f) M6

(g) M7 (h) M8

(i) M9 (j) M10

Figure 6.12 Synthesized controller of the running example

6.6 Case Studies

We implemented the proposed approach and tested it on four applications that are specified as SARA SDF graphs and we have synthesized their controllers. The first test graph is the SARA SDF running example detailed in this chapter. The others are a system specification of an MP3 decoder application, a system specification for an MPEG-4 SP decoder, and a system specification of a printer image processing pipeline taken from a real industrial case study. The PC used

for the experiments is a 64-bit Linux system with an Intel 2.8GHz Core™ i7 with 8GB memory.

Table 6.1 shows the experimental results of the controller synthesis for four cases. The SARA SDF graphs are shown in the appendix. We use similar techniques to those in Chapter 5 to prune the design space. Limits were imposed on the iteration depth and on schedule branches to reduce the size of the iteration state space, and therefore also the size of the synthesized controller. The synthesized controllers can reach the throughputs given in the table. The throughput results are optimal. For MP3 and MPEG-4 SP, the synthesized controllers reach the same throughputs as we know from [62]. MPEG-4 SP has a long exploration time due to its large repetition vectors (many actor firings need to be scheduled in each iteration). The execution time results show that the algorithm spends most of its time on constructing the game graph and that the controller synthesis part is only a small fraction of the total analysis time. A more efficient game graph construction method would be a beneficial future development. Note that the refinement of execution times with smaller time units and the involvement of more resources will lead to a larger state space size.

	Example	**MP3**	**MPEG-4 SP**	**Printer**
Iteration depth limit	2	3	3	5
Branching limit	4	4	4	4
State space size	4	309	26	609
Game graph size	12	1854	260	3654
Graph construction time (ms)	409	126387	224927	74881
Synthesis time (ms)	1	36	5	89
Throughput	1.18×10^{-1}	1.71×10^{-7}	1.41×10^{-3}	1.42×10^{-3}

Table 6.1 Controller synthesis results

6.7 Related work

The work in this chapter has developed based on integrating results from the following three fields: decision and game theory, max-plus algebra and automata theory or more specifically dataflow models.

The game we investigated can be classified as a non-cooperative game, as introduced by John Nash [114]. Our environment and controller converge to fixed policies that represent a Nash equilibrium point, which means neither

can improve their strategy on the current opponent strategy. [46] introduces mean payoff games and shows the existence of optimal positional (memory-less) strategies. Our throughput game is modeled as a so-called infinite mean payoff game with perfect information. [180] investigates the complexity of solving mean payoff games and shows the complexity is in $NP \cap co-NP$ by reducing it to a simple stochastic game. Karp's algorithm [94] can be used for playing against a known positional strategy, which amounts to computing the maximal cycle mean (MCM) of the game graph.

The technique of policy iteration was invented by Howard to solve stochastic control problems [82], so-called Markov Decision Processes [12]. Later, it was generalized to stochastic games [80]. However, the generalization requires strictly positive transition probabilities and cannot be directly applied to deterministic games. The applications of policy iteration to deterministic games appeared later in the study of min-max functions [28, 53, 73, 74].

A game-theoretic approach is also widely used in controller synthesis for automata. The approach using games as a model for the synthesis problem has been proposed in [21] in which the specification is translated into a deterministic automaton. The synthesis problem is reduced to the computation of a strategy on a finite game graph. Priced timed automata [126], for instance, are used to model costs and real-time constraints and it is shown that the problem of finding a winning strategy can be modeled as a reachability problem.

In the study of control theory, max-plus algebra [6] is widely used in performance analysis of systems with synchronization primitives, such as timed Petri nets [52, 54, 55]. Heaps-of-Pieces is a model that combines Max-plus analysis and automata for performance analysis [52, 55]. Spectral analysis techniques [6, 29, 54] are used to analyze the asymptotic timing behavior of such timed Petri nets. [28, 29, 42, 53] provide practical algorithms for spectral analysis, to compute the max cycle means of graph which are faster than Karp's algorithm in practice.

SDF is a special subclass of Petri nets (corresponding to Weighted Marked Graphs) and the data and resource consistency can be expressed as place invariants of Petri nets. [151] introduces scenarios of dataflow behavior in SADF graphs using Markov Chains and introduces state-based methods to analyze performance. [13] introduces parameterized SDF graphs for functional

analysis of different scenarios. [62] utilizes the iteration concept and uses max-plus algebra and max-plus automata to analyze the performance of SADF graph in worst-case scenarios.

The work described in this chapter proposed a new dataflow model, i.e., so-called SARA SDF to model streaming applications with dynamic inputs and resource sharing. By reducing the controller synthesis problem to a mean-payoff game on a bipartite game graph converted from the iteration state-space explored, we can directly use the algorithm developed in [30, 42] to find the Nash equilibrium, i.e., giving the throughput and strategy for both controller and environment.

6.8 Summary

The chapter introduces a new design approach from a game-theoretic viewpoint to tackle the controller synthesis problem of an embedded system with dynamic inputs. The novelties of our method are the following: modeling of dynamics in resource-sharing streaming applications, and capturing the environment-controller interaction as a mean-payoff game, yielding a method to synthesize a throughput-guaranteeing controller and to identify the worst-case situation of scenarios. The experimental results show synthesis results; the controller synthesis time is only a small fraction of the construction time of the game graph. Faster construction of the game graph by limiting the scheduling options or iterating known scheduling techniques will be investigated in the future. Analyzing each scenario separately and combining their iteration state space together to generate a game graph can be a very good candidate approach. The adversarial environment player can be replaced by a stochastic environment player; the game then becomes a stochastic game that can be used to analyze average performance of a system with known input distribution. Existing results for Markov Decision Processes and reinforcement learning can be investigated for this type of game. This chapter and the discussion of related work show strong links among studies in game theory, max-plus algebra and automata models such as SDF. Since the future of embedded system design will be challenged by more and more dynamic environments, a systematic way of applying the results of the three fields to tackle the challenges seems very promising.

7 Conclusions and Outlook

"The philosophers have only interpreted the world; the point is to change it."

-Karl Marx

In this chapter, we give our conclusions on our work and discuss interesting future work directions.

7.1 Conclusions

The better your understanding about your system, the more efficient system you can design. In order to design streaming applications efficiently, we have to take the context of these streaming applications into consideration, i.e., the resource constraints on them and the inputs for them, For this reason, we explicitly model resources and inputs with Synchronous Dataflow (SDF) graphs and introduce two new variants of SDF graphs in this thesis: Resource-Aware SDF (RASDF) graphs and Scenario- and Resource-Aware SDF (SARA SDF) graphs. By carefully choosing the operational semantics, i.e., fixed rates for resource production and consumption and the timing properties for RASDF graphs and SARA SDF graphs, the consistency and iteration concept known from SDF graphs are preserved into the new models.

This thesis studies the Design Space Exploration (DSE) problem, i.e., how to find non Pareto-dominated design points (so-called trade-off points) in a multi-dimensional metric space, for a streaming application system modeled by RASDF graphs. Throughput and resource usage are chosen as two metrics for trade-off analysis of a streaming application. Trade-off analysis techniques for RASDF graphs are developed in Chapter 4 in the time domain. The execution of an RASDF graph is interpreted as an alternating sequence of states and transitions. The introduction of resource sharing to SDF graphs provides opportunities of more resource-efficient executions; it also implies that the execution path is no longer unique but that there may be multiple possible schedules with different actor firing orders. The Pareto-optimality criterion helps us to accelerate analysis by pruning redundant executions. The case studies show explorations with larger but still very reasonable exploration times, normally a few minutes, caused by the much larger state space due to resource sharing options. The case studies also show that resource sharing can reduce resource usage by more than 50% compared to the non-sharing

resource case. In order to efficiently explore resource configurations for RASDF graphs, we developed a bottleneck-driven approach for design-space exploration. The bottleneck-driven approach can be seen as a feedback driven approach in which the algorithm identifies the resource bottleneck in periodic executions by constructing dependency graphs and detecting dependency cycles in the dependency graphs. These potential bottlenecks (resources in the dependency cycles) are used to determine the next resource configuration to be explored. This bottleneck-driven approach significantly reduces the exploration time compared to a non-bottleneck-driven approach.

In Chapter 5, we try to solve the trade-off-analysis problem from a different angle, i.e., the iteration-based point of view. We only have to store the iteration states, i.e., the states of every new iteration start rather than all the states within every iteration. This reduction leads to a lower memory footprint for analysis and speeds up the analysis. However, there are some disadvantages too. The separation between iterations at their boundaries does not allow us to interleave activities from different iterations for potentially more efficient resource usage. This may cause the iteration-based analysis to lose some trade-offs. On the other hand, also the time-based analysis may not be able to find all trade-off points, because also the time-based analysis needs pruning heuristics in order to scale to realistic models. Designers can decide whether a schedule with interleaving is allowed or not. The choice of applying time-based, iteration-based or the combination of the methods is a trade-off that can be made by designers themselves. An important advantage of the iteration-based approach is that it provides the basis to analyze dynamic behavior of streaming applications caused by different input types in Chapter 6,

Dynamic behavior such as data-dependent execution times is always a challenge for performance analysis providing both guaranteed and tight bounds on timing and resource usage. Iteration-based analysis allows to capture data-dependent or, more precisely, scenario-dependent behavior. The scenario concept helps us to describe behaviors that are dependent on some parameters that are stable for some period; the period can be expressed either in terms of time or in terms of iterations. SARA SDF graphs combine the scenario concept of Scenario-Aware Dataflow (SADF) and resource-awareness of RASDF. Performance analysis of SARA SDF graphs is different from existing performance analysis of FSM-SADF graphs or SADF graphs. Due to multiple possible execution paths caused by arbitration decisions of conflicting actor firings on shared resources, the performance is not only decided by the

environment input but also by the decisions of controllers, i.e., the firing order of actors on shared resources. How to find a controller that guarantees a certain throughput no matter what kind of input is given by the environment? Chapter 6 investigates this question from a game-theoretic view and reduces the controller synthesis problem to the problem of finding an optimal positional strategy for a mean-payoff game on a bipartite game graph. By translating the iteration state space of a SARA SDF graph to a bipartite game graph of a mean-payoff game played by the environment and the controller, a throughput-guaranteed controller can be synthesized from a policy iteration algorithm developed in the field of game theory.

In summary, we have made important steps in the development of analysis and synthesis methods for SDF models of dynamic streaming applications executing on multi-processor platforms with resource sharing. The methods provide a basis for platform dimensioning and design-space exploration, and for synthesizing controllers that provide guaranteed performance under workload variations.

7.2 Open Questions and Future Work

The work in this thesis provides some interesting observations and open questions for future investigations.

Fixed point analysis: The throughput analysis, resource usage evaluation, and controller strategy synthesis developed in this thesis can be classified as fixed point computation problems, i.e., the value being computed is over time converging to a specific fixed point, in performance, resource space or strategy space. One interesting direction for future work is to investigate the findings in fixed point theory and to find mappings between the results of fixed point theory in mathematics to the analysis of properties in the embedded systems domain.

Parameterized analysis: Many important system parameters are changing with time or inputs. We can either know the range of parameter values or the rule of how they change. One exhaustive way to analyze such a system is to list values for these parameters and to iterate each combination of parameters to evaluate system. Compared to such an exhaustive analysis, a smarter way would be to analyze the system symbolically, i.e., analyze executions of a system on symbolic parameters directly. For throughput and consistency analysis of SDF graphs, some work in this direction was already done [13, 63].

Interfacing theory for different dataflow analysis models: There exist quite a lot of different models of computation for performance analysis, Petri nets, SDF graphs, Events models, Network Calculus, Real-time Calculus, etc. There will likely be many more new models in the future. While new models may be useful, because they can express some aspects more efficiently or allow analysis more efficiently, they also lack capabilities to efficiently analyze or exactly analyze problems fitted better into other models. The designers have to make a trade-off and choose a specific model for their purposes. For example, SDF graphs are more efficient for performance analysis while Petri nets are more expressive; Events models allow easy analysis of jitter in systems while it is difficult to handle loop-carried dependences. It would be beneficial if we can interface different models seamlessly and use strong aspects of different models to solve different problems. For instance, the resource usage requests in RASDF graphs can be viewed as an arrival curve in Network Calculus; the throughput analysis of SDF graphs can be viewed as the frequency analysis of linear bounds in max-plus algebra for Petri nets. By developing interfacing theory, we may model sub systems or different levels of systems with their most appropriate models. Tools such as Ptolemy II project [23] in Berkeley and Octopus toolset [10] from TU/e are examples of research work that try to interface different models.

Besides the general questions raised by the research work presented in this thesis, we also see some specific questions related to RASDF and SARA SDF.

Probabilistic analysis: It will be a very interesting extension to SARA SDF to allow analysis on scenario-FSMs that are annotated with probabilities on every scenario transition, i.e., using Markov chains to model inputs transitions. How will this influence our controller synthesis to reach better performance and resource usage? Markov decision processes may be used for controller synthesis for the probabilistic analysis case.

Parametric and multi-objective analysis based on a game-theoretic approach: The interaction between dynamic inputs and the controller of an embedded system is viewed as a game between two players in this thesis. We find some interesting directions to continue this approach. For instance, we may treat parametric analysis of RASDF graphs as a parametric game on a parametric bipartite game graph. The multi-objective analysis may be treated as a multi-player game in which multiple players compete with their individual goals. The trade-offs in multi-dimensional space may be interpreted as multiple Nash

equilibriums in the strategy space of the multiple players, compared to only one Nash equilibrium for the controller synthesis problem discussed in this thesis. The players can cooperate for switching among trade-off points (from one Nash equilibrium to another).

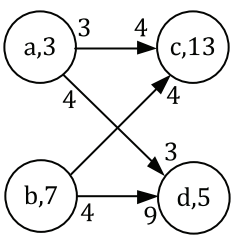
Hierarchical modeling and analysis: Real-world systems can be very large and the modeling process may involve many designers. Can we build RASDF or SARA SDF models for subsystems, analyze them compositionally, connect the component analyses together, and generate the results (performance, resource trade-offs, controllers) for the whole system?

A model-based approach to system design allows us to have a deeper understanding of existing questions and find interesting new questions. The goal of modeling embedded systems is not only to interpret the real system, but also to change the existing design process and allow better designs. The complexity of embedded systems and the context they are embedded in makes it impossible for our thesis to cover all interesting aspects. Instead, streaming applications and SDF models were chosen as our focus. You can never reach perfection in every aspect; there are only trade-offs!

APPENDIX

In this appendix, we include six SDF graphs: an artificial bipartite graph from [16], a sample rate converter [16], a modem [16], a satellite receiver [131], an MP3 decoder [144] and an H.263 decoder [144]. Since their corresponding RASDF graphs contain a lot of request edges, we show these request edges in the tabltabular formate for clarity reasons. For each SDF graph, there are two corresponding RASDF graphs, one with only one shared buffer and the other with two or three shared buffers. We also put the three SARA SDF graphs used in Chpater 6 in this appendix and provide their scenario parameters. All models are also available through http://www.es.ele.tue.nl/sdf3.

Bipartite Graph



- One shared buffer: Buffer 1

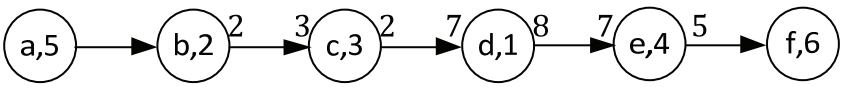
actor	resource	claim	release
a	Buffer 1	7	0
b	Buffer 1	5	0
c	Buffer 1	0	8
d	Buffer 1	0	12

- Two shared buffers: Buffer 1 and Buffer 2

actor	resource	claim	release
a	Buffer 1	3	0
a	Buffer 2	4	0
b	Buffer 1	1	0
b	Buffer 2	4	0
c	Buffer 1	0	8

d	Buffer 2	0	12

Sample Rate Graph



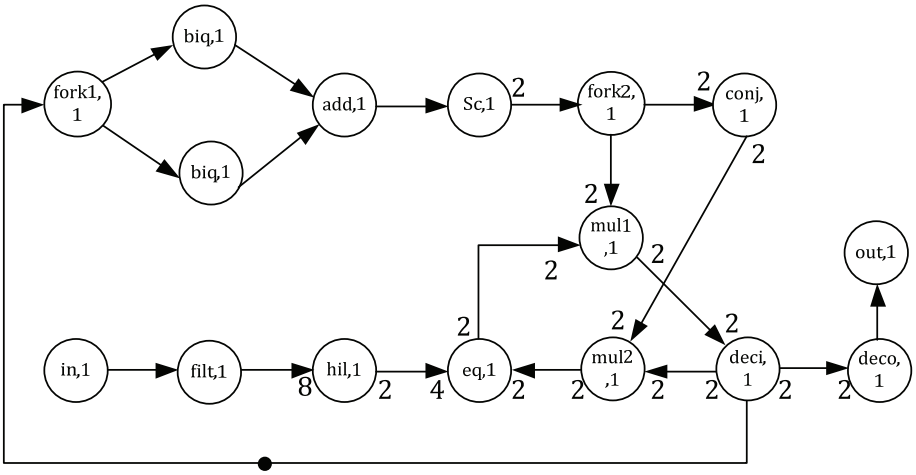
- One shared buffer: Buffer 1

actor	resource	claim	release
a	Buffer 1	1	0
b	Buffer 1	2	1
c	Buffer 1	2	3
d	Buffer 1	8	7
e	Buffer 1	5	7
f	Buffer 1	0	1

- Three shared buffers: Buffer 1, Buffer 2 and Buffer 3

actor	resource	claim	release
a	Buffer 1	1	0
b	Buffer 1	2	1
c	Buffer 1	2	3
d	Buffer 1	0	7
d	Buffer 2	8	0
e	Buffer 2	0	7
e	Buffer 3	5	0
f	Buffer 3	0	1

Modem Graph



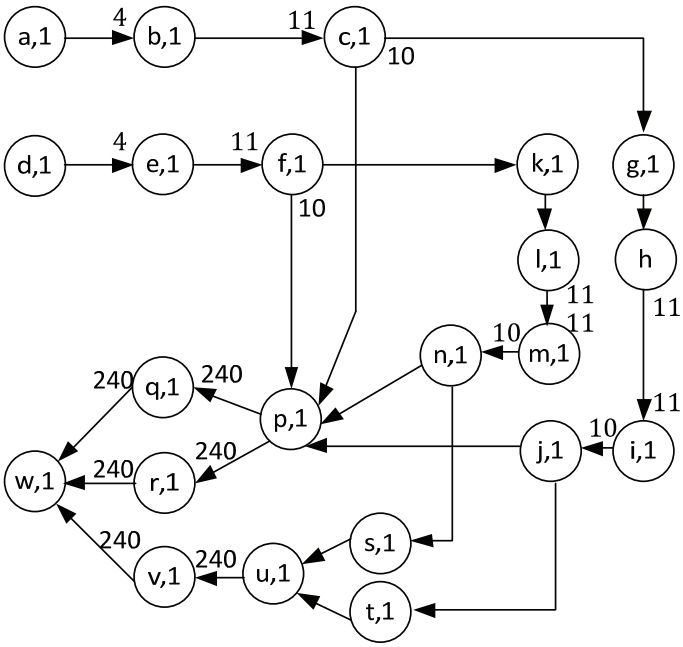
- One shared buffer: Buffer 1

actor	resource	claim	release
fork1	Buffer 1	2	1
biq1	Buffer 1	1	1
biq2	Buffer 1	1	1
add	Buffer 1	1	2
sc	Buffer 1	2	1
fork2	Buffer 1	2	1
conj	Buffer 1	2	2
mul1	Buffer 1	2	4
mul2	Buffer 1	2	4
in	Buffer 1	1	0
filt	Buffer 1	1	1
hil	Buffer 1	2	8
eq	Buffer 1	2	6
deci	Buffer 1	5	2
deco	Buffer 1	1	2
out	Buffer 1	0	1

- Three shared buffers: Buffer 1, Buffer 2 and Buffer 3

actor	resource	claim	release
fork1	Buffer 1	2	1
biq1	Buffer 1	1	1
biq2	Buffer 1	1	1
add	Buffer 1	1	2
sc	Buffer 1	2	1
fork2	Buffer 1	2	1
conj	Buffer 1	2	2
mul1	Buffer 1	2	4
mul2	Buffer 1	2	4
in	Buffer 2	1	0
filt	Buffer 2	1	1
hil	Buffer 1	2	0
hil	Buffer 2	0	8
eq	Buffer 1	2	6
deci	Buffer 1	3	2
deci	Buffer 3	2	0
deco	Buffer 3	1	2
out	Buffer 3	0	1

Satellite Graph



- One shared buffer: Buffer 1

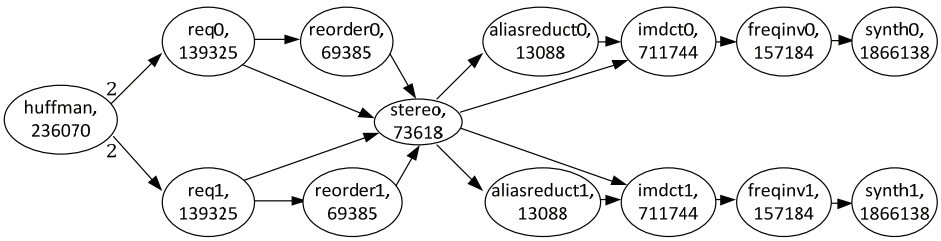
actor	resource	claim	release
a	Buffer 1	1	0
b	Buffer 1	1	4
c	Buffer 1	11	11
d	Buffer 1	1	0
e	Buffer 1	1	4
f	Buffer 1	11	11
g	Buffer 1	1	1
h	Buffer 1	11	1
i	Buffer 1	10	11
j	Buffer 1	2	1
k	Buffer 1	1	1
l	Buffer 1	11	1
m	Buffer 1	10	11
n	Buffer 1	2	1
p	Buffer 1	2	4

q	Buffer 1	240	240
r	Buffer 1	240	240
s	Buffer 1	1	1
t	Buffer 1	1	1
u	Buffer 1	1	2
v	Buffer 1	240	240
w	Buffer 1	0	3

- Three shared buffers: Buffer 1, Buffer 2 and Buffer 3

actor	resource	claim	release
a	Buffer 1	1	0
b	Buffer 1	1	4
c	Buffer 1	11	11
d	Buffer 2	1	0
e	Buffer 2	1	4
f	Buffer 2	11	11
g	Buffer 1	1	1
h	Buffer 1	11	1
i	Buffer 1	10	11
j	Buffer 1	2	1
k	Buffer 2	1	1
l	Buffer 2	11	1
m	Buffer 2	10	11
n	Buffer 2	2	1
p	Buffer 1	1	2
p	Buffer 2	1	2
q	Buffer 1	240	240
r	Buffer 2	240	240
s	Buffer 2	0	1
s	Buffer 3	1	0
t	Buffer 1	0	1
t	Buffer 3	1	0
u	Buffer 3	1	2
v	Buffer 3	240	240
w	Buffer 1	0	1
w	Buffer 2	0	1
w	Buffer 3	0	1

MP3 Graph



- One shared buffer: Buffer 1

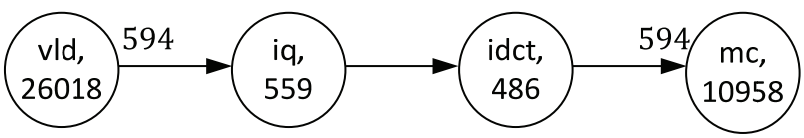
actor	resource	claim	release
huffman	Buffer 1	4	0
req0	Buffer 1	2	1
reorder0	Buffer 1	1	1
req1	Buffer 1	2	1
reorder1	Buffer 1	1	1
stereo	Buffer 1	4	4
aliasreduct0	Buffer 1	1	1
imdct0	Buffer 1	1	2
freqinv0	Buffer 1	1	1
synth0	Buffer 1	0	1
aliasreduct1	Buffer 1	1	1
imdct1	Buffer 1	1	2
freqinv1	Buffer 1	1	1
synth1	Buffer 1	0	1

- Three shared buffers: Buffer 1, Buffer 2 and Buffer 3

actor	resource	claim	release
huffman	Buffer 1	4	0
req0	Buffer 1	2	1
reorder0	Buffer 1	1	1
req1	Buffer 1	2	1
reorder1	Buffer 1	1	1
stereo	Buffer 1	0	4
stereo	Buffer 2	2	0

stereo	Buffer 3	2	0
aliasreduct0	Buffer 2	1	1
imdct0	Buffer 2	1	2
freqinv0	Buffer 2	1	1
synth0	Buffer 2	0	1
aliasreduct1	Buffer 3	1	1
imdct1	Buffer 3	1	2
freqinv1	Buffer 3	1	1
synth1	Buffer 3	0	1

H.263 (QCIF) Graph



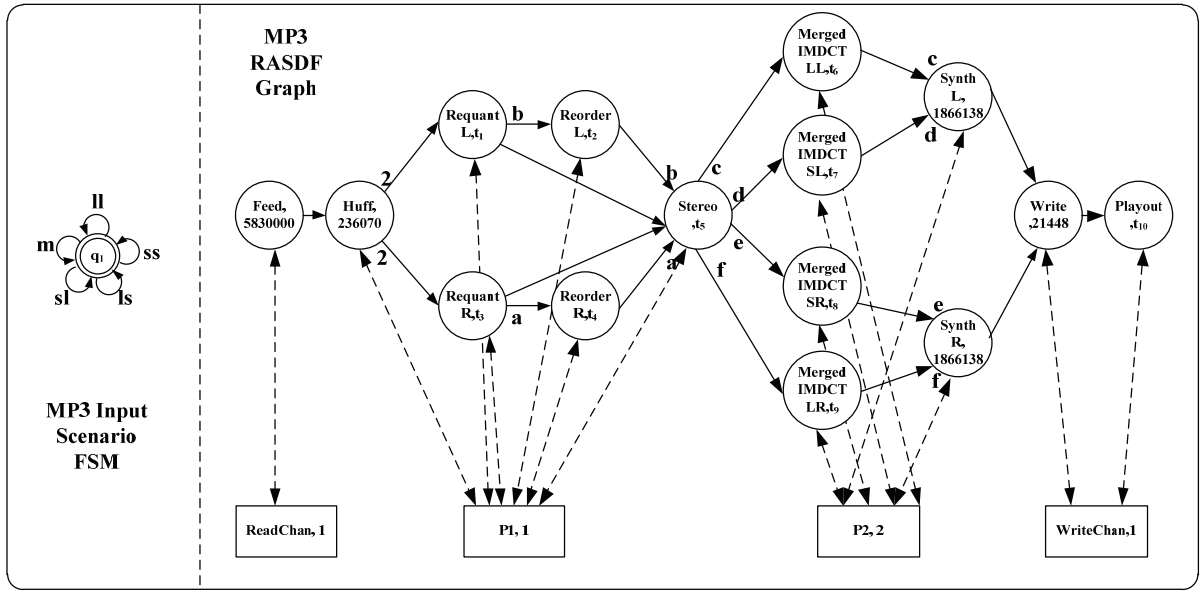
- One shared buffer: Buffer 1

actor	resource	claim	release
vld	Buffer 1	594	0
iq	Buffer 1	1	1
idct	Buffer 1	1	1
mc	Buffer 1	0	594

- Two shared buffers: Buffer 1, Buffer 2

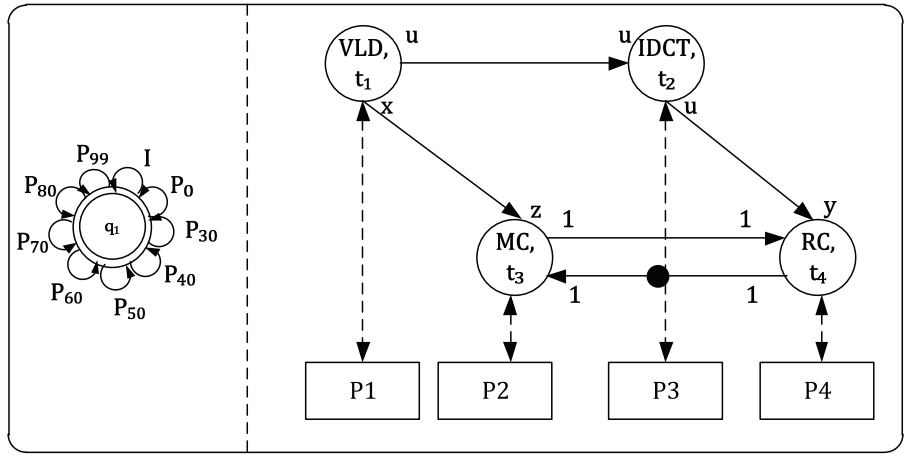
actor	resource	claim	release
vld	Buffer 1	594	0
iq	Buffer 1	0	1
iq	Buffer 2	1	0
idct	Buffer 1	1	0
idct	Buffer 2	0	1
mc	Buffer 1	0	594

MP3 SARA SDF



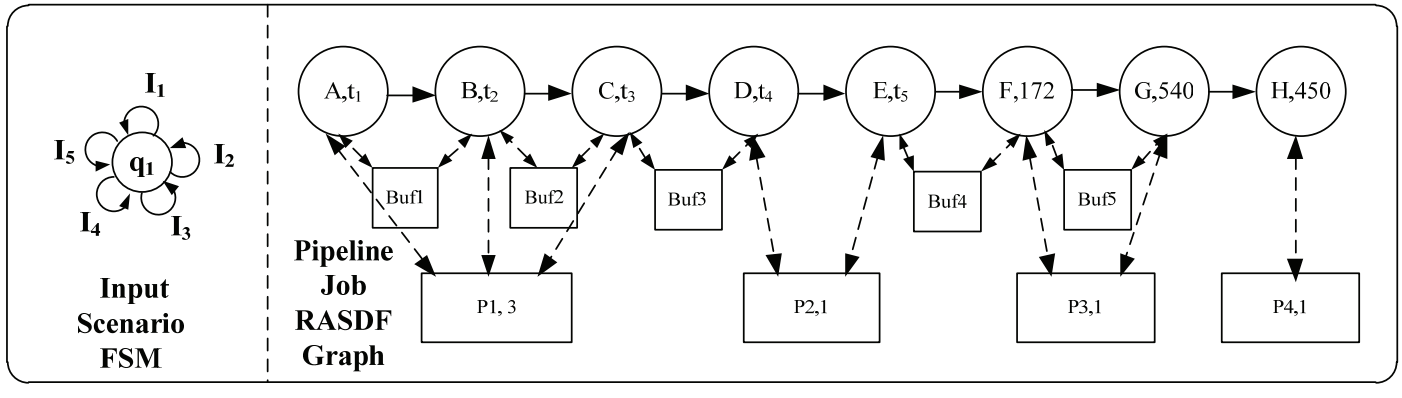
	a	b	c	d	e	f	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
ss	1	1	0	1	1	0	139325	69385	139325	69385	58239	NA	1005408	1005408	NA	2915000
ll	0	0	1	0	0	1	110785	NA	NA	110785	73618	407520	NA	NA	407520	2320000
ls	0	1	1	0	1	0	110785	NA	139325	69385	19	407520	NA	1005408	NA	2915000
sl	1	0	0	1	0	1	139325	69385	NA	110785	19	NA	1005408	NA	407520	2320000
m	0	0	1	1	1	1	60594	26269	64527	942570	64527	942570	25470	25470	942570	2880000

MPEG-4 SARA SDF



	x	u	z	y	t_1	t_2	t_3	t_4
I	0	1	0	99	40	17	0	350
P0	0	0	0	0	40	17	0	0
P30	1	1	30	30	40	17	90	250
P40	1	1	40	40	40	17	145	250
P50	1	1	50	50	40	17	190	250
P60	1	1	60	60	40	17	265	320
P70	1	1	70	70	40	17	235	300
P80	1	1	80	80	40	17	310	320
P99	1	1	99	99	40	17	390	320

Printer Pipeline SARA SDF



	t_1	t_2	t_3	t_4	t_5
I_1	303	105	208	82	82
I_2	254	198	115	44	44
I_3	433	169	519	228	228
I_4	600	422	633	177	177
I_5	35	25	21	11	11

BIBLIOGRAPHY

[1] Abdeddaïm, Y. et al. 2001. Job-shop scheduling using timed automata. *13th International Conference on Computer Aided Verification, CAV '01, Proceedings, in LNCS, Springer* (Berlin, Heidelberg, Jul. 2001), 478-492.

[2] Agarwal, M. and Frank, M.I. 2009. SPARTAN: A software tool for Parallelization Bottleneck Analysis. *2nd ICSE Workshop on Multicore Software Engineering, Proceedings, IEEE* (May. 2009), 56-63.

[3] Ahmad, I. and Jiancong, L. 2006. On using game theory to optimize the rate control in video coding. *IEEE Transactions on Circuits and Systems for Video Technology*. 16, 2 (Feb. 2006), 209-219.

[4] Alexander, P. 2007. Rosetta: language support for system-level design. *22nd International Conference on Automated software engineering, ASE '07, Proceedings, ACM* (New York, New York, USA, Nov. 2007), 577-577.

[5] Altisen, K. et al. 1999. A framework for scheduler synthesis. *20th IEEE Real-Time Systems Symposium, RTSS '99, Proceedings, IEEE* (1999), 154-163.

[6] Baccelli, F.L. et al. 1993. *Synchronization and Linearity: An Algebra for Discrete Event Systems (Wiley Series in Probability and Statistics)*. John Wiley & Sons.

[7] Baier, C. and Katoen, J.-P. 2008. *Principles of Model Checking*. The MIT Press.

[8] Balarin, F. et al. 1997. *Hardware-software co-design of embedded systems: the POLIS approach*. Kluwer Academic Publishers.

[9] Barrera, B. and Lee, E. a. 1991. Multirate signal processing in Comdisco's SPW. *16th International Conference on Acoustics, Speech, and Signal Processing, ICASSP '91, Proceedings, IEEE* (1991), 1113-1116.

[10] Basten, T. et al. 2010. Model-driven design-space exploration for embedded systems: the octopus toolset. *4th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA '10, in LNCS 6415,* (2010), 90-105.

[11] Bekooij, M. and Wiggers, M. 2006. *Latency-Rate servers & Dataflow models*.

[12] Bellman, R. 1957. A Markovian Decision Process. *Journal of Mathematics and Mechanics 6*. (Apr. 1957).

[13] Bhattacharya, B. and Bhattacharyya, S.S. 2001. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*. 49, 10 (2001), 2408-2421.

[14] Bhattacharyya, S.S. et al. 1993. A scheduling framework for minimizing memory requirements of multirate DSP systems represented as dataflow graphs. *Workshop on VLSI Signal Processing, Proceedings, IEEE* (1993), 188-196.

[15] Bhattacharyya, S.S. et al. 1996. *Software Synthesis from Dataflow Graphs (The Springer International Series in Engineering and Computer Science)*. Springer.

[16] Bhattacharyya, S.S. and Murthy, P.K. 1999. Synthesis Of Embedded Software From Synchronous Dataflow Specifications. *Journal on VLSI Signal Process. Syst.* 21, 2 (1999), 151-166.

[17] Bini, E. et al. 2011. Resource Management on Multicore Systems: The ACTORS Approach. *IEEE Micro*. 31, 3 (2011), 72-81.

[18] Black, D.C. et al. 2009. *SystemC: From the Ground Up, Second Edition [Hardcover]*. Springer; 2nd Edition. edition.

[19] Le Boudec, J.Y. and Thiran, P. 2001. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag.

[20] Bouillard, A. et al. 2009. *Service curves in Network Calculus: dos and don'ts*.

[21] Buchi, J.R. and Landweber, L.H. 1969. Solving Sequential Conditions by Finite-State Strategies. *Transactions of the American Mathematical Society*. 138, (Apr. 1969), 295.

[22] Buck, J. and Vaidyanathan, R. 2000. Heterogeneous modeling and simulation of embedded systems in El Greco. *8th International Workshop on Hardware/Software Codesign, CODES '00, Proceedings, ACM* (2000), 142-146.

[23] Buck, J.T. et al. 1994. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*. 4, (1994), 155-182.

[24] Buck, J.T. and Lee, E.A. 1993. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. *IEEE International Conference on Acoustics Speech and Signal Processing*. (1993), 429-432 vol.1.

[25] Butkovi, P. 2010. *Max-linear Systems: Theory and Algorithms (Springer Monographs in Mathematics)*. Springer.

[26] Calvez, J.-P. and Perrier, V. 2005. *Digital Multimedia System Architecting With CoFluent Studio: A CoFluent Design White Paper*.

[27] Chakraborty, S. et al. 2003. A general framework for analysing system properties in platform-based embedded system designs. *6th Conference on Design Automation and Test in Europe, DATE '03, Proceedings, IEEE* (2003), 190-195.

[28] Cochet-Terrasson, J. et al. 1999. A constructive fixed point theorem for min-max functions. *Dynamics and Stability of Systems*. 14, 4 (Dec. 1999), 407-433.

[29] Cochet-Terrasson, J. et al. 1998. Numerical computation of spectral elements in max-plus algebra. *IFAC Conference on Syst. Structure and Control* (1998), 1-7.

[30] Cochet-Terrasson, J. and Gaubert, S. 2000. Policy iteration algorithm for shortest path problems. (2000), 1-14.

[31] Commoner, F. et al. 1971. Marked directed graphs. *Journal of Computer and System Sciences*. 5, 5 (1971), 511–523.

[32] Compton, K. and Hauck, S. 2002. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys*. 34, 2 (Jun. 2002), 171-210.

[33] Cruz, R.L. 1991. A calculus for network delay. I. Network elements in isolation. *IEEE Transactions on Information Theory*. 37, 1 (1991), 114-131.

[34] Cruz, R.L. 1991. A calculus for network delay. II. Network analysis. *IEEE Transactions on Information Theory*. 37, 1 (1991), 132-141.

[35] Cuenot, P. et al. 2011. The EAST-ADL Architecture Description Language for Automotive Embedded Software. *Model Based Engineering of Embedded Real-Time Systems*. H. Giese et al., eds. Springer. 297-307.

[36] Czyżżak, P. and Jaszkiewicz, A. 1998. Pareto simulated annealing—a metaheuristic technique for multiple - objective combinatorial optimization. *Journal of Multi - Criteria Decision Analysis*. 7, 1 (Jan. 1998), 34-47.

[37] Damavandpeyma, M. et al. 2012. Modeling Static-Order Schedules in Synchronous Dataflow Graphs. *15th Design, Automation & Test in Europe Conference, DATE '12, Proceedings,* (2012), 775-780.

[38] Dantu, K. and Pedram, M. 2002. Frame-based dynamic voltage and frequency scaling for a MPEG decoder. *International Conference on Computer Aided Design, ICCAD '02, Proceedings, IEEE* (2002), 732-737.

[39] Dasdan, A. 2004. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems*. 9, 4 (Oct. 2004), 385-418.

[40] Dasdan, A. and Gupta, R.K. 1998. Faster maximum and minimum mean cycle algorithms for system-performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 17, 10 (1998), 889-899.

[41] Deb, K. et al. 2000. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. *6th Conference on Parallel Problem Solving from Nature, PPSN '00, Proceedings* (2000), 849-858.

[42] Dhingra, V. and Gaubert, S. 2006. How to solve large scale deterministic games with mean payoff by policy iteration. *1st International Conference on Performance evaluation methodolgies and tools , Valuetools '06, Proceedings* (New York, New York, USA, 2006), 12.

[43] Dijkstra, E.W. 1968. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*. 8, 3 (1968), 174–186.

[44] Dijkstra, E.W. 1970. *Notes on structured programming*. Eindhoven University of Technology.

[45] Edwards, S. et al. 1997. Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*. 85, 3 (Mar. 1997), 366-390.

[46] Ehrenfeucht, A. and Mycielski, J. 1979. Positional strategies for mean payoff games. *International Journal of Game Theory*. 8, 2 (Jun. 1979), 109-113.

[47] Etessami, K. et al. 2008. Multi-Objective Model Checking of Markov Decision Processes. *Logical Methods in Computer Science*. 4, (2008), 21.

[48] Fang, Z. and Bensaou, B. 2004. Fair bandwidth sharing algorithms based on game theory frameworks for wireless ad-hoc networks. *23rd International Conference of the IEEE Computer and Communications Societies, INFOCOMM' 04, Proceedings, IEEE* (2004), 1284-1295.

[49] Fernandez, M. 2009. *Models of Computation: An Introduction to Computability Theory (Undergraduate Topics in Computer Science)*. Springer.

[50] Fisher, J.A. et al. 2005. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier.

[51] Gao, F. and Sair, S. 2006. Long-term Performance Bottleneck Analysis and Prediction. *24th International Conference on Computer Design, ICCD '06, Proceedings, IEEE* (Oct. 2006), 3-9.

[52] Gaubert, S. 1995. Performance evaluation of (max,+) automata. *Automatic Control, IEEE Transactions on*. 40, 12 (1995), 2014–2025.

[53] Gaubert, S. and Gunawardena, J. 1998. The duality theorem for min-max functions. *Comptes Rendus de l'Académie des Sciences-Series I-Mathematics*. 326, 1 (1998), 43–48.

[54] Gaubert, S. and Mairesse, J. 1999. Asymptotic analysis of heaps of pieces and application to timed Petri nets. *Proceedings 8th International Workshop on Petri Nets and Performance Models (Cat. No.PR00331)* (1999), 158-169.

[55] Gaubert, S. and Mairesse, J. 1999. Modeling and analysis of timed Petri nets using heaps of pieces. *IEEE Transactions on Automatic Control*. 44, 4 (Apr. 1999), 683-697.

[56] Geilen, M. et al. 2007. An algebra of Pareto points. *Fundamenta Informaticae*. (2007), 88-97.

[57] Geilen, M. et al. 2005. Minimising buffer requirements of synchronous dataflow graphs with model checking. *42nd Design Automation Conference, DAC '05, Proceedings, ACM* (New York, New York, USA, Jun. 2005), 819-824.

[58] Geilen, M. 2009. Reduction techniques for synchronous dataflow graphs. *46th Design Automation Conference, DAC '09, Proceedings, ACM* (2009), 911-916.

[59] Geilen, M. 2010. Synchronous dataflow scenarios. *ACM Transactions on Embedded Computing Systems*. 10, 2 (Dec. 2010), 1-31.

[60] Geilen, M. et al. 2011. The earlier the better: a theory of timed actor interfaces. *14th International Conference on Hybrid systems: computation and control, HSCC '11, Proceedings, ACM* (2011), 23–32.

[61] Geilen, M. and Basten, T. 2004. Reactive process networks. *4th International Conference on Embedded software, EMSOFT '04, Proceedings, ACM* (New York, New York, USA, 2004), 137-146.

[62] Geilen, M. and Stuijk, S. 2010. Worst-case performance analysis of synchronous dataflow scenarios. *8th International Conference on Hardware/software codesign and system synthesis, CODES+ISSS '10, Proceedings, IEEE* (2010), 125–134.

[63] Ghamarian, A.H. et al. 2008. Parametric throughput analysis of synchronous data flow graphs. *11th Conference on Design, Automation & Test in Europe, DATE '08, Proceedings, IEEE* (2008), 116-121.

[64] Ghamarian, A.H. et al. 2006. Throughput Analysis of Synchronous Data Flow Graphs. *6th International Conference on Application of Concurrency to System Design, ACSD'06, Proceedings, IEEE* (2006), 25-36.

[65] Ghamarian, A.H. 2008. *Timing analysis of synchronous data flow graphs*. Technische Universiteit Eindhoven.

[66] Gheorghita, S.V. et al. 2005. Automatic scenario detection for improved WCET estimation. *42nd Design Automation Conference, DAC '05, Proceedings, ACM* (New York, New York, USA, 2005), 101-104.

[67] Girault, A. et al. 1999. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 18, 6 (1999), 742–760.

[68] Govindarajan, R. et al. 1994. Minimizing memory requirements in rate-optimal schedules. *International Conference on Application Specific Array Processors, ASAP'94, Proceedings, IEEE* (1994), 75-86.

[69] Gresser, K. 1993. An Event Model for Deadline Verification of Hard Real-Time Systems. *5th Euromicro Workshop on Real-Time Systems, ECRTS '93, Proceedings, IEEE* (1993), 118-123.

[70] Gries, M. 2004. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal*. 38, 2 (Dec. 2004), 131-183.

[71] Gries, M. and Keutzer, K. 2005. *Building ASIPs: the MESCAL methodology*. Springer.

[72] Guha, R. et al. 2009. Resource management and task partitioning and scheduling on a run-time reconfigurable embedded system. *Computers Electrical Engineering*. 35, 2 (2009), 258-285.

[73] Gunawardena, J. 1994. Cycle times and fixed points of min-max functions. *11th International Conference on Analysis and Optimization of Systems* (1994), 266–272.

[74] Gunawardena, J. 2003. From max-plus algebra to nonexpansive mappings: a nonlinear theory for discrete event systems. *Theoretical Computer Science*. 293, 1 (Feb. 2003), 141-167.

[75] Hahn, J. and Chou, P.H. 2007. Buffer optimization and dispatching scheme for embedded systems with behavioral transparency. *7th International Conference on Embedded software, EMSOFT '07, Proceedings, ACM* (New York, New York, USA, Sep. 2007), 94-103.

[76] Hamann, A. et al. 2004. Symta/s-symbolic timing analysis for systems. *16th Euromicro Conference on Real-Time Systems, ECRTS'04, Proceedings, IEEE* (2004), 17–20.

[77] Heidergott, B. et al. 2005. *Max Plus at Work: Modeling and Analysis of Synchronized Systems: A Course on Max-Plus Algebra and Its Applications (Princeton Series in Applied Mathematics)*. Princeton University Press.

[78] Hennessy, J.L. and Patterson, D.A. 2006. *Computer Architecture: A Quantitative Approach, 4th Edition [Paperback]*. Morgan Kaufmann; 4 edition.

[79] Hiroyuki Tomiyama, A.H. 1999. Architecture Description Languages for Systems-on-Chip Design. *6th Asia Pacific Conference on Chip Design Language, ACM* (1999), 109-116.

[80] Hoffman, A.J. and Karp, R.M. 1966. On Nonterminating Stochastic Games. *Management Science*. 12, 5 (Jan. 1966), 359-370.

[81] Hoffmann, A. et al. 2001. A methodology for the design of application specific instruction set processors (ASIP) using the machine description language LISA. *International Conference on Computer-aided design, ICCAD '01, Proceedings, IEEE* (Nov. 2001), 625-630.

[82] Howard, R.A. 1960. *Dynamic Programming and Markov Processes (Technology Press Research Monographs)*. The MIT Press.

[83] Hsu, C.-J. et al. 2005. Software synthesis from the dataflow interchange format. *Workshop on Software and compilers for embedded systems, SCOPES '05, Proceedings, ACM* (New York, New York, USA, 2005), 37-49.

[84] Hu, J. et al. 2006. System-Level Buffer Allocation for Application-Specific Networks-on-Chip Router Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 25, 12 (Dec. 2006), 2919-2933.

[85] Hung, W.-L. et al. 2005. Thermal-Aware Task Allocation and Scheduling for Embedded Systems. *8th Conference on Design Automation and Test in Europe, DATE '05, Proceedings, IEEE* (2005), 898-899.

[86] ITRS 2009. International Technology Roadmap for Semiconductors, 2009 Edition. *Executive Summary. Semiconductor Industry*. (2009).

[87] Igna, G. et al. 2008. Formal modeling and scheduling of datapaths of digital document printers. *6th International Conference on Formal Modeling and Analysis of Timed Systems, FORMATS '08, Proceedings,* (2008), 170-187.

[88] Jain, R.K. 1991. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley.

[89] Jantsch, A. 2003. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation (Systems on Silicon)*. Morgan Kaufmann.

[90] Johnson, G.W. 1997. *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control*. Mcgraw-Hill.

[91] Jonsson, B. et al. 2008. Cyclic dependencies in modular performance analysis. *8th International Conference on Embedded software, EMSOFT'08, Proceedings, ACM* (2008), 179–188.

[92] Jr., E.M.C. et al. 1999. *Model Checking*. The MIT Press.

[93] Kahn, G. 1974. The semantics of a simple language for parallel programming. *Information Processing, Proceedings, North Holland, Amsterdam* (1974), 471-475.

[94] Karp, R.M. 1978. A characterization of the minimum cycle mean in a digraph. *Discrete mathematics*. 23, 3 (1978), 309–311.

[95] Karp, R.M. and Miller, R.E. 1966. Properties of a model for parallel computations: Determinancy, termination, queueing. *SIAM Journal on Applied Mathematics*. 14, 6 (1966), 1390–1411.

[96] Keutzer, K. et al. 2000. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 19, 12 (2000), 1523-1543.

[97] Kiczales, G. et al. 1997. Aspect-oriented programming. *11th European Conference on Object-Oriented Programming, ECOOP '97, Proceedings, volume 1241 in LNCS, Springer* (1997), 220–242.

[98] Kienhuis, B. et al. 1997. An approach for quantitative analysis of application-specific dataflow architectures. *International Conference on Application-Specific Systems, Architectures and Processors, ASAP' 97, Proceedings, IEEE* (1997), 338-350.

[99] Lee, C. et al. 2009. A Systematic Design Space Exploration of MPSoC Based on Synchronous Data Flow Specification. *Journal of Signal Processing Systems*. 58, 2 (Mar. 2009), 193-213.

[100] Lee, E.A. 1991. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems*. 2, 2 (1991), 223–235.

[101] Lee, E.A. and Messerschmitt, D.G. 1987. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*. C-36, 1 (Jan. 1987), 24-35.

[102] Lensen, B.J.P. 2010. *Automated Design-Space Exploration of resource-aware Synchronous Dataflow Graphs using multi-objective evolutionary algorithms*. Eindhoven University of Technology.

[103] Lombardo, A. et al. 1998. An accurate and treatable Markov model of MPEG-video traffic. *17th International Conference of the IEEE Computer and Communications Societies, INFOCOMM' 98, Proceedings, IEEE* (1998), 217-224.

[104] Lotze, J. et al. 2011. A Model-Based Approach to Cognitive Radio Design. *IEEE Journal on Selected Areas in Communications*. 29, 2 (2011), 455-468.

[105] Lukasiewycz, M. et al. 2008. Efficient symbolic multi-objective design space exploration. *Asia and South Pacific Design Automation Conference, ASP-DAC' 08, Proceedings, IEEE* (Jan. 2008), 691-696.

[106] Madisetti, V. and Arpnikanondt, C. 2005. *A platform-centric approach to system-on-chip (SoC) design*. Springer.

[107] Martin, G. 2006. Overview of the MPSoC design challenge. *43rd Design Automation Conference. DAC '06, Proceedings, ACM* (2006), 274-279.

[108] Martin, G. 2002. UML for Embedded Systems Specification and Design: Motivation and Overview. *5th Conference on Design, Automation & Test in Europe, DATE' 02, Proceedings, IEEE* (Mar. 2002), 773-775.

[109] Mitchell, R. and McKim, J. 2001. *Design by Contract, by Example*. Addison-Wesley Professional.

[110] Moreira, O. and Valente, F. 2007. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. *7th International Conference on Embedded software, EMSOFT '07 Proceedings, ACM* (New York, New York, USA, 2007), 57-66.

[111] Murthy, P.K. et al. 2001. System Canvas: a new design environment for embedded DSP and telecommunication systems. *9th International Symposium on Hardware/Software Codesign. CODES '01, Proceedings, ACM* (2001), 54-59.

[112] Murthy, P.K. and Bhattacharyya, S.S. 2006. *Memory Management for Synthesis of DSP Software*. CRC Press.

[113] Murthy, P.K. and Bhattacharyya, S.S. 2001. Shared buffer implementations of signal processing systems using lifetime analysis techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 20, 2 (2001), 177-198.

[114] Nash, J. 1951. Non-cooperative games. *The Annals of Mathematics*. 54, 2 (1951), 286–295.

[115] Neumann, J.V. and Morgenstern, O. 2007. *Theory of Games and Economic Behavior (Commemorative Edition)(Princeton Classic Editions)*. Princeton University Press.

[116] Ni, J. et al. 1996. Source modelling, queueing analysis, and bandwidth allocation for VBR MPEG-2 video traffic in ATM networks. *IEE Proceedings - Communications*. 143, 4 (1996), 197-205.

[117] Parks, T. et al. 1995. A comparison of synchronous and cyclo-static dataflow. *29th Asilomar Conference on Signals, Systems and Computers, ASILOMAR '95, Proceedings, IEEE* (1995), 1-7.

[118] Peterson, J.L. 1977. Petri nets. *Computing Surveys*. 9, 3 (Jan. 1977), 223-252.

[119] Petri, C.A. 1962. *Communication with Automata (in german)*. Institute für instrumentelle Mathematik.

[120] Pillai, P. and Shin, K.G. 2001. Real-time dynamic voltage scaling for low-power embedded operating systems. *ACM SIGOPS Operating Systems Review*. 35, 5 (Dec. 2001), 89.

[121] Pimentel, A.D. et al. 2006. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*. 55, 2 (Feb. 2006), 99-112.

[122] Pino, J.L. and Kalbasi, K. 1998. Cosimulating synchronous DSP applications with analog RF circuits. *32nd Asilomar Conference on Signals, Systems and Computers, Asilomar '98, Proceedings, IEEE* (1998), 1710-1714.

[123] Powell, D.B. et al. 1992. Direct synthesis of optimized DSP assembly code from signal flow block diagrams. *17th International Conference on Acoustics, Speech, and Signal Processing, ICASSP '93, Proceedings, IEEE* (1992), 553-556.

[124] Puri, A. and Tripakis, S. 2002. Algorithms for routing with multiple constraints. *Workshop on Planning and Scheduling using Multiple Criteria, AIPS '02, Proceedings* (2002), 7–14.

[125] R.Apt, K. and Gradel, E. eds. 2011. *Lectures in Game Theory for Computer Scientists*. Cambridge Press.

[126] Rasmussen, J.I. et al. 2004. Resource-optimal scheduling using priced timed automata. *Tools and Algorithms for the Construction and Analysis of Systems*. (2004), 220–235.

[127] Richter, K. 2004. *Compositional scheduling analysis using standard event models*. Technical University of Braunschweig.

[128] Richter, K. and Ernst, R. 2002. Event Model Interfaces for Heterogeneous System Analysis. *5th Conference on Design, Automation & Test in Europe, DATE' 02, Proceedings, IEEE* (Mar. 2002), 506-513.

[129] Ritz, S. et al. 1992. High level software synthesis for signal processing systems. *International Conference on Application Specific Array Processors, ASAP '92, Proceedings, IEEE* (1992), 679–693.

[130] Ritz, S. et al. 1993. Optimum vectorization of scalable synchronous dataflow graphs. *International Conference on Application Specific Array Processors, ASAP '93, Proceedings, IEEE* (1993), 285-296.

[131] Ritz, S. et al. 1995. Scheduling for optimum data memory compaction in block diagram oriented software synthesis. *20th International Conference on Acoustics, Speech, and Signal Processing, ICASSP '95, Proceedings, IEEE* (1995), 2651-2654.

[132] Robinson, D. 2007. *Aspect-Oriented Programming with the e Verification Language: A Pragmatic Guide for Testbench Developers*. Morgan Kaufmann.

[133] Rox, J. and Ernst, R. 2008. Modeling event stream hierarchies with hierarchical event models. *11th Conference on Design, Automation and Test in Europe, DATE'08, Proceedings, IEEE* (2008), 492–497.

[134] Sangiovanni-Vincentelli, A. and Martin, G. 2001. Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of Computers*. 18, 6 (2001), 23-33.

[135] Satish, N.R. et al. 2008. Scheduling task dependence graphs with variable task execution times onto heterogeneous multiprocessors. *7th International Conference on Embedded software, EMSOFT '08, Proceedings, ACM* (New York, New York, USA, 2008), 149-158.

[136] Savage, J.E. 1998. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Pub (Sd).

[137] Schlichter, T. et al. 2006. Improving System Level Design Space Exploration by Incorporating SAT-Solvers into Multi-Objective Evolutionary Algorithms. *Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures, ISVLSI'06, Proceedings, IEEE* (2006), 309-316.

[138] Schliecker, S. et al. 2007. Performance analysis of complex systems by integration of dataflow graphs and compositional performance analysis. *10th Conference on Design, Automation and Test in Europe, DATE '07, Proceedings, IEEE* (Apr. 2007), 273-278.

[139] Silage, D. 2006. *Digital Communication System Using System VUE [Paperback]*. Laxmi Publications.

[140] Siyoum, F. et al. 2011. Resource-Efficient Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. *17th International Conference on Embedded and Real-Time Computing Systems and Applications,RTCSA '11, IEEE* (Aug. 2011), 309-318.

[141] Skelly, P. and Schwartz, M. 1993. A histogram-based model for video traffic behavior in an ATM multiplexer. *Networking, IEEE/ACM*. 1, 4 (1993), 446-459.

[142] Sriram, S. and Bhattacharyya, S.S. 2009. *Embedded Multiprocessors: Scheduling and Synchronization, Second Edition (Signal Processing and Communications)*. CRC Press.

[143] Stiliadis, D. and Varma, A. 1998. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE Transactions on Networking*. 6, 5 (1998), 611-624.

[144] Stuijk, S. et al. 2006. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. *43rd Design Automation Conference, DAC '06, Proceedings, ACM* (2006), 899-904.

[145] Stuijk, S. 2007. *Predictable mapping of streaming applications on multiprocessors*. Technische Universiteit Eindhoven.

[146] Stuijk, S. et al. 2006. SDF^3: SDF For Free. *6th International Conference on Application of Concurrency to System Design, ACSD' 06, Proceedings, IEEE* (2006), 276-278.

[147] Stuijk, S. et al. 2011. Scenario-Aware Dataflow: Modeling, Analysis and Implementation of Dynamic Applications. *11th International Conference on*

Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS '11, Proceedings, IEEE (2011), 404-411.

[148] Stuijk, S. et al. 2008. Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs. *IEEE Transactions on Computers*. 57, 10 (Oct. 2008), 1331-1345.

[149] Sung, W. et al. 1997. Demonstration of hardware software codesign workflow in PeaCE. *5th International Conference on VLSI and CAD, ICVC '97, Proceedings, IEEE* (1997).

[150] Teruel, E. et al. 1992. On weighted T-systems. *Application and Theory of Petri Nets 1992, in LNCS, Springer Berlin*. 616, (1992), 348–367.

[151] Theelen, B.D. et al. 2006. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. *4th International Conference on Formal Methods and Models for Co-Design, MEMOCODE '06, Proceedings, IEEE* (2006), 185-194.

[152] Thiele, L. et al. 2000. Real-time calculus for scheduling hard real-time systems. *International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century, Proceedings, IEEE* (2000), 101-104.

[153] Thiele, L. et al. 2011. Thermal-aware system analysis and software synthesis for embedded multi-processors. *48th Design Automation Conference , DAC '11, Proceedings, ACM* (2011), 268.

[154] Thies, W. et al. 2002. StreamIt: A language for streaming applications. *11th International Symposium on Compiler Construction, CC '02, Proceedings, volume 2304 in LNCS, Springer-Verlag,* (2002), 179-196.

[155] Vink, J.P. et al. 2008. Performance analysis of SoC architectures based on latency-rate servers. *11th Conference on Design, Automation & Test in Europe, DATE '08, Proceedings, IEEE* (New York, New York, USA, 2008), 200-205.

[156] Wandeler, E. 2006. *Modular performance analysis and interface-based design for embedded real-time systems*. SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH.

[157] Wandeler, E. et al. 2006. System architecture evaluation using modular performance analysis: a case study. *International Journal on Software Tools for Technology Transfer*. 8, 6 (Jul. 2006), 649-667.

[158] Wang, Z. and O'Boyle, M.F.P. 2010. Partitioning streaming parallelism for multi-cores: a machine learning based approach. *19th Internation Conference on Parallel Architecture and Compiler Techniques, PACT '10, Proceeding, IEEE* (2010), 307–318.

[159] Wattanapongskorn, N. and Coit, D. 2007. Fault-tolerant embedded system design and optimization considering reliability estimation uncertainty. *Reliability Engineering System Safety*. 92, 4 (2007), 395-407.

[160] Werner, J. et al. 2007. Integrating Security Modeling in Embedded System Design. *14th Conference on Engineering of Computer-Based Systems, ECBS '07, Proceedings, IEEE* (2007), 221-228.

[161] Wiegand, T. et al. 2003. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*. 13, 7 (Jul. 2003), 560-576.

[162] Wiggers, M.H. et al. 2008. Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE* (Apr. 2008), 183–194.

[163] Wiggers, M.H. et al. 2010. Buffer capacity computation for throughput-constrained modal task graphs. *ACM Transactions on Embedded Computing Systems*. 10, 2 (Dec. 2010), 1-59.

[164] Wiggers, M.H. et al. 2007. Modelling run-time arbitration by latency-rate servers in dataflow graphs. *Workshop on Software & compilers for embedded systems - SCOPES '07* (New York, New York, USA, 2007), 11-22.

[165] Wiggers, M.H. and Bekooij, M.J.G. 2009. Monotonicity and run-time scheduling. *7th International Conference on Embedded software, EMSOFT '09, Proceedings, ACM* (2009), 177-186.

[166] Wolf, W.H. 1994. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*. 82, 7 (Jul. 1994), 967-989.

[167] Wonyong, S. and Soonhoi, H. 2000. Memory efficient software synthesis with mixed coding style from dataflow graphs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. 8, 5 (Oct. 2000), 522-526.

[168] Wu, D. et al. 2003. Scheduling and Mapping of Conditional Task Graph for the Synthesis of Low Power Embedded Systems. *Computers and Digital Techniques IEE Proceedings*. 150, 5 (2003), 262-273.

[169] Yang, Y. et al. 2010. Automated bottleneck-driven design-space exploration of media processing systems. *13th Conference on Design, Automation & Test in Europe, DATE '10, Proceedings, IEEE* (2010), 1041–1046.

[170] Yang, Y. et al. 2009. Exploring trade-offs between performance and resource requirements for synchronous dataflow graphs. *7th Workshop on Embedded Systems for Real-Time Multimedia, ESTMEDIA '09, Proceedings, IEEE* (Oct. 2009), 96-105.

[171] Yang, Y. et al. 2011. Iteration-based Trade-off Analysis of Resource-aware SDF. *14th Euromicro Conference on Digital System Design, DSD '11, Proceedings, IEEE* (2011), 567-574.

[172] Yang, Y. et al. 2012. Playing Games with Scenario- and Resource-Aware SDF Graphs Through Policy Iteration. *15th Conference on Design, Automation & Test , DATE '12, Proceedings, IEEE* (2012).

[173] Zafar, S. and Dromey, R.G. 2005. Integrating Safety and Security Requirements into Design of an Embedded System. *12th Asia Pacific Software Engineering Conference, APSEC '05, Proceedings, IEEE* (2005), 629-636.

[174] Zhai, J.T. et al. 2011. Modeling adaptive streaming applications with Parameterized Polyhedral Process Networks. *47th Design Automation Conference, DAC '11, Proceedings, ACM* (2011), 116-121.

[175] Zhang, G. 2009. *More than Moore: Creating High Value Micro/Nanoelectronics Systems*. Springer-Verlag.

[176] Zitzler, E. et al. 2000. Evolutionary algorithms for the synthesis of embedded software. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. 8, 4 (Aug. 2000), 452-455.

[177] Zitzler, E. et al. 2003. Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Transactions on Evolutionary Computation*. 7, 2 (Apr. 2003), 117-132.

[178] Zitzler, E. et al. 2001. SPEA2 : Improving the Strength Pareto Evolutionary Algorithm. *Evolutionary Methods for Design, Optimisation and Control with*

Application to Industrial Problems, EUROGEN '01, Proceeding (2001), 95-100.

[179] Zonghua, G. et al. 2007. Optimization of Static Task and Bus Access Schedules for Time-Triggered Distributed Embedded Systems with Model-Checking. *44th Design Automation Conference, DAC '07, Proceedings, ACM* (2007), 294-299.

[180] Zwick, U. 1996. The complexity of mean payoff games on graphs. *Theoretical Computer Science*. 158, 1-2 (May. 1996), 343-359.

ACKNOWLEDGEMENTS

PhD study is like a thousand miles trip to an unknown destination. Without the help from many people, I might still wander halfway or even have lost my faith to finish it. I would like to thank all those people who have supported me during my PhD study.

In the past five years, I have been coached by my first promoter Twan Basten. Since the first day that he picked me up at the train station and drove me to my first apartment in Eindhoven, he has provided a lot of advices both on my research and my life. One thing that I appreciate the most is his tireless efforts on my draft conference papers and this thesis. From his corrections of my grammar mistakes, his checking on the consistency of terminologies and his insightful comments, I learned how to do research in a serious way.

I would also like to thank my second promoter Henk Corporaal. He was always able to point out some ignored points in my draft papers in a short time so that I could improve the drafts before submitting them to a conference. I would also like to thank my co-promoter and daily supervisor Marc Geilen. The daily discussions with him always shed a light on my research. And he always asked a lot of 'why's on my drafts and pushed me to answer them in a way that could satisfy him and finally also the reviewers.

I would also like to thank Marco Bekooij, Kees van Berkel and Samarjit Chakraborty for being part of my PhD committee. Your thorough review of, and constructive comments on, the draft version of this thesis were very helpful. Also Ton Backx is thanked for being the chairman of the PhD committee.

I would also like to thank my two officemates Sander Stuijk and Ahsan Shabbir. We spent more than four years in the same office, PT 9.10. They set an example of hard working for me. They helped me on many things, from improving my research to fixing my broken bicycle.

In the last few years, I have had a lot of discussions with people in the weekly Promes Meetings. I would like to thank Bart Theelen, Amir Hossein Ghamarian, Majid Nabi Najafabadi, Rob Hoes, Morteza Damavandpeyma,

Maarten Wiggers, Marcel Steine, Milos Blagojevic, and Nikola Trcka. I also learned a lot from them.

Working in the Electronic Systems Group has been quite fun to me. I always enjoyed discussion with our group leader Ralph Otten. I would like to thank secretaries Marja and Rian, they were always helpful when I forgot my office key and when I was filling the reimbursement forms. I would also like to thank my Chinese friends in our group. Hao Hu and Yu Pu gave me suggestions at the beginning of my PhD study. Yifan He and his wife Songyue Chen invited me to their house many times and I really enjoyed their hospitality and delicious food. I also enjoyed my discussions with Dongrui She on a wide range of topics. I would also like to thank Zhenyu Ye, Wei Tong and Bo Liu. I also want to thank the other group members at the coffee table. The conversations with Akash Kumar, Mathias Funk, Raymond Frijns, Bart Mesman, Yahya Jan, Cedric Nugteren, Roel Jordans, and other members were relaxed and fun.

I would like to thank many people that I worked with in the Octopus project. Frans Reckers took me to the R&D department of Océ many times with his car. I would like to thank Roelof Hamberg and Jacques Verriet for their feedback on my presentations given at Océ. I would like to thank Lou Somers and Sebastian de Smet for hosting me at Océ, for their feedback, and their help to get my papers approved by Océ. I would also like to thank Georgeta Igna, Venkatesh Kannan, and Marc Voorhoeve who also worked on the LoA1 research track. The discussions on different modeling tools with them helped me develop my understanding about my own research. I would like to thank Klemens Schindler for his ResVis tool and Brian Lensen for his work on genetic algorithms.

I would like to thank my former colleague Chunyang Gou, who gave me suggestions on PhD study in Netherlands. I would also like to thank my Chinese friends in Eindhoven, especially the people in the Chinese student and scholar association, ACSSE. They are Ping Li, Jing Li, Chenyang Ding, Xiaoping Chen, Wei Xu, Yuzhong Lin, Kang Zhao, Ziqiang Yan, Dujuan Yang, Zhengjie Lu, Jinfan Man, Peng Zhang, Rui Zhang, Hao Gao, Shiqi Li, Yi Wang, You Peng. Special Thanks to Rui Zhang and Kang Zhao, I visited their apartment many times and had many delicious dinners there. They partially contributed to my weight gain.

Finally, I would like to thank my family. I would like to thank my twin brother He Yang and his wife Yalan Hu and their support in these years. No matter how far we are from each other, our hearts are always close. Their encouragements always give me strength to move on. I would like to thank my grandma, may you find peace in heaven. I would like to thank my parents Jianxing Yang and Wanhua Shi, their support and sacrifices make what I am today. I am always in debt to them.

在这里，我要感谢我的奶奶，愿您在天堂安息。我要感谢我的爸爸和妈妈，没有你们的牺牲和支持，就没有今天的我。

我谨以这本书，献给我的爸爸、妈妈和奶奶。

Yang Yang May 20, 2012

杨阳 写于 2012 年 5 月 20 日

CURRICULUM VITAE

Yang Yang was born in Jin Tang county, Si Chuan Province, China, on Oct 2nd, 1981. He got his Bachelor degree in Electrical Engineering at the Beijing Normal University in 2004. Then he started a Master study in the High-speed Signal Processing Lab, Electrical Engineering Department of Tsinghua University. He received his Master degree in Electrical Engineering at the Tsinghua University in 2007.

In October 2007, he started working towards a Ph.D. degree within the Electronics Systems group at the department of Electrical Engineering of the Eindhoven University of Technology. His research was funded by the Octopus project with Océ Technologies B.V. under the responsibility of the Embedded Systems Institute. This project was partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program. The research focused on trade-off analysis for multicore embedded systems. It has led among others to several publications and this thesis.

LIST OF PUBLICATIONS

First author

- Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal, "*Playing throughput games with scenario and resource-aware SDF graphs through policy iteration*", in 15th Conference on Design, Automation and Test in Europe, DATE '12, Proceedings, IEEE, 2012. (pp. 194-199)

- Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal, "*Iteration-based trade-off analysis of resource-aware SDF*", in 14th Euromicro Conference on Digital System Design, DSD '11, Proceedings, IEEE, 2011. (pp. 567-574)

- Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal, "*Automated bottleneck-driven design-space exploration of media processing systems*", in 13th Conference on Design, Automation and Test in Europe, DATE '10, Proceedings, IEEE, 2010. (pp. 1041-1046).

- Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal, "*Exploring tradeoffs between performance and resource requirements for synchronous dataflow graphs*", in 7th Workshop on Embedded Systems for Real-Time Multimedia, ESTIMedia '09, Proceedings, IEEE, 2009. (pp. 96-105).

Co-author

- T. Basten, E. van Benthum, M. Geilen, M. Hendriks, F. Houben, G. Igna, F.J. Reckers, S. de Smet, L.J.A.M. Somers, E. Teeselink, N. Trcka, F.W. Vaandrager, J.H. Verriet, M. Voorhoeve, Y. Yang *"Model-driven design-space exploration for embedded systems: the Octopus toolset*" in 4th International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation, ISoLA '10, Proceedings, Springer, in Lecture Notes in Computer Science, Vol. 6415, (pp. 90-105).

- G. Igna, K. Venkatesh, Y. Yang, T. Basten, M. Geilen, F. Vaandrager, M. Voorhoeve, S. de Smet, L.J.A.M Somers "*Formal modeling and scheduling of datapaths of digital document printers*", in 6th International Conference

on Formal Modeling and Analysis of Timed Systems, FORMATS '08, Proceedings, Springer, in Lecture Notes in Computer Science, Vol. 5215 , (pp. 170-187).