Extracting SDF from sequential applications for MSPoC and implementation on FPGA

Gielen, T.P.C.

Award date:
2008

TU/e technische universiteit eindhoven

Faculty of Electrical Engineering
Section Electronic Systems (ICS/ES)
ICS-ES 901

Master's Thesis

EXTRACTING SDF FROM SEQUENTIAL APPLICATIONS FOR MSPOC AND IMPLEMENTATION ON FPGA.

TPC Gielen

Supervisor:	prof.dr.H. Corporaal
Coach:	dr.ir. Y. Ha
	MTD A. Kumar
	Dr.ir. T.P. Stefanov
Date:	December 2008

Abstract

Future embedded systems have to combine low energy consumption with high computational power. Multi-processors provide a solution and therefore SoC platforms become increasingly multi-processor architectures. Furthermore the number of processors in these architectures rises as well. This implies a shift in emphasis from processor design towards compiler design. A major challenge is parallelising applications to run on these MPSoCs and most methodologies rely on error-prone and time consuming design by hand. This thesis presents a design methodology to extract the parallelism from an application. Synchronous Dataflow (SDF) is used to express the parallelism because it allows easy worst-case behaviour analysis. The SDF graph is derived from an intermediate analysis of the application provided by a tool called DAT. This DAT tool profiles the application and generates an intermediate graph with data dependencies. From that intermediate graph a new tool called C2SDF derives the SDF graph. This include the actors, channels and various properties of the SDF. This output can be used by the MAMPS framework to implement and map the application directly to a MPSoC on FPGA. MAMPS has been prepared to take the generated parallel source code into account. For simple cases the DAT and C2SDF flow can derive an SDF graph. Improvements can be made to the handling of conditional statements and the sharing of variables between potential actors in the input application. The profiling approach is unsafe and requires checking of input applications for conditions on input data values.

Acknowledgement

I was very lucky to get the opportunity to work for one year on my graduation project at the National University of Singapore. The research done in this excellent environment eventually led to this thesis. I would like to thank my supervisor and graduation professor Henk Corporaal for arranging that great experience for me. I would also like to thank my NUS supervisor Dr. Ha Yajun of NUS for providing a place for me at NUS and giving me valuable feedback in our many meetings.

Special thanks goes to Akash Kumar for helping me in the first months in Singapore and introducing Singapore and his many friends to me. Next to his personal help he also offered me a lot of feedback in numerous discussions allowing me to greatly improve my understanding of the problem. Thanks to him this year in Singapore and the travels to India became a very useful and enjoyable experience.

Furthermore I would like to thank Sean Rul for giving his research tool and support to me. I would also like to thank Dr. Todor Stefanov for giving me insight into the world of models of computation and the discussions about my work. I finish with thanking Shakith Fernando, Abhinav Krishna, Priyantha De Silva and Zhu Guolei for bringing some life into the lab and the members of the ES group for their support and help.

Contents

Chapter 1

Introduction

1.1 Background

In recent years we have seen an explosion of new devices for domestic or for mobile use. The numerous new devices include smart phones, set-top boxes, navigation systems, multimedia players, PDAs and many others. These types of electronic devices are called *embedded systems*. An embedded system can be defined as a computer system dedicated to a single task or multiple tasks. They are characterised by containing one or more processors. Typically it is part of a larger machine with other mechanical, optical or electronic parts. Basically every modern device in a common household contains one or more processors combined with some of the other parts and can be considered an embedded device. General purpose computers like a Personal Computer are not considered embedded devices because they can be programmed to perform any arbitrary task.

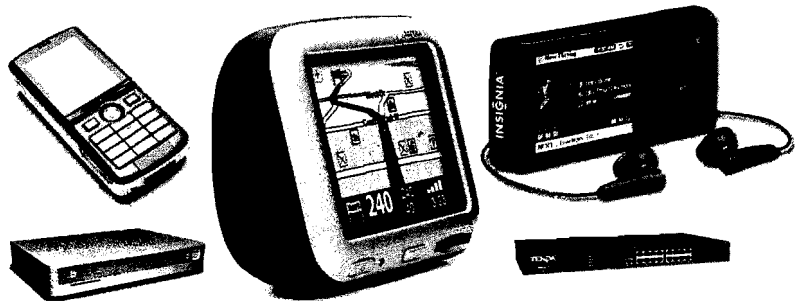


Figure 1.1: Examples of Embedded Systems

If we compare embedded systems designed a decade ago with those designed nowadays, we can see a major difference. In the past the overall majority of embedded devices consisted of devices where the processing unit would take care of the user interface and settings management but most signal processing would be done by independent electronics. Modern embedded systems tend to integrate all functions in a single chip. Additionally there is a shift towards digital signal processing for tasks

that would have been done with analog electronics before. So overall we see a trend towards sharply increasing demands for processing power.

A second trend is the demand for less power consumption. From the early nineties onwards we saw a huge increase in the use of mobile phones which exploded to 3.3 billion mobile phones at the end of 2007 [1]. In recent years we saw a similar trend with portable multimedia players. Many new devices are handhelds running on batteries where the time between battery recharges and weight of the handheld are major points of competition. Therefore when designing these systems we need to consider how to minimise the energy usage.

Third, where in the past a device would have to provide a single application, nowadays it has to provide many. For example a modern multimedia player supports several audio and video standards standards, recording functionality, FM radio and numerous smaller applications like organizers. And these specifications can change with time like recently many manufacturers adding support for MPEG4 in their players.

Concluding we can identify three major trends, increasing processing power, less energy consumption and a high level of flexibility. A possible solution for these challenges is the use of ASICs that implement the tasks in fixed hardware being more power efficient. This has as disadvantage its limited flexibility. Consumer and commercial demands for product specifications may change and might require changes to the underlying system design. The solution central to this thesis, is the use of multi-processor systems.

In modern chip design the power dissipation determines the limit on integration on a single chip and not the available area. This implies a design choice for multiple processors running on lower clock speed instead of a single processor running on a high clock speed. The former occupies more area, which is not a limiting factor nowadays, and thus the generated energy can dissipate over a larger area. Another dimension can be the use of multiple simple processors instead of one complex processor.

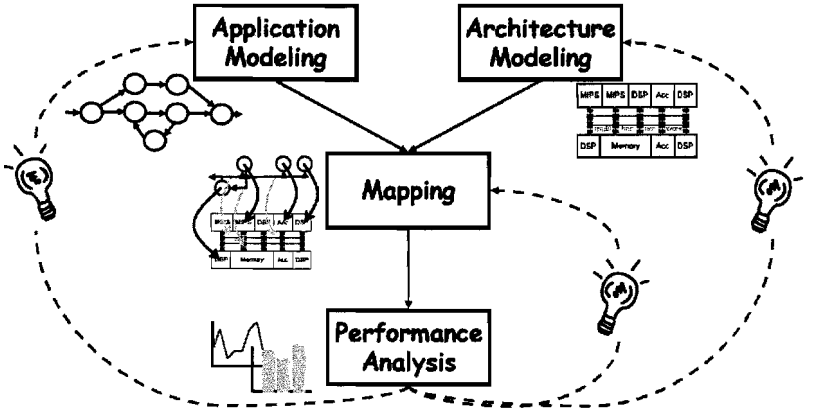


Figure 1.2: Y-chart for DSE

Many parameters come together in the design of new embedded systems. Next to the previously mentioned energy consumption, processing requirements and flexibility, we also must consider limitations due to resource constraints, like available memory, heterogeneous processors (different types of processors in one system), multiple applications and several more. All these lead to a complex multi-dimensional design

space. The process of determining a viable solution for designing an embedded system with certain design constraints is called *Design Space Exploration*. A typical approach often followed, is shown in figure 1.2. Exploring by hand is error-prone and time consuming for large complex designs and thus a need exists for an automated approach. The Multi-Application Multi-Processor Synthesis [21](MAMPS) framework is such an automated approach and provides the base of the work outlined in this thesis.

The trend to multi-processor architectures combines the solutions for many of the current challenges, but unfortunately this change does not come for free. The problem we face is the way most software has been written. Typically programmers write software for single processor systems in a sequential way. Except for some approaches with parallel programming languages or parallel extensions to popular existing languages, writing software for parallel multi-processor systems is not a common procedure. If we want to use the huge amount of existing software or offer the possibility to the programmer to write in a familiar style, we have to find a way to split the software over the multiple processors automatically.

To split the software we have to find those parts of it that can be executed at the same time, otherwise it would not lead to a speed-up. We have to analyse the program and extract the parallelism. In order to find the parallelism we need to determine the dependencies in the code. This can be done by either *static or dynamic analysis*. The former analyses the source code where the program is not compiled or executed, the latter measures the compiled program. Static analysis tends to return conservative results because it cannot take input data into account and thus has to cover all possible control paths. On the other hand dynamic analysis, or profiling, knows which control paths have been taken because it executes the program. The problem here is to judge if the followed control paths are constant for different inputs. Certain dependencies only occur when a condition tests true. If that condition depends on the value of input data some dependencies might disappear with different inputs.

From the analysis we need to derive the actual parallel parts of the program. From the profiling information we can extract the sequence of writing to and reading from memory locations and from that the dependencies. Finally we need a way to present the extracted parallel information. A typical representation used for that is a *Model of Computation* (MoC). We define an MoC as a formal mathematical structure that reflects some aspects of the behaviour of a real-time computational system. One particular MoC and the subject of this thesis is a model called Synchronous Data Flow (SDF). In this thesis I explain how to derive an SDF graph from sequential C source code. Figure 1.1 shows an example of an SDF graph (SDFG).

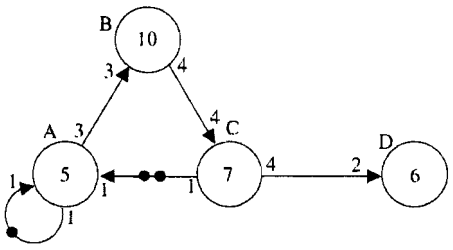


Figure 1.3: Example of a Synchronous Data Flow Graph

1.2 Problem Definition

The subject of the work presented in this thesis is about filling the gap between the applications and the SDF specification. More specifically, determining an approach to extract the parallelism from an application written in the C language and specify an SDF graph as output. The tool from this research should be able to generate an SDF description file which can directly be used by MAMPS, which in turn can generate the MPSoC design for synthesis on an Xilinx FPGA. In fact the resulting tool will become an integral part of MAMPS to form a single Design Space Exploration tool. Central to the research is to define how an SDF can be derived from its corresponding source code and which limitations are involved.

The work can be split in two parts. The first part deals with finding a method to extract the parallelism from the application and express it in an intermediate format. A number of parallelisation engines exist already as partially discussed in Chapter 4, like FP-MAP and PN-gen. Considering the available time for the work and the complexity of the problem making use of one of the existing parallelisation engine seems the preferred choice. Considering the inherent limitations of SDFGs, the set of accepted input applications can also be limited, more particularly limited to DSP and/or multimedia applications.

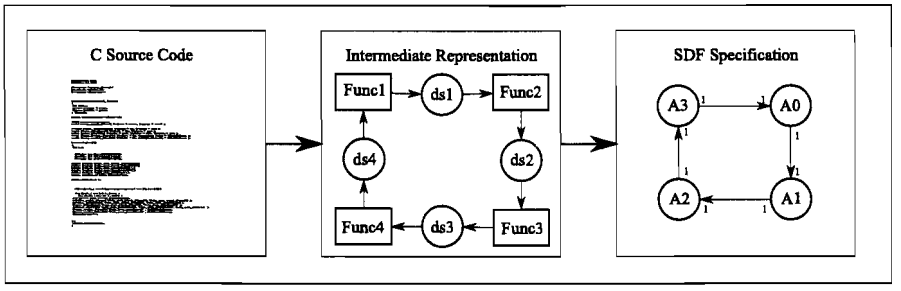


Figure 1.4: Problem overview.

The second part involves deriving the SDF representation from the intermediate format from part 1. The SDF should contain all the common properties and have some support for further automatisation of the flow. Common properties are the actors, channels, worst-case execution time for actors and most challenging, the channel rates. Figure 1.4 illustrates the problem description.

1.3 Thesis Organisation

This thesis is organised as follows. Chapter 2 discusses the various aspects of partitioning. Especially it focuses on the underlying hardware and the communication between parts. Chapter 3 describes the problem definition with a introduction about Models of Computation and describing the limitations of expressing parallelism of both practical and fundamental nature. It continues with an explanation on how to extract the parallelism from the source code and distil an SDF graph. Chapter 4 gives an overview of the related work regarding parallelisation and design tools with multi-processor approaches and explains which tool has been chosen for experimenting. The actual implementation is presented in chapters 5 and 6. Chapter 5 covers the modifications made

to the existing MAMPS code and chapter 6 presents the new C2SDF tool. The short chapter 7 gives an overview of the remaining related work. Finally chapter 8 concludes the thesis and gives recommendations for future work.

Chapter 2

Aspects of Partitioning

In this chapter I provide some foundations for the next chapters. I start with an more extensive overview of the various trends introduced in the previous chapter. Then I continue with common hardware architectures in embedded multi-processor systems. After that I describe the two main programming models used in multi-processor systems, Message Passing and Shared Memory Architectures.

2.1 Trends

In Chapter 1 I have identified three major trends in embedded systems.

- First I mentioned *rapidly increasing computational power demands*. This has various reasons but the main reason is the implementation of many new multimedia technologies in embedded devices. For example where television for decades were largely analog devices with analog signal processing, we see that nowadays LCD televisions require extensive digital video processing. This is illustrated by example 2.1 which shows the schematic of a chip used in LCD televisions. As can be seen in the schematic, the list of supported techniques for video processing is extensive and includes among others MPEG2, MPEG4 and audio processing. The amount of required computational power for these applications contrasts heavily with the analog televisions of earlier decades. A second reason for increased demand for computational power is the digitalisation of analog signal processing techniques.

- Second, a trend towards mobile devices is obvious. In recent years especially we have seen an explosive growth of mobile devices such as mobile phones and multimedia players. Figure 2.2 illustrates the growth over the period from 2000 until the end of 2007. We can see identical trends for multimedia devices like iPods. Typical for mobile devices is the dependency on batteries as energy source and therefore it is important for mobile embedded system designer to minimise the energy consumption. A mobile device that needs recharging too soon can hardly be considered mobile. The time between recharging and the weight of the device (largely due to battery weight) is a major point of competition between manufacturers. That means designers have to *limit the energy consumption* as much as possible.

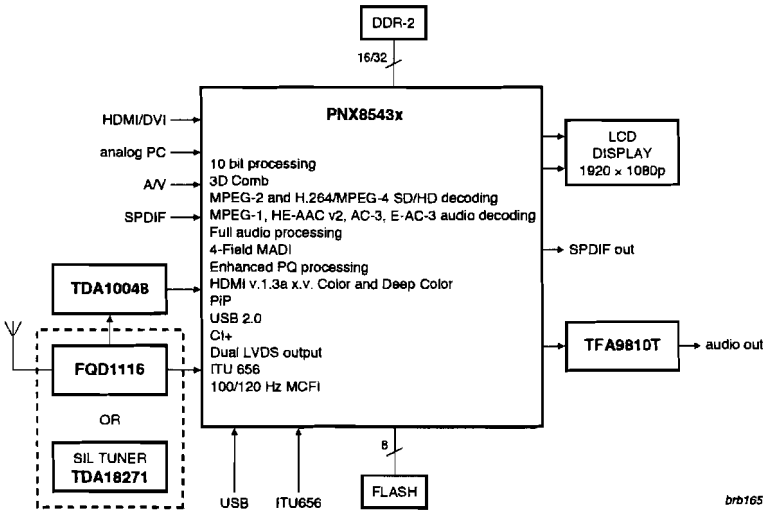


Figure 2.1: Schematic of NXP PNX8543x Home Entertainment Engine

- Third, as already could be seen in the example of the first point, modern devices have to support many different standards and applications. The example of figure 2.1 shows a chip schematic but we can expand that to the feature list of a modern multimedia device such as in figure 2.3. The list of supported standards in the example is extensive and includes next to obvious applications such as video and audio playback also applications like a task planner and a contact organiser. The combination of many different applications on a single embedded device is the third trend we can identify. Additionally the set of applications is subject to changing demands due to marketing and technological advances in general. Manufacturers attempt to share a common platform among all similar products to limit the design cost. However that *requires a certain degree of flexibility.*

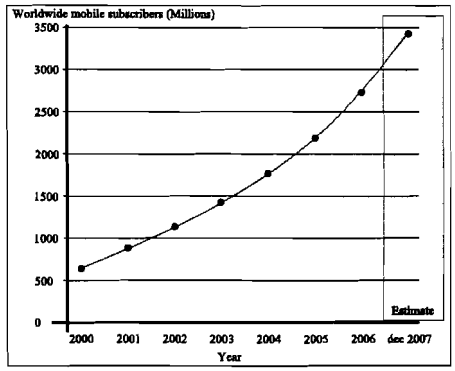
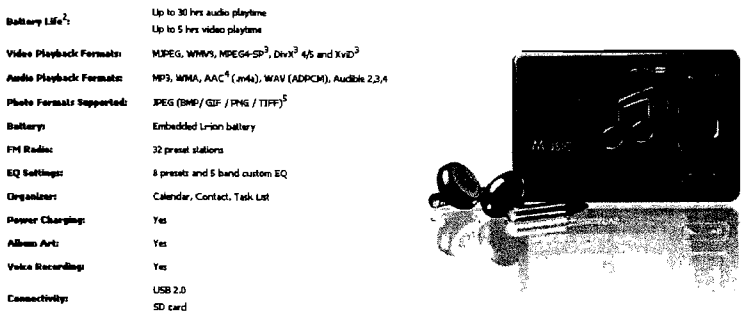


Figure 2.2: Growth of mobile subscriptions 2000-2007, source: ITU



Battery Life[2]:	Up to 30 hrs audio playtime Up to 5 hrs video playtime
Video Playback Formats:	MJPEG, WMV9, MPEG4-SP[3], DivX[3] 4/5 and XviD[3]
Audio Playback Formats:	MP3, WMA, AAC[4] (.m4a), WAV (ADPCM), Audible 2,3,4
Photo Formats Supported:	JPEG (BMP/ GIF / PNG / TIFF)[5]
Battery:	Embedded Li-ion battery
FM Radio:	32 preset stations
EQ Settings:	8 presets and 5 band custom EQ
Organizer:	Calendar, Contact, Task List
Power Charging:	Yes
Album Art:	Yes
Voice Recording:	Yes
Connectivity:	USB 2.0 SD card

Figure 2.3: Features of a common multimedia player. Source: Creative

2.2 Architecture

For efficient mapping of applications to an embedded system we need to take the target platform architecture into regard. In general we can split the architecture in two parts. First the processing units like processors, specific hardware accelerators or I/O devices and second the connections between the first. The connections can consist of point to point connections, buses, crossbar switches and Networks-on-Chip (NoCs). The last category contains meshes, tori and cubes.

2.2.1 Processors

We have identified three major trends which led to demands for increasing computational power, less energy consumption and a high level of flexibility. A possible solution for these demands lies in the use of ASICs that implement the tasks in fixed hardware being more power efficient. This has as disadvantage its limited flexibility. Consumer and commercial demands for product specifications may change and might require changes to the underlying system design. ASIC designs usually cost several millions and this amount increases with newer technologies like smaller feature sizes. It only pays off to design an ASIC for large production volumes. Another solution uses FPGAs for implementation. This offers maximum flexibility but scores worse on energy consumption, which makes it unsuitable for mobile applications. Also the relatively high pricing of FPGAs make them only suitable for smaller niche markets and not for the typical high volume multimedia consumer products. Yet another option is the implementation of all applications on a single processor. Unfortunately general purpose CPUs don't score very well on energy consumption for specialised tasks. Although general purpose CPUs maximise the flexibility. Figure 2.4 summarises various trade-offs between technologies. The ideal solution would be placed in the upper right corner, maximum flexibility with maximum efficiency. All proposed solutions can be considered a variant of *Systems-On-Chip* (SoC). These are usually defined as integration of computer components on a single integrated circuit. The components are not limited to digital parts only, but can also consist of analog and mixed-signal parts.

Several solutions have been proposed to approach the trade-off between flexibility and efficiency. Figure 2.5 shows the most common as proposed in [31]. These solu-

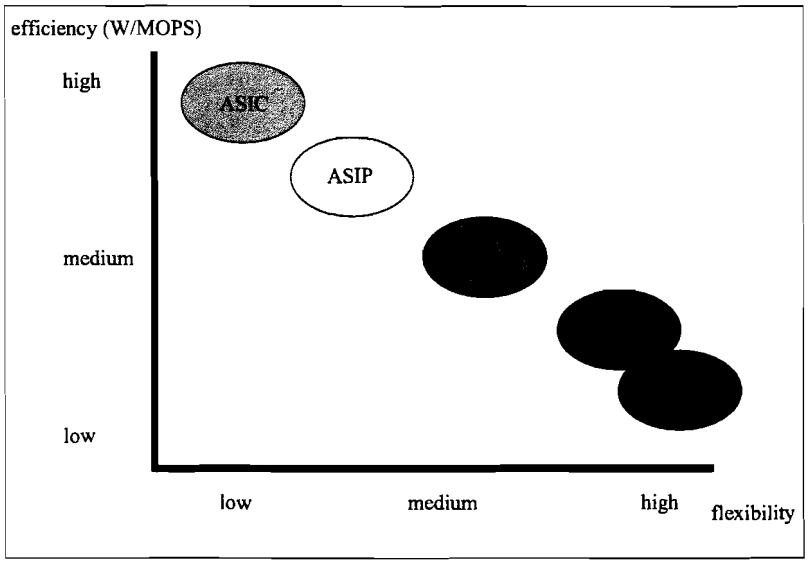


Figure 2.4: Flexibility versus Efficiency for various processing technologies. Source: [14]

tions are called *heterogeneous* System-on-Chip designs which are characterised by a combination of two or more blocks from Figure 2.4.

The processor/accelerator couple gains from the speed-up of certain tasks by the accelerator. Data traffic between the two is dependent on the given application. If we use a loosely coupled accelerator, it can be used by several units in the system coupled by a common bus. This set-up is suitable for a low-traffic application. In case of a high-traffic application a tightly coupled accelerator is preferable, directly connected to a host processor and triggered by specific instructions in that processor. Drawbacks we find in the lack of flexibility of the architecture. If we replace the accelerator by an FGPA we partially eliminate the drawback of the combination. We still face the fixed interface between the two parts which restricts optimisation of the architecture and design space. FPGA vendors offer FPGA with integrated processors and often other building blocks such as DSP modules. Unfortunately it hardly provides an optimal solution with regards to efficiency. Application-Specific Instruction-Set Processors gain from the elimination of the interface between accelerator and processor, but require a much more complex design process, including the development of specific compilers.

Trying to reach the optimal solution in the corner of Figure 2.4, a comprised solution central to this thesis, is the use of *multi-processor systems*. It offers lower energy use than FPGAs and general purpose CPUs while retaining some of the flexibility that ASICs lack. It also implies a shift from hardware to software design. The advantage of multiprocessor systems comes from two sources. On one side splitting the computational load over multiple processor allows these processors to run on a lower clock speed and thus be more efficient than a comparable single CPU. This is related to limitations on chip design. Modern applications in chips are limited by the amount of heat that can be dissipated and not by the available chip surface area. Basically the computational power of a chip design is limited by the energy it can dissipate per surface area

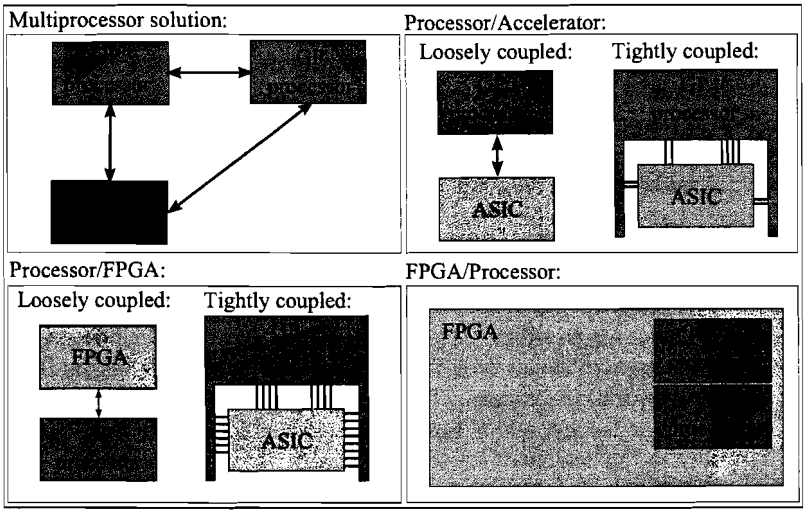


Figure 2.5: Heterogeneous architecture types as SoC building blocks. Source: [31]

available. This is advantageous for multiprocessors because running on a lower clock speed while using more chip area reduces the energy dissipation per square surface area. Another advantage lies in the adaptation of certain processors in the multiprocessor design for their particular task. Since specialised hardware in general is much more energy conserving than general purpose CPUs, we can save in that area as well. However it requires that we map that task to that processor, sacrificing some flexibility for energy conservation. A system with only one type of processor is called homogeneous, with multiple types it is called heterogeneous.

Drawbacks are the lack of linear scaling of the performance with the number of processors and the impact of the communication interface on the whole system. On one hand the amount of parallelisation is limited to some degree. I will explain this subject in sections 3.1, 3.1.3, and 3.1.3. On the other hand the performance of the interconnection network will worsen significantly if we keep scaling up. More on that in the next section (2.2.2).

2.2.2 Communication

In a multi-processor system running parallel programs we need to transfer information from a source processor to a destination processor. Three main criteria are considered for this task:

1. Minimize latency.
2. Support a high amount of concurrent transfers.
3. The network should be inexpensive compared to the processing units.

In this context we define a number of concepts. A *link* is the physical wire on which a signal gets transmitted. On one end a transmitter converts digital values into analog signal which are converted back by the receiver at the other end. In the OSI model

of network design this protocol for conversion forms the lowest layer in addition to all electrical and physical properties. The combination of the transmitter, link and receiver is called a channel. The link-level provides the segmentation into packets or messages for the network. Channels can be connected to switches as well and those switches use the information in the packets for forwarding to the final destination.

I will discuss shortly the most commonly used connection approaches [5].

- First there is the well-known *bus*. It typically uses a single shared medium for communication between the connected devices which is split in control, data and address signals. For deciding access to the bus it requires an arbiter following a protocol for priorities. The arbiter decides which device becomes master when two or more device require simultaneously access. A bus can be either synchronous or asynchronous. The former shares a common clock among all devices on the bus but allows for less complicated logic. The latter gives more flexibility to the connected devices but requires handshaking protocols implemented [7].

- Another approach is the *crossbar network* which avoids the main disadvantage of the bus. It allows multiple concurrent communications instead of a single as in the case of the bus. The crossbar consists of many small switches organised into a grid and has N inputs and M outputs. That allows for $min(N, M)$ of connections without overlapping. Typically these crossbars are used in small-scale shared memory multiprocessor systems to let each process read from a different memory at the same time. A simple arbiter is included for the occasional situation when two processors read from the same memory.

- The third category are *Networks-on-Chips* (NoC), to which several types belong. Some of the more important ones are the *mesh*, *torus*, *cube* and the *tree*. As shown in figure 2.6 a mesh is a n-dimensional network and can be defined as an interconnection structure that has $K_1 \times K_2 \times \cdots \times K_n$ nodes where n is the number of dimensions of the network and K_i is the radix of the dimension i.

  The torus has wrap-around connections to turn every axis into a circular bus. The cube is a particular form of a 3-dimensional mesh. All these networks require some kind of routing mechanism. For example *dimension-ordering routing* the message travels in dimensions in a consecutive order. Other routing mechanisms include *dimension reversal routing*, the *turn model routing*, and *node labelling routing*. MPSoCs with mesh networks perform efficiently for many scientific computations and scale very well. Finally in the tree network, nodes exist in a hierarchical network. Every node (except the root) can have multiple children and a single parents. Communication between children always requires sending a packet to the common parent first.

2.3 Programming Model

In this section I discuss the general concepts relevant for the work in this thesis. The first concept is the programming model. Culler [5] describes the programming model as the conceptualization of the machine that the programmer uses in coding applications. It specifies how parts of the program running in parallel communicate information to one another and what synchronization operations are available to coordinate

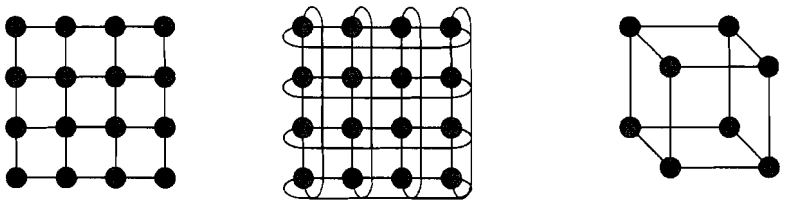


Figure 2.6: Mesh, Torus and Cube. Source: [5]

their activities. On top of the programming model the designer writes the applications. Programming models range from a system similar to modern multi-processor desktop systems, where multiple processors execute a number of independent sequential programs to more advanced parallel programming models such as shared memory/address, message passing and data parallel programming.

These models can be described as:

- Shared address programming where one can communicate with others by posting information at known, shared location.
- Message passing where information from a specific sender is sent to a specific receiver.
- Data parallel processing where several processing units perform an action on separate elements of a data set simultaneously and then exchange information globally before continuing.

These programming models offer us the possibility to implement parallel applications on a parallel system. However they don't give us a way to express the parallelism in an application. The following sections describe the two most common basic programming models for multi-processor systems, *Shared Memory* and *Message Passing Architecture*. When mapping an application to a multi-processor system we face a dilemma. If we consider a computer program as an assembly of dependent tasks, we see that a task might require data from another task. We call this requirement a *dependency*. This dependency represents actual data which we have to transfer between the tasks. In a single processor system the program runs with a single memory and there is obviously no requirement for transferring the data. However when we run the parallel program on a multi-processor system we need to actually transfer the data to other processors. The first proposed architecture is very similar to the original single-processor system. All processors share a common memory accessed through an interconnection network as shown in Figure 2.7. Communication between processors occurs through shared memory locations. From the figure we can easily see the major disadvantage of this solution, all memory communications share a single connection. The memory access becomes quickly a bottleneck when scaling the design. Subsection 2.3.1 gives a more detailed overview of the shared memory architecture and its (dis)advantages.

An alternative architecture is called *Message Passing Architecture* (MPA), in which all processors have their own local memory. The bottleneck to a single memory does not exist but the communication between tasks/processors becomes much more complicated. As shown in Figure 2.8 a typical MPA system consists of multiple processors

with each its own memory, all connected by an interconnection network. Because it lacks a shared memory, data must be passed to other processors by other means. Every processor can send packets with data to the other processors through the interconnection network. Typically some addressing system is required to assure packets arrive at the right destination. Subsection 2.3.2 deals with MPA in more detail. When using a suitable interconnection network MPA scales much better than Shared Memory but it is not straightforward to get functional communication between the processors. Every processor requires code to be added for handling the communication.

2.3.1 Shared Memory

A shared memory model is defined as a model in which the processors communicate by reading and writing locations in shared memory. These locations are equally accessible to all processors. Next to the common memory processors can have local memories as well, for instances caches are widely used in Shared Memory models for performance reasons. We can identify 3 properties that a shared memory system needs to function correctly.

- *Access control* manages the access of processes to resources. If a processor wants to access a shared memory location the access control checks whether it is possible. It usually keeps track of accesses in a table and blocks any attempt until the resource has been cleared.
- *Synchronisation* provides a set of rules to ensure system functionality and that information flows properly. For example it can limit the time processors can use a resource.
- *Protection* takes care of shielding processes from memory locations of other processes.

The two main challenges for designing a shared memory system are performance degradation due to contention, and coherence problems. When too many processors want to access the memory it becomes a bottleneck. The most common solution adds caches to the processors to solve the problem. In an extreme case contention can be reduced to zero when all data is stored in the local caches. Unfortunately this creates the other problem, coherence. If there are multiple copies of data, in the caches and memory, it might lead to coherence problems. Incoherent means that not all copies are equal because a processor has overwritten a particular copy. Several solutions exist to solve this problem, but these are beyond the scope of this thesis.

Hardware

A simple shared memory system consists of a single dual-ported memory connected to 2 processors. An arbiter inside the memory decides which processor gets access to the memory. In case only a single request is received, it will be granted and the processor gains access to the memory. The status of the memory will be set to busy. Any request from the other processor will be put on hold as long as the other is being serviced.

We can identify two major classes of Shared Memory architectures, Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA). In an UMA system all processors can access the shared memory through an interconnection network. In

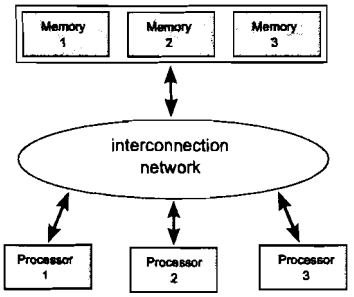


Figure 2.7: Shared Memory Architecture.

this way it is very similar to the way a single processor system accesses its memory. The interconnection network can be a bus, multiple buses or a crossbar switch. Because of the balanced memory access these systems are referred to as Symmetric Multi-Processor systems (SMP). It is a very popular architecture in shared memory systems. The second class NUMA differs from UMA in the way the memory is organised. For NUMA each processor has its own part of the shared memory directly attached, but still the whole system has a single address space and therefore all processors can access any memory location. Obviously the access times depend on the distance to the processor and that leads to non-uniform access times. The interconnection network can be formed by a tree or a hierarchical bus network.

Software

The parallel programming of Shared Memory systems is similar to the programming of operating systems and general multiprogramming. It can be done by extensions of existing programming languages, operating systems, and code libraries. In a parallel program we must be able to identify three constructs:

1. *Task Creation* We can divide the task creation in coarse- and fine-grained levels. On the coarse level the shared memory system can provide time-sharing, where new processes are assigned to the processor with the least amount of work. In case the process is a large task with page tables, memory and file descriptions in addition to the program code and data the process will have a high overhead and is most suitable for heterogeneous tasks. The fine-grained tasks are more suitable for homogeneous tasks and allow parallelism in single applications. This pattern is called supervisor-workers model.

2. *Communication* Parallel and sequential processes don't have identical memory spaces. Where a sequentially executing process has 3 segments called text, data and stack, a parallel process has an additional segment called shared data. In the text segment the process stores the binary code for execution. In data the program's data is stored like variables and in stack dynamic data is stored, like the data to be stored when calling a function. The latter two expand and contract while executing the program and have a gap between them. There is no sharing of addresses between processes, except for parallel process by means of their shared data section. This shared data section is located between the data and

stack segments and can also change sizes. Communication can be performed by writing or reading to variables in these shared data sections.

3. *Synchronisation* Because it is undetermined what happens if two processes write the same variable simultaneously, we require a mechanism to avoid this. This synchronisation mechanism must provide mutual exclusive access to processes. It also coordinates the execution of parallel processes and can synchronise at certain points in execution. It uses two main synchronisation constructs, locks and barriers. A lock blocks access to other processes while a process accesses the resource. A barrier blocks a process until all other processes reach the same barrier.

2.3.2 Message Passing Architecture

The main difference of the Message Passing Architecture (MPA) model to the Shared Memory model lies obviously in the lack of a shared memory. Every processor in MPA has its own local memory and communication has to happen through send and receive operations. The combination of a processor and its memory is typically called a node. A node has the ability to store message temporarily in buffers until it can be sent or received. Processing can happen at the same time as sending or receiving. The nodes are connected by an interconnection network which can range from an architecture-specific structure to a world-spanning network like the internet. A major advantage is the high scalability without significant decreases in efficiency.

Hardware

Various networks are used for connecting the nodes, these include hypercubes, 2D and 3D mesh networks. When designing these networks the most important design factors to consider are link bandwidth and network latency. We define the link bandwidth as the number of bits that can be transmitted per unit time. The message latency stands for the time needed to complete a message transfer. On top of this principal improvements have been made such as wormhole routing where messages are sent in parts to reduce the required buffer size.

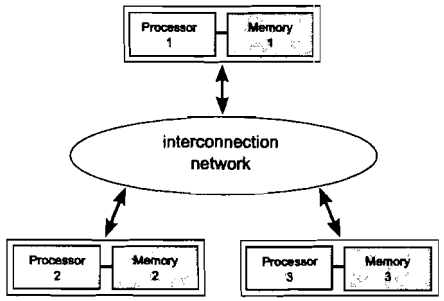


Figure 2.8: Message Passing Architecture.

When executing an application, the program is divided into concurrent processes that are executed on separate processors. In case that there are more processes than

processors, some processes have to share a processor in a time-sharing fashion. Communication between processes running on the same processor use internal channels to exchange messages. Communication with processes on other processors use external channels. We define a message as a logical unit for inter-node communication. It is considered a collection of related information travelling together. It can consist of instructions, data, synchronisation, or interrupt signals. There is no sharing of data, only copying and sending in messages. This way of communication leads to performance gains because there is no need for synchronisation constructs. Another advantage is the excellent scalability. A large number of nodes can be included in the system without significant performance issues. A node can execute multiple processes and they usually are of medium or coarse granularity.

The technique to find a path in the network from origin to destination is called routing. A more formal definition according to el-Rewini: routing involves the identification of a set of permissible paths that may be used by a message to reach its destination, and a function, h, that selects one path from the set of permissible paths. One way to classify routing techniques is whether they are adaptive or deterministic. Adaptive techniques decide their path based on current network conditions, deterministic use only the source and destination information. Deterministic techniques can possibly be inefficient. Another classification method is based on where the routing decision is made. Either centralised, where the path is decided before sending or distributed, where every node decides itself about the next destination. Centralised routing require knowledge of the whole network contrary to distributed that only requires knowledge of the neighbouring nodes.

Routing is not without problems. The three most important problems include *deadlock*, *livelock* and *starvation*. Deadlock can occur when two messages hold the resources required by the other one. This happens whenever a cyclic dependency exists for resources. A flow control mechanism can be used to avoid this situation. The most simple but inefficient solution is to allow re-routing of messages or resending messages. The second problem livelock happens in an adaptive routing system when a message never reaches its destination because of continuous re-routing due to network contention. It can only occur when using dynamic injection routing, which allows nodes to transmit messages at any arbitrary time. A possible solution makes use of priorities for message routing. Finally starvation occurs when a node never gets the opportunity to send messages. Also starvation can only occur in dynamic injection systems. The solution here is to keep track of a queue for send and receive operations.

The mechanisms used for remove data from an input channel and place it on an output channel we call switching mechanisms. The chosen mechanism has a high influence on the latency. I give a short overview of the various mechanisms:

- In *circuit-switching* the whole path for the message is determined beforehand and all links are reserved. This eliminates the need for buffering. After the transfer the path is released for other messages. It guarantees a maximum routing latency and a certain bandwidth. It is advantageous to use it in a system with a large number of message transfers.

- Alternatively in *store-and-forward (packet-switched)* switching the message is split in smaller parts, called packets. A complete path might not be available at transmission start, but packets are moved from node to node as links become available.

- *Store-and-forward (virtual cut-through)* is similar to packet-switched but when a

packet arrives at a node, it is forwarded if channels are available without waiting for the whole message. Because unnecessary buffering is avoided this gives an advantage regarding latency.

- With *wormhole* routing the packet is further split into smaller units called flits. The first (header) flit finds a way through the network and the following flits cannot overtake it. It results in an latency independent of the path length and requires less storage than the other techniques.

Software

Next to the underlying hardware we also require software support to run the communication. Usually a message passing architecture uses a set of primitives that allow the processors to communicate. Typical primitives are send, receive, broadcast and barrier. The basic programming model attempts to match a send in one processor with a receive in another processor. These commands are blocking, meaning that send blocks until receive is executed. In a blocking architecture we need to implement a 3-way communication protocol, involving a *request-to-send* by the sender, a *ready to receive* reply by the receiver and finally the *data transfer*. This protocol takes a long time to complete because it involves a full round trip. That means the sender and receiver are blocked during that time and for that reason most MPAs utilise a non-blocking operation. The send appears immediate for the user program, although the message layer buffers the message until it can be transmitted. At the receiver's side the message is buffered until it can be received in the program. Popular software implementations are *Pthreads*, *CSP* and *MPI*.

The first implementation *POSIX Threads* (Pthreads) is a library that offers programmers the possibility to create, manipulate and manage threads, as well as techniques for synchronisation between threads. Threads can be described as the execution state of a program instance with just enough properties to be able to run independently. Pthreads provides a standardised implementation of threads for UNIX, BSD and Windows operating systems. It requires design by hand and offers little analysability.

Another approach is *Communicating Sequential Processes* (CSP) [29]. CSP uses formal methods to describe a system in terms of component processes and events. These processes are independent and interact through message-passing communication. Processes can be both sequential and parallel. The designer specifies the whole behaviour of the system in terms of algebraic operators. The events represent communications or interactions. CSP is strong on modelling and analysing systems with complex communication behaviour.

Finally it is worth mentioning the *Message Passing Interface* (MPI). MPI provides a standard library of routines for writing portable and efficient message passing programs. It is not a language but programs can call its functions to simplify the programming of message passing architectures. Included are many point-to-point communication routines and operations for data movement, global computation, and synchronisation. MPI has been extended in the MPI-2 standard which adds support for dynamic processes, client-server support, one-sided communication, parallel I/O, and non-blocking collective communication functions. An MPI application consists of a collection of concurrent communicating tasks. This program includes code, written by the application designer, that calls standard functions in the MPI library. Every tasks receives a unique rank to identify it in the network. This rank can be seen as an address. The tasks run both on single processors and multi-processors.

Chapter 3

Problem Definition

3.1 Parallelism

The trend to multi-processor architectures offers a promising solution for many of the current challenges, but unfortunately this change does not come for free. The problem we face is the way most software has been written. Typically programmers write software for single processor systems in a sequential way. Except for some approaches with parallel programming languages or parallel extensions to popular existing languages, writing software for parallel multi-processor systems is not a common procedure. If we want to use the huge amount of existing software or offer the possibility to the programmer to write in a familiar style, we have to find a way to split the software over the multiple processors automatically.

Many computer programs used in embedded and desktop systems fall into the category of sequential programs. They have not been optimized for use in a multiprocessor system. This wastes a large part of the potential computing power of the multiprocessor system because the program only keeps a single processor occupied. Obviously this could have a negative impact on the performance of the system as well. Some code might execute faster on a certain type of processor. In this and the following subsections I will provide a theoretical foundation for parallelism.

3.1.1 Dependencies

Data Dependencies

We can view a sequential program as long chain of instructions. In a fully parallelised program all these could be executed independently. The other extreme is the situation where all instructions would depend on each other. The defining factor here is whether instructions read and write the same data. In other words, dependencies between instructions exist when one writes to a particular data location and the other reads from it [2] [16]. I'll explain here the work of Bernstein.

Let us consider a sequential program which consists of a set of $P_1, P_2, ..., P_k$ of program fragments. Every fragment can contain multiple instructions. For every fragment P_i we define an input set I_i with all variables read by P_i. Similarly we define an output set O_i for all variables written by P_i. Bernstein considers two fragments P_i and P_j independent and parallel executable if they satisfy his three conditions:

$$1)\ I_j \cap O_i = \emptyset,$$

$$2)\ I_i \cap O_j = \emptyset \text{ and}$$
$$3)\ O_i \cap O_j = \emptyset.$$

which means that the inputs of one fragment cannot be the output of another fragment if the fragments run in parallel (first 2 conditions). The third condition specifies that the outputs of parallel fragments must be different.

A violation of each Bernstein condition leads to a certain type of dependency:

1. Flow dependence: when the second fragment uses an result produced by the first fragment. Violation of the first condition.
2. Anti-dependence: when the second fragment overwrites the result of the first. Violation of the second condition.
3. Output dependence: when both fragments write to the same variable the second fragment must provide the final value. This is a violation of the last condition.

Control Dependencies

Next to data dependencies we also have to take control dependencies into account. They form a problem when the condition is data-dependent. It occurs when a statement gets executed only if a condition is true and this condition is determined in a previous statement. It is possible to convert a control dependency to a data dependency. In some languages a special keyword exists like where in Fortran 90 to to split the original if-then-else contruct. In C no such keyword exists, but one can split the if-else construct into loose and independent if statements. In the case of nested if statements the new conditions will be the logical-and of all nested if statements.

Resource Dependencies

The third kind of dependency is about the underlying hardware. In case code requires access to a certain type of hardware like a divider or FFT block, two fragments cannot access at the same time and thus cannot be considered to be parallel. It applies mainly to heterogeneous SoCs with dedicated hardware for certain tasks. It is considered more a scheduling problem than a parallelising problem.

Variable Names and Pointers

Up to this point it we did not discuss how to deal with the variables in the input and output sets. These variables have been kept abstract but in fact they represent a location in the system's memory. At compile time variable names get assigned to this memory location and the program uses that information every time it is executed. However this does not cover all situations. In C the use of pointers is very common and usually pointers get assigned at run-time. Because of this connection between names and memory locations we cannot determine all dependencies at compile-time. Signal processing applications depend heavily on arrays for their algorithms and in C the difference between pointers and arrays is small. Thus for this class of applications the problem could be considerable.

3.1.2 Granularity

Loops

Loops are especially for signal processing applications of high importance. Many applications require repeated execution of the same processing steps, so obviously these programs contain loops. For a loop we can define a dependence distance between two statements. The formula for this distance D is equal to $D = \lambda - \kappa$, where λ and κ stand for integers calculated by taking the indices of the arrays called in the loop. D being equal to zero corresponds to a intra-loop dependency, i.e. one that would also exist without a loop. D being one or larger means that the dependence is loop-carried or an inter-loop dependency, i.e. it will carry on to one of the consecutive iterations of the loop. A D of 2 means that a particular array location gets read 2 iterations after it was written.

Nested Loops

For nested loops the problem gets more complicated. In the equation for simple loops we substitute all variables by vectors. In addition to the dependence distance vector D_{Dist} we also get a dependence direction vector D_{DirV}. The latter indicates how a particular dependence in the D_{Dist} vector behaves, i.e. if it crosses over to other iterations. If no dependence distance vector exists the statements can be executed safely in parallel. In general there are two widely used graphs to show these dependencies:

Dependencies and FIFOs

Loops follow a certain pattern for accessing memory locations in their body. Often this pattern is straightforward with the index variable running from 0 to N incrementing by 1 every iteration. With a nested loop the pattern follows usually row after row. If we have two or more consecutive nested loops and a statement in the second depends on a statement the first sharing the same memory location the situation can occur that the second nested loop requires data from the first which has not been processed. In a sequential program the problem does not occur because the first loop completes before the second. Obviously you need then a relatively large buffer for the intermediate data. In a parallel system where we cannot assume that the previous loop has been completed we have to assure that the second doesn't have to wait for the first to send the correct data packets. In practice this can mean that the sequence of data calculation should be carefully scheduled according to the read and write sequence of the two loops or that a large enough buffer gets included which would at the same time eliminate the parallelism.

Functions

Next to loops we can look at the structure of a program also in a different way. A function is defined as a portion of a program which performs a certain task relatively independent. Functions serve both the readability of a program and the reduction of code size and its compiled binaries. A function takes a number of parameters as input and can return a single value back to the calling function. Parameters can act as inputs or outputs by using call by reference.

Statements

Another level of granularity are statements or the set of statements which we call snippets. They form a set because they can be grouped together with a common set of dependencies. A function consists of a number of statements. If no dependencies interfere a number of snippets can be executed in parallel, similar to functions and loops.

3.1.3 Measuring Parallelism

Amdahl's Law

One of the most fundamental questions for parallelisation is how much parallelism we can expect. Amdahl's law is a simple and general model for the expected performance after parallelisation. Basically it provides a measure for the percent speed-up of the parallelised performance compared to the full sequential performance. It is important to realize that not every part of a program can be fully parallelised. The following formula is Amdahl's law for the speed-up:

$$S_{Parallel}(f,n) = \frac{1}{(1-f) + \frac{f}{n}}$$

- with n the number of processors.
- with f the fraction of a program which can be parallelised.

A modern version of Amdahl's original law is [15] :

$$S_{enhanced}(f,S) = \frac{1}{(1-f) + \frac{f}{S}}$$

with S the speed-up of fraction f.

Figure 3.1 shows the speed-up for different percentages of the parallelised part.

For the problem definition we can draw the conclusion that parallelising a small portion of the total program already can lead to a significant speed-up, but the total execution time of the program after parallelisation can never be smaller than that part that cannot be parallelised.

Gustafson's Law

The disadvantage of Amdahl's law is that it assumes a constant problem size. In practice we solve larger problems with larger computers. Gustafson [13] proposed his law where the sequential part is independent of the problem size. He assumes that the parallelism increases with the problem size. Where Amdahl assumed the execution on the serial system to take 1 unit of time, Gustafson assumes that the execution on the parallel system takes 1 unit of time. The time spent on the serial system is equal to $N(1 - F_s) + F_s$, which leads to a speed-up of:

$$S = N(1 - F_s) + F_s$$

F_s is the time spent in sequential code. N is the number of processors.
Now the speed-up scales up with the number of processors.

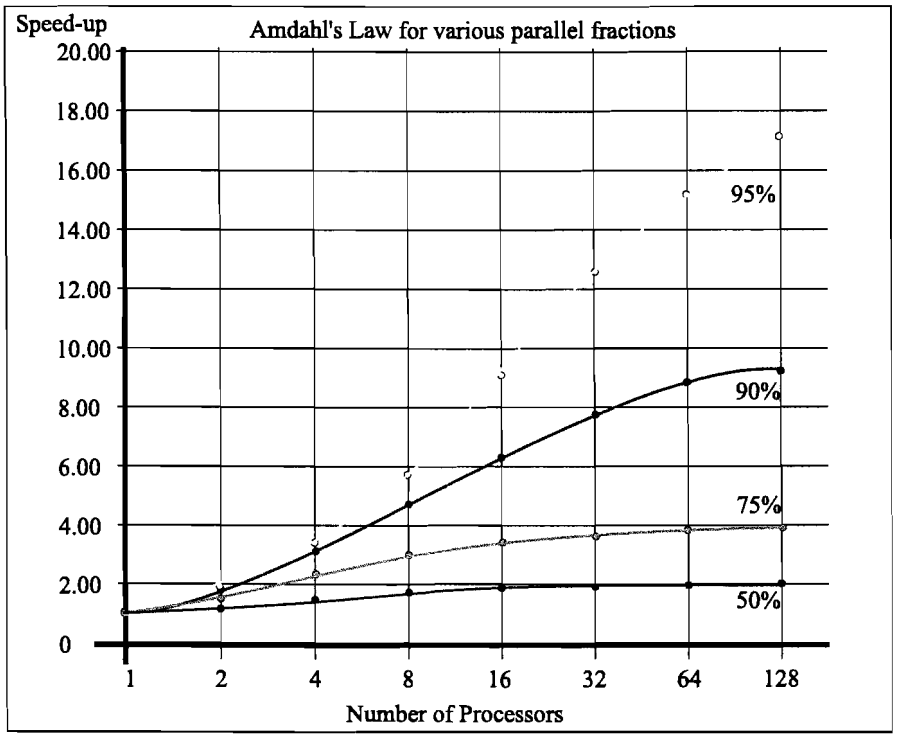


Figure 3.1: Speed-up for different percentages of the parallelised part. Source: [15]

3.1.4 Partitioning

Generally parallelism can be exploited on several levels. The most common are the Instruction-Level, Data-Level and Task-Level parallelism.

Instruction-Level Parallelism

Instruction-Level Partitioning (ILP) exploits the situation where two or more instructions are independent of each other. It allows these independent instructions to be executed simultaneous or in reversed order. As the name suggests this parallelism occurs on the instruction-level and is typically addressed on the processor architecture level. This includes both improvements in the field of compilers and processor architecture design. Typically scientific applications have a large amount of ILP contrary to application like cryptography. Several techniques and processor architectures have been proposed to exploit the ILP. These include the well-known Instruction Pipelining where multiple instructions executed in parallel by using different parts of the processor. It makes more efficient use of the processor resources by keeping them almost fully occupied. Superscalars where the processor contains multiple instantiations of each of the basic building blocks like instruction fetching unit or ALUs. Out-of-order Execution allows changing the program execution order to avoid idle cycles because of delay due other instructions.

Data-Level Parallelism

Data-Level Parallelism (DLP) exploits the parallelism in program loops. The idea is to split the data among separate processing units performing the same calculations on different parts of the input data. For example processor 1 can process the first 50% of the data set, processor 2 takes care of the second 50%. Both processors execute in parallel and optimally they finish in half the time.

Task-Level Parallelism

In contrast with DLP *Task-Level Parallelism* (TLP) splits along the lines of different tasks. It assigns in general different tasks to each of the various processors. For example in a program with two tasks, task A and B, processor 1 will process task A, task B will be processed by processor 2. Most applications show some degree of both Task-Level and Data-Level Parallelism, including the work presented in this thesis.

3.2 Models of Computation

In the previous sections I elaborated about models for implementing parallel applications. These models are of high practical use but are limited in analysing the properties and behaviour of a parallel program. The numerous steps in the design of a new product often require some form of modelling. We use functional modelling for specifying the functional behaviour and performance modelling for determining other characteristics of the system. The correct behaviour of the final implementation is validated to its specifications.

Since the real product is not yet available while designing and prototypes are prohibitively expensive we have a clear need for models. Modern electronic products might need hundreds of models which makes prototypes even less attainable, but we still want a optimum solution with correct timing and minimum use of resources. This would lead to a time consuming and error-prone design space exploration (DSE) by hand. Furthermore modern designs are so complex that we need mathematical models to realise predictable designs. The formal features of mathematical models guarantee certain properties and allow us to use efficient synthesis tools. Before we give a definition of such models of computation we first explain the building blocks.

The basic concepts of MoCs are processes, events and signals. Events are the elementary units of information exchange between processes. Processes receive or consume events, and they send or emit events over a medium called signal. One process at a time can only emit into a signal but multiple can receive. Also the order in which events are received is preserved. This is not unlike how digital logic behaves.

Internally processes are divided into evaluation cycles. This cycle consists of a consumption of input events, computation of the new internal state and emission of the output events. Only the rules of the MoC limit how a process can be modelled. Components of the process are the initial state, an output encoding function and a next-state function. During an evaluation cycle the process consumes exactly one input event for each incoming signal and produces exactly one output event for each outgoing signal. There are two more concepts to introduce: Process constructors and combinators. The former is similar to the C++ constructor because it defines a process and its properties. Examples include a constructor with one input and one output, another one with no internal state and one with an internal state. The properties are determined by the parameters of the constructor and include how the next-state function, the output

encoding function and other properties behave. The latter, combinators, allow us to compose process networks out of simpler parts. The combinators are parallel composition, sequential composition and a feedback operator. We are now ready to define a Model of Computation:

A Model of Computation is defined as the set of processes and process networks that can be constructed by a given set of process constructors and combinators.

Furthermore, we also define a untimed MoC:

An untimed Model of Computation is characterised by the absence of timing information for use by processes.

Finally we need to define the behaviour of a process and how it interacts with other the signals. We represent this functional behaviour by the characteristic function and the signature of a process. The characteristic function determines the behaviour in each evaluation cycle. The process signature describes the input and output behaviour. Uprating a process means that it consumes and produces twice as many events in each cycle, identical to execute the process twice.

Processes communicate with other processes by writing to and reading from signals. Events represent the value or are values themselves. We mention three types of events: Untimed, Timed, and Synchronous events. Untimed events only contain values, timed and synchronous events include a time value, where timed events occur at a much finer granularity. Timed events are used for psychical times like nanoseconds while synchronous are used for abstract time values such as clock cycles. Signals are sequences of events. Other properties are that they are ordered, can be finite or infinite, and three types exist: untimed, synchronous and timed. We can consider processes as functions because for a particular input signal we'll always get the same amount of output signal. This still allows processes to have an internal state. The resulting output event might not have the same value at different times though.

Three important properties of processes are *monotonicity*, *continuity*, and *sequentiality*.

- *Monotonicity* indicates that more input data leads to more output data without changing the data that has been output already. For example a sorting process is not monotonic because a new input value would change the sequence of the current output.
- *Continuity* allows a process to start producing output before all input events have been received.
- *Sequentiality* involves processes with more than one input and a process is sequential if the inputs are not independently processed. It means that if a process is waiting for an event on one signal, it does not consume more events on any other signals. We say the process blocks then.
- Another concept related to processes is the *characteristic function*. This function defines the relationship between the input and output signals. The function generates all output signals of the process and a process can have numerous characteristic functions.

Several models of computation are suitable for designing multimedia applications. These models differ in aspects like expressiveness and analysability. Obviously designers have to make trade-offs between thee aspects. We will take a closer look at two MoCs, Synchronous Dataflow and Kahn Process Networks.

3.2.1 SDF

Synchronous Dataflow (SDF) is a special case of the untimed model of computation, where, to avoid confusion, synchronous has the meaning of static or regular. Its main advantage lies in the suitability for multimedia applications due to its high level of analysability, but its disadvantage is the limited expressiveness. Typical of an SDF is the constant relationship between input and output. Processes always consume and produce the same number of events on their signals, in other words the data rates are constant. This constant behaviour makes SDF very suitable for modelling applications. SDFs are monotonic, sequential but not continuous.

monotonicity, *continuity*, and *sequentiality*

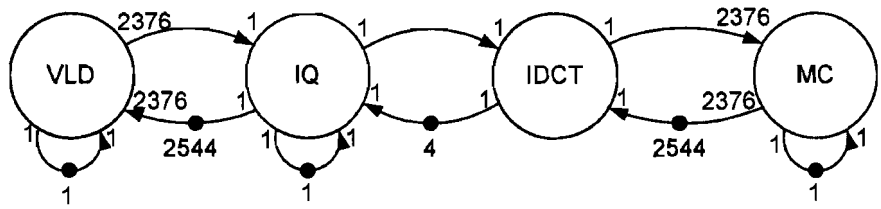


Figure 3.2: The SDF model of an H.263 decoder. Source: [12]

Depending on available resources each process can either be mapped to its own resource like a processor or custom hardware block, or usually several processes will share a single resource. Basically possibilities range from one resource per process to all processes on a single resource. When sharing resources we need a way to divide the resources. The two principal ways of *scheduling* are *static* and *dynamic* scheduling. Static scheduling precomputes a schedule while dynamic computes a schedule during runtime. Obviously dynamic scheduling is more suitable for less predictable systems but also causes significant overhead. The predictability of static scheduling fits SDF quite well. In fact if a static schedule exist for SDF, it can always be found. The schedule will be applied periodically since inputs streams will be very long or possibly infinite.

The naming for SDFs is different from the untimed MoC introduced in the previous section. We call processes in an SDF graph (SDFG) *Actors*, signals become *Channels* and events become *Tokens*. Tokens already present at the start are called *Initial Tokens*. Figure 3.2 shows an example of an SDF graph from [12]. This SDFG represents an H.263 decoder, which is a video codec often used for video-conferencing. We see four actors VLD, IQ, IDCT and MC each performing a part of the frame decoding. The channels between the actors have two numbers assigned, the channel rates. Actor VLD produces 2376 tokens for every firing and consumes 2376 on the incoming channel. A firing is a single execution of an actor and for each firing it also consumes a token on the self-channel which represents the data crossing the iteration boundary. Actor

firings are atomic operations and cannot be interrupted. The duration of the firing is set by a property called execution time. A token models a particular amount of data, which is not necessarily identical for all channels in the graph. Initial tokens are modelled by the black bullets on the channels. They are necessary to execute the SDF graph because without any initial tokens no actor would have tokens on its inputs and would thus not fire. The four initial tokens on the channel from IDCT leftwards to IQ model the buffer sizes of the rightward channel from IQ to IDCT. In case IQ fires more often than IDCT it can consume the four initial tokens. After it runs out of these four it has to wait until IDCT outputs a new token. In this way we model a four-position buffer for the rightward channel because the channels in SDFGs have no value for buffer size. The H.263 example displays the expressiveness of SDF graphs for multimedia applications. Most multimedia applications will be similar to the H.263 example and therefore can be modelled by SDFGs.

An SDF graph has to follow a schedule which can be expressed by a vector. The *repetition vector* is one particular instance of the vector. The existence of this vector proves that the SDF graph has a correct schedule and therefore is correct itself. We can calculate the repetition vector by satisfying this balance equation [23]:

$$\Gamma \times \vec{q} = \vec{0}$$

with Γ representing the topology matrix. This topology matrix is a compact description of the SDF graph. In the matrix there is one column for each actor and one row for each channel of the SDF graph. There are 3 possible entries for each position in the matrix:

1. A zero means there is no edge to or from the actor.
2. A positive number means there is an outgoing edge from the actor.
3. A negative number means there is an incoming edge to the actor.

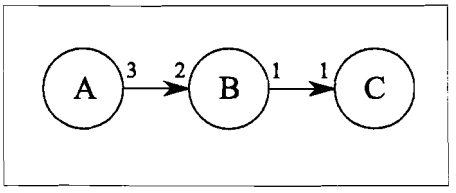


Figure 3.3: SDF example.

The SDF graph in Figure 3.3 can be represented by the following topology matrix:

$$\Gamma = \begin{pmatrix} 3 & -2 & 0 \\ 0 & 1 & -1 \end{pmatrix} \text{ which leads to a repetition vector } \vec{q} = \begin{pmatrix} 2 \\ 3 \\ 3 \end{pmatrix}.$$

3.2.2 CSDF

Cyclo-Static Data Flow [27] (CSDF) is a generalisation of SDF. CSDF allows actors to have cyclicly changing firing rules (or functions). In SDF an actor behaves exactly the

same for each firing, but in CSDF an actor follows a sequence of one rule per firing. The actor switches to the next rule after a firing until all rules have been used. Then it starts over. This property can be used to model behaviour that happens less often than the general flow of an application. In a similar fashion as with SDF we can also construct the topology matrix for CSDF. The elements in the matrix stand for the cycle times of the various actors instead of the channel rates in the SDF case. The resulting $\vec{q}$ vector contains then the number of cycles instead of the number of actor firings. Finally it is worthwhile to mention that CSDF graphs can be transformed to SDF graphs. The scheduling of actor firings is exponential relative to the number of nodes in an SDF graph and CSDF only makes this worse because of the scheduling of the cyclic firings. The targeted architecture might not be able to support this amount of parallelism. By transforming CSDF into SDF we can reduce that parallelism. In [27] it is explained how this can be done.

3.2.3 KPN

Kahn Process Networks (KPN) is another MoC developed by G. Kahn [17]. Just as with SDF FIFOs channels are unbounded, the writing to channels is non-blocking and the reading from channels is blocking, but actors have no firing rules. KPNs can be seen as a generalisation of SDFs. In other words SDF is a subset of KPN. KPNs lack the properties worst-case execution time and firing rate for actors, which are called processes. It does not define channel rates. KPN can model dynamic behaviour but evaluating their correctness and performance is in general undecidable.

3.2.4 SADF

While SDF lacks in expressing dynamic behaviour but is strong in analysis of performance and correctness properties, the opposite is true for KPN. SADF [34] is a method to support dynamic behaviour within the framework of SDF. It uses the concept of scenarios to allow a process to have varying execution times. The terminology follows SDFs except for some modifications for the scenarios. Channels can have rates of zero to model that they are inactive in certain scenarios. It uses a stochastic approach to model the scenarios while keeping the strong analysis of correctness and performance of SDF.

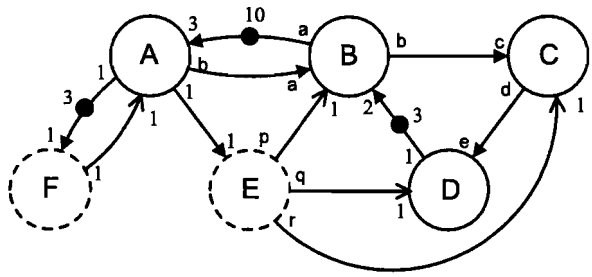


Figure 3.4: Scenario-aware data flow example.

Figure 3.4 shows a typical SADF graph. There are two types of processes, kernels and detectors. *Kernels* represent the data processing part of the streaming application. *Detectors* model the control part. In the figure actors A, B, C and D are of the type kernel and E and F are of the type detector. The detectors are connected to kernels by means of control channels, indicated with open arrowheads. The detectors send on every firing a control token to the kernel actors. The kernel actors decide then based on that control token how to behave that firing. When the kernel actor A has fired it produces a token which is sent to the detector E which then uses a stochastic model to determine the next control token.

3.3 SDF Generation

In this section I define what we should expect as SDF given a particular source code input. Note that initial tokens are not included until section 3.3.6.

3.3.1 Program Analysis

In order to derive an SDF graph from an application, we first need to analyse that application. In general we can identify two types of applications and two methods for analysing an application.

Static and Dynamic Programs

Programs can be categorised in several ways. For this work the Static and Dynamic categories are most important. A *static program* always executes in the same way. Every statement in the program will be executed equally often for every execution. On the other hand a *dynamic program* will have variation in these numbers. This is because dynamic programs have conditions depending on the value of input programs. For example a condition depends on the value of a pixel in an image processing application. If the pixel value is higher than a threshold a certain part of the code is executed. It is obvious that different iterations of the program on different input data sets will lead to a different execution pattern. Static program lack any dependencies on input data and always perform identical. Static programs tend to be more easily analysable than dynamic programs.

Static and Dynamic Program Analysis

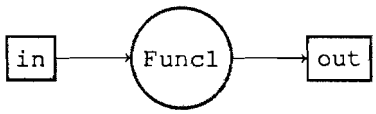
When analysing a computer program, there are two main approaches. The first *Static Program (or Code) Analysis* attempts to determine the behaviour of an application by analysing the source code or object code. The latter is a compiled version of the source code containing machine code but also comments, variable names and stack information among others. Static Code Analysis often make use of formal methods depending heavily on mathematical techniques to extract the behaviour of a program. Alternatively one can use *Dynamic Program Analysis* to analyse a program when it is actually being executed. Another more common name for Dynamic Program Analysis is *Profiling*. It measures the behaviour of a program while it executes. Metrics measured include execution time of functions, the number of function calls and memory use. Typically profiling tools generate a call graph to present their results.

3.3.2 Functions

First we'll discuss functions and how to extract an SDF actor out of them. A function or subroutine gets usually defined as a piece of code that performs a certain task with some degree of independence. Functions make suitable actors because it is clearly defined what is consumed and produced. We define every parameter in the function call which is read inside as an incoming token. Every parameter written to (inside the function) is considered an outgoing token. The following function illustrates this idea:

```
int Func1 (int * in, int * out)
```

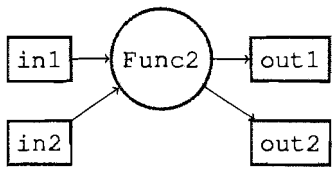
It is assumed that some statement in the function body reads from the parameter `in` and writes to `out`. The parameters `in` and `out` do not describe a single data unit, but usually stand for an array of data units. When deriving the SDF for the function, we equate every execution of the function with a firing of the corresponding actor. The execution of a function leads to a fixed amount of writes and reads, just as the firing of an actor leads to a fixed amount of token consumption and token production. Functions which cannot guarantee that these amounts are equal for every execution cannot be represented in SDF.
The derived SDF graph is shown in the figure below. The derivation is very straightforward in this simple case. Note that in and out are not actors in the SDF but just labels for the channels.



In the previous case we did not discuss multiple input or output parameters. The following example will explain it. We take the function

```
int Func2 (int * in1, int * in2, int * out1, int * out2)
```

which, similar as in the first example, leads to the following SDF graph:

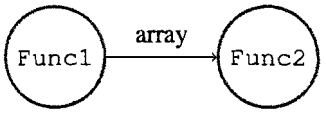


An obvious next step is to see what happens when we have more than one function where one reads data generated by the other. We define a producing function `int Func1 (int * out)` and a consuming function `int Func2 (int * in)`. For data transfer we also define an array `int array[10]`. The size of 10 is arbitrarily chosen and it has no influence on the SDF extraction, even not for calculating the channel rates. The following C code illustrates how it all connects together:

```
int Func1 (int * out);
int Func2 (int * in);

int main (void)
{
   int array[10];
```

```
    Func1(array);
    Func2(array);

}
```

which gives this SDF graph (without channel rates):

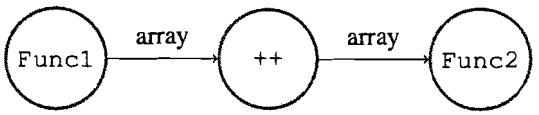


3.3.3 Statements

Functions seem decent basic units for deriving SDFGs, however a program does not only contain functions. A requirement that a program is designed in such way that all processing has been concentrated in functions, might be too much for most programs and would limit the input applications we can process. The problem is illustrated by the following code:

```
int Func1 (int * out);
int Func2 (int * in);

int main (void)
{
    int array[1];

    Func1(array);
    array[0]++;
    Func2(array);

}
```

When only taking functions into account we would miss the actor for incrementing in the SDF. The statement between the two function calls should also be represented in the SDF. The result should look as follows:



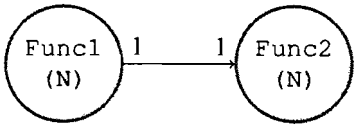
3.3.4 Loops

In the previous sections we have seen how to deal with statements and functions. In DSP applications a very common construct is the loop. I will explain what we should expect from loops for SDFGs starting with a simple 1-dimensional loop.

Single loops

```
int Func1 (int * out);
int Func2 (int * in);

int main (void)
{
   int array[N];

   for (int i=0;i<N;i++)
   {
      Func1(&array[i]);
      Func2(&array[i]);
   }
}
```

Here we face loops for the first time. Actors fire according to a schedule that has to be valid. In the SDF graphs we add a number for this firing rate. In this example it is obvious that both functions are executed N times. But if we look at the amount of data that is transferred each firing of the actors then 1 data token (in this case with the size of an integer) is communicated between the two actors. The data rates of the channel are both 1, 1 token produced and consumed on this channel for every firing of the respective actors. Important to note is that we equate one iteration of the SDF with one execution of the program, or more specific, of `main`. This leads to the following SDFG:



It is silently assumed that the data rates are constant, which is a requirement for SDF. The amount of data a function consumes can vary depending on the control flow inside the function. This problem will be discussed in Section 3.3.5.

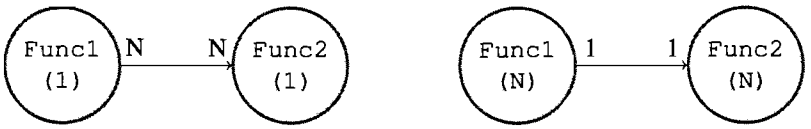
Double loops

In the this example I put each function in its own loop. On first sight the difference with the previous example seems to be rather small. This example seems to contain much less parallelism because the first loop needs to finish before the second can start. But if we map this to an SDF then the second actor can actually fire when it receives the required tokens on its input. Thus the only difference with the previous example is that this one requires in the source code a larger buffer, but in fact it takes exactly the same amount of space. The situation is somewhat ambiguous, because we could say the first actor (representing $Func1$) fires once and outputs then N tokens. The second consumes then N tokens in a single firing. On the other hand we could put the firing rate of both actors to N and the rates to 1 token. The following example illustrates the issue.

```
int Func1 (int * out);
```

```
int Func2 (int * in);

int main (void)
{
   int array[N];

   for (int i=0;i<N;i++)
   {
      Func1(&array[i]);
   }

   for (int i=0;i<N;i++)
   {
      Func2(&array[i]);
   }
}
```

which can lead to the following two SDFGs:



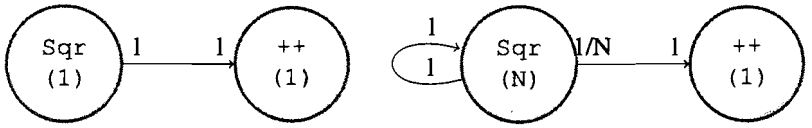
Loops with cross-iteration dependencies

The ambiguity of the previous example requires more investigation. Although until now it was not a major problem to generate a correct SDFG but now we will analyse an example for which it is a problem. I take a function called `Sqr` which takes the square of the input and writes it back to that same input pointer. This function gets called N times with an integer `a` and after the loop the value of `a` is incremented by 1. If we would have a firing rate of N for the actor we would cause a problem with the channel to the incremental actor, because that actor fires once and consumes 1 token. Actor `Sqr` also puts one token into this channel at the end of the loop. The final value of `a` of the N square operations is offered to the incremental statement. If we want one token from actor `Sqr` we have to set the firing rate to one as well, which then influences the self-edge of actor `Sqr`. Self-edges usually represent data dependencies over iteration borders. It transfers data from a previous iteration to the current. In this case it holds the value of a calculated in the previous loop iteration. In this case we can choose 2 approaches again. Either we consider the `for`-loop including its body an actor with firing rate 1 and thus we should have no self-loops, or we consider the function an actor and end up with an actor with an firing rate of N and a self-loop with rates 1.

```
void Sqr (int * in)
{
    *in = (*in)*(*in);
}

int main (void)
{
```

```
   int a=0;

   for (int i=0;i<N;i++)
   {
      Sqr(&a);
   }

   a++;
}
```

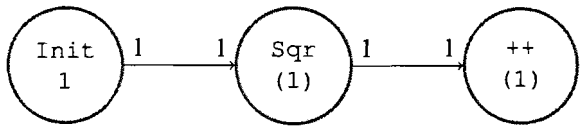
which leads to the following 2 possible SDFGs:



Note that I left out the initial value of the variable a. For correct representation we would have to add an actor that takes care of the initial value of a (like a value of 2). In the next example I add the initialisation of the variable. I will explain the $1/N$ notation further on in this chapter.

```
void Sqr (int * in)
{
    *in = (*in)*(*in);
}

int main (arg)
{
   int a = arg;

   for (int i=0;i<N;i++)
   {
      Sqr(&a);
   }

   a++;
}
```

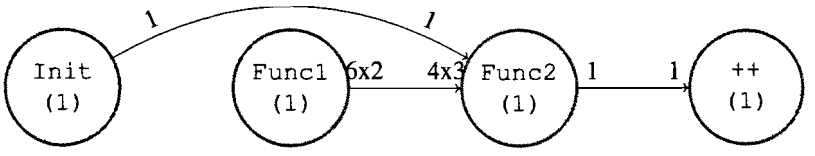
The first actor called $Init$ sets the value of a to the value of the main input argument. Between $Init$ and Sqr in the loop only a single time data is transferred. We should set both channel rates to one. The rest of the SDFG should not change as can be seen in the following figure.



Transferring data into loops

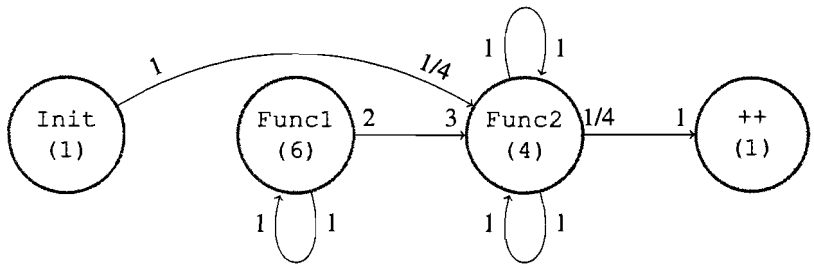
```
int Func1 (int * in)
{
   static i;

   in[i++] = rand();
   in[i++] = rand();

}

int Func2 (int *in, int * out)
{
   static i;

   *out = *out + in[i] + in[i+1] + in[i+2];
}

int main (void)
{
   int a[];
   int sum=0;

   for (int i=0;i<6;i++)
   {
      Func1(a);
   }

   for (int i=0;i<4;i++)
   {
      Func2(a, &sum);
   }
   sum++;
}
```

This example is considerably more complex and we end up with 4 actors. The buffer a has no defined size and there is a variable called sum which is set to zero in the first actor. The function $Func1$ fills the buffer every iteration with two values and the function $Func2$ reads three values every iteration and additionally it updates a running sum called sum. Each function gets its own actor and the fourth actor takes care of the final `sum++` statement. The resulting SDFG:



Again I start at the end with building up the SDFG. The final actor is fired once, consuming one token. That implies that Actor $Func3$ should produce 1 token as well. Since that function is executed 4 times, it would have to produce and consume 4 tokens on its self-edge. Sum is being read once as well so another channel comes in from

the $Init$ actor with rate 1. It also consumes 3 tokens per iteration for the value of a. Actor $Func1$ produces 2 tokens each of its 6 iteration. The self-edge for actor $Func1$ represents the variable i; Although this SDF seems correct, another problem surfaces here. There is hardly any parallelism in the SDF. That is especially visible for actors $Func1$ and $Func2$, where $Func2$ can only fire when actor $Func1$ has finished. If we take a look at the source code, it is not difficult to see that $Func1$ doesn't need to finish for $Func2$ to start. After two firings of actor $Func1$ should actor $Func2$ be able to start since it needs 3 tokens. That expression of parallelism has been lost in this representation. A much more parallel model would be the following:



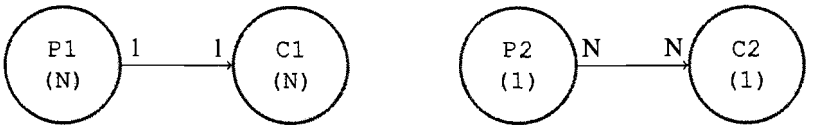
The new SDFG offers a much better expression of the parallelism between actors $Func1$ and $Func2$. Unfortunately I had to introduce the fractional data rates. These are defined as a token that is either only consumed in the first firing or only produced in the last firing. This makes the SDFG almost useless for automatic implementation and probably it is better to merge actor $Init$ into actor $Func2$. If I round the fraction up to 1 the SDF schedule becomes incorrect. Another option is to include an actor in between that generates (in this case) 3 additional dummy tokens. But in fact the piece of code we discuss is hardly worth the trouble because it's importance in computational time and firing rates is negligible. As previously mentioned SDF is more aimed at high-level modeling and not so much at these low-level details.

Data sequence

As I explained in section 3.1.2 not all communications between two actors follow the pattern of a FIFO. The following example shows source code in which that is not true.

```
int main (void)
{
   int a[N], b[N];

   for (int i=0;i<N;i++)
      Producer(a);

   for (int i=0;i<N;i++)
      Consumer(a);

   for (int i=0;i<N;i++)
      Producer(b);

   for (int i=N-1;i>=0;i--)
      Consumer(b);
}
```

Assuming that the consumer functions consume 1 token per iteration, we see that in the case of variable a some parallelism exists. The consumer function can actually start before the Producer function has finished. For the second variable b this cannot happen. Since Consumer starts with the last value of the array, it has to wait until Producer has finished completely. Thus there is no parallelism in the second part. When we look at a SDF representation of these examples, it turns out that they can have identical SDFGs, because SDF does not specify what a token contains. As we have seen before, the right SDFG in the figure below gives a valid representation of the first and second parts. However the left representation offers a much better parallel performance but is not valid for the second part. All this means that we have to test for this non-FIFO channel occurrence or requires a major manual rewrite of the source code.

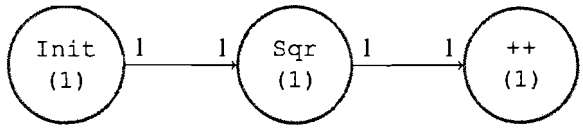


3.3.5 Conditional Statements

This sections explains how we should handle conditional statements, i.e. IF and CASE statements. An important property of SDFGs is that actors fire when there are sufficient tokens on their input channels and enough buffer space on their output channels. The firing schedule of the actors is fixed, in other words there is no conditionality for actors. Let us see what happens with the following source code:

```
int main (arg)
{
   int a = arg;

   if(a>10)
     Sqr(&a);

   a++;
}
```

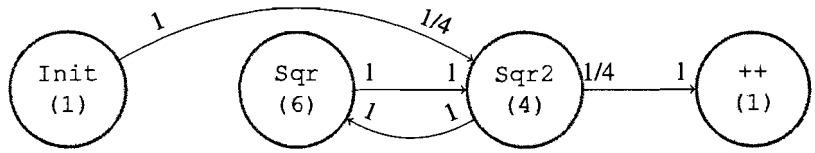
Here we can construct the SDF by taking one actor for the initialisation, one for the Sqr function and one for the increment. The question is what to do with the conditional statement? Whether the function is called depends on the value of the variable *a*. If the value is smaller than 10 the function won't be called. One possible solution is to bring the condition into the actor. The only difference between the two possible control flows would be a change of execution time for this actor. Fortunately actors have a property called *worst-case execution time* which should be set to the highest value. In this case that means setting it to the time that Sqr need to execute plus the time to check the condition. Obviously this procedure leads to inaccurate modelling and eventually an inefficient use of resources.



In the previous example we considered a condition which tests against a data value. Another situation are conditions testing against index values. Consider the following example:

```
int main (arg)
{
   int a = arg;

   for(i=0;i<N;i++)
   {
     if(i>N/3)
       Sqr(&a);
     Sqr2(&a);
   }
   a++;

}
```

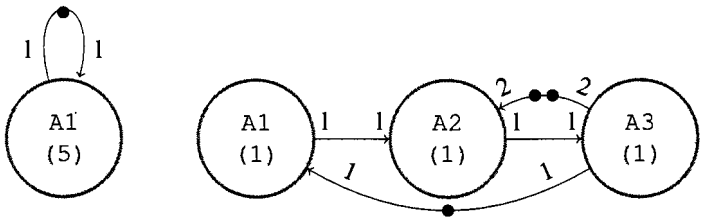
In this example the functions `Sqr` and `Sqr2` are identical, the names only differ for clarity. We consider the conditional statement part of actor Sqr. I use the $1/N$ notation for ease of construction. We end up with the following SDFG:



Also in this case the SDFG can be generated as long as we keep conditions inside an actor, which leads to more data communication in the system. If a condition happens in the highest level it eliminates possibly a lot of parallelism. After all we have to include all statements under this condition into the actor. In an extreme case that would merge the whole program into a single actor. Unfortunately this behaviour is inherent to SDF.

3.3.6 Initial Tokens

In previous sections I did not explicitly mention where initial tokens have to be put in an SDFG. Basically we have to put an initial token when we want to model a delay or a dependency on a previous iteration. The consumption rate of the channel determines how many tokens are required on a channel. The following example shows some cases.



3.3.7 Rates

In this section I will explain how we can derive the channel rates from source code. If we look at the source code of an application and we split that code into groups of statements, then it is not hard to see that parts of that application communicate data with other parts. This data traffic can be measured during profiling and is basically accounting of reads and writes. Next to how often a particular memory location has been written or read, we also need to know how often that particular piece of code has been executed, since we cannot know beforehand how often a program is called for a certain input file. For example, for an MP3-decoder the times that the actual decode function is called depends heavily on the size of the input file, i.e. it might be called once for every frame. A simple division would then return the actual rates for a single firing/execution of the actor/function. However we need to check if rates are constant. As we have seen conditions checking on data variables instead if index variables cause problems in this regard. One has to count all data going into the condition and its body.

Since the data unit of an SDFG is the token and software typically uses bytes as units, we should define the token size in bytes. Most programs use a mix of bytes, words and long words variables (8,16,32 bytes). That means we have to watch carefully what unit is used for the tokens. Note that often in SDFGs tokens can represent frames or other large quantities of data.

3.3.8 Worst-case Execution Time

Determining the worst-case execution time can be done by using profiling information. Typically the execution time spend in a particular part of the application is shown as a percentage of the total. From that information we can calculate the worst-case execution time for each part in time units. Of course one needs to know how long it takes to execute the application once.

3.3.9 SDF Verification

Finally after generating an SDFG, we need to verify if the result is equivalent in functionality to the original. This can be done in multiple ways, but the most straightforward is running both the original and the parallelised version on the same input data files, for example an MP3 file for an MP decoder or a picture for a Sobel filter. That would require implementation of both on the same platform. Disadvantage of this approach is the considerable amount of work to get both running. A more simple approach involves calculating if the SDFG has a valid schedule. In this chapter we have seen how to verify a schedule of an SDFG. If the schedule turns out to be valid it gives a strong indication that the result is correct. If the designer has good knowledge of the

applications involved he can try to judge the correctness by hand. Obviously that is not the optimal solution we should aim for.

Chapter 4

Parallelisation engines

This chapter covers similar work done on parallelisation and design exploration tools. The first tool to discuss is FP-MAP [18] by Ireneusz Karkowski and Henk Corporaal of Delft University of Technology. Increasing the amount of instruction level parallelism does not necessarily lead to a higher performance, but it probably leads to high costs because of inefficient use of the hardware. The highly complimentary functional pipelining approach they propose maps embedded programs written in ANSI C onto a pipeline of application specific processors. Their algorithm has a low computational complexity and aims to be a parallelisation engine for a (semi)-automatic system for multi-processor embedded system design. Section 4.1 covers FP-MAP in detail.

Second, I go into details about PN-GEN [35] by Sven Verdoolaege, Hristo Nikolov and Todor Stefanov from LIACS. Continuing the research done for the Compaan project [20] they present compiler techniques for extracting the parallelism from sequential applications. For expressing the parallelism they use the process model network model of computation. In particular the MoC is a Kahn Process Network (KPN). After generating a KPN graph they try to optimise the KPN for improved performance. Finally they calculate the minimal buffer sizes for the FIFOs on the channels at compile-time. PN-GEN is explained in Section 4.2.

The output of the previous tools is mainly a mapping to a multi-processor system. The third work SPRINT [4], by Johan Cockx, Kristof Denolf, Bart Vanhoof and Richard Stahl from IMEC in Belgium, generates a hardware description in SystemC as output. Another difference is that it relies on manual instead of automatic partitioning. The user is expected to define the parallel parts by adding user directives to the source code. SPRINT uses these directives to extract the partitions and static analysis to derive the SystemC description. SPRINT is covered in Section 4.3.

4.1 FP-MAP

Research with trace analysis [36] has shown that the theoretical upper bound for parallelism in a program lies around 60, split between coarse and fine grain parallelism. With Instruction Level Parallelism alone we can reach a speed-up in the order of 2-5. Higher speed-ups are hindered by limits in the area of software parallelization techniques and in the architecture of the modern ILP processors like superscalar and VLIW. Many methods for ILP exploitation have been proposed in the past, mainly applied to loops. The most popular method for the latter is software pipelining [28] but it has

some limitations. These include difficulties scheduling loops high in the call tree of the program and loops containing control constructs. Some approaches proposed are Hierarchical Reduction [22], All Paths Pipelining [33] and Enhanced Pipeline Scheduling [9] [8]. Another problem forms loops and function calls in the loop's body. Although loop unrolling and function inlining can help in certain cases, there is no universal solution. On the downside, often the code size can expand substantially, which is a major drawback in embedded systems. To get a high level parallelism all possible control paths have to be scheduled separately, which leads to even larger code sizes. For some loops predicting the loop counts at compile time is difficult or even impossible. Finally the hardware causes some problems because the interconnection network of the known ILP processors becomes rapidly more complex with the number of available FUs. All these problems mentioned, lead to a limit on ILP exploitation. To reach a higher level of parallelisation we need to combine fine- and coarse-grain parallelism. Since chip area is no longer the most important limitation on design, embedded multi-processors have become a feasible and attractive option. We can gain even more by using hetergeneous processors optimised for the particular tasks they have to perform. This allows fine- and coarse-grain parallelism to be exploited efficiently. To exploit coarse-grain parallelism an algorithm is needed to partition the application onto different multiprocessors architectures. Previous solutions were based on a shared memory approach which leads to high hardware demands and a complex memory system. Also searching for data parallelism only in loops might be a too limited approach. There is a need for a coarse-grain parallelism exploitation method to use in simple multi-processor architectures. FP-MAP is a tool that implements a parallelisation technique called functional pipelining, which maps a selected loop nest in a sequential program onto a pipeline of processors. The result is a pipelined style of execution. It takes a selected loop nest in a ANSI C program and maps it to a pipeline of application specific processors. It is especially aimed at loop nests high in the call hierarchy of a program and containing a variety of control statements.

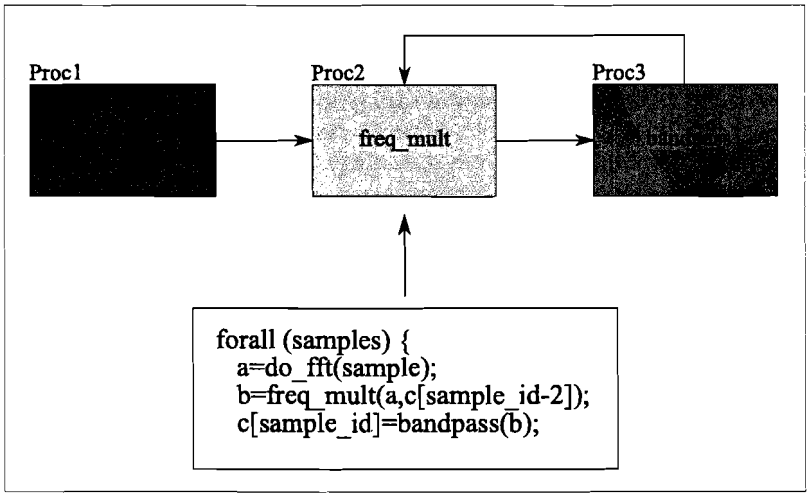


Figure 4.1: Example of mapping code to processors

The following example illustrates the method of FP-MAP. The code of figure 4.1 is

mapped to the 3 processor system in the same figure. The *do_fft* function maps to the first processor in the chain, the *freq_mult* to the second one and *bandpass* to the third one. Communication channels are put in place when there is a dependency between two functions executing on different processors (as is the case for every function here). The method consists of 3 steps.

1. Build a dependency graph $G_{LR}(V, E)$.
2. Perform the mapping.
3. Estimate the speedup.

Figure 4.2 shows a graphical overview of the method.

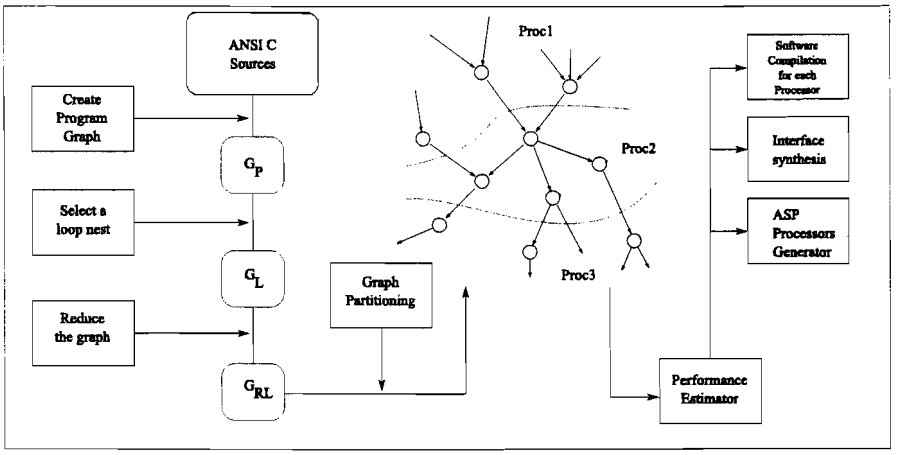


Figure 4.2: Overview of the FP-MAP method

The dependency graph $G_{LR}(V, E)$ is a reduced directed graph, with V the set of vertices and E the set of edges. It is called reduced because it does not equal every operation in the code with a single vertex and unimportant operations are left out of the graph. Every vertex has a number of properties, the time necessary to perform the operation(s) modeled by the vertex (in clock cycles), the probability that a given operation(s) will be executed during a single iteration, id of the conditional statement to which the vertex belongs, the value of the conditional statement test expression, which selects the branch where the vertex belongs and finally a variable that is true when the vertex is on the longest conditional branch. There is a single property for an edge, namely the iteration distance vector. To create the vertices they follow a set of rules. All loop and call statements are mapped to single vertices with the latency set to the maximum time of an execution of the loop. For every conditional statement a test vertex is created with latency set to the time needed to test the condition. The remaining statements are mapped each to a vertex with latency set by the most critical path in the instruction level data dependency graph. Every vertex is assigned the execution probability as measured by profiling. An edge is added for every data or control dependency. Vertices are considered atomic operations. More optimisations are made by removing vertices that have a low execution probability or small latency. This is done to simplify the scheduling. For executing the less common branches they stall the pipeline.

The mapping in FP-MAP is done such that a pipelined style of execution is obtained, while leading to maximal speed-up compared to the sequential case. This come down to minimising the *initiation interval* of the pipeline, which is equal to the time the most loaded processor needs to execute its task. Basically their algorithm attempts to spread the load as much out as possible over the available processors. Furthermore they take conditional statements into regard by forwarding these statements to follow-up processors while mapping. For the actual mapping they use a modified bin-packing algorithm with complexity $\mathcal{O}(|V||E|)$.

4.2 Compaan & PNgen

The motivation behind their research is the large focus on hardware multiprocessor architectures but neglect of compiler techniques to efficiently program these architectures. Especially the mapping of applications to multiprocessor systems needs more attention. Difficulties for programmers are mainly caused by the way applications are defined. The applications are usually defined as sequential programs using imperative languages such a C/C++ or Matlab. Although writing a program sequentially is convenient for the programmer, it does not reveal any parallelism. Harder for the programmer, easier for mapping to a multiprocessor system, is specifying an application using a parallel MoC. Mapping the application from such a MoC can be done by automatic mapping tools. The gap between a sequential program and a parallel model of computation was the motivation behind the research resulting in Compaan and its follow-up PNGEN. They chose the Kahn Process Network as parallel MoC because its operational semantics are simple but it is still very suitable for the stream-oriented data processing application domain they are aiming for. Additionally many researchers have shown that process networks can be mapped efficiently to multi-processor systems. They present in [35] compiler techniques for deriving process networks from a limited set of input applications. These must belong to the class of programs called static affine nested loop programs (SANLPs), mostly used in scientific, matrix computation and multimedia and adaptive signal processing applications. SANLPs can be represented by the polytope model from [10]. A SANLP has the following attributes (from [35]):

1. It consists of a set of statements, each possibly enclosed in a loop and/or guarded by conditions.
2. The loops need not be perfectly nested.
3. All lower and upper bounds of the loops as well as all expressions in conditions and array accesses can contain enclosing loop iterators and parameters as well as modulo and integer divisions, but no products of these elements. Such expressions are called quasi-affine. The parameters are symbolic constants, that is, their values may not change during the execution of the program fragment.

To convert the SANLPs to process networks they use the following steps:

1. Modified Dataflow Analysis

   They perform a data-flow analysis according to [11], but in order to reduce communication between different nodes they also take into account all previous reads

from the same statement. So the problem comes down to finding for a particular read from an array element the last write to or read from to that element. They use a technique called Parametric Integer Programming (PIP) to solve this problem.

2. Determine Channel Types

   In the next step PNGEN analyses which type of channel has been derived. Not all channels will be FIFOs. If the order in which a channel is written differs from which it is being read it is called a reordering channel. PNGEN checks for this condition by solving another PIP problem. In this process they also eliminate so called multiplicity from the channels. Multiplicity is the situation when the same data is read multiple times from the channel. They split the channel in that case.

3. Optimize Self Loops

   PNGEN analyzes the channels using PIP and where applicable it replaces a FIFO by either a shift register, a simple register and something called a "sticky FIFO". Elaborating on these replacements is beyond the scope of this text.

4. Computing Channel Sizes

   The final step in their flow consists of determining the optimal size for the remaining FIFOs.

5. The Output

   PNGEN is part of a larger tool set, which uses the output of PNGEN as input to the next tool in the flow called ESPAM [25]. Their output consists of a graph with the functions as vertices and dependencies between the vertices as edges. Every edges gets a number attached which represents the buffer size required for this channel. Figure 4.3 shows an example of the KPN output of PN-GEN. The 2-dimensional Discrete Wavelet Transform (DWT) application is divided into processes. Communication between those parts are translated into edges in the KPN. On the edges it outputs the calculated optimal buffer sizes for the channels. The partial source code is as follows:

```
for (i = 0; i < 2*Nrw; i++)
  for (j = 0; j < 2*Ncl; j++)
    image[i][j] = ReadImage();

for (i = 0; i < Nrw; i++)
  for (j = 0; j < 2*Ncl; j++) {
    tmpLine = (i==Nrw-1) ? image[2*i][j] : image[2*i+2][j];
    Hf[j] = high_flt_vert(image[2*i][j],image[2*i+1][j],tmpLine);
    tmp = (i==0) ? Hf[j] : oldHf[j];
    low_flt_vert(tmp,image[2*i][j],Hf[j],&oldHf[j],&Lf[j]);
}
```

From the source code we see that the 3 functions $ReadImage$, $high_flt_vert$ and low_flt_vert become processes in the KPN, but only the $image$ array becomes a channel. The buffer sizes are optimised for size and these sizes are dependent on the number of iterations of the loops. That makes sense because the more columns in the image the more we need to buffer. Self-channels represent inter-iteration buffering, i.e. data from a previous iteration is being re-used in the next.

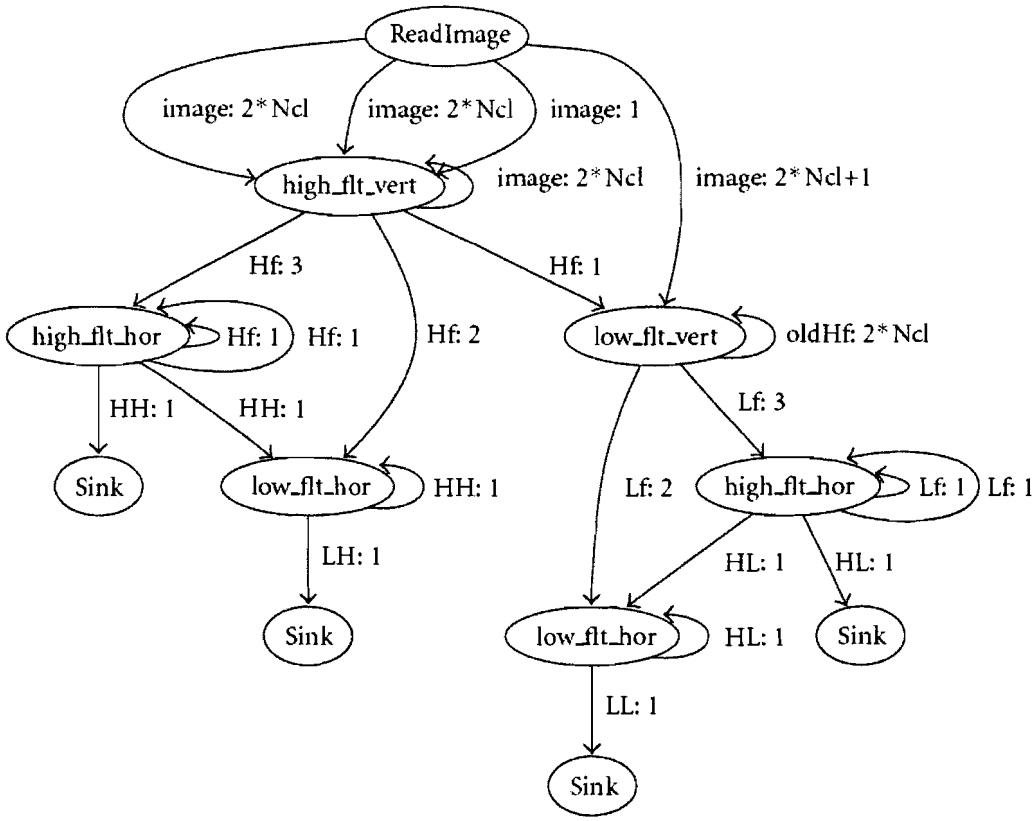


Figure 4.3: PN-GEN generated PN for an 2D-DWT example.

4.3 SPRINT

The third tool I discuss in this chapter is SPRINT [4] by Johan Cockx, Kristof Denolf, Bart Vanhoof and Richard Stahl of IMEC in Belgium. They also identify the conflicting constraints between high computational demands and minimal energy consumption. They propose a multiprocessor implementation as solution pointing out that multiple processors operating on a lower clock frequency provide the same performance as a single processor at a higher frequency, while consuming less energy. Further optimisations can be attained by using specialised processors for each task. They identify

two main challenges for efficient implementation. First, as with the other approaches, the parallel tasks must be identified and extracted for the sequential specification. The extracted tasks and available architecture must match as optimal as possible to reduce performance losses, decrease in resource utilisation and reduced energy efficiency. Second, a major part of the power is consumed by the bus/communication network and it is crucial to optimise this part. Exploring the various program partitions is currently one of the major challenges. For evaluation we need a concurrent executable model and they chose a register transfer level HDL model. This model is for implementation in custom hardware or for coding a programmable multiprocessor. Creating such a model is very time consuming to explore multiple alternatives. They mention attempts with transaction level modelling, but point out that they are still time consuming and error-prone. The solution they propose is SPRINT, a tool that automatically generates a SystemC model from sequential code and user-defined directives. To that model SPRINT can add derived delay estimates from the source code based on the number and types of operations. SPRINT is part of a C-based design flow targeting the implementation of advanced streaming applications on multiprocessor platforms. The capabilities of the flow include preprocessing and high-level optimizations applied to the C code before partitioning. Their tool is fast, validating the concurrent behaviour in less than 6 minutes including generation of the model. Figure 4.4 shows an overview of the SPRINT flow for a small example.

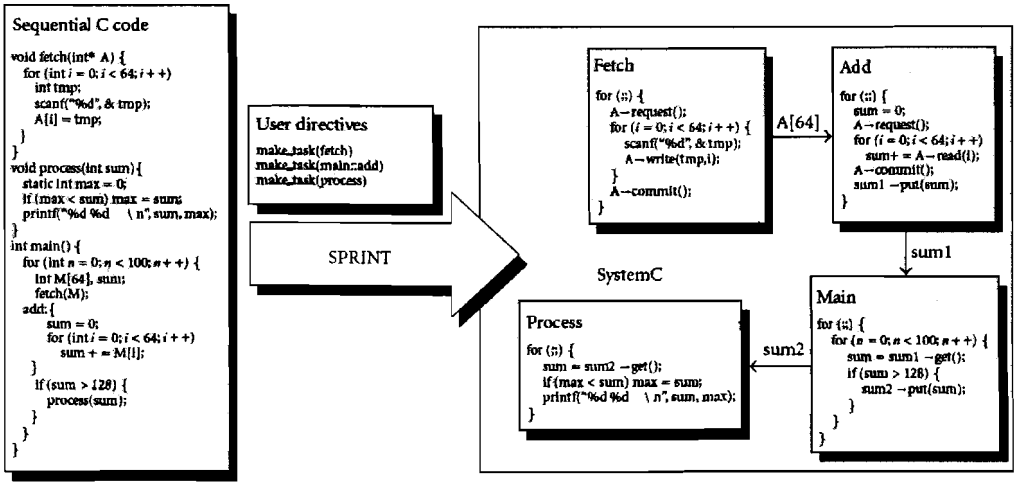


Figure 4.4: Example how SPRINT transforms the input code into parallel SystemC code.

SPRINT aims at multimedia applications for which standardisation bodies usually provide code. This code consists of a wide range of options, so called profiles and levels and it serves the goal of verification of a particular profile. It is therefore not very suitable for direct parallelisation and it would lead to an inefficient implementation. To solve this problem a systematic design approach is needed. The authors propose the approach of figure 4.5, which starts with high-level optimizations and parallelisation and ends with a structured verification.

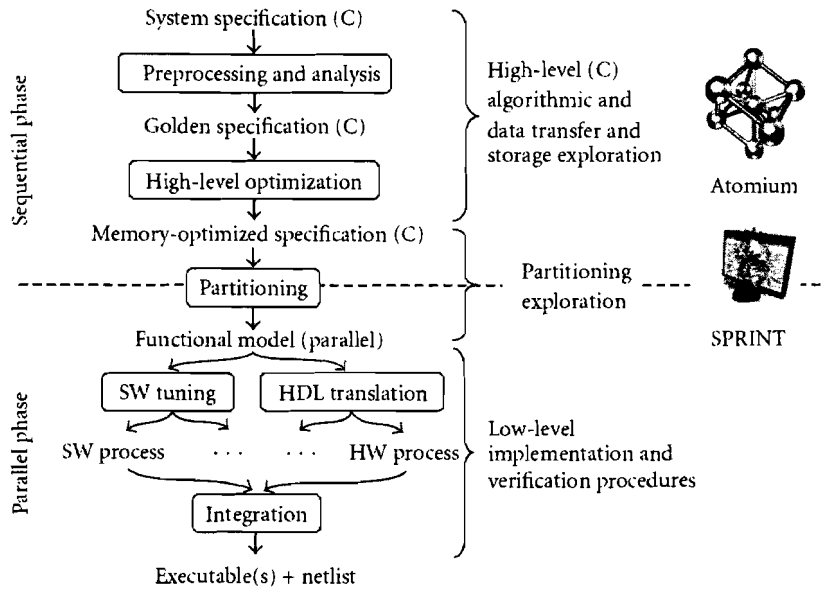


Figure 4.5: Example how SPRINT transforms the input code into parallel SystemC code.

Two major phases can be identified in the flow, a sequential phase and a parallel phase. The sequential phase takes care of the preprocessing, analysis and high-level optimisation of the initial specification; the parallel phase of the division of the application into parallel processes and refinement into register transfer level for hardware implementation or implementation code for software implementation. The whole flow is built around five steps.

1. Preprocessing and analysis removes all code which is irrelevant for the chosen profile. It also provides candidates for optimisation by discovering bottlenecks.
2. High-level optimization simplify input to minimise the data communication and memory requirements since they have a large impact on the energy consumption.
3. Partitioning splits the code and inserts the required communication channels. Throughput requirements, power requirements, design complexity and the exploitation of heterogeneous processors determine the result of this step.
4. Software tuning and HDL translation refines each task independently. Tasks are either mapped to processors or written as HDL for synthesis.
5. Integration gradually recombines the tasks to a complete system. Also it takes care of the testing.

SPRINT automates the generation of a concurrent model and provides the appropriate communication primitives for the designer-specified task boundaries. SPRINT

eliminates the need for manually derivation of the communication channels between the tasks. The speedup gained by this automation enables design space exploration.

A lot of research has been done on data parallelism, where loop iterations are divided over multiple processors. SPRINT on the other hand focuses on task-parallelism in order to exploit the efficiency gains of a heterogeneous system. Task-parallelism or functional parallelism leads naturally to pipelined execution and therefore we often call it functional pipelining. The tasks can be considered as pipeline stages communicating through FIFO-like channels. SPRINT does not cover fine-grained forms of functional parallelism such as instruction level parallelism (ILP).

After the designer adds the task boundaries, SPRINT inserts the necessary communication channels as detected. The tasks use a few well defined interface functions to communicate over the channels. The use of these functions allows the separate refinement of tasks and channels. There are two classes of channels: unidirectional point-to-point FIFO channels and shared data channels. FIFO channels are the default option and they are available in scalar and block form. The result with tasks connected by FIFOs can be interpreted as Kahn Process Networks (KPN). The behaviour is determined by outputs dependent on inputs and not on the relative order of execution. They mention that correct insertion of FIFO channels to be equivalent to the original code is non-trivial and requires complicated data-flow analysis. The shared data channel is inserted when the communication pattern cannot be covered by a equivalent FIFO. Detecting when a shared data channel can be replaced by a FIFO requires sophisticated program analysis techniques. For that reason SPRINT leaves it to the designer to explicitly add them. The resulting output can be tested by simulating the transaction level model. SPRINT completes the model by adding timing information. For computation delays in the tasks wait statements can be inserted into the SystemC code. The delay length depends on the execution time of adjacent statements on the target architecture or on a counting of the various types of statements. Alternatively the designer can add the delays based on profiling for existing or on experience for custom hardware. For the communication channel delay a generic implementation is used in the channel interface functions and can be easily replaced.

4.4 Data Analysis Tool

This sections introduces a tool called Data Analysis Tool (DAT) by Sean Rul [30] of Ghent University, Belgium. They start with identifying a major problem with automatic parallelisation, being the correct static analysis of both control flow and data flow. To bypass this problem they assume perfect knowledge of these flows by using dynamic analysis, profiling, instead. The measured control and data flow returns the opportunities for parallelisation, but it might be unsafe. Possible solutions they propose are thread-level speculation and designer validation for correctness. Central to their approach are memory-dependencies, since they impede parallelism. The granularity is largely set to fine-coarse because of this method. For their needs that is too small and they look for a method to combine smaller operations to thread-level partitions. The thread-level partitions should be in the order of one million instructions and functions offer for that a sufficiently detailed overview of the program. Profiling returns all functions and data structures used in a program. From that information it is possible to extract which functions communicate data to other functions. Finally from these dependencies they can identify the parallelism between the functions. They start by recording all function calls and returns, because these give a first restriction

on parallelism. The recording contains how often a function is called, the number of different functions it calls and the fraction of execution time it takes. From this data a *Call Graph* is constructed.

Function	Operation	Current producer
F1	store	??
F2	load	F1
F2	store	F1
F2	load	F2
F3	load	F2
F2	store	F2
F2	load	F2
F3	load	F2

Address 0x1000				
Producer	Times produced	Consumer		
		F1	F2	F3
F1	1			
F2	2	1	2	
F3			2	

Figure 4.6: Example of DAT memory structure for determining data dependencies. Source: [30]

The results of the dependency measurement are recorded in a matrix-like data structure, one for each memory address. $cell(i, j)$ shows how many times function i read a value produced by function j. Figure 4.6 gives the table (on the right) of the data structure for address $0x1000$ according to a write and read pattern (on the left). Functions that access this memory location end up in the columns Consumer and Producer. The number of times a function reads from the address will be put in the consumer column, how often it writes in the column called Times Produced.

DAT generates three graphs as output. These are the previously mentioned *Call Graph*, the *Interprocedural Data Flow graph* (IDFG) and the *Data Sharing graph* (DSG). Figure 4.7 shows the latter two. From information in a table similar to the table in Figure 4.6 the IDFG is constructed. This graph is used to cluster functions according to their communication pattern, where the nodes are the functions and the edges the data streams. The edges can be divided into inter-cluster data streams and intra-cluster data streams. Clusters are defined as sets of nodes which share a large amount of data among the nodes itself. Inter-cluster streams are edges between nodes in different clusters, and intra-cluster streams are edges between nodes in a single cluster. In Figure 4.7 the clusters are indicated by gray rectangles.

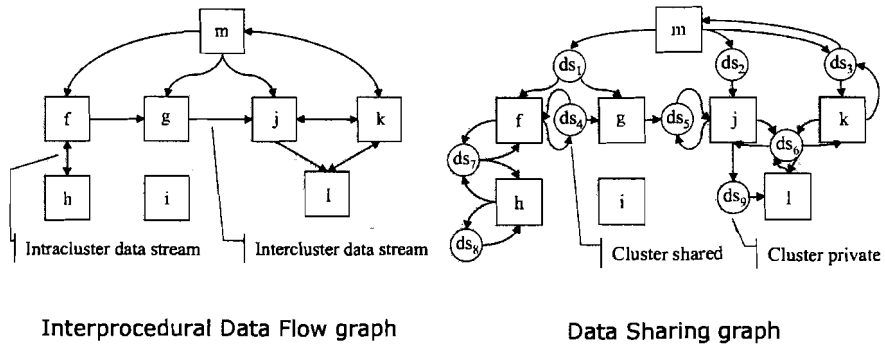


Figure 4.7: Examples of IDFG (left) and DSG (right). Rectangular nodes process data and circular nodes represent data. Source: [30]

The method followed to determine the clusters is to limit the bi-directional inter-

cluster data streams. Functions that show a high level of bi-directional data traffic are merged into the same cluster. Only when both functions take a high fraction of the total execution time of the program, they should be split. The Call graph takes a large part in determining the IDFG. The third graph, the Data Sharing graph provides next to the functions also the data that is communicated between these functions. Is contains two kinds of nodes, the already mentioned function nodes and the new data-nodes. Edges from a function node to a data node indicates a write access, an edge from a data node to a function node a read access. Memory address used by the same functions are merged into a single data node to reduce the amount of information. Four types of data usage can be identified, a consumer is a function that reads data written by another function, a producer writes the data to be read by another function, private consumption happens when a function reads its own previously written data and finally data without source is considered a constant. The resulting data sharing graph as in figure 4.7 on the right side, contains rectangular nodes for functions and elliptic nodes for data structures. Functions that communicate by passing parameters through the stack or registers have no connections to the rest of the graph.

The actual parallelisation involves defining and marking parts of the program code depending on the results of the previous analysis. The memory addresses are mapped on the corresponding data structures from the original program. For static structures it is trivial, but for dynamic structures it requires recording of the addresses during profiling. The next step is specifying the shared data structures to limit the amount of communication and sequentiality. These shared structures require locking to avoid data races. Possibly multiple instances can be created for this structure to bypass the problem. Finally the initialisation and start-up code for the threads is generated to preserve correct execution of the program and new data structures are introduced for communication between threads.

4.5 Parallelisation Extraction

In the previous section we have seen a number of available parallelisation tools. All of these were considered for forming the core of the work in this thesis. I will start with an overview of the requirements for the parallelisation part. After that I will compare the various tools and explain finally the choice for DAT as parallelisation engine.

4.5.1 Requirements

I have mentioned the various properties and parts of SDF graphs, i.e. actors, channel, channel rates and worst-case execution times. When deriving these one by one we can best start with the actors and channels, because they form the basic components. Actors are atomic operations for which the number of input and output channels is constant for each firing. When we look at C-source code we can investigate a number of candidates on different levels of granularity. There are instructions, statements, functions and loops and combinations of these, complicated by control-flow statements. The parallelisation engine should provide a set of atomic parallel executable operations. A fine-grained granularity is preferable here because it allows better mapping to a target architecture later on. SDF cannot handle control-flows very well because it is aimed at modelling DSP applications where control-flow is less important. SDF is particularly aimed at modelling high-level behaviour, as seen in the H.263 example (figure 3.2), that only contains tokens for handling the video information. Based on source code

the resulting SDFs will hardly be so streamlined, because typical DSP application will contain some degree of control-flow. Furthermore we must find a method to translate the iterations of a program to an firing of the complete SDF. DSP Applications often have a main loop whose parts should roughly correspond to the actors in the SDF. As we have seen with the SPRINT flow, they have a separate tool to clean the code from unnecessary code to avoid inclusion in the end result. After we have derived the basic structure of the graph, we should fill in the details. Either by static analysis or profiling we should derive numbers for the channel rates and worst-case execution time. Especially the latter seems suitable for profiling. In the next sections I recall the previously discussed parallelisation engines and argue why DAT is most suitable for our requirements.

4.5.2 Comparison and reasons for choosing DAT

The three tools considered for using as parallelisation engine in this work are FP-MAP, PN-GEN and DAT. Not only scientific reasons were involved in the choice but also practical issues like availability and its suitable for the range of applications we had in mind. Preferably the tool would support a wide range of input applications, so we could abstract several issues later in the method. Next to that input criterion, the generated output is also very important. We need output that can easily be converted into an SDF graph, thus it preferably has to contain all information we need to derive the properties. Where the generation of actors and channels should be obvious for the tool, it should also provide the data we need for properties like the channel rates and execution times. I will now give an overview of how the tools score on suitability. This only covers FP-MAP, DAT and PNGEN because they were the only ones that were available to various degrees. Table 4.1 gives an overview of the various properties of the tools discussed and Figure 4.8 provides a graphical overview.

As introduced in this chapter FP-MAP is a tool that implements a parallelisation technique called functional pipelining, which maps a selected loop nest in a sequential program onto a pipeline of processors. The result is a pipelined style of execution. It takes a selected loop nest in a ANSI C program and maps it to a pipeline of application specific processors. It is especially aimed at loop nests high in the call hierarchy of a program and containing a variety of control statements. FP-MAP seemed very promising in providing the necessary functionality but unfortunately it was not in working order. In fact it became never available. Another problem is the fact that it was written a decade ago, probably causing problems with supporting tools.

PNGEN and the commercial COMPAAN are tools for deriving process networks from a limited set of input applications. These must belong to the class of programs called static affine nested loop programs (SANLPs), mostly used in scientific, matrix computation and multimedia and adaptive signal processing applications. The output consists of a Kahn Process Network expressed in YAPI [6] format. The considerable source code of PNGEN was available to me for evaluation. Since KPN is similar to SDF it seemed a promising candidate. However PNGEN does not have a profiling part, which would require a separate effort to get profiling results and cause difficulties in matching the two unrelated approaches. The main problem with PNGEN is that it requires the user/designer to define which functions to parallelise. PNGEN based on that definition generates the KPN. Additionally it doesn't take sub-functions into account, because it only accepts defined functions on the highest level in main. These two properties make it not very suitable to parallelise applications from random sources, because usually source code has been divided up into different source code file (as this

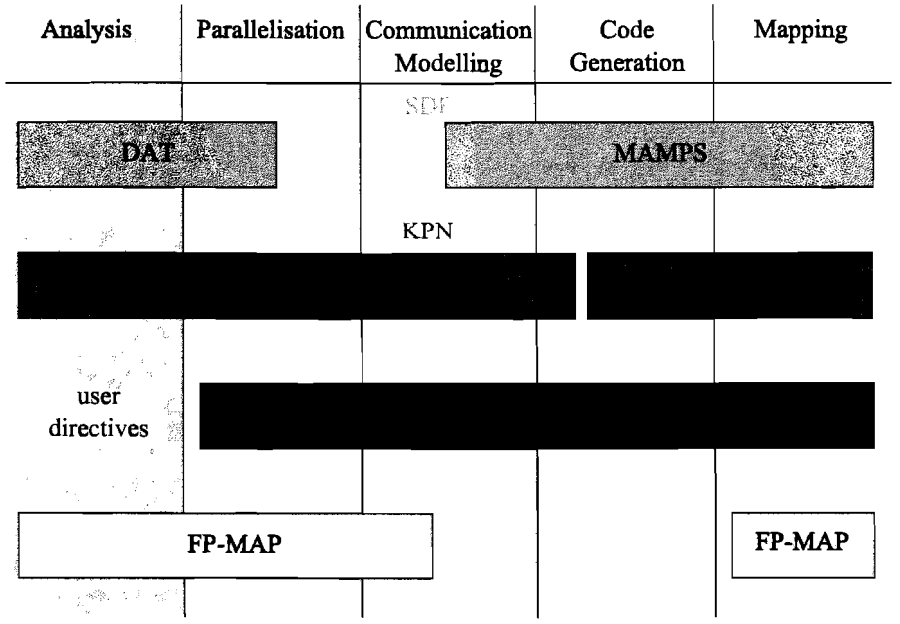


Figure 4.8: Graphical overview of the parallelisation engines. Note that the tools do not perform exactly the same task in each part. On the left C-code enters the flow.

is good coding practice). It would require extensive modifications to the code to fit it into the SANLP frame.

	Input	Partitioning	Data-flow analysis	Output
PN-GEN	ANLP	Automatic	Static	array YAPI KPN
FP-MAP	C	Automatic	Dynamic	None
SPRINT	C	User-defined	Static, scalar	SystemC TLM
DAT	C	Automatic[1]	Dynamic[2]	Dataflow graphs

Table 4.1: Summary of Design Tools. [1] Provides basic information. [2] Incomplete, it does not handle inter-iteration distance.

Finally DAT seemed also very promising, but also not without problems. Although it took a long time to get hold of DAT because it was not as ready as the authors claimed in their paper, it returned good results regarding the acceptance of input code. In fact it accepts everything that can be compiled by GCC. The automatic clustering was initially not available but in a later update it was, but it turned out to be less useful than as previously thought. DAT can generate all information we need for constructing the SDFG. The DSG can be used to derive the actors and channels, the same graph can also be used to calculate the channel rates and the call graph can be used to determine the worst-case execution time as I will show in the next chapter. The main problem with DAT was that it used functions as the basic unit for granularity. That would not cover all code in a program. After some minor modifications it was able to use statements as level of granularity. Overall DAT gives the best result considering the requirements discussed in section 4.5.1.

Chapter 5

Multi-Application Multi-Processor Synthesis

This chapter starts with a short introduction to Design Space Exploration. It continues with a description of the MAMPS framework. Finally in Section 5.3 I will give an overview of the modifications I made to MAMPS in order to interface it to the new front-end. This front-end will be discussed in the next chapter.

5.1 Design Space Exploration

In previous sections we have seen that we have to consider numerous aspects in the design process of an embedded system. Not only technical issues like energy consumption, computational power and flexibility but also commercial issues like resource constraints, marketing and product feature decisions are involved in the design process. All these variables leads to a complex multi-dimensional design space. Evaluation by hand is very time-consuming and error-prone for large complex systems and thus there is a clear need for an automated methodology. In order to evaluate the available options in a structured way we use an approach called *Design Space Exploration* (DSE). The Multi-Application Multi-Processor Synthesis [21] (MAMPS) framework is such an automated approach and provides the base of the work outlined in this thesis. I will discuss MAMPS in Section 5.2. In this section I will continue explaining a particular DSE approach called the Y-chart approach (Figure 5.1) as presented in [19].

When designing an embedded system the designer has to take a lot of variables into regard. Many of these variables will influence the performance of the system and utilisation of resources. If we have to map multiple applications the process is further complicated, since certain choices might work out bad for one application but good for another. The basic goal for the designer is to attain sufficient performance of the system for an acceptable price. This involves trade-offs between various parameters of the design which have to be explored by the designer. Figure 5.2 shows a typical relationship between parameter and performance. The parameter can be regarded as the amount of allocated resources such as chip area. On the left side the performance gain is large, but overall unacceptable, on the right a large allocation leads to little gain, and in the middle both trends seem to give a reasonable result. The trade-off point is often called *knee*.

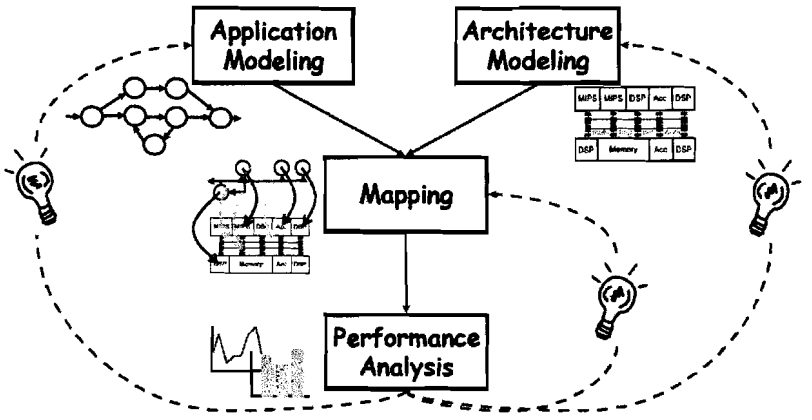


Figure 5.1: Y-chart for DSE

To assist the designer in evaluating all choices the design space exploration approach must provide an abstraction of current low-level design methods. The Y-chart approach models the architectures to retrieve a performance model. Evaluation of this model provides the various performance metrics of a processor. By changing the design choices in a systemic way the designer should be able to explore part of the design space, and in the process gaining the insight needed for making the right trade-offs. Result are attained efficiently by applying the performance at different levels of detail. By making the right choices at lower levels of detail, the design space at higher levels can be reduced and expensive large scale exploration is avoided. The Y-chart approach has been developed for programmable architectures and takes its three core issues into account: the architecture, the mapping and the set of applications.

5.2 MAMPS

In this section I discuss the MAMPS framework (MAMPS) [21]. As explained in this thesis, design space exploration has been proposed as a solution to reduce time-to-market by assisting in finding an optimal cost-efficient solution for embedded multi-processor designs. Hereby it attempts to meet constraints such as low-power consumption and low chip area. Because modern embedded systems have to support multiple applications the design problems extends to the performance evaluation of multiple use-cases. A use-case is defined as the combination of applications that are active in a set of all possible applications. In order to find a efficient solution we have to explore the hardware and software implementation of these use-cases. The shaded part of figure 5.3 shows the method to do that. Currently tools only support single applications and force designers to resort a manual approach for certain parts of the design.

MAMPS contributes to these tools by providing a design flow that generates a complete MPSoC design based on the input specification. The resulting design can be readily synthesised for use in an Xilinx FPGA. The specification is in the form of Synchronous Data Flow (SDF) which, as explained in section 3.2.1, is very suitable for modelling DSP applications. MAMPS ensures concurrent execution of applications first by using non-blocking reads and writes to avoid deadlocks, second by including

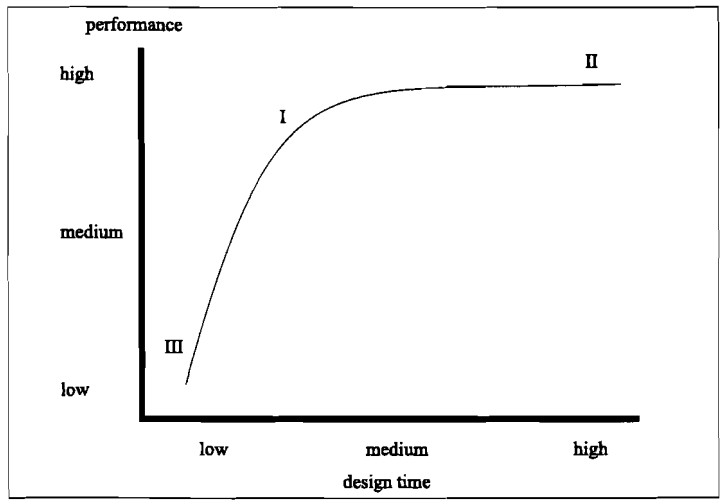


Figure 5.2: A typical idealised relationship between performance and value of a particular parameter, in this case design time. Point I indicates the best trade-off, Point II gives little gain for a large value increase, and in Point III the performance is not acceptable. Source: [19]

an arbiter to enforce fairness and third by mapping channels to individual FIFOs to avoid head-of-line blocking.

Figure 5.4 shows a small example of a typical design flow in MAMPS. From the application description specified in SDF, MAMPS generates the hardware topology. It also generates the project files for the Xilinx EDK software. The example involves two input applications, Appl0 with four actors, and Appl1 with 3 actors. From there MAMPS generates the multi-processor system including the software. In the design, we find 4 processors with actors a0 and a1 sharing Proc0, while Proc3 only runs actor d0. The edges are mapped onto point-to-point connections with FIFO buffers. The specification of the applications in SDF are specified in XML format and can be derived from profiling or from the source code directly. The specification supports all properties of a typical SDF graph, such as actors and channels. MAMPS also allows

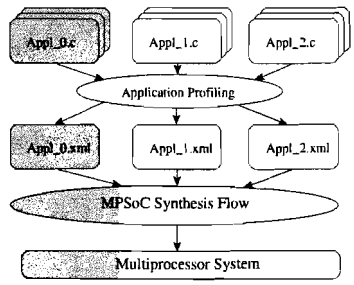


Figure 5.3: MAMPS overview. Source: [21]

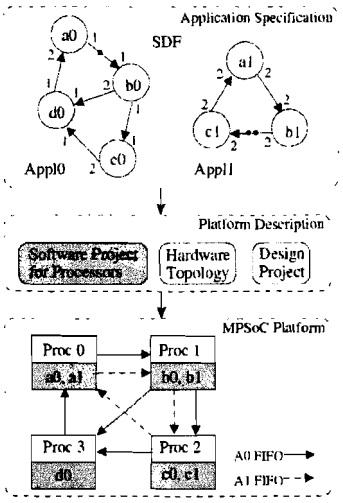


Figure 5.4: Example of MAMPS design flow. Source: [21]

use-case depiction, in case a system supports multiple applications, but only few are active at a given time.

As we have seen in the previous example MAMPS maps for single applications each actor to its own processor but for multiple applications it lets actors from different applications share processors. MAMPS maps the channels to dedicated connections which are not shared in case of multiple applications. This assures that communication delays have no negative effect on overall system performance and that head-of-line blocking does not occur. According to the SDF theory actors can only fire when sufficient tokens are available on its inputs. In case of multiple actors mapped to the same processor that could lead to a blocking of ready actors. MAMPS solves this problem by executing the other idle actors in case of unsuccessful read or write.

MAMPS maps the processors in the MPSoC flow to Xilinx Microblaze processors. The FIFO channels it maps to Fast Simplex Links (FSL). The buffer-size, as specified in the SDF input, sets the depth of the FSLs. Every actor is mapped to its own processor and gets its own memory. Only the first processor, due to limitations of the Xilinx system, also gets peripherals such as timers, a UART, a SysACE flash controller, and a DDR RAM interface. The timer helps in profiling the application, the SysACE supports writing and reading files and the DDR RAM can be used as memory for algorithms. The generated software includes functions for modelling the behaviour of the tasks mapped onto a processor. At times of the publication it only supported simple delays as tasks, but part of the work in this thesis added support for including real externally specified tasks. This is presented in section 5.3.

5.3 Modifications

When I began with this project the MAMPS tool lacked a number of features. These features included the actual sending of data tokens between processors on multi-rate channels and the ability to execute designer-provided source code. In order for sec-

ond stage of MAMPS to generate a MPSoC system including the software for each processor, it needs two inputs. First there is the SDF specification in XML to derive the architectural design. Second we have to specify which source code should end up in which location. I modified the MAMPS application to allow these changes. The second change involves the communication between processors (or actors). I added to MAMPS the possibility to actually send tokens with real data. I will explain the changes in more detail now.

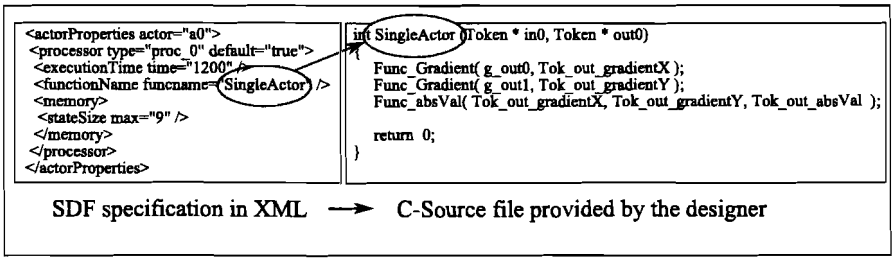


Figure 5.5: A reference in the XML file points to a function in the source code files.

1. Adding support for designer supplied application code.

   As illustrated in figure 5.5, the XML file contains a reference to a function name. Each actor refers to a single function, which should contain all functionality to be executed in that actor. In this case actor $A1$ should execute the function $gradientX$. The source file should be accompanied with a header file with the same name. In the header file the function should be declared with the following format:

```
Function_Name ( Token * in0,..., Token * inN,
                Token * out0,..., Token * outM);
```

   Obviously the function definition should be identical. For every incoming data variable we define a pointer to a array of the type Token called `in` with a sequential number. Similarly we define a pointer `out` for the outgoing data variables. In this case we have two parameters, one input token called $in0$ one output token called $out0$. The designer is responsible for modifying the functionality of the actor such that it communicates through these tokens. The `Token` structure definition can be arbitrary but also needs to be defined in the source files provided to MAMPS. MAMPS will take care of any declaration of these functions in the final project. Eventually the code generated by MAMPS will look like the following code for the input of figure 5.5. Previously MAMPS only put a delay statement in place of the functionality.

```
int task_Sobel()
{
 Token tok_out0[1];
 Token tok_in0[6];
```

```
 if(write_output_tokens_Sobel(tok_out0) == FAILED)
        return FAILED;
 if(get_input_tokens_Sobel(tok_in0 ) == FAILED)
        return FAILED;
 int ret = GradientX(tok_in0, tok_out0);
 init_token_Sobel();
 write_output_tokens_Sobel( tok_out0);
 return SUCCESS;
}
```

The implemented change automates the manual copying and modifying of source code in the Xilinx project. With an eye on full automatisation of the whole design flow I offer now an easy to use interface to the back-end of MAMPS. Basically a front-end tool needs to generate a single source file per actor with a function body and a corresponding function declaration in a header file.

2. Adding support for real data communication while taking channel rates into regard.

   Previously the processors in a MAMPS generated design did not actually send and receive data tokens. In order to test real applications it was desirable to add this functionality. I needed to define the send and receive functions and modify the actor task function as seen in the previous example. In that example two array have been added to the task for communication. The array $tok\_in0[6]$ represents an incoming channel with rate 6, the array tok_out0 an outgoing channel with rate 1. The task consists of a number of subtasks representing the (atomic) behaviour of an actor. First it attempts to send any tokens that haven't been sent in the last iteration. If it fails the task starts over until it succeeds. The next operation receives incoming tokens and also this should succeed or the task restarts. If it succeeds we should have an array full of received data tokens which are then fed into the function, in this case $GradientX$. This function will return the token for the outgoing channel in the array called tok_out0. If this is the first iteration then we need to set the initial tokens if they are defined for the outgoing channels. Finally the outgoing tokens are written to their respective channels. The code for multiple outgoing channels can be seen in the following example.

```
int task_Sobel()
{
 Token tok_out0[6];
 Token tok_out1[6];
 Token tok_in0[1];

 if(write_output_tokens_Sobel(tok_out0, tok_out1) == FAILED)
    return FAILED;
 if(get_input_tokens_Sobel(tok_in0 ) == FAILED)
    return FAILED;

 int ret = ReadWrite(tok_in0, tok_out0, tok_out1);
 init_token_Sobel();
 write_output_tokens_Sobel( tok_out0, tok_out1);
```

```
 return SUCCESS;
}
```

This modification finally allowed us to demonstrate the back-end part of MAMPS in working order. We now can run actual applications on the MPSoC in an FPGA. I have designed a demo-application which tests all necessary functionality of an SDF graph, i.e. multiple incoming, multiple outgoing channels, and multiple rates. MAMPS has been implemented now so far that we can use it for any design. We only need to consider the special resources in an Xilinx FPGA. These include serial ports, disk access and SDRAM memory access. These resource are only available for a single actor, most probably the first processor in the design. This is a Xilinx issue and beyond our control.

Chapter 6

C2SDF Implementation

In Section 5.3 I gave an overview of the modifications I made to MAMPS in order to interface it to the front-end. This front-end consists of a tool called C2SDF that converts the DAT output into an SDFG. I will discuss that part in this chapter. I finish this chapter with a number of experiments and results. Also I will analyse the found results in the final section.

6.1 C2SDF

In this section we explain the implementation of the front-end of MAMPS that gives the opportunity to split any C program into a parallel parts and generate simultaneously the hardware for the multi-processor system. It is a preprocessor for the hardware generating part of MAMPS and the XML input file with the SDF specification has been modified to support the software part. In the previous section I explained how I have added a property called `functionName` to the XML file which serves as the task name for the application running on each processor. A function with this name will be called from the main loop running on the processor. The number of arguments to this function is equal to the sum of channel inputs and outputs. The port rate determines the size of the inputs, i.e. a rate of 6 means that one input consists of a pointer to an array of size 6.

6.1.1 Output

First I will introduce the format of the XML file and its components. As can be seen in figure 6.1, the XML file with the SDF specification can be split into two parts. The first part describes the actors including its ports, the second part the properties of each actor such as execution time and function name. The whole description is encapsulated in a *root element* called $< applicationGraph >$. Under the root element we find two subelements called $sdfname$ and $sdfProperties$. The former contains 1 subelement for each actor in the graph. This actor has one subelement for each incoming or outgoing channel called $portname$, which includes a property for the value of the channel rate. Also $sdfname$ contains an element for each channel in the graph. This channel element has 4 properties, source actor, source port, destination actor and destination port. Using these elements the structure of the whole graph can be described. The latter element $sdfProperties$ defines the properties of each actor.

These are the processor type, the worst-case execution time, the already previously mentioned function name and the memory requirements.

```
<sdfMapping version="1.0">
 <applicationGraph>
  <sdf name="g" type="G">
   <actor name="a0" type="A1">
    <port name="IN" type="in" rate="0" />
    <port name="OUT" type="out" rate="0" />
   </actor>
   <channel name="X_X" srcActor="a0" srcPort="OUT" dstActor="
       a0" dstPort="IN" />
  </sdf>

  <sdfProperties>
   <actorProperties actor="a0">
    <processor type="proc_0" default="true">
     <executionTime time="1200" />
     <functionName funcname="SingleActor" />
     <memory>
      <stateSize max="9" />
     </memory>
    </processor>
   </actorProperties>
  </sdfProperties>
 </applicationGraph>
</sdfMapping>
```

Figure 6.1: Partial SDF description in XML

In figure 6.1 we see a single actor called $a0$ with 2 ports, one for an outgoing channel and one for an incoming channel. These channels have rates 0, thus basically they have no function. The actor task in the generated design will call a function named *SingleActor* and it will have an execution time of 1200 microseconds. The other properties are not very relevant at this development stage of MAMPS.

6.1.2 Input

Now we know which output we have to generate, we can examine the input for the C2SDF tool more closely. As we have seen in section 4.4 I have chosen DAT for my experiments. I recall that DAT is a tool developed by Rul et al. from Ghent University that provides a framework for extracting potential parallelism. It analyses the read and write pattern of memory location for determining dependencies. To avoid the difficult static analysis they use a profiling approach, recognising that it is potentially unsafe. Further, they aim to parallelise large chunks of code and use functions as their basic building block. So they measure the dependencies between functions and more particularly the data structures that the program uses. Basically they fill a matrix for each memory location with the read and write information.

DAT outputs 3 graphs, first a call graph showing the hierarchical relationship between the functions. It also contains how often the function is called, the number of different functions it calls and the fraction of execution time it consumes. The call

graph is useful for the SDF extraction because it allows us to calculate the worst case execution time and it provides how often a function has been called, which we need for determining the channel rates. The second graph is the inter-procedural data flow graph and is a visual representation of the memory location matrix analysis. It shows the dependencies between the functions in a directed graph. The third graph is called data sharing graph. It adds the data structures between functions to the inter-procedural data flow graph. It contains two sorts of nodes, one for the data structures (DS node) and another for the functions (ID node). If there is an edge from an ID node to a DS node to an ID node then we've found a dependency. Next to the graphical content, this graph also contains the numbers we need to calculate the channel rates.

I have made a small modification to DAT so that it doesn't work on a function level but on a statement level. Not only functions but also single statements show up in the output graphs. This avoids the problem that functions do not cover the whole program. For example before the first function call in main there can be several other statements, or between two functions. For determining parallelism that is sufficient, but not for SDFs. After the modification we end up with a graph where each statement is represented, including among others function calls, conditional statements and loops.

6.1.3 Actor values

After running DAT and having generated the graphs we can start extracting the SDF graph. Since communication resources don't come for free in an actual hardware design we are only interested in treating the DS-nodes as dependencies. They represent large structures, usually arrays and thus large transfers of data. Considering our final FPGA design, it doesn't really pay off to send for example index variables around between processors. Let us take a detailed look at the DAT *Data Sharing* output graph in figure 6.2.

We can identify the two types of nodes, simply called DS- and ID-nodes. DS-nodes represent data structures in the application and can be considered data dependencies. ID-nodes represent statements and functions in the application. If we examine ID-node ID150, we 2 incoming edges and 1 outgoing edge. The dashed incoming line comes from a DS-node, the dashed outgoing edge also goes to a DS-node. The solid line, although not visible in the figure, comes from another ID-node. Dashed edges always run (both ways) between DS- and ID-nodes, solid lines only run between ID-nodes. Where dashed lines indicate data dependencies, solid lines indicates a hierarchy of statements. This hierarchy is illustrated in more detail in figure 6.3, where I show how the following double FOR loop is represented in the DS-graph. A node with a name starting with a capital L followed by a number is a loop statement, as in the nodes with ID = 118 and ID = 121. Node ID118 (or L1766) has 4 children, ID119, ID120, ID205 and another loop statement called ID121. The number behind ID indicates the sequence in which these statements were first called. So we can conclude that 2 statements were called before the second loop was entered and 1 afterwards. That can be related to the 3 parts of a loop: an initial assignment, a comparison and an increment.

```
for (int j=2;j<=M-1;j++)
{
  for (int i=2;i<=N-1;i++)
  {
   Task1();
     .
```

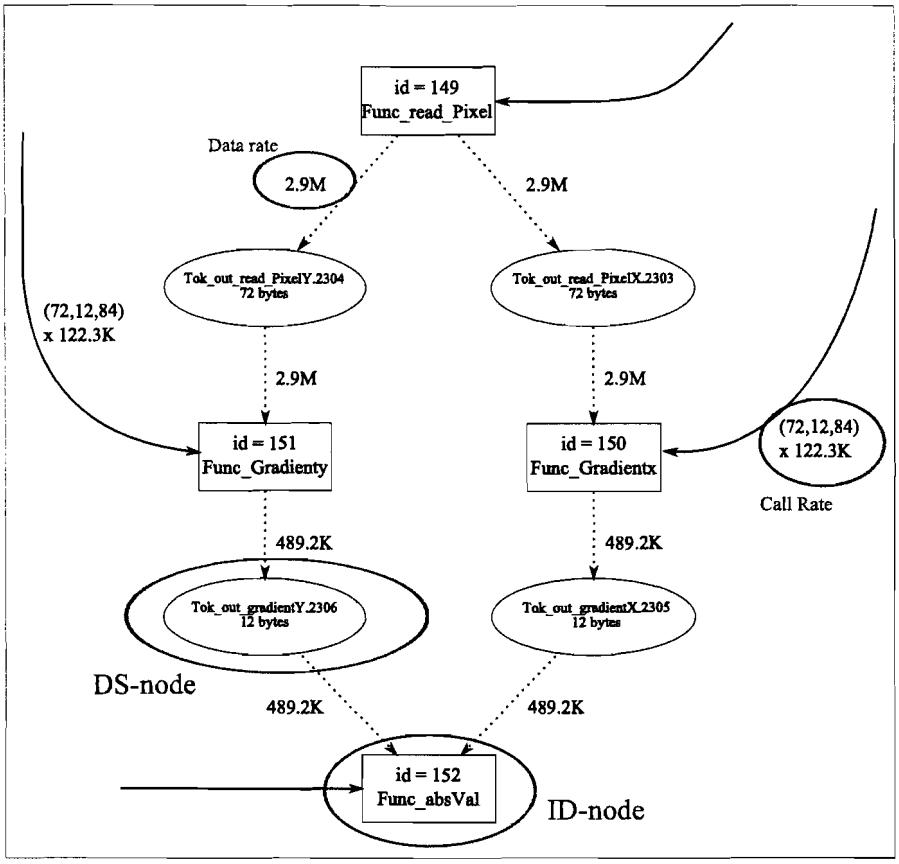


Figure 6.2: Detailed view of DS-graph.

```
    .
    .
  TaskN();
 }
}
```

Each solid edge has several numbers as property. For example ID150 has the values (72,12,84) x 122304 on its call edge. The numbers between round brackets show the memory use of the node and the second number how often the statement of the node was called. The memory use is not relevant for our needs, but the node call rate is very important. First we return to the dashed lines. The number next to the dashed lines stands for the amount of data read by the ID-node towards the edge points. For example the incoming dashed edge into node ID150 has the value 489.2K. The destination ID node reads this amount of data in bytes from the data structure represented by the DS-node. For our example that means that the code represented by node ID150, the function $Func\_Gradientx$ reads 489200 bytes from the data structure with the name $Tok\_out\_read\_PixelX.2303$. In fact the number 489200 is the result of rounding by DAT. I modified the DAT program such that it writes the actual values instead of the

rounded value as we can see in other example in the remaining part of this chapter.

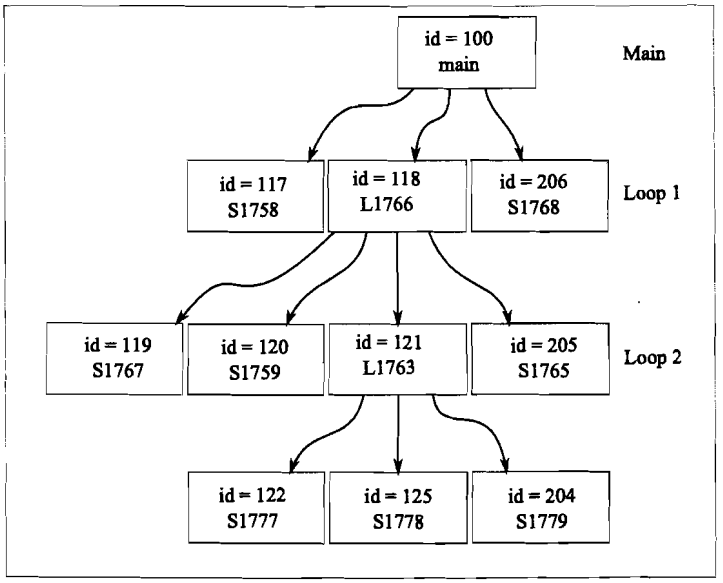


Figure 6.3: Double loop in a DS-graph.

We have now a measure for the data that is communicated over data dependencies in a program and we have a measure for the number that a statement is called by its parent. The latter tells us how often that statement is called and we know the sum of all data used by that statement. If we divide the amount of daat by the times the statement is called we get the following formula for the data per execution:

$$Data\,rate = \frac{Total\,amount\,of\,communicated\,data}{Times\,statement\,called}$$

The formula is valid both for outgoing edges and incoming edges. It gives us exactly that behaviour that is essential for an actor. The number of consumed and produced tokens in an actor is constant for all firings, just as it is for the statements in the DS-graph. Figure 6.4 shows a simple case of deriving an SDFG from the DS-graph.

Note that the formula applied to this example returns 4 as answer but in the SDFG we see 1 as rates. This is due to the unit used for the SDFG. DAT counts the data sizes in bytes, while we are more interested in integers for the SDFG. That means we have to divide the rates by 4 in this case. The formula can be modified as follows:

$$Data\,rate = \frac{Total\,amount\,of\,communicated\,data}{Times\,statement\,called * Sizeof(int)}$$

6.1.4 SDF Graph structure

Now we know how to derive the rates, the next step is how to derive the total graph. The approach I followed puts the data dependencies and therefore the DS-nodes cen-

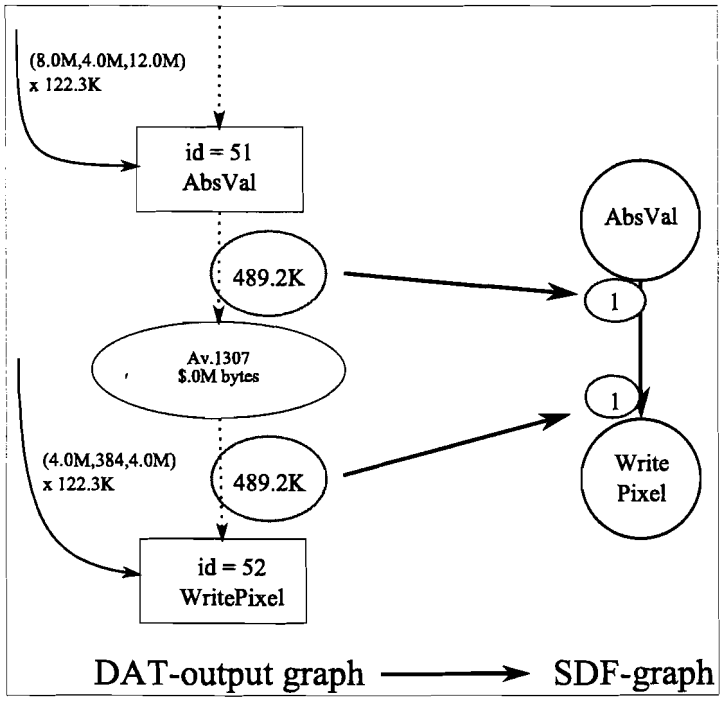


Figure 6.4: Deriving an simple SDFG.

tral. I set each data dependency equal to a channel in the final SDFG. After making that assumption, the problem reduces to merging the actors around these channels. Basically every statement of group of statements represented by ID-nodes needs to merge such that they only communicate over channels, i.e. elimination of all dashed lines in the DS-graph. The following algorithm takes a greedy approach to solve the merging problem:

The first step is to sort the nodes such that they follow a sequence of increasing numbering. In fact DAT numbers the nodes according to the discovery sequence. This sequence is important for the algorithm because it implies that two sequential nodes correspond to consecutive statements in the source code. This is important because of the atomic requirement of actors. We can only merge statements that are called consecutively in the program code because merging statements after another statement has been called in another actor does not lead to atomic behaviour. After all there is no data dependency between the two parts of the actor. At the same time we have to visit every node to set the initial value of the actor number to zero. This variable actor number is only a number to identify to which actor a node belongs. In the second step the algorithm visits all nodes again and assigns the current value of the actor number to each node. When it finds a DS-node it will increment the value of the actor number by 1. From that point onwards it will assign the new value to the ID-nodes. Note that DS-nodes do not get this value assigned because they are not part of any actor, but they become the channels between the actors. After we have visited all nodes we end up with all nodes assigned an actor number. The overall structure of the SDFG is known now. In the third step the gathered information is used to construct a real SDF

Algorithm 1 $SDF - Extract()$

```
1: Parse the DAT files.
2: for all nodes do
3:     Sort nodes according to discovery sequence.
4:     Set actor number to 0.
5: end for
6: for all nodes do
7:     if type(node) = DS then
8:         Increment Actor number.
9:     else
10:        Assign actor number to node.
11:    end if
12: end for
13: for all nodes do
14:    if type(node) = ID then
15:        Create Actor for each unique actor number.
16:    else if type(node) = DS then
17:        Create Channel.
18:    end if
19: end for
20: for all channels do
21:    Calculate channel rates.
22:    Calculate worst-case execution time.
23: end for
```

graph with actors and channels. All ID-nodes with the same actor number are merged into a single actor. The channels can directly be derived from the DS-nodes. The statement (or ID-node) that was discovered first in that actor provides the name for the new actor. The function name that will be called by the actor is also derived from that same ID-node. After this step has been finished we have a SDF graph with all actors and channels included. We still lack in the numerical part, in other words we have not yet added the channel rates, the worst-case execution times and the actor fire rates. In the fourth and final step we fill in the numerical gaps of the newly constructed SDFG. The channel rate will be calculated as previously explained using the sending ID-node's call rate and the data transfer rate on the edge to the connected DS-node. From the Call Graph we can derive the execution times and assign them to the actor. The worst-case execution time for the merged statements is equal to the sum of all execution times of the individual statements. The actor firing rate can be derived from the call rates of the statements in the actors. In case of multiple values inside the same actor we can take the greatest common divider. This does not take the conditional statements into account. However we can easily expand the algorithm since we can modify the tools such that if-statements can be identified. The graphs could be modified such that call edges go from the if-node to the nodes that are included in the condition in the actual source code. It is then relatively easy to merge all nodes under an if statement. Another problem is the sequence of writing in a producing actor and sequence of reading in the consuming actor. In a software program the communication between 2 functions in this situation is solved by having large buffers. In an SDF graph we cannot express that buffer directly and we have to modify either the code inside an actor or the channel

rates. The situation for 2 consecutive nested for-loops, where $func1$ writes to an array that $func2$ reads in opposite sequence currently cannot be detected by the tool.

6.2 Experiments, results and evaluation

In this section I will illustrate the whole process of SDFG deriviation with a number of experiments. After these experiments I will give an overview of my findings and finally finish with an evaluation of my results.

6.2.1 Experiment I : A Sobel filter with a single double-for-loop

In these experiments I will often refer to a Sobel filter application. This Sobel filter was supplied with PN-GEN but I modified it extensively to perform my experiments with DAT. In the original form it consisted of 5 double for-loops with each a function call. These calls are to functions called `readPixel`, two times to `gradient`, `absVal`, and finally `writePixel`. The special feature of this application is the structure of the data dependencies. It is not straightforward to put the function calls together in a single double for-loop. The data dependencies are such that the application cannot easily be pipelined. It requires extensive buffering in part to allow this pipelined execution. The data that is required in the next function is usually not all yet calculated in the previous function. PN-GEN's strenth lies in solving these kinds of dependencies and also in calculating the minimal amount of buffer space necessary in the actor to enable optimal performance. For my experiments I modified the Sobel example such that these deep dependencies have been eliminated. Unfortunately that led to increased data traffic between actors, because it requires resending of the same data packets. Furthermore I replaced the used integer variables by small array sufficiently sized to hold the data for a single iteration of a function. Basically I simulated already a message passing system in the program. The idea was to simplify in the beginning the test application for DAT such that we can understand the results. Finally I merged the 5 double loop of the original into a single double for-loop. This way the pipelined behaviour is very obvious and should also be easily recognisable in the output. The double loops look as follows:

```
static Token Tok_out_read_PixelX[6];
static Token Tok_out_read_PixelY[6];
static Token Tok_out_gradientX[1];
static Token Tok_out_gradientY[1];
static Token Tok_out_absVal[1];

for (j=2; j <= M-1; j++) {
  for (i=2; i <= N-1; i++) {
    if (j==2 && i==2) InitImage();
    Func_read_Pixel( Tok_out_read_PixelX, Tok_out_read_PixelY );
    Func_Gradientx( Tok_out_read_PixelX, Tok_out_gradientX );
    Func_Gradienty( Tok_out_read_PixelY, Tok_out_gradientY );
    Func_absVal( Tok_out_gradientX, Tok_out_gradientY, Tok_out_absVal  );
    Func_write_Pixel( Tok_out_absVal );
  }
}
```

I had to add this $InitImage$ function for easy reading from disk. It fills a global array with the data from a picture file. $Readpixel$ can read from that array to return the correct pixel for use in the gradient functions. The contents of the called functions can be seen in the appendix, but it should not be very relevant here. It is sufficient to know that both parameters for function $ReadPixel$ are being written in the function body. For $WritePixel$ the parameter is only being read. For the other functions the first parameter gets read, the second gets written in the respective functions. The arrays are sized such that the communicated data per iteration exactly fits. For example for the first array that means that $ReadPixel$ fills it with 6 pixels and $Gradientx$ reads 6 pixels from it. The following SDFG is a simplified output result of the tool. I left out the index variables and file pointers. DAT includes all variables that are static as DS-nodes. The C@SDF tool outputs the SDFG of Figure 6.5.

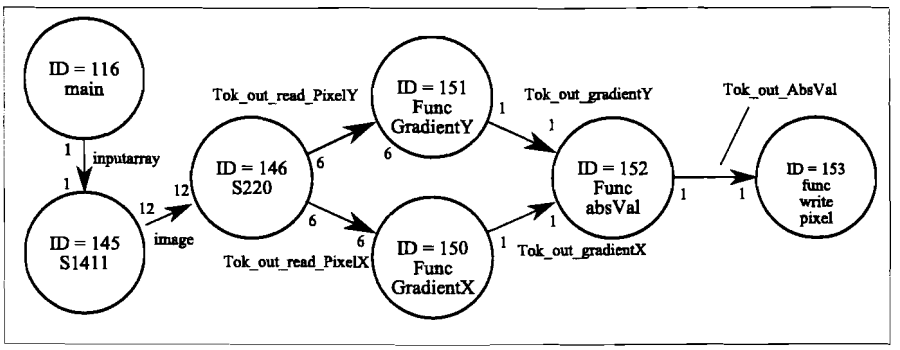


Figure 6.5: SDF for Experiment 1.

The resulting SDFG points us to a number of issues. First this SDFG would never fire because there is no initial token. I think this is not so critical at this point of research because we can easily add a self-edge to the first actor with input rate 0 and output rate 1 and a single initial token. That would get the SDFG started. A second issue to note is that this graph only fires once for a single iteration of the original double loop. In fact we should have some technique to make it fire as many times as the loop nest. One could include in the self-edge as many initial tokens as necessary, one for each iteration. Maybe a better solution would be to assign the call rate of the statements to the fire rate of the actor, but that would also require the same amount of initial tokens on the self-edge. In the actor I have put a reference to the function call. For example one of the actors includes a function call to a function `Func\_Gradienty` which in this case is only code in that function in the application. However the first actor calls `main`, which includes also all statements up to the point in the program where `inputarray` is being written. From that point onwards all statements are included in the next actor `InitImage`. Regarding the channel rates we see that they are exactly equal to the array locations written in the functions. Between actor 146 and 151 an array of 6 Tokens has been communicated. We see that actor 146 produces 6 tokens and actor 151 consumes 6 tokens.

6.2.2 Experiment II : Multiple double-for-loops

In this example I divide the function calls over multiple loop nests. Communication runs through 3 large arrays and the hidden global array from experiment 1.

```
int i, j;
static int Jx[1000][1000];
static int Jy[1000][1000];
static int av[1000][1000];

InitImage();

for (j=2; j <= M-1; j++) {
  for (i=2; i <= N-1; i++) {
  gradient( &image[j-1][i-1], &image[j][i-1],
  &image[j+1][i-1], &image[j-1][i+1],
  &image[j][i+1], &image[j+1][i+1], &Jx[j][i] );
}}

for (j=2; j <= M-1; j++) {
  for (i=2; i <= N-1; i++) {
    gradient( &image[j-1][i-1], &image[j-1][i],
    &image[j-1][i+1], &image[j+1][i-1],
    &image[j+1][i], &image[j+1][i+1], &Jy[j][i] );
}}

for (j=2; j <= M-1; j++) {
  for (i=2; i <= N-1; i++) {
    absVal( &Jx[j][i], &Jy[j][i], &av[j][i] );
}}

for (j=2; j <= M-1; j++) {
  for (i=2; i <= N-1; i++) {
    writePixel( &av[j][i] );
}}
```

Not surprisingly the result is almost exactly the same, but at one point we need to be careful. Due to the way DAT works we end up with a situation as in figure 6.7. Here the two gradient functions share a common source. The common formula does not work for this situation. After all the same data would count for both dependencies and we would calculate 12 instead of 6 as channel input rate for both. The C2SDF tool needs to take this possibility into account.

figure 6.6 shows the resulting SDF graph.

6.2.3 Experiment III : Double-for-loops with a condition

Since conditional statement tend to be problematic for SDF as we have seen in the previous chapter, it is worthwhile to investigate what happens when encounter a conditional statement in an application. For this example I have made a small modification to the previous example by putting an if-condition in front of the second gradient function call. The condition `if(image[j][i] < 10)` tests if one of the data values is smaller than 10. We should realise that this is not allowed in an SDF graph, because

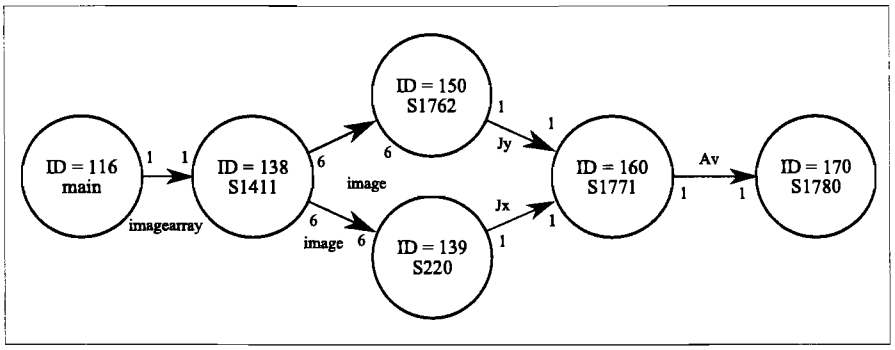


Figure 6.6: SDF for Experiment 2.

it makes the SDF schedule unpredictable. The solution is to make sure that every statement within the condition is merged into the same actor. Figure 6.8 illustrates how DAT handles such a condition and Figure 6.9 shows the resulting SDFG.

Unfortunately currently DAT does not generate enough information for us to be able to derive the correct actors. Two requirements would solve the problem, first, show the statements under the condition in a similar way as currently happens with loops, i.e. a solid line from the conditional ID-node to the ID-node of each statement in the body of the condition. Second, we must be able to actually identify a condition in the DS-graph. Also this can be handled similarly as with loops, by changing the ID name to a pattern like CXXX with XXX an unique number for the statement. For loops it follows the pattern LXXX.

Also here the calculation of the channel rates fails for the same reason. In fact sharing between three nodes makes the problem worse. In the SDFG I have changed the wrong values of 10 to the correct values of 6. I have pictured the channel for the dependency to the condition separately, but in fact it can merged into the normal image channel. Because the same location is also read for the operation in the gradient function we should leave out this extra channel. PN-GEN fails to produce an output graph for a condition testing on data values.

6.2.4 Experiment IV : Limits of DAT

In the previous example we were confronted with a problem regarding conditional statements. In this example we will take a look at another problem regarding the program analysis with DAT. I use the following source for this example:

```
int main(void)
{
int i, j;
static Token Tok_out_read_PixelY[6];
static Token Tok_out_gradientY[1];
static unsigned long CheckSum;

 for (j=2; j <= M-1; j++) {
   for (i=2; i <= N-1; i++) {
     Func_Gradient( Tok_out_read_PixelY, Tok_out_gradientY );
```

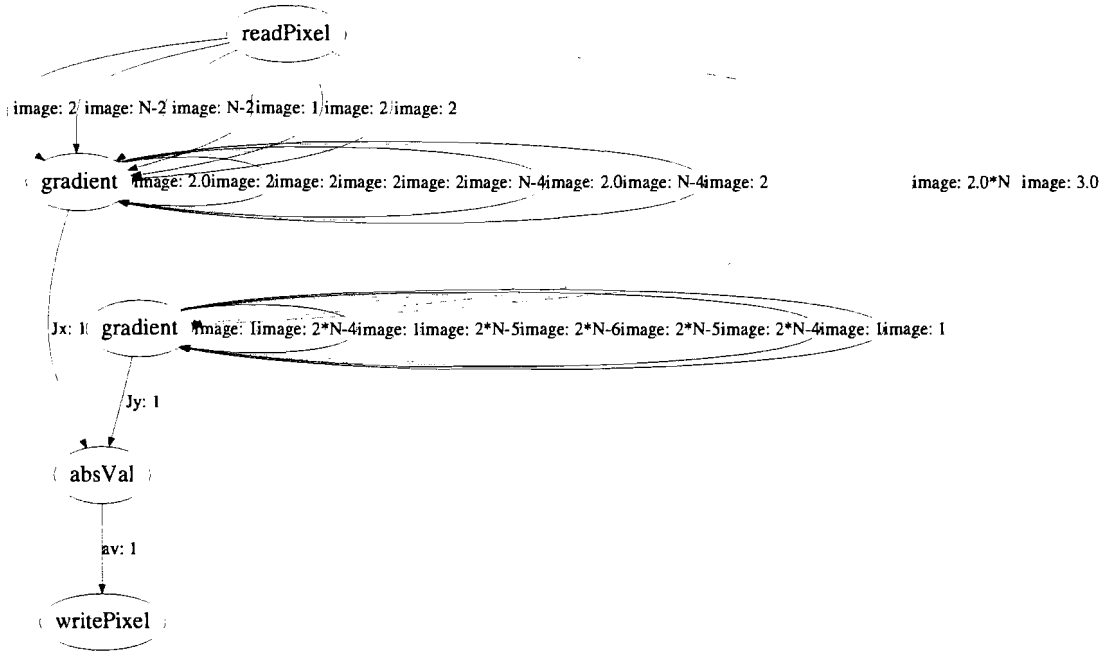


Figure 6.7: PN-GEN output for Experiment 2.

```
    Func_write_Pixel( Tok_out_gradientY , &CheckSum);
    CheckSum++;
 }}

return CheckSum;}
```

I have removed some of the functions from the previous examples and modified the *writepixel* function. It now updates a checksum variable every iteration of the loop. That means it reads and writes to that variable. This variable is incremented in the next step in that loop body and finally after the loop finished it gets read one last time before the program finishes in the return statement. The gradient function has been retained here to get an idea of data flow again. Figure 6.13 shows the resulting output of DAT for the given source code.

I focus here on DS-node $CheckSum.2306$ because there we run into the limits of DAT. We have 2 nodes writing and reading from this node and a single node that only reads from the DS-node. The problem we face is how to distil the channel information from the graph. In fact this situation is the limit of what we can extract from DAT in the current version. Since ID-node 126 is outside the double for-loop (which we can detect), we can simplify the problem to 2 nodes. A 2-node problem is easily disentangled. Of course this is only possible because we know which statements belong to the body of the loop. The derived SDFG is shown in Figure 6.10.

Now I will illustrate with figure 6.11 the situation for which we cannot derive a correct SDFG with the current version of DAT. I made a small modification to the pro-

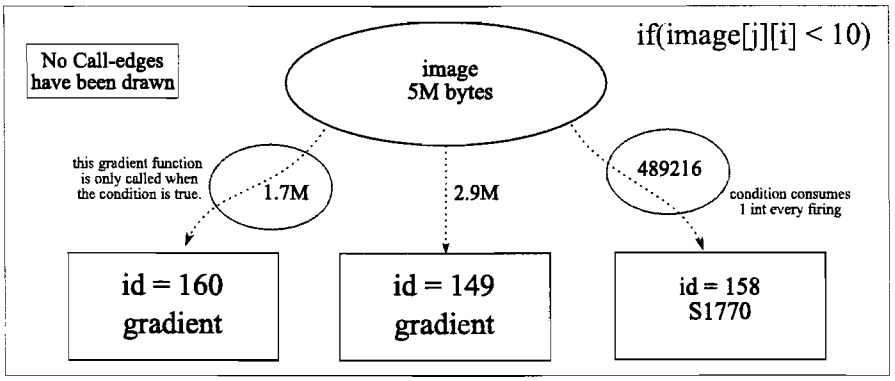


Figure 6.8: Dependency with condition.

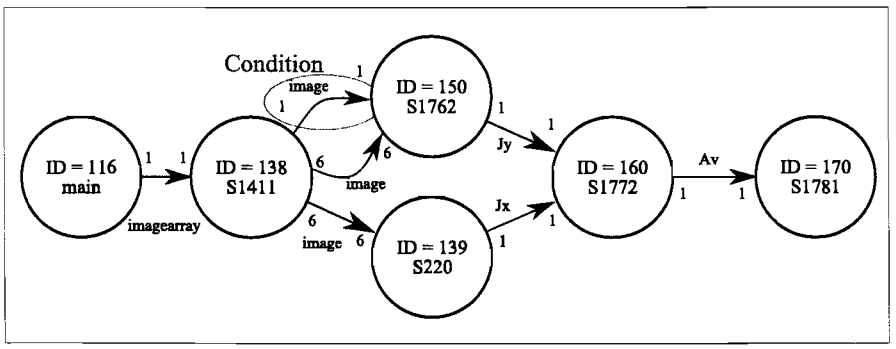


Figure 6.9: SDf for Experiment 3.

gram that splits the call to $WritePixel$ into two identical functions called WritePixel1 and WritePixel2. Both read and write again to $CheckSum$. In figure 6.11 we clearly see that it is very difficult to determine the sequence of reading and writing. Without additional information it is impossible to derive the correct SDFG with certainty. Here it helps that we know the sequence of calling in the program, but we would not be able to detect complex read and write patterns. We can only assume a simple pattern with first reading and then writing. The worst-case would be were all statements read and write to the same buffer in arbitrary write and read sequences.

We can conclude that we can derive SDFGs for relatively simple program constructs. Unfortunately DAT does not offer enough detail to discover more complex read and write patterns. There is clearly a need for more detailed accounting on behalf of DAT.

PN-GEN on the other hand can handle this situation. The source code from experiment 2 has been modified slightly. I have added to the code from experiment 2 a variable called $CheckSum$ which function $absVal$ reads and writes to repeatedly. Furthermore function $WritePixel$ reads this variable also once and finally it is returned from $main$. The array $image$ is re-used in the last loop-nest to investigate how PN-GEN handles the shared variable problem. The following source has been stripped

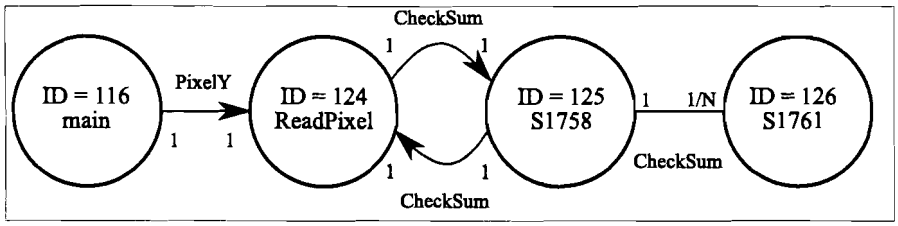


Figure 6.10: SDF for Experiment 4.

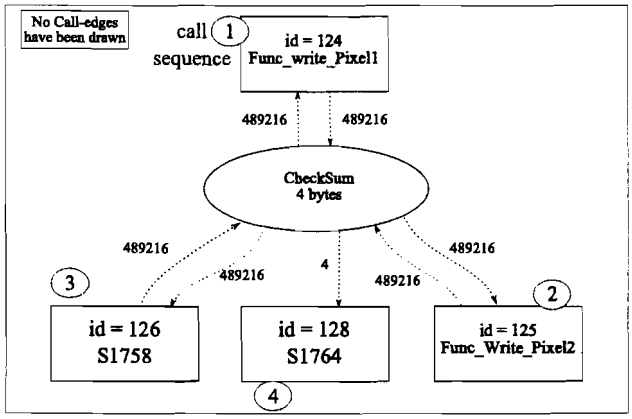


Figure 6.11: Shared variable problem.

of the unnecessary parts.

```
int main(void)
{
 int i, j;
 static int CheckSum[1000];
 static int image[1000][1000];

 for (j=1; j <= M; j++)
  for (i=1; i <= N; i++)
   readPixel(&image[j][i]);

      .
      .

 for (j=2; j <= M-1; j++) {
  for (i=2; i <= N-1; i++) {
   absVal( &Jx[j][i], &Jy[j][i], &image[j][i] , &CheckSum[0]);

 for (j=2; j <= 2; j++) {
  writePixel( &CheckSum[j-2] );}
```

```
 for (j=2; j <= M-1; j++) {
  for (i=2; i <= N-1; i++) {
   writePixel( &image[j][i] );

return (CheckSum[0]);
}
```

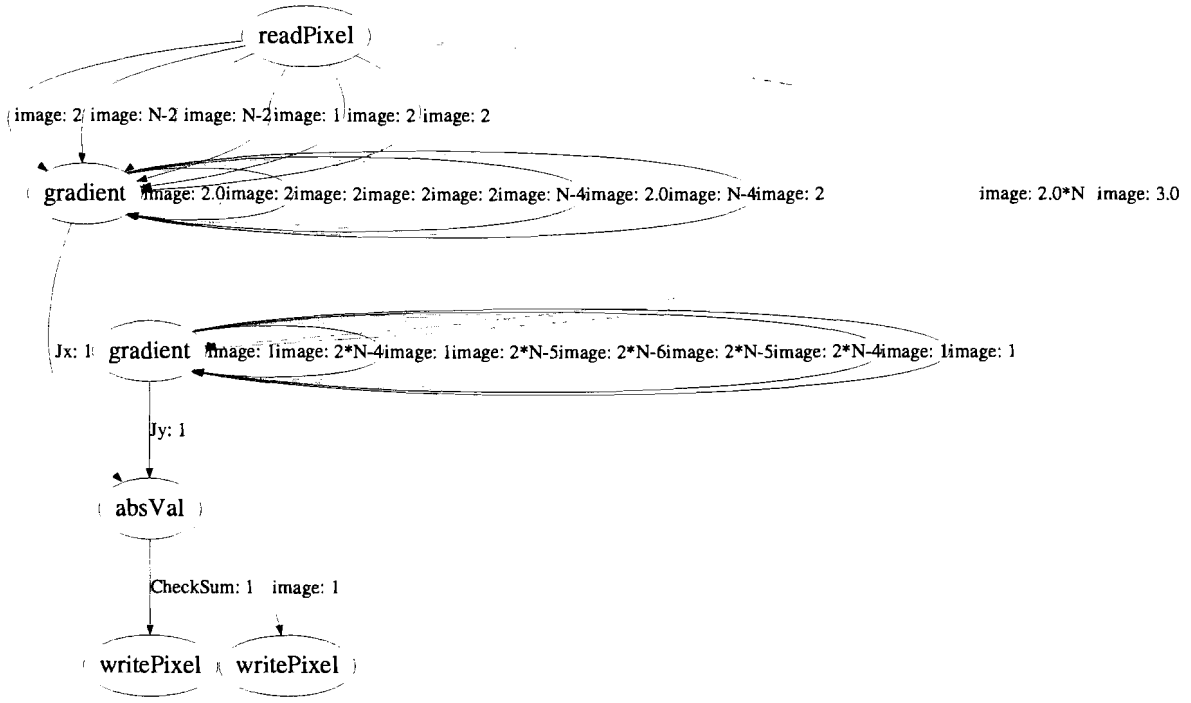


Figure 6.12: PN-GEN for shared variables.

As Figure 6.12 shows, PN-GEN understands the re-use of the array *image* in the last loop-nest. It also handles the variable $CheckSum$ correct, unless one expects a self-loop at the actor $AbsVal$ for the cross-iteration dependencies. We can assume that it is included in the actor. Because firing rates and channel rates are not in a KPN it is hard to judge if the KPN is suitable for transformation into an SDF.

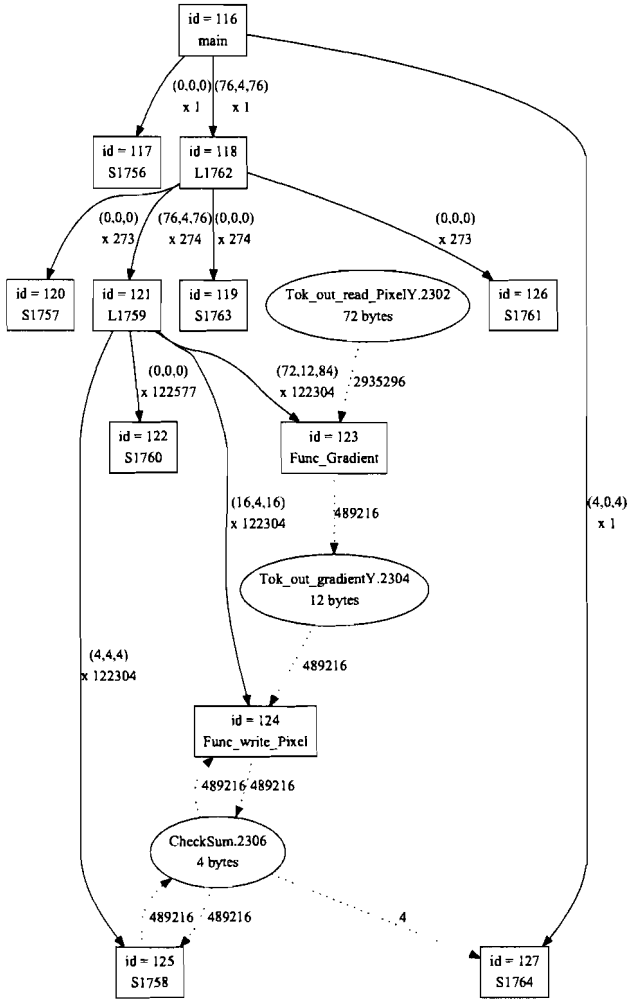


Figure 6.13: DAT output for Experiment 4.

6.2.5 Experiment V : Dynamic memory allocation

In the previous examples I have used static arrays for communication between the functions. In practice applications often use dynamic allocation of memory. Pointers are used in combination with the well-known *malloc*90/*free*() combination. DAT can detect dependencies with dynamically allocated memory locations. Every call to *malloc*() will produce a single DS node. When this happens often, a lot of DS-nodes will be constructed. Fortunately these nodes can be merged with relative ease. I use the following code to illustrate the result of *malloc*() occurrences.

```
#define N 5
#define M 5
```

```
void Func_Gradient( Token * in, Token * out )
{
  out[0].Data  = ( ((in[0].Data)+((in[1].Data)<<1)+(in[2].Data)) ) -
                 ( ((in[3].Data)+((in[4].Data)<<1)+(in[5].Data)) );
}

static Token Tok_out_read_Pixel1Y[6];
static unsigned long CheckSum;
Token * tmp;

for (j=2; j <= M-1; j++) {
      for (i=2; i <= N-1; i++) {
       tmp = malloc(sizeof(Token));
       Func_Gradient( Tok_out_read_Pixel1Y, tmp );
       Func_write_Pixel1( tmp , &CheckSum);
       Func_write_Pixel2( tmp , &CheckSum);
       CheckSum++;
       free(tmp);
}}
```

And the resulting output is shown in figure 6.14. In total DAT constructs nine DS-nodes (not all of them are shown here) and we can see how it numbers them. Therefore merging these nodes should be trivial.

However note that the input rate for the *malloc* channel is equal to 8; 4 bytes for each iteration of a function that reads from this channel. I need to modify the actor merging algorithm such that it merges the DS-nodes and calculates the channel rates correctly.

6.3 Evaluation

In the previous sections I have presented a number of experiments to illustrate the possibilities and limits of using DAT as parallelisation engine. I will summarise them in this section. As we have seen DAT provides a detailed overview of a program. It clearly presents all data dependencies in the whole program. We also have seen that it provides all necessary values for calculating the channel rates which make it especially suitable for SDF generation. I have shown how I can derive correctly channel rates for most programs. Every variable used for communication between different statements or functions end up as a special node in the output.

The critical factor for successful SDF extraction is the presence of separate variables for communication between actors. As long the same variable (this can be an array or a scalar) is not used for communication by more than 2 actors, it is possible to derive an SDFG. If 3 functions use the same variable consecutively it is impossible to extract the correct sequence from the DAT output. To resolve this limitation we need to modify DAT such that it keeps better track of access sequences of various functions or statements. This should be possible since it keeps track of these patterns already. It should largely be a matter of better accounting. PN-GEN is much stronger in dependency analysis.

DAT can both handle static variable and dynamically allocated variables. The former works very straightforward and can easily be converted to a channel in an SDFG.

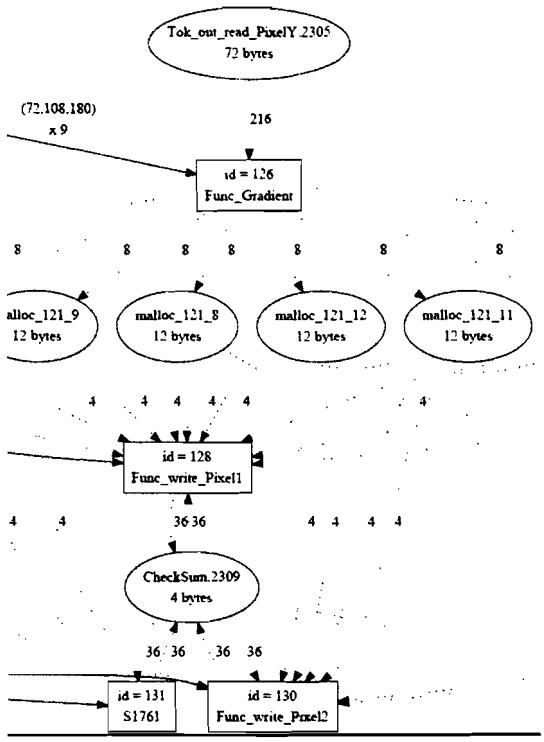


Figure 6.14: Output when using malloc.

In the latter case the output becomes somewhat more complicated but can be simplified with relative ease.

The program analysis technique used by DAT is profiling. The disadvantage of profiling is that it is potentially unsafe. Unsafe because program execution patterns might vary depending on the input data for the program. This has as consequence that the output of DAT is not correct. With some modifications to DAT it should be possible to detect this situation and take it into account for deriving the SDFG. An SDF model can handle this situation as long as the condition which causes this is hidden in an actor including all statements in the body of that condition.

The experiments shows that SDF might not be the most suitable model for expressing these low level programs. SDF is usually used for modelling high-level applications like the example of an H.263 decoder. Since we can merge actors into new actors it might be possible to construct an SDF from the bottom up. This way it would not become a huge SDFG for large programs, but this does not solve the problem for small low-level applications. An alternative MoC is CSDF, Cyclo-Static Data Flow which is better suited for this problem but tends to result into larger graphs than SDF. SDFs can be converted into CSDF and the other way around, so it is also possible to express the problem in SDF but at the cost of an larger graph. This might not be worth it.

Chapter 7

Related Work

In chapter 4 we have already seen a number of tool for parallelisation. The field of (semi-)automatic parallelisation tools is much wider than the three approaches that were considered. Many researchers propose a tool-chain comparable to the total MAMPS flow. First there is a tool called ESPAM [24] which, similar to MAMPS, automatically maps a MoC specification to a MPSoC for implementation on FPGA. They also rely on the commercial Xilinx tools for the final step, but they use KPN as MoC. It is from the same group as PN-GEN. A work closely related to the PNGEN/ESPAM tool-chain is the now commercialised COMPAAN/LAURA [20] [32] combination. The two approaches differ in their target platform. Where PNGEN/ESPAM targets true MPSoC platforms, COMPAAN/LAURA targets an architecture of a processor with a co-processor. Furthermore COMPAAN also supports matlab input. Ottoni et al. presented in [26] their work about *Decoupled Software Pipelining* (DSWP). It especially takes inter-thread communication latency into account. It also aims to parallelise ordinary programs and not only DSP-like applications. Recently published work by Ceng et al. from RWTH Aachen is called MAPS [3] which uses a combination of static and dynamic analysis to find parallelism. The analyse the input source code, extract the parallel tasks and generate the output source code for these tasks. A framework called TCT compiles the output and maps it to a MPSoC utilising a so-called *Tightly-Coupled-Thread* (TCT) programming model.

Chapter 8

Conclusion and Recommendations

- I could choose between PN-GEN and DAT as parallelisation engine. Many researchers are currently working on automatic parallelisation and more tools come available continuously. Other approaches include MAPS, SPRINT, and DSWP.
- DAT provided suitable output and because of its relatively low complexity easy to modify. Another advantage is that it performs the analysis on the whole program. The main alternative PN-GEN only accepts ANL programs and only takes the main function of a program into account.
- For the work in this thesis I have mainly focused on DAT for experimenting. That led to some promising results but it is important to take PN-GEN again into account in the future. Its static analysis and especially solving the shared variable problem are crucial to success.
- Profiling is unsafe, if we continue with this approach we need to detect if an application is dynamic or static. Therefore we need to modify DAT such that it presents conditions in a different and more clear way.
- The accounting in DAT needs to become more detailed. Being able to differentiate which pattern a variable is written to and read from is essential for solving the shared variable problem.
- For an application that strictly uses a variable only for one channel C2SDF can derive an SDF graph.
- Eventually the resulting actors need to be implemented in code again. That requires a table where statement nodes are linked to statements in code and keeping track of which statements end up in an actor. Putting the bulk of the program in its correct place should be straightforward but finishing the detailed parts like variable declarations might be challenging.
- I have completed and then extended MAMPS from the SDF specification onwards. It now can generate project files for synthesis which, complete with fully functional source code, can be implemented on an FPGA.

- Probably CSDF a more suitable MoC for expressing the extracted parallelism from low-level source code than SDF. Nevertheless we can convert the problematic SDFs (due to program constructs) to correct SDFs.

Bibliography

[1] http://www.chinapost.com.tw/business/global-phone.htm.

[2] A. Bernstein, "Analysis of programs for parallel processing," *Electronic Computers, IEEE Transactions on*, vol. EC-15, no. 5, pp. 757–763, Oct. 1966.

[3] J. Ceng, J. Castrillon, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda, "Maps: an integrated framework for mpsoc application parallelization," in *DAC '08: Proceedings of the 45th annual conference on Design automation*. New York, NY, USA: ACM, 2008, pp. 754–759.

[4] J. Cockx, K. Denolf, B. Vanhoof, and R. Stahl, "Sprint: a tool to generate concurrent transaction-level models from sequential code," *EURASIP J. Appl. Signal Process.*, vol. 2007, no. 1, pp. 213–213, 2007.

[5] A. G. D. Culler, J. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 2006.

[6] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink, "Yapi: application modeling for signal processing systems," in *DAC '00: Proceedings of the 37th conference on Design automation*. New York, NY, USA: ACM, 2000, pp. 402–405.

[7] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks: An Engineering Approach*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1997.

[8] K. Ebcioğlu and T. Nakatani, "A new compilation technique for parallelizing loops with unpredictable branches on a vliw architecture," in *Selected papers of the second workshop on Languages and compilers for parallel computing*. London, UK, UK: Pitman Publishing, 1990, pp. 213–229.

[9] K. Ebcioğlu, "A compilation technique for software pipelining of loops with conditional jumps," *SIGMICRO Newsl.*, vol. 19, no. 3, pp. 36–41, 1988.

[10] P. Feautrier, "Automatic parallelization in the polytope model," in *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*. London, UK: Springer-Verlag, 1996, pp. 79–103.

[11] ——, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, 1991.

[12] A. Ghamarian, "Phd thesis, timing analysis of synchronous data flow graphs," Ph.D. dissertation, Technische Universiteit Eindhoven, 2008.

[13] J. L. Gustafson, "Reevaluating amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[14] B. M. H. Corporaal, J. van Meerbergen, *Lecture Notes TU/e course Processor Architectures and Program Mapping (5KK10).* Technische Universiteit Eindhoven, 2006.

[15] M. Hill and M. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, July 2008.

[16] L. E. Jordan and G. Alaghband, *Fundamentals of Parallel Processing.* Prentice Hall Professional Technical Reference, 2002.

[17] G. Kahn, "The semantics of a simple language for parallel programming," *Information Processing*, pp. 471–475, 1974.

[18] I. Karkowski and H. Corporaal, "Fp-map - an approach to the functional pipelining of embedded programs," in *HIPC '97: Proceedings of the Fourth International Conference on High-Performance Computing.* Washington, DC, USA: IEEE Computer Society, 1997, p. 415.

[19] A. Kienhuis, *Design Space Exploration of Stream-based Dataflow Architectures : Methods and Tools.* Delft University of Technology, Delft, The Netherlands, 1999.

[20] B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: deriving process networks from matlab for embedded signal processing architectures," in *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign.* New York, NY, USA: ACM, 2000, pp. 13–17.

[21] A. Kumar, S. Fernando, Y. Ha, B. Mesman, and H. Corporaal, "Multi-processor system-level synthesis for multiple applications on platform fpga," in *Proceedings of the 17th International Conference on Field Programmable Logic and Applications*, 2007, pp. 92–97.

[22] M. S. Lam, *A Systolic Array Optimizing Compiler.* Norwell, MA, USA: Kluwer Academic Publishers, 1989.

[23] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987.

[24] H. Nikolov, T. Stefanov, and E. Deprettere, "Multi-processor system design with espam," in *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis.* New York, NY, USA: ACM, 2006, pp. 211–216.

[25] ——, "Multi-processor system design with ESPAM," in *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis.* New York, NY, USA: ACM, 2006, pp. 211–216.

[26] G. Ottoni, R. Rangan, A. Stoler, M. J. Bridges, and D. I. August, "From sequential programs to concurrent threads," *IEEE Comput. Archit. Lett.*, vol. 5, no. 1, p. 2, 2006.

[27] T. M. Parks, J. L. Pino, and E. A. Lee, "A comparison of synchronous and cyclo-static dataflow," in *ASILOMAR '95: Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers (2-Volume Set)*. Washington, DC, USA: IEEE Computer Society, 1995, p. 204.

[28] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *MICRO 14: Proceedings of the 14th annual workshop on Microprogramming*. Piscataway, NJ, USA: IEEE Press, 1981, pp. 183–198.

[29] A. W. Roscoe, *The theory and practice of concurrency*. Prentice Hall, 1998.

[30] S. Rul, H. Vandierendonck, and K. D. Bosschere, "Function level parallelism driven by data dependencies," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 55–62, 2007.

[31] O. Schliebusch, H. Meyr, and R. Leupers, *Optimized ASIP Synthesis from Architecture Description Language Models*. Springer Netherlands, 2007.

[32] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System design using kahn process networks: The compaan/laura approach," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2004, p. 10340.

[33] M. G. Stoodley and C. G. Lee, "Software pipelining loops with conditional branches," in *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 262–273.

[34] B. Theelen, M. Geilen, T. Basten, J. Voeten, S. Gheorghita, and S. Stuijk, "A scenario-aware data flow model for combined long-run average and worst-case performance analysis," *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, pp. 185–194, July 2006.

[35] S. Verdoolaege, H. Nikolov, and T. Stefanov, "pn: a tool for improved derivation of process networks," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 19–19, 2007.

[36] D. W. Wall, "Limits of instruction-level parallelism," *SIGPLAN Not.*, vol. 26, no. 4, pp. 176–188, 1991.