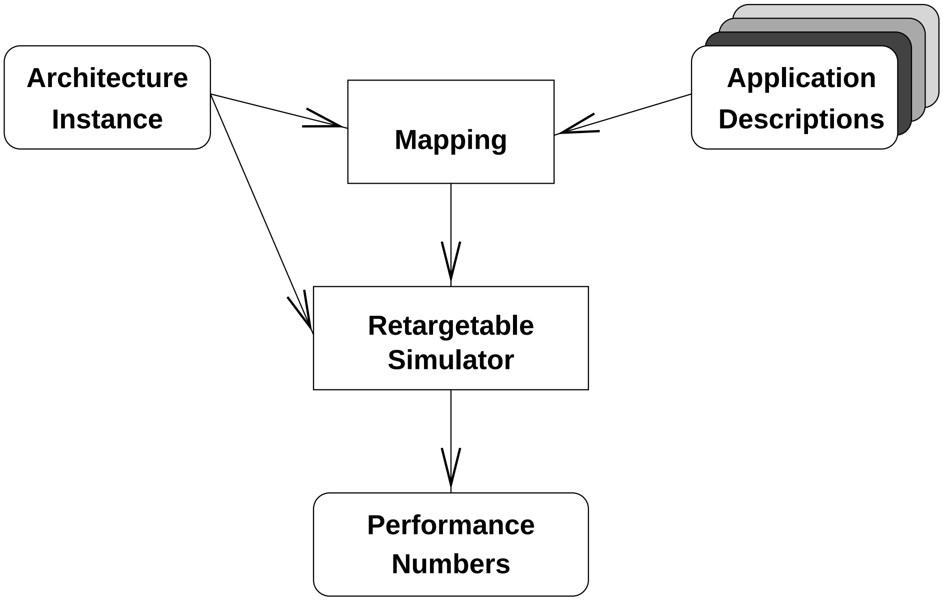
Design Space Exploration of Stream-based Dataflow Architectures

Methods and Tools



A.C.J. Kienhuis

Design Space Exploration of Stream-based Dataflow Architectures

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie,
door het College voor Promoties aangewezen,
op vrijdag 29 Januari 1999 te 10:30 uur

door

Albert Carl Jan KIENHUIS

elektrotechnisch ingenieur
geboren te Vleuten.

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. P.M. Dewilde

Toegevoegd promotor: Dr.ir. E.F. Deprettere.

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter	
Prof.dr.ir. P.M. Dewilde,	Technische Universiteit Delft, promotor
Dr.ir. E.F. Deprettere,	Technische Universiteit Delft, toegevoegd promotor
Ir. K.A. Vissers,	Philips Research Eindhoven
Prof.dr.ir. J.L. van Meerbergen,	Technische Universiteit Eindhoven
Prof.Dr.-Ing. R. Ernst,	Technische Universität Braunschweig
Prof.dr. S. Vassiliadis,	Technische Universiteit Delft
Prof.dr.ir. R.H.J.M. Otten,	Technische Universiteit Delft

Ir. K.A. Vissers en Dr.ir. P. van der Wolf van Philips Research Eindhoven, hebben als begeleiders in belangrijke mate aan de totstandkoming van het proefschrift bijgedragen.

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Kienhuis, Albert Carl Jan

Design Space Exploration of Stream-based Dataflow Architectures : Methods and Tools
Albert Carl Jan Kienhuis. -
Delft: Delft University of Technology
Thesis Technische Universiteit Delft. - With index, ref. - With summary in Dutch
ISBN 90-5326-029-3
Subject headings: IC-design; Data flow Computing; Systems Analysis

Printed in the Netherlands

Contents

Chapter 1

Introduction

Contents

THE increasing digitalization of information in text, speech, video, audio and graphics has resulted in a whole new variety of digital signal processing (DSP) applications like compression and decompression, encryption, and all kinds of quality improvements [Negroponte, 1995]. A prerequisite for making these signal processing applications available to the consumer market is their cost-effective realization into silicon. This leads to a demand for new application-specific architectures that are increasingly *programmable* i.e., architectures that can execute a set of applications instead of only one specific application. By reprogramming these architectures, they can execute other applications with the same resources, which makes these programmable architectures cost-effective. In the near future, these architectures should find their way into consumer products like TV-sets, set-top boxes, multi-media terminals, and other multi-media products, as well as in wireless communication and low cost radar systems.

The trend toward architectures that are more and more programmable represents, as argued by Lee and Messerschmitt [1998], the first major shift in electrical engineering since the transitions from analog to digital electronics and from vacuum tubes to semiconductors. However, general and structured approaches are lacking for designing application-specific architectures that are sufficiently programmable.

The current practice is to design application-specific architectures at a detailed level using hardware description languages like VHDL [1993] (Very high speed IC hardware description Language) or Verilog. A consequence of this approach is that designers work with very detailed descriptions of architectures. The level of detail involved limits the design space of the architectures that they can explore, which gives them little freedom to make trade-offs between programmability, utilization of resources, and silicon area. Because designers cannot make these trade-offs, designs end up underutilizing their resources and silicon area and are thus unnecessarily expensive, or they cannot satisfy the imposed design objectives.

The development of programmable architectures that execute widely different applications has already been being worked on for decades in the domain of general-purpose processors. These processors can execute a word-processing application or a spreadsheet application or can even simulate some complex physical phenomenon, all on the same piece of silicon. Currently, these processors are designed by constructing performance models of processors at different levels of abstraction ranging from instruction level models to Register Transfer Level (RTL) models [Bose and Conte, 1998; Hennessy and Heinrich, 1996]. These *performance models* can be evaluated to provide data for various *performance metrics* of a processor, like resource utilization and compute power while processing a workload. A workload is typically a suite of *benchmarks*, i.e., a set of typical applications a processor should execute. Measuring the performance of a processor delivers *quantitative* data. This data allows designers to explore the design space of processors at various levels of detail and to make trade-offs at the different levels between, for example, the utilization of resources and performance of processors. Moreover, quantitative data gives designers insight, at various levels of detail, into how to further improve architectures and serves as an objective basis for discussion of possible architecture improvements.

The benchmark approach practiced in general-purpose processor architecture design leads to finely tuned architectures targeted at particular markets. When the benchmark approach was initially introduced at the beginning of the 1980s, it revolutionized general-purpose processor design, which resulted in the development of RISC-style processors [Patterson, 1985]. These processor architectures were smaller, faster, less expensive and easier to program than any conventional processor architecture of that time [Hennessy and Patterson, 1996].

General-purpose processors, although programmable, are not powerful enough to execute the digital signal processing applications we are aiming at, as we will explain soon. Special application-specific architectures are therefore required that are nonetheless programmable to some extent. The benchmark approach used in the design of general-purpose processors is also useful in the design of the programmable application-specific architectures emerging now, as we show in this thesis. We will develop and implement in this thesis a benchmark approach, which we call the Y-chart approach, for a particular class of programmable application-specific architecture called *Stream-Based Dataflow Architecture*. The benchmark approach we develop results in an environment in which designers are able to perform design space exploration for the stream-based dataflow architecture at a level of abstraction that is higher than that offered by standard hardware description languages.

In Section 1.1 of this chapter we explain further why programmable application-specific architectures will emerge for digital signal processing applications. Following this, in Section 1.2 we discuss as an example the video signal processing architecture in a modern TV-set. Based on this example, we illustrate how the trend towards programmable architectures will affect the next generation of TV-sets and explain why general-purpose processors are not capable of providing the required performance at acceptable cost. We focus on the stream-based dataflow architecture in TV-sets. In Section 1.3 we indicate how we are going to explore the design space of this architecture at a high level of abstraction. This chapter concludes with the statement of the main contributions of this thesis in Section 1.4 and the further outline for the thesis in Section 1.5.

1.1 Motivation

New advanced digital signal processing applications like signal compression and decompression, encryption, and all kinds of quality improvements become feasible on a single chip because the number of transistors on a single die is still increasing, as predicted by Moore's law. Digital signal processing applications involve *real-time* processing, which implies that these applications take in and produce samples at a particular guaranteed rate, even under worst-case conditions. In addition, these applications are very demanding with respect to *computational power*, i.e., the number of operations performed in time, and *bandwidth*, i.e., the amount of data transported in time.

Architectures that realize the new applications cost-effectively in silicon must be able first of all to deliver enough processing power and bandwidth to execute the applications and secondly to satisfy the real-time requirements of the applications. The design of such new architecture configurations is becoming an increasingly intricate process. Architectures are becoming increasingly programmable so that they can support *multi-functional* products as well as *multi-standard* products (like a mobile telephone operational worldwide). In the design of these architectures, it is no longer the performance of a single application that matters, but the performance of a *set of applications*. This impedes the design of architectures that must satisfy all the given constraints such as real-time processing, utilization of the resources, and programmability.

Before we look in more depth into the design problems of these new architectures, we illustrate the trend towards more programmable application-specific architectures by looking at the video signal processing architecture inside a modern television. In this domain, the need for architectures that can execute a set of applications is clearly present.

1.2 Video Signal Processing in a TV-set

The trend towards devising new signal processing architectures that are programmable is clearly visible in the domain of consumer market TV-sets, where the digital revolution started in the 1980s and early 1990s. In this period, the signal processing architecture inside a TV-set moved from analog processing to digital processing, whereby analog functions were replaced by digital functions. The all-digital TV architecture together with the expected continuation of transistor miniaturization in the semiconductor industry has led to increased demand for new innovative signal processing architectures.

1.2.1 TV-set

The signal path within a TV-set consists of three sections: a *front-end and baseband processing section*, a *video signal processing section*, and a *display section*, as shown in Figure 1.1. The first section transforms a TV-signal coming from an outside source like an aerial antenna into a signal that the display section visualizes on a display device, for example a Cathode Ray Tube (CRT). In between these two sections is an analog video signal processing section that makes a received signal suitable for display.

As the video signal processing section evolves from being analog to fully digital, a whole range of new applications becomes available through digital signal processing. Examples of new digital applications that improve the quality of images are luminance peaking [Jaspers and de With, 1997], noise reduction, and ghost image cancellation. Examples of complete new applications are picture reduction (Picture in Picture), picture enlargement (Zoom) [Janssen et al., 1997], and motion compensated 100Hz image conversions [de Haan et al., 1996].

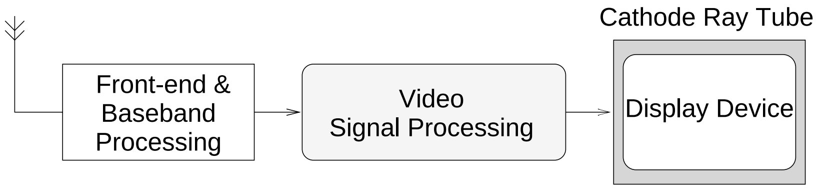


Figure 1.1. The signal path within a TV-set consists of three sections: a *front-end and baseband processing section*, a *video signal processing section*, and a *display section*.

A TV-set must also be compatible with an increasing variety of standards. Signals containing content to be displayed may be delivered through many different sources, like cable TV, a satellite dish, a computer, video recorder (VCR), or a set-top box. Traditionally these formats have complied with conventional formats like NTSC (National Television System Committee), PAL (Phase Alternating Line), or SECAM (Sequentiel à Mémoire), but increasingly often they are now also complying with emerging standards like various types of MPEG (Moving Picture Expert Group) and computer standards like SVGA (Super Video Graphics Array).

The increasing demand for new digital applications by consumers and the need for TV-sets to comply to different standards has made a more complex video signal processing section necessary in TV-sets [Claasen, 1993]. The video signal processing section of a modern high-end TV-set is shown in Figure 1.2. Each of the applications shown has its own hardware processing unit, When new applications are added, new hardware units are incorporated into the video signal processing architecture to support them. Since the customer does not select all applications at the same time, the architecture uses these units uneconomically.

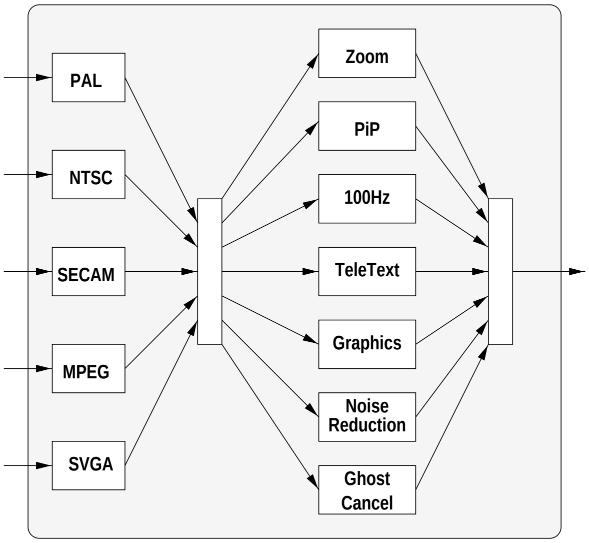


Figure 1.2. Video signal processing section of a modern high-end TV-set.

Because each hardware unit is dedicated to one particular application, it is not possible for applications to share hardware units. However, if the hardware units were made less dedicated (which we discuss later in this chapter), hardware units could be used to support more than one application. These programmable architectures of tomorrow could be reprogrammed to execute other applications with the same amount of resources, thus improving the utilization of silicon in implementing architectures. Thus, these architectures become cost effective for a set of applications.

1.2.2 Video Processing Architectures in the TV of the Future

The architecture most likely to be found in the TV-set of tomorrow [Claasen, 1993] will look similar to the structure illustrated in Figure 1.3. This architecture is built around a programmable communication network to which various elements connect. The video-in processor consists of several input channels. The video-out processor takes an output signal to the display section. A collection of processing elements (PEs) operate as hardware-accelerators, and a general purpose processor and a large high-bandwidth memory are present as well. The set of processing elements have their own controller and operate very independently (but not completely) of the general-purpose processor.

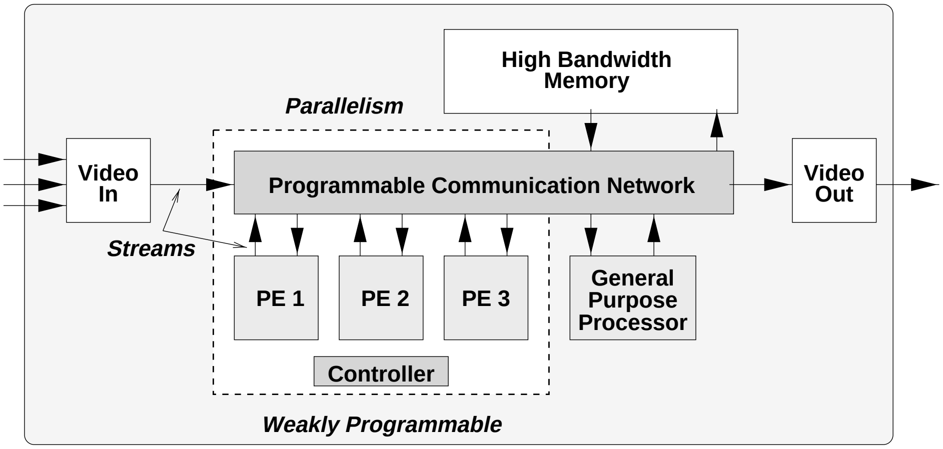


Figure 1.3. TV architecture of tomorrow.

The architecture in Figure 1.3 combines two architecture concepts: a general-purpose processor and the architecture enclosed by the dashed line. These two concepts are needed because of the large variety of timing constraints present in a TV-set, as we will show. The general-purpose processor processes reactive tasks (e.g., a user pressing on the volume button on the remote control) and control tasks (e.g., controlling the menus displayed on the screen and their functions). The dedicated processing elements, on the other hand, execute data processing tasks, i.e., the digital signal processing applications.

The large variations in timing constraints are caused by the structure of TV-signals. For example, a standard PAL video signal consists of a sequence of frames presented at a rate of 25 interlaced video *frames* per second. Each frame consists of two *fields*: an even field containing the even video lines and an odd field containing the odd video lines. A field has 312 video *lines*, and each line consists of 864 video *pixels*. Algorithms that operate on PAL video signals with different repetition periods – either pixels, lines, fields, or frames – result in very different computational requirements. Suppose an algorithm consists of 300 RISC-like operations and operates periodically on pixels, lines, fields, or frames. The vastly different computational requirements are given in Table 1.1. An algorithm operating at a field rate would perform 50×300, or 15,000 operations per second. An algorithm operating at a pixel rate would perform 13.5M×300 operations per second, which is 4 Giga operations per second.

A general-purpose processor (e.g., a RISC processor) is considered powerful enough to execute

Type of repetition rate for performing an operation	Computational requirements (Operations per second)	
Pixels	$4.05 \times 10^9 =$	300×13500000
Lines	$4.8 \times 10^6 =$	300×16000
Fields	$15 \times 10^3 =$	300×50
Frames	$7.5 \times 10^2 =$	300×25

Table 1.1. Computational requirements for different types of repetition rates for performing operations.

periodically scheduled algorithms with repetition cycles based on field or frame rates and it can also perform all kinds of reactive and control tasks. The efficiency of general-purpose processors, as measured in terms of performance per dollar, is increasing at a dazzling rate [Hennessy and Patterson, 1996]. However, they are still not powerful enough to execute periodically scheduled high-quality video algorithms with repetition cycles based on pixel or line rates.

It is inevitable that architectures used in TV-sets and in computers will eventually be merged to create new products like the PCTV [Rose, 1996] that are centered around one or more powerful general-purpose processors. Many high-end general-purpose processors already demonstrate a clear trend towards real-time processing using special multi-media instructions [Diefendorff and Dubey, 1997]. However, these general-purpose processors will not be able to furnish the processing power and communication bandwidth in the near future that is required by current and forthcoming TV applications. Consequently, at least for the coming decade, TV architectures will use a general-purpose processor in conjunction with a special architecture for the high-performance, real-time video processing.

1.2.3 Stream-Based Dataflow Architecture

The architecture enclosed by the dashed line in Figure 1.3 describes a programmable, application-specific architecture for high-performance, real-time video applications. We call this architecture *Stream-Based Dataflow Architecture* and we consider this architecture and its design in detail in this thesis.

A *Stream-Based Dataflow Architecture* consists of a global controller, a communication structure and a set of parallel-operating processor elements that are weakly programmable. These processing elements operate on streams that they exchange among each other via the communication structure under control of the global controller. The architecture exploits the following characteristics so as to be programmable, efficient, and able to satisfy real-time constraints.

Stream-based processing The processing elements operate on streams: these are a natural form with which to represent video as a one-dimensional sequence of video pixels [Watlington and Michael Bove Jr, 1997].

Coarse-grained parallelism The processing elements exploit inherent coarse-grained parallelism available within video applications by executing coarse-grained functions to achieve the required computational performance.

Weakly Programmable The processing elements implement a limited set of coarse-grained functions that provide the processing elements with the flexibility required to support a set of applications.

The granularity of the functions implemented by the processing elements highly influences the efficiency of stream-based dataflow architectures. The granularity of the functions has two extremes, as illustrated in Figure 1.4.

At one end of the spectrum, there are Application Specific Integrated Circuits (ASICs). Here, each ASIC implements one video application like luminance peaking, noise reduction, or picture in picture. An ASIC can only execute one application very efficiently in terms of silicon use. At the other end of the spectrum are programmable domain specific architectures like the VSP [Vissers et al., 1995] or Paddi [Chen, 1992]. These architectures use processing elements that implement small sets of fine-grained functions like add, subtract, and compare. These fine-grained functions allow the architectures to execute a wide range of video applications belonging to a specific application domain while satisfying real-time constraints. However, in order to support this programmability, these architectures dedicate a substantial part of their silicon area to control and communication and less to the actual computation.

For some video applications a gap of a factor of ten to twenty is found in silicon efficiency between an ASIC solution and a programmable solution [Lippens et al., 1996, 1991]. If multiple applications execute at the same time, a collection of ASICs results in the most efficient solution. However, for a given set of applications of which only one or a few execute simultaneously, the efficiency is no longer that high. In that case, a domain-specific programmable architecture is not efficient either, because it possesses more flexibility than required, i.e., the architecture can execute more applications than are present in the set of applications. This surplus of programmability is present at the expense of extra silicon.

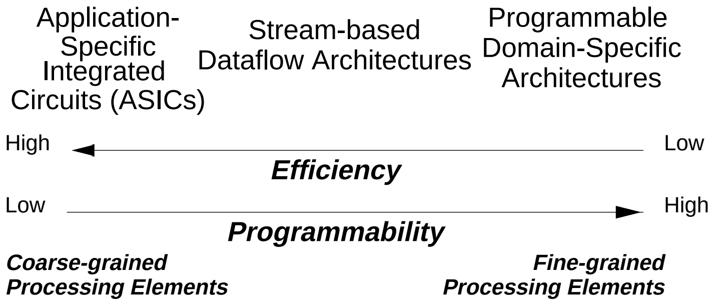

Application-Specific Integrated Circuits (ASICs)
Stream-based Dataflow Architectures
Programmable Domain-Specific Architectures
High
Low
Efficiency
Low
High
Programmability
Coarse-grained Processing Elements
Fine-grained Processing Elements

Figure 1.4. The relationship between granularity of the processing elements and the efficiency and programmability for high-performance, real-time signal processing applications. At one extreme of the spectrum, we find application-specific architectures that use very coarse-grained processing elements. They are very efficient but cannot be programmed. At the other end of the spectrum, we find programmable domain-specific architectures that use very fine-grained processing elements. They are highly programmable but have a low efficiency. Stream-based dataflow architectures result in the best balance between efficiency and programmability.

Lieverse et al. [1997] have investigated the relationship between the granularity of the processing elements and the efficiency and programmability of a stream-based dataflow architecture. For a limited set of applications, coarse-grained processing elements (i.e., processing elements implementing coarse-grained functions) result in the best balance between efficiency and programmability for high-performance, real-time signal processing applications. Therefore, we investigate stream-based dataflow architectures that make use of coarse-grained processing elements.

1.3 Design Space Exploration

In the design of programmable application-specific architectures like the stream-based dataflow architecture, a designer has to make many choices, like the granularity of the functions that the processing elements implement. Other choices include the number of processing elements to use or how much bandwidth to allocate to the communication network. These and many more choices have an effect on the overall performance of architectures in terms of utilization of the resources and throughput of the various processing elements. Furthermore, because the architecture has to execute a set of applications, particular choices might be excellent for one application in the set, but bad for another.

Nevertheless, a designer has to make choices such that the performance of the architecture is satisfactory for the set of applications while being cost effective. To achieve this goal, a designer has to make *trade-offs*, i.e., weigh one choice against another and come to a compromise. A designer must know what the design space of architectures looks like in order to make trade-offs. He acquires this knowledge by *exploring* how a particular performance metric depends on a particular parameter.

Design space exploration typically results in graphs like the one shown in Figure 1.5. It shows a simplified, idealized relationship between a measured performance of the architecture for a range of parameter values that each represent particular choices.

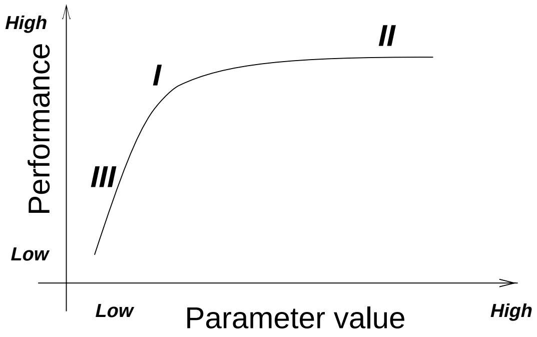


Figure 1.5. The relationship between a measured performance in the architecture and a range of parameter values. Point I indicates the best trade-off between a particular performance and parameter values. Point II shows a marginal increase of the performance at a large cost and point III shows a deterioration of the performance.

In such graphs, there is often a point (I) representing the best trade-off between a particular performance and parameter value; it is this point, the so-called knee, that a designer seeks. Selecting a larger parameter value (i.e., closer to II) would result in a marginal increase in the performance at a large cost. On the other hand, selecting a lower parameter value (i.e., closer to III) would result in a deterioration of the performance.

We observed designs of programmable application-specific architectures at both Philips Research and Delft University of Technology. Based on these observations, we conclude that it is current practice to design these architectures at a low level of detail. Architectures are described in a standard hardware description language like VHDL or Verilog and consequently the designer ends up with too much detail. The level of detail involved limits the design space of these architectures that the designer can explore to look for better trade-offs. Therefore, a designer has difficulty finding a balance among the many choices present in architectures during the design process. This makes it hard to produce architectures that are both cost effective and programmable enough to support a set of applications.

In this thesis we present a design approach called the “Y-chart approach” that overcomes the limitations introduced by the low level of detail currently involved in the design of programmable architectures and which makes it possible to make better trade-offs in architectures. This approach leads to an environment in which designers can first exercise architecture design, making design choices in architectures quantitative using *performance analysis*.

This involves the modeling of architectures to determine a performance model and the evaluation of this performance model to determine performance numbers, thus providing data for various performance metrics of a processor. In addition, by systematically changing design choices in a Y-chart environment, a designer should be able to systematically explore part of the design space of an architecture. This exploration provides the designer with the insight required for making the trade-offs necessary for a good architecture for a given set of applications.

Performance analysis can take place at different levels of detail and designers should exploit these different levels to narrow down the design space of architectures in a stepwise fashion. A Y-chart environment is used in each step, but at different levels of detail. Therefore, when the modeling and evaluation of architectures is relatively inexpensive, a large part of the design space can be explored. By the time the modeling of an architecture as well as the evaluation of this model become expensive, the design space has been reduced considerably and it contains the interesting design points.

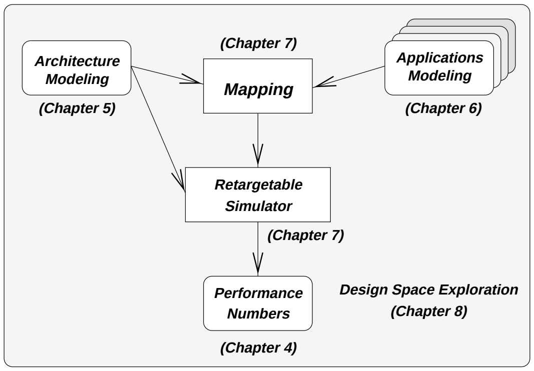


Figure 1.6. The Y-chart environment we develop in this thesis. The highlighted components of the chart are labeled with references to the chapters in which they are elaborated.

The Y-chart approach structures the design process of programmable architectures. It takes into account right from the beginning the three core issues that play a role in finding good programmable application-specific architectures, i.e., the architecture, the mapping, and the set of applications.

A general outline for the Y-chart environment as we will develop it in this thesis for stream-based dataflow architectures in order to explore their design space is shown in Figure 1.6. We will do this at a level of abstraction higher than that offered by standard hardware description languages.

1.4 Main Contributions of Thesis

The main contributions of this thesis are:

Y-chart Approach The pivotal idea in this thesis is to provide a means with which the effect of design choices on architectures can be quantified. This resulted in the formulation of the Y-chart approach, which quantifies design choices by measuring the performance.

We implemented the Y-chart approach in a Y-chart environment for the class of stream-based dataflow architectures. This led to the following contributions:

Architecture Template for Stream-based Dataflow Architectures We present the stream-based dataflow architecture as a class of architectures. We show that all choices available within this class of architectures can be described by means of an architecture template that has a well defined design space. We derive architecture instances from the architecture template.

Modeling Architectures in a Building Block Approach We use a high-level performance-modeling tool to render performance analysis at a high abstraction level. Using this method, we model the complete class of stream-based dataflow architectures while still obtaining cycle-accurate results. We use object oriented programming techniques extensively together with the performance-modeling tool to construct building blocks. Using these building blocks, we construct executable architecture instances of the architecture template of stream-based dataflow architectures.

Stream-based Functions (SBF) Model We develop a new model of computation: the Stream-based Functions (SBF) Model. This model combines Kahn Process Networks [Kahn, 1974] with the a specialization of the Applicative State Transition (AST) Model proposed initially by Backus [1978]. The SBF model is well suited for describing digital signal processing applications at different levels of granularity, ranging from fine-grained to coarse-grained. We also develop a simulator called SBFsim for the SBF model.

ORAS We develop the Object Oriented Retargetable Simulator (ORAS). Because ORAS is retargetable, it can derive a full functional simulator for all feasible architectures from the class of stream-based dataflow architectures. The derived simulator operates very fast in terms of real computer time while it also executes the correct functional behavior of an application. The execution speed is a prerequisite to performing an exploration of the design space of the class of stream-based dataflow architectures in a limited amount of time.

Mapping Approach We introduce the notion of the model of architecture. Using this notion, we formulate a mapping approach in which we postulate that the model of computation should match the model of architecture of stream-based dataflow architectures. Only in this way is a smooth mapping possible. Moreover, it leads to an interface between applications and architecture. This interface permits the execution of applications onto an architecture instance without its being necessary to modify the original application when mapping the application onto an architecture instance.

Design Space Exploration We use a generic design space exploration environment to perform an exploration of stream-based dataflow architectures. We also formulate the problem of selecting a set of parameters that result in a particular architecture which satisfies the design objectives for a set of applications.

Different Design Cases We use the Y-chart approach in two different design cases of programmable architectures. One design case is the *Prophid* video-processor architecture [Leijten et al., 1997] for high-performance video applications and the other is the *Jacobium* processor architecture [Rijpkema et al., 1997] for array signal processing applications.

1.5 Outline of the Thesis

The organization of this thesis is as follows. We present the class of stream-based dataflow architectures in detail and formulate the main problem statement in Chapter 2. Our solution approach – the Y-chart approach – is presented and discussed in Chapter 3. The chapters that follow each discuss a particular aspect of the Y-chart environment for the class of stream-based dataflow architectures.

We explain what performance analysis entails in Chapter 4. We look into the aspects that determine the performance of a system, thus laying the foundation for performance analysis at a high level of abstraction. We use a high-level performance analysis method to carry out the performance analysis. Using this method, we set up an object oriented modeling approach leading to the notion of building blocks.

In Chapter 5, we look at how to model stream-based dataflow architectures using the building blocks discussed in Chapter 4. We construct the building blocks of the stream-based dataflow architecture in detail. We explain how we describe a class of architectures using a parser. Finally, we explain how to program stream-based dataflow architectures such that they execute a particular application.

To model applications, we introduce a new model of computation, called *Stream-Based Functions* (SBF). In Chapter 6, we explain what the SBF Model of computation comprises. We also explain how the SBF Model is embedded in other well-established models of computations. How we implemented this model of computation using C++ and Multi-threading is also described.

In Chapter 7 we combine aal these aspects to construct the *Object oriented Retargetable Architecture Simulator* (ORAS). We combine the work on architecture modeling presented in Chapter 5 with the work on application modeling in Chapter 6 to construct a retargetable simulator that executes at high speed. We also look in detail how we can easily map an application onto an architecture instance.

In Chapter 8 we explain what design space exploration (DSE) implies. We embed the ORAS developed in Chapter 7 in a generic design space exploration environment. We elaborate on the statistical tools that the generic DSE environment uses to perform design space exploration efficiently. We also explain how we actually embed the ORAS in the generic DSE environment.

We investigate in Chapter 9 two cases in which a programmable architecture is developed and we use in their design the Y-chart environment developed in this thesis. One case concerns the Prophid architecture for high-performance video application developed at Philips Research; the other concerns the Jacobium architecture for a set of array signal processing applications developed at the Delft University of Technology.

We conclude this thesis in Chapter 10 with our conclusions.

Bibliography

John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613 – 641, 1978.

Pradip Bose and Thomas M. Conte. Performance analysis and its impact on design. *IEEE Computer*, 31(5):41 – 49, 1998.

D.C. Chen. *Programmable Arithmetic Devices for High Speed Digital Signal Processing*. PhD thesis, University of California at Berkeley, California, Department of Electrical Engineering and Computer Science, 1992.

T.A.C.M. Claasen. Technical and industrial challenges for signal processing in consumer electronics: A case study on TV applications. In *Proceedings of VLSI Signal Processing, VI*, pages 3 – 11, 1993.

G. de Haan, J. Kettenis, and B. Deloore. IC for motion compensated 100hz TV, with smooth motion movie-mode. In *IEEE Transactions on Consumer Electronics*, volume 42, pages 165 – 174, 1996.

Keith Diefendorff and Pradeep K. Dubey. How multimedia workloads will change processor design. *IEEE Computer*, 30(9):43 – 45, 1997.

John Hennessy and Mark Heinrich. Hardware/software codesign of processors: Concepts and examples. In Giovanni De Micheli and Mariagiovanna Sami, editors, *Hardware/Software Codesign*, volume 310 of *Series E: Applied Sciences*, pages 29 – 44. NATO ASI Series, 1996.

John L. Hennessy and David A. Patterson. *Computer Architectures: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., second edition, 1996.

Johan G.W.M. Janssen, Jeroen H. Stessen, and Peter H.N. de With. An advanced sampling rate conversion algorithm for video and graphics signals. In *IEE Sixth International Conference on Image Processing and its Applications*, Dublin, 1997.

Egbert G.T. Jaspers and Peter H.N. de With. A generic 2-D sharpness enhancement algorithm for luminance signals. In *IEE Sixth International Conference on Image Processing and its Applications*, Dublin, 1997.

Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

Edward A. Lee and David G. Messerschmitt. Engineering and education for the future. *IEEE Computer*, 31(1):77 – 85, 1998.

Jeroen A.J. Leijten, Jef L. van Meerbergen, Adwin H. Timmer, and Jochen A.G. Jess. Prophid: A heterogeneous multi-processor architecture for multimedia. In *Proceedings of ICCD'97*, 1997.

P. Lieverse, E.F. Deprettere, A.C.J. Kienhuis, and E.A. de Kock. A clustering approach to explore grain-sizes in the definition of weakly programmable processing elements. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 107 – 120, De Montfort University, Leicester, UK, 1997.

Paul Lippens, Bart De Loore, Gerard de Haan, Piet Eeckhout, Henk Huijgen, Angelica Loning, Brian McSweeney, Math Verstraelen, Bang Pham, and Jeroen Kettenis. A video signal processor for motion-compensated field-rate upconversion in consumer television. *IEEE Journal of Solid-Sate Circuits*, 31(11):1762 – 1769, 1996.

P.E.R. Lippens, J.L. van Meerbergen, A. van der Werf, W.F.J. Verhaegh, B.T. McSweeney, J.O. Huisken, and O.P. McArdle. PHIDEO: A silicon compiler for high speed algorithms. In *Proc. EDAC*, pages 436 – 441, 1991.

Nicholas Negroponte. *Being Digital*. Knopf, 1995.

D.A. Patterson. Reduced instruction set computers. *Comm. ACM*, 28(1):8 – 21, 1985.

Edwin Rijpkema, Gerben Hekstra, Ed Deprettere, and Ju Ma. A strategy for determining a Jacobi specific dataflow processor. In *Proceedings of 11th Int. Conference of Applications-specific Systems, Architectures and Processors (ASAP'97)*, pages 53 – 64, Zurich, Switzerland, 1997.

Frank Rose. The end of TV as we know it. *FORTUNE*, pages 58 – 68, 1996.

VHDL. *IEEE Standard VHDL Language Reference Manual*. IEEE Computer Service, 445 Hoes Lane, P.O. Box 1331, Piscataway, New Jersey, 08855-1331, 1993. IEEE Std 1076-1993.

K.A. Vissers, G. Essink, P.H.J. van Gerwen, P.J.M. Janssen, O. Popp, E. Riddersma, and J.M. Veendrick. *Algorithms and Parallel VLSI Architectures III*, chapter Architecture and programming of two generations video signal processors, pages 167 – 178. Elsevier, 1995.

John A. Watlington and V. Michael Bove Jr. Stream-based computing and future television. *SMPTE Journal*, 106(4):217 – 224, 1997.

Chapter 2

Basic Definitions and Problem Statement

Contents

STREAM-BASED dataflow architectures were briefly introduced in the previous chapter. We showed that such architectures will be used for the high-performance video signal processing section in the TV-sets of the near future. In this chapter, we look in more detail at the structure and behavior of stream-based dataflow architectures.

Stream-based dataflow architectures are not one particular architecture, but rather a class of architectures. This class is described using an architecture template to characterize the class in a parameterized form. The architecture template has an associated design space and the design of architectures becomes the selection of parameter values representing a particular architecture within the design space. The problem designers face, however, is how to select these parameter values. How do designers select these parameter values such that they result in architectures which satisfy the many design objectives involved, such as real-time constraints, throughput of the architecture and the efficiency of resources? At the same time, these architectures also need to be programmable enough that they can execute a set of applications.

We start in Section 2.1 by defining what a stream-based dataflow architecture is. We introduce definitions of terms to clarify what we understand in the context of this thesis by specific terms. We also describe the structure and behavior of stream-based dataflow architectures and introduce the many choices present in both the structure and behavior of stream-based dataflow architectures. All these choices together characterize the class of stream-based dataflow architectures.

To describe the class of stream-based dataflow architectures, in Section 2.2 we introduce the architecture template, which characterizes this class of architectures in a parameterized form. By assigning values to all parameters in the architecture template, we can derive a particular architecture instance that makes up a design. This brings us in Section 2.3 to the goal of this thesis, which is to provide a systematic methodology for finding parameter values for an architecture template.

Dataflow architectures have already been around for many years in many different forms. We conclude this chapter in Section 2.4 by presenting related work on dataflow architectures. We identify known problems within dataflow architectures and indicate to what extent stream-based dataflow architectures exhibit these problems and how they cope with these problems.

2.1 Stream-based Dataflow Architectures

In the application domain of high performance real-time digital signal processing like video applications, the required processing power is in the order of hundreds of RISC-like operations per pixel, while data streams are in the range of 10 to 100 Msamples per second. Consequently, this kind of signal processing requires architectures that perform 10 to 100 billion operations per second and have an internal communication bandwidth of 1 – 10 Gbytes per second. By their nature, stream-based dataflow architectures comply with such requirements.

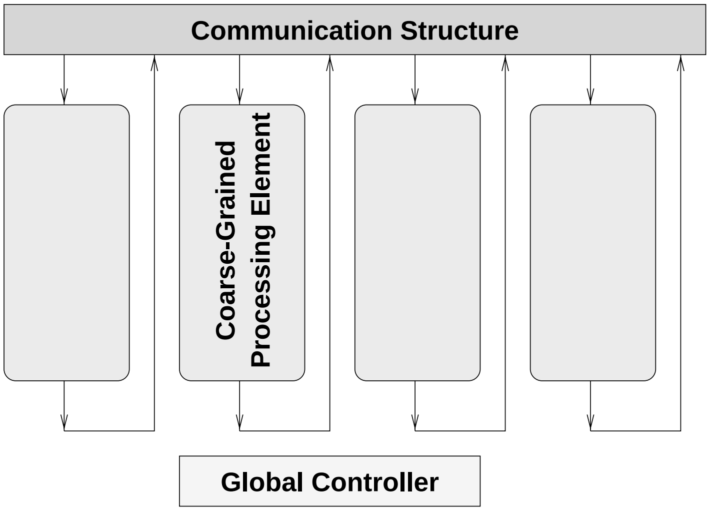


Figure 2.1. A stream-based dataflow architecture consisting of a number of *Processing Elements*, a *Communication Network*, and a *Global Controller*.

2.1.1 Definitions

A schematic representation of the structure of a stream-based dataflow architecture is depicted in Figure 2.1. It consists of a number of *Processing Elements*, a *Communication Network*, and a *Global*

Controller. The processing elements operate concurrently on streams, which we define as

Definition 2.1. STREAM
A *stream* is a one-dimensional sequence of data items. □

Unless stated differently, a data item represents a sample. In the case of video, a sample is typically a video pixel and in the case of radar, a sample is typically an integer or fixed-point value. A stream can be broken down into packets of finite length, resulting in a packet stream. We define a packet as

Definition 2.2. PACKET
A *packet* is a finite sequence of data items and is a concatenation of a header and a data part. □

In the architecture, a processing element executes a pre-selected function operating on one or more streams and producing one or more output streams. The pre-selected function is one of a finite – typically, small – number of functions present in a processing element. These functions define the function repertoire of a processing element.

Definition 2.3. FUNCTION REPERTOIRE
The *function repertoire* of a processing element describes a finite number of different pre-defined functions that the processing element can execute. □

Each processing element has a function repertoire that typically, but not necessarily, differs from the function repertoire of every other processing element. The grain sizes of the functions of the function repertoire are a measure of their complexity.

Definition 2.4. GRAIN SIZE
A function has a *grain size* expressed in terms of the equivalent number of representative RISC-like operations, like *Add*, *Compare*, and *Shift*. RISC-like functions have a grain size of one, by definition. □

The grain size provides a metric allowing us to quantify the complexity of functions. A function with a grain size of 100 is supposed to have an executable specification in terms of approximately 100 RISC-like operations. [For more information on RISC instructions, see Appendix C of Hennessy and Patterson, 1996]. We say that functions with a grain size of one are *fine-grained*. Similarly, functions with a grain size between 1 and 10 are *medium-grained* functions, and functions with a grain size larger than 10 are *coarse-grained* functions.

Although processing elements execute in parallel, each and every processing element executes only one function from its function repertoire at a time. A processing element can switch at run-time between the functions of the function repertoire, which leads to the notion of weakly programmable processing elements.

Definition 2.5. WEAKLY PROGRAMMABLE PROCESSING ELEMENT
A *weakly programmable processing element* can switch at run-time between a fixed number of pre-defined functions present in the function repertoire in such a way that only one function of the function repertoire is active at a time. □

Although the processing elements can execute functions ranging from fine-grained to coarse-grained, the functions that it executes are typically coarse-grained, to balance best between programmability and efficiency, as shown in Figure 1.4. When the granularity of functions increases, they become more dedicated and can, therefore, only be used to execute particular applications that belong to a set of applications. Consequently, coarse-grained functions are more specific than fine-grained functions used in fully programmable architectures. The weakly programmable processing elements and the grain size of the functions allow the architecture to provide just enough flexibility to support a set of applications. We assume that only one application is executed on the architecture at a time.

The global controller controls the flow of packets through the architecture. It contains a *Routing Program* with which to control the flow. This routing program indicates which processing element processes which stream, using which function from the function repertoire. By changing the routing program, we can *reprogram* the architecture to execute another application.

We define stream-based dataflow architectures as follows:

Definition 2.6. STREAM-BASED DATAFLOW ARCHITECTURES

A *Stream-based Dataflow Architecture* consists of a set of weakly programmable processing elements operating in parallel, a communication structure and a global controller. The processing elements operate on packet streams that they exchange among themselves via the communication structure controlled by the global controller. □

Stream-based dataflow architectures have a particular hierarchical structure and a particular behavior in time. We will now describe the structure and behavior of the architecture in more detail, whereby we make use of the terminology that Veen [1986] uses to describe dataflow architectures in general.

We want to emphasize that the architecture concept shown in Figure 2.1 was proposed by Leijten, van Meerbergen, Timmer, and Jess [1997] and is further developed and discussed in greater detail in Leijten [1998].

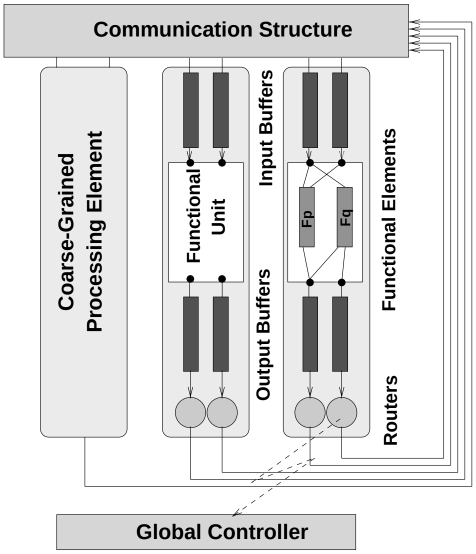


Figure 2.2. A detailed representation of the structure of a stream-based dataflow architecture. The architecture consists of a set of *processing elements*, a *communication structure*, and a *global controller*.

2.1.2 Structure

A stream-based dataflow architecture is given in Figure 2.2. The architecture consists of a set of *processing elements*, a *communication structure*, and a *global controller*. A processing element consists of a number of input and output *buffers* and *routers* and a *functional unit*. The routers interact with the global controller. The functional unit consists of a number of *functional elements* (i.e., FE_p and FE_q). Each functional element executes a function and the functions of the functional elements make up the function repertoire of a processing element. The communication structure interconnects the processing elements so they can communicate packet streams with each other under the control of the global controller.

Packets

A *packet* consists of a *data* part D and a *header* part H, as shown in Figure 2.3. The data part contains a limited amount of data samples of, for example, a sampled video signal or radar signal. The header part contains information needed to route packets through the architecture and identifies the function that needs to operate on the data part. In the stream-based dataflow architecture, we use a header format that consists of four fields, namely the base field H_b, the source field H_s, the function field H_f, and the length field H_l. They appear in the header part in the order given. The base and source fields take part in the routing of packets. The function field indicates which function of the function repertoire should process the data part of the packet. Finally, the length field indicates the number of samples contained in the data part of a packet. The lengths of the data parts of the packets do not need to be the same. A field in the header part takes the same amount of space as a sample in the data part and we distinguish between data samples and header samples.

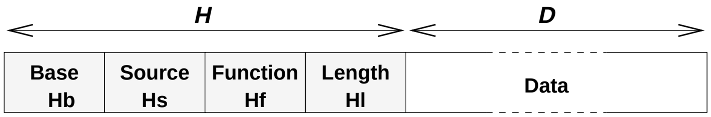


Figure 2.3. The structure of a packet consists of a *data* part D and a *header* part H.

Processing Elements

A *processing element* consists of a number of input and output *buffers,* a number of *routers* and one *functional unit,* as shown in Figure 2.4. The functional unit is the central element of a PE. It has a number of input and output ports. The input ports connect to the communication structure via buffers. The output ports connect to routers via output buffers. Each port, whether an input or an output port, connects to its own single buffer. Output buffers belonging to the same functional unit may or may not share routers.

Both input and output buffers store samples temporarily to smooth out peak in the flow of data in the architecture. Typically, though not necessarily, a buffer is a *First-In-First-Out* (FIFO) buffer. Each buffer can hold a particular number of samples. If it can hold only one sample, then the buffer is a *handshake* buffer. If it can hold a finite amount of samples larger than one, then it is a *bounded* FIFO buffer. Although it is not done in practice, theoretically a buffer can hold an unbounded number of samples, in which case the buffer is an *unbounded* FIFO buffer.

In theory, each buffer has a side from which to *read* and a side to which to *write*. In Figure 2.4, the write side is at the top of the buffers and the read side is at the bottom of the buffers. Consequently,

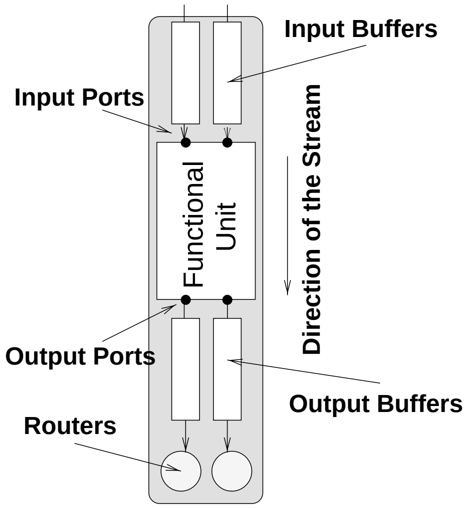


Figure 2.4. A processing element consists of a number of input and output *buffers*, a number of *routers* and one *functional unit*.

streams flow through the functional unit from top to bottom.

Routers connect to the read side of output buffers and, via the communication structure, to the input side of some input buffer. At run-time, the routers update the information in the four header fields of a packet. This affects the routing of packets and the function operating on the data part of a packet. Routers interact with the global controller to obtain this new header information. Besides changing the header fields, the router also checks whether the communication structure provides a path to the correct input buffer. If such a path exists, the router uses it to transport a packet to its new destination in the architecture.

There are two special kinds of PEs: a source PE and a sink PE. The source and sink processing elements interact with the external world of the architectures. A source processing element produces packet streams, whereas the sink processing element consumes packet streams. A regular PE connects to the communication structure with both its inputs and its outputs. A source PE connects only with outputs of the communication structure and a sink PE connects only with inputs to the communication structure.

Functional Units

A *functional unit* consists of the set $\mathcal{F}$ of *functional elements* (FEs), a *local controller*, and input and output ports, as shown in Figure 2.5. Functional Elements also have input and output ports, which bind statically to the input and output ports of the functional unit. The set $\mathcal{F}$ specifies the *function repertoire* of a processing element.

$$\mathcal{F} = \{FE_1, FE_2, \ldots, FE_x\} \tag{2.1}$$

A functional unit switches between the functional elements at run-time. This switching takes place under the supervision of the local controller. Based on the value of the function field $\mathsf{H_f}$ in the header of a packet, the local controller resolves which functional element it should activate. Although processing elements, and thus also functional units, operate concurrently, inside a functional unit only a single functional element is active at a time.

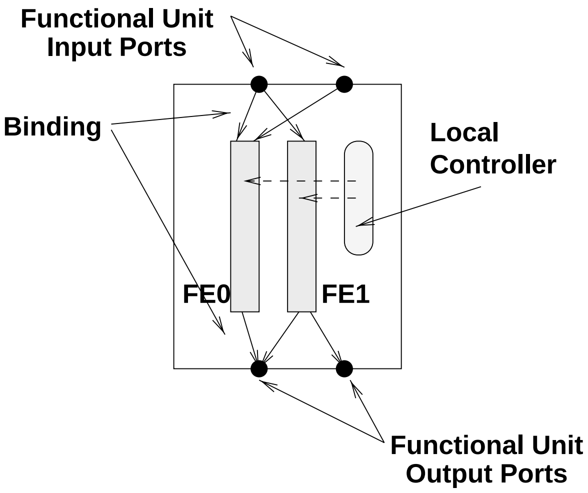


Figure 2.5. A functional unit consists of a set of *functional elements* (FE), a *local controller*, and input and output ports.

In principle, functions implemented by functional elements can have a grain size that ranges from fine-grained to coarse-grained. Fine-grained functions are RISC-like functions like *Addition*, *Subtraction*, or *Comparison*. Coarse-grained functions range from, e.g., *Finite Impulse Response* (FIR) filters and *Sample-Rate Conversion* (SRC) filters to even coarser functions like *Discrete Cosine Transformation* (DCT) functions, or *Variable Length Decoding* (VLD) functions. The functions implemented on the functional element can have parameters. A parameter of a FIR-filter, for example, could be the number of filter coefficients. A sample rate converter may have a down-sampling factor as a parameter.

A functional unit need not have both input ports and output ports. A source functional unit has no input ports and a sink functional unit has no output ports. Only source and sink processing elements contain sink and source functional units.

Functional Elements

A *functional element* consists of input and output ports, as shown in Figure 2.6, and implements a *function* using all kinds of computational elements. The function reads data from the input ports and writes data to the output ports. On the functional element, the function can have a pipelined implementation and can maintain state information.

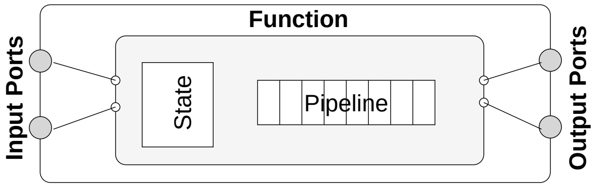


Figure 2.6. A functional element consists of input and output ports and implements a *function* using all kinds of computational elements.

The function that a functional element implements consumes samples from the input ports at a particular rate and produces samples on the output ports at a particular rate. We express this *rate* in terms of samples consumed or produced per unit time, where time is expressed in either seconds or cycles. A *cycle* is equal to a multiple of a clock-cycle. The rate at which a function reads input data or writes output data defines the *throughput* of the function. The reciprocal of throughput defines the *initiation period*. Thus, if a function has an initiation period of 2 cycles, and thus a throughput of $\frac{1}{2}$ a sample per cycle, the function reads (or writes) a single sample every 2 cycles. When a function *executes*, it produces results that correspond to the currently consumed input arguments. This takes a certain amount of time, which is defined as the *latency* of that function. A function can have a *pipelined* implementation on a functional element, which means that the function operates concurrently on different sets of input samples where the function consumes each set of input samples at different time instances. In a non-pipelined implementation of a function, the initiation period is equal to the latency of the function. Pipelining functions leads to a reduction in the initiation period of a function while the latency remains the same or increases. A pipeline has a *pipeline depth*, which indicates how many sets of samples a function concurrently operates on. The pipeline depth is equal to latency divided by initiation period.

A functional element need not have both input ports and output ports. A source functional element has no input ports and a sink functional element has no output ports. Only source and sink functional units can contain source and sink functional elements respectively.

Global Controller

The *global controller* interacts with all routers in the architecture. It provides the routers with the information they need to place in the header part of a packet. By changing the header information of packets, routers change the routing of packets through architectures and identify the function that is to operate next on the data part of a packet. The information the global controller stores via the routers in the headers is contained in a routing program that is part of the global controller. By changing this routing program, we reprogram an architecture instance to let it execute a different application.

Communication Structure

The *communication structure* consists of a number of input and output ports, and between these input and output ports it realizes communication *channels*, as shown in Figure 2.7. The channels provide parallel paths over which packets are communicated from an input port to a specific output port. All routers connect to the input ports of the communication structure. The write sides of the input buffers of all processing elements connect to the output ports of the communication structure.

The communication structure is *reconfigurable*, i.e, it can provide a channel from any input port to any output port. Routers interact with the communication structure to obtain a channel from a specific input port to a specific output port. Within the communication structure, the number of channels present indicates how many samples per cycle the communication structure can communicate in parallel from its input ports to its output ports.

2.1.3 Behavior

Besides having a particular structure, stream-based dataflow architectures also have a particular behavior in time. We first discuss the behavior of stream-based dataflow architectures in general terms. This is followed by a more detailed discussion on different behaviors that architectures can exhibit in time.

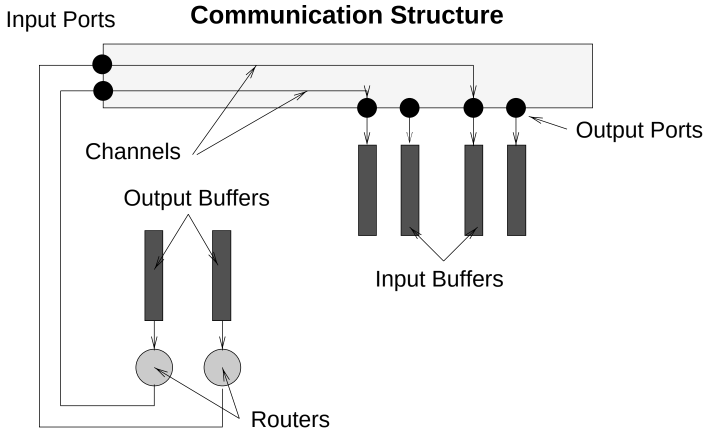


Figure 2.7. The communication structure.

Dataflow Architecture

To understand what a dataflow architecture is, we first have to look at how parallel computations are expressed as a directed graph. The earliest reference to a comprehensive theory in which parallel computations are expressed as a directed graph in which the nodes represent functions and the arcs represent data dependencies is in Karp and Miller [1966]. The functions operate concurrently on data that is represented as a *token*, which is an arbitrary data structure treated as a monolithic entity. Tokens move from one node to another over the arcs that queue tokens. This model was named *Dataflow* model of computation for the first time by Adams [1968]. A *dataflow architecture* implements the dataflow model of computation directly into hardware [1].

Definition 2.7. DATAFLOW ARCHITECTURE

A *dataflow architecture* is an implementation of the dataflow model of computation. □

A dataflow architecture can be classified as either data-driven or demand-driven [Jagannathan, 1995]. In *data-driven* dataflow architectures, the node activation, or firing, is determined solely by the availability of input data. In *demand-driven* dataflow architectures, the need for data activates a node. This node propagates its demand for data to other nodes and activates (fires) when its demands are satisfied.

Furthermore, a dataflow architecture is classified as either static or dynamic [Jagannathan, 1995]. The term *dynamic dataflow* was introduced for the first time by Dennis [1974]. In a *static* dataflow architecture, a function fires when tokens are present for all its arguments. In a *dynamic* dataflow architecture, tokens carry a *tag* that uniquely identifies a token in a stream. A function fires when tokens are present for all its arguments whose tags are identical. A dynamic dataflow architecture differs from a static dataflow architecture in its ability to execute recursion and data-dependent conditionals not known at compile-time.

A stream-based dataflow architecture is a true dataflow architecture because it implements a dataflow model of computation. A functional element implements a function that executes when tokens are present for all its input arguments. In contrast, in 'Von Neumann' architectures a flow of instructions (i.e. operators) operates on data stored in memory [Jagannathan, 1995]. The functional

[1] In addition to dataflow architectures, dataflow machines can also be discussed. We limit our discussion to dataflow architectures to indicate that our only concern is the architecture

elements in stream-based dataflow architectures operate not on individual tokens, but on streams of tokens wrapped up in packets – hence the name *stream-based* dataflow architectures.

A stream-based dataflow architecture is a data-driven dataflow architecture. The activation of the functional unit as well as of the functional element is solely determined by the availability of input tokens; i.e., functional units and functional elements schedule themselves in time on the availability of data. Stream-based dataflow architectures therefore operate without the need for a global controller that explicitly governs the order in which functional units and functional elements execute. The global controller present in stream-based dataflow architectures only governs the flow of the packets through the architecture.

A stream-based dataflow architecture is a dynamic dataflow architecture. It uses packets, each of which has a header. This header tags the data present in the data part of a packet. Because of this header, a functional unit knows which function of the function repertoire to activate. We discuss this behavior in more detail later. The dynamic behavior of functions manifests itself in two ways: functional units dynamically produce either more packets or packets of variable length.

The Sharing of Functional Units

By exploiting parallelism in computer architectures, computer scientists try to minimize the processing time of a given workload. They traditionally focus on *speed-up*, which is a measure of relative utilization, e.g., it indicates how effectively an architecture uses its resources when extending these resources while processing a workload. A linear speed-up is most desirable, implying that the architecture fully utilizes newly added resources like processing elements while processing a workload, so that it processes the workload in less time. These architectures normally process a confined workload, but in the case of the stream-based dataflow architecture process, they process a continuous workload. In addition, stream-based dataflow architectures process applications having *r*eal-time constraints, which defines a pre-defined rate at which an architecture must take in samples or produce samples, even under worst case conditions. Architectures with such real-time constraints should process streams at a specific rate – neither slower nor faster.

Due to the presence of real-time constraints, we focus in stream-based dataflow architectures on sharing, instead of processing a workload as fast as possible. In the context of stream-based dataflow, *sharing* means that a functional unit can operate on more than one stream multiplexed in time. For a functional unit we define the *sharing factor* α as

$$\alpha = \frac{e}{r} \tag{2.2}$$

where e is the rate at which a functional unit can consume tokens and r represents the rate at which the functional unit consumes a stream. The sharing factor α indicates how many streams of rate r the functional unit can consume multiplexed in time. The sharing factor is an upper bound. Switching between streams involves an overhead that reduces the number of streams which the functional unit multiplexes in practice.

When a particular architecture executes an application, adding functional units reduces the efficiency of the architecture since the architecture cannot utilize the added resource. After all, it already satisfies the real-time constraint of the application and processing a stream faster than the pre-defined rate violates the real-time constraint. Making functional units share more streams, on the other hand, allows the architecture to use fewer functional units to execute the same application. Sharing increases the *efficiency* of architectures. Since each functional unit has a function repertoire, a functional unit can share streams while at the same time each stream can select a different function from the function repertoire. The function repertoire increases the *flexibility* of architectures.

Multiplexing Different Streams on a Functional Unit

Sharing streams on a functional unit requires multiplexing of these streams in time. This multiplexing is determined either at compile-time or at run-time. In static dataflow architectures the multiplexing of streams is solved at compile-time, as, for example, is done in the Video Signal Processing (VSP) architecture [Vissers et al., 1995]. In dynamic dataflow architectures, the multiplexing is determined at run-time. Since the stream-based dataflow architecture is a dynamic dataflow architecture, the functional units have to multiplex streams at run-time.

Run-time multiplexing requires that functional units can discriminate between different streams. In stream-based dataflow architectures, streams are partitioned and each partition is embedded in a packet. Data streams are thus converted into packet streams. Apart from the data part, each packet also carries a header that allows a functional unit to discriminate between different streams. Because a packet consists of a header and a data part, the processing of a packet happens in two phases. In the first phase, the header processing takes place, and in the second phase, the data processing takes place. In stream-based dataflow architectures, the functional units take care of the first phase – the header processing – and the functional elements take care of the second phase – the data processing.

On stream-based dataflow architectures, these two phases are strictly separated. Functional units only read and write the samples of the header part, whereas functional elements only read and write samples of the data part (see Figure 2.3 showing the structure of a packet). As a result of this strict separation, functional elements seemingly consume and produce only continuous streams of data samples.

Header processing A functional unit takes care of the header processing, which involves the processing of headers at both input ports and output ports. The local controller starts reading a header from an input buffer we call the *opcode buffer* and removes the header from a packet, making the data of a packet accessible for further processing by a functional element.

Because a functional element can read samples from more than one input buffer, the local controller has to remove the headers from all the functional element's input buffers, not only from the one which serves as opcode buffer. The local controller ignores the information stored in headers read from buffers other than the opcode buffer; only the information of the header read from the opcode buffer is significant for further processing. After the local controller removed the headers from all input buffers, the functional element reads only the samples that belong to the data part of packets.

A functional element produces one or more streams on the output buffers. However, stream-based dataflow architectures operate on packets, so a stream of samples needs to be partitioned into a header part and a data part. A part of a stream of samples needs a header prefix. The local controller prepares these new headers, using the information in the opcode buffer. The local controller writes new header samples to the output buffers of the functional element. By the time the functional element is activated and writes samples to its output buffers, these data samples are preceded by new header samples and the data samples automatically form the data part of the newly created packets.

Data processing Functional elements take care of the data processing. Because the functional units take care of the headers, the functional elements read only samples contained in the data part of packets. Functional elements read data samples from their input buffers, process them, and write results to their output buffers. Because the functional unit has already written headers to these output buffers, the new data samples are automatically preceded by a header. Due to the strict separation of header and data processing, the functional elements are unaware of the presence of headers and seem to consume and produce samples of continuous streams of data.

Granularity of Switching

A functional unit switches between the functional elements in time. This switching can happen in two different modes: a packet-switching mode and a sample-switching mode. In *packet-switching* mode, a functional unit switches between functional elements at the granularity of packets. In *sample-switching* mode, a functional unit switches between functional elements at the granularity of samples. The packet-switching mode provides a coarse-grained scheduling mechanism generating less switching in time. The sample-switching mode results in a fine-grained scheduling mechanism generating more switching in time.

Packet-Switching In packet-switching mode, the functional unit switches between functional elements at the granularity of packets, as illustrated in Figure 2.8. This figure shows a functional unit with a function repertoire of two functional elements, FE_0 and FE_1. Functional element FE_0 has one input port and two outputs ports. Functional element FE_1 has two input ports and one output port because the functional elements share input and output buffers (see Section 2.1.2), the functional unit has two input buffers (B_0 and B_1) and two output buffers (B_2 and B_3). The figure shows which data pass through the buffers in time. The shaded areas in the figure represent complete packets.

In the figure, buffer B_0 is the opcode buffer of the functional unit. It contains a sequence of light gray and dark gray packets. The function field H_f of the light gray packets is equal to 0, activating FE_0. The function field H_f of the dark gray packets is equal to 1, activating FE_1. The sequence on the opcode buffer thus activates first FE_0, then FE_1, and so forth.

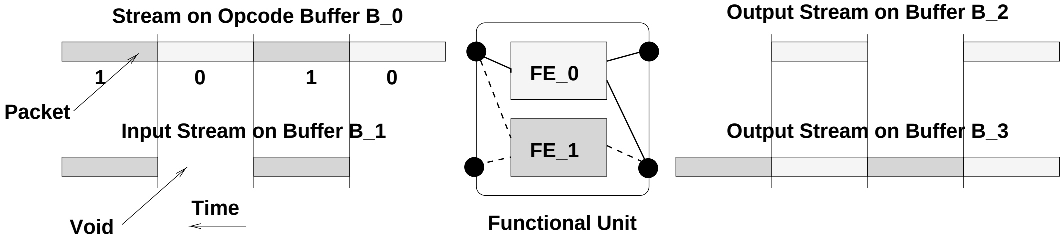

Stream on Opcode Buffer B_0
1
0
1
0
Packet
Input Stream on Buffer B_1
Void
Time
FE_0
FE_1
Functional Unit
Output Stream on Buffer B_2
Output Stream on Buffer B_3

Figure 2.8. A functional unit in packet-switching mode switches between functional elements at the granularity of packets.

In packet-switching mode, the local controller reads the header from the opcode buffer containing the function field H_f. Based on H_f, the local controller resolves which functional element to activate. Activating a functional element means that the local controller passes on the control to that functional element. The one having *control* is the only one being active in time. When the functional element finishes processing, it returns control to the local controller of the functional unit. The functional element finishes processing when it has read all data samples from the packet of the opcode buffer as indicated by H_l, the length field present in the header of the opcode buffer. By passing control back and forth between the functional unit and functional element, the functional unit switches in time at the granularity of packets between functional elements.

Suppose the local controller reads a header with H_f equal to 0. This will cause the activation of FE_0. After the local controller has completely read the header it produces two new headers on respectively buffer B_2 and buffer B_3 and passes on control to FE_0. The functional element consumes all data samples from the opcode buffer and produces new data samples on both output buffers B_2 and B_3. After FE_0 has consumed H_l data samples, it returns control to the local controller, which starts to read the next header from the opcode buffer. The next header activates FE_1. Because FE_1 has two input ports, the local controller must remove the header from the packet present in buffer B_1 as well. When the data part of the packets is accessible on both B_0 and B_1, the local controller activates FE_1. Again, FE_1 consumes the data part on both input buffers while producing new data on buffer B_3. It returns control to the local controller after it has read H_l samples from the opcode buffer B_0.

Sample-Switching In sample-switching mode, the functional unit switches between functional elements at the granularity of samples as illustrated in Figure 2.9. This figure shows a functional unit with a repertoire of two functional elements, FE_0 and FE_1. Functional element FE_0 has one input port and two output ports. Functional element FE_1 has two input ports and one output port. The functional elements in sample-switching mode must have their own input and output buffers and thus the functional units have three input buffers (B_0, B_1 and B_2) and three output buffers (B_3, B_4, and B_5). The figure shows what passes through these buffers in time. The shaded areas in the streams in the figure represent single samples that can be either a data sample or a header sample.

In sample-switching mode, each functional element must have its own input and output buffers since an individual sample cannot be associated with a particular stream. Sending a sample to a particular buffer identifies the stream to which a sample belongs and also automatically indicates which functional element should process the sample. In Figure 2.9, only light gray samples are processed by the light gray FE_0. The dark gray samples are only processed by the dark gray FE_1. In this figure, the light and dark samples do not share any buffers.

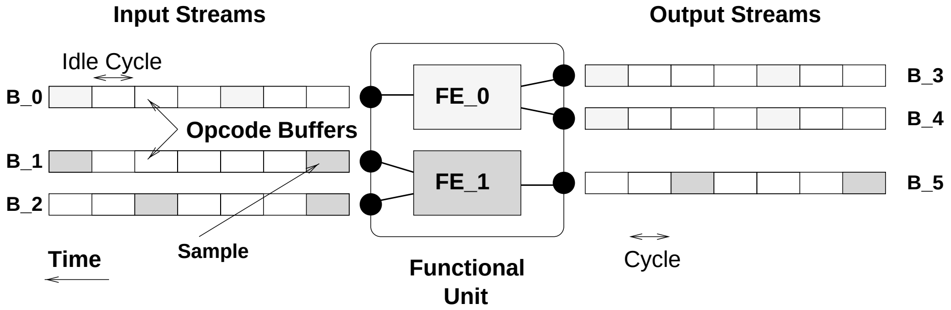


Figure 2.9. A functional unit in sample-switching mode switches between functional elements at the granularity of samples.

In sample-switching mode, each functional element has its own opcode buffer. In Figure 2.9, buffers B_0 and B_1 are the opcode buffers of FE_0 and FE_1, respectively. The local controller performs the header processing for both functional elements. After the local controller finishes header processing for one of the functional elements, it passes on control to that functional element. Contrary to packet-switching mode, in sample-switching mode more than one functional element can have control at the same time. Nonetheless, only one functional element is allowed to actually execute at a time. The local controller uses a scheduler with which to decide which functional element executes.

A functional element is active only when the functional element is ready to execute, i.e. when all its input ports contain data samples. The functional element informs the local controller that it is ready by sending a *request* for permission to execute. The local controller can receive such requests from other functional elements at the same time. Since only one functional element is allowed to execute, the local controller grants one request at a time. To decide to which functional element it should grant the request, the local controller uses a scheduler; for example, a Round Robin scheduler.

When a *granted* functional element executes, it consumes the data samples from the input buffers and produces samples on the output buffers. After a functional element has executed and its input buffers again contain data samples, it makes a new request to the local controller. After all H_l data samples have been read from the opcode buffer, a functional element returns control to the local controller of the functional element's functional unit.

In Figure 2.9, we assume that both FE_0 and FE_1 have control and read data samples from the input buffers. FE_1 has all its input data available and requests the local controller to let it execute. The local controller grants the request of FE_1 to execute, which consumes the two samples from buffer B_1 and B_2, producing a new sample on buffer B_5. Next, an *idle cycle* follows, which means that none of the functional elements of the function repertoire can execute during that cycle. After the idle cycle, the next cycle starts and FE_0 reads samples and issues a request to the local controller to let it execute. The local controller grants this request, making sure only one functional element executes at a time. This process is repeated again and again.

Communication Structure

Within the communication structure, a number of channels are present to communicate packets in parallel from input ports to output ports. By changing the number of channels, the communication structure implements different communication mechanisms ranging from a simple bus structure up to a fully interconnected network structure like the *switch matrix* [Vissers et al., 1995]. In the case of a *bus* structure, the communication structure provides only a single channel. In the case of a *switch matrix*, the communication structure provides as many channels as needed such that all output buffers can have a connection to input buffers at the same time. The bus structure and switch matrix structure both represent extremes: any structure in between could serve as a communication structure.

The samples of packets are communicated over the communication structure using a particular protocol. In stream-based dataflow architectures, two examples of protocols are a first-come-first-served protocol and a time-division-multiplexed protocol.

When the communication structure employs a *First-Come-First-Served* (FCFS) protocol, its channels make up a pool of available channels. As soon as a router requires a channel for communication, it claims an available channel. The communication structure takes a channel from the pool and sets up a path from the input port that is connected to the router to the output port that is connected to the input buffer of a processing element. The router claims the channel until it has completely transferred a packet to the input buffer. If all channels are in use and the communication structure cannot set up a path, the router has to wait until a channel becomes available.

When the communication structure employs a *time division multiplexed* (TDM) protocol, the capacity of a channel is multiplexed in time. The bandwidth of each channel is divided into N time slots of x cycles. All the routers are pre-assigned a time slot on a channel that they can use to communicate samples. The router receives the time slot even if it does not have to communicate samples at all. If a router is given its time slot, a channel is set up from the input port connected to the router to the output port connected to the input buffer of a processing element the router wants to communicate with. The router can communicate samples for x cycles. After the x cycles of the time slot have passed, the same channel is set up for another router.

The TDM protocol guarantees that the communication capacity assigned to a router is a fraction of the channel capacity. This capacity is pre-assigned and routers might not use all the assigned capacity to transport samples. Although the FCFS protocol might be able to use the communication structure more efficiently than the TDM protocol does, it is more difficult to determine whether the FCFS protocol satisfies the real-time constraints of architectures, because it does not guarantee communication capacity in time.

In the TDM protocol, routers communicate samples only in a specific time slot. This forces the communication in architectures to take place in a particular rhythm. The global controller must inform each router when it can communicate. This results in large amount of control interaction between the global controller and the routers, which makes the global controller more complicated. In the FCFS protocol, the routers themselves determine when to communicate samples and they do not interact with the global controller. This leads to a simpler global controller.

Source/Sink Modes

The source and sink processing elements interact with systems external to the architecture. A source processing element produces packet streams, whereas a sink processing element consumes packet streams.

The source processing element consists of a source functional unit and output buffers that connect to routers. A source functional unit wraps up an external stream of data samples (e.g. a TV-signal) into a packet stream. The functional unit writes headers to its output buffers followed by a certain amount of data samples (i.e. the packet length $\mathsf{H_l}$) from the external stream.

The sink processing element consists of a sink functional unit and input buffers. A sink functional unit removes the headers from a packet stream, producing a continuous stream of data samples (e.g. an output TV-signal). The functional unit reads packets from its input buffers, removes the headers and writes the data samples into an external stream of data samples.

A source functional unit consumes an external stream at a particular rate while producing a packet stream at a particular rate. Within stream-based dataflow architectures, the source functional unit uses two different modes to produce packet streams, namely a stream mode and a burst mode.

In *stream mode*, the source functional unit produces a packet stream at the same rate at which it consumes the external stream. In *burst mode*, the source functional unit produces a packet stream at a higher rate than that at which it consumes the external stream. In this case, the source functional unit accumulates samples until it can send out a packet at the higher rate.

In the stream mode, the source functional unit generates packets in a continuous stream, whereas in burst mode the source functional unit generates packets in bursts. Both modes are illustrated in Figure 2.10. Every black dot shown represents a data sample; every light gray dot, a header sample; and every square, a cycle. At the left-hand side are two external streams with a rate of 1 sample per 4 cycles each. These streams are consumed by two source functional units that produce packet streams in burst mode (i.e., the box at the top of Figure 2.10) and in stream mode (i.e., the box at the bottom of Figure 2.10). In burst mode, the source functional unit produces a packet stream at a rate of 1 sample per cycle. The source functional unit has packed the data samples closer together. The data of the external stream is preceded by 4 header samples. In stream-mode, the source functional unit produces a packet stream at a rate of 1 sample per 4 cycles, which is the same rate as the external stream. Again, the data samples are preceded by 4 header samples, of which only the last is still visible in the figure.

The mode in which a source functional unit operates affects the multiplexing of streams on functional units. The stream mode is used when functional units operate in sample-switching mode. In Figure 2.10, samples are produced in stream mode at a rate of 1 sample per 4 cycles. If a functional unit has a throughput of 1, it can multiplex at maximum of four packet streams each having a rate of 1 sample per 4 cycles.

If the same functional unit were to operate in packet-switching mode, it would generate samples at a rate that is simply too slow. For every sample that the functional unit consumes, it has to wait 3 cycles before it can read the next sample. However, when functional units operate in packet-switching mode,

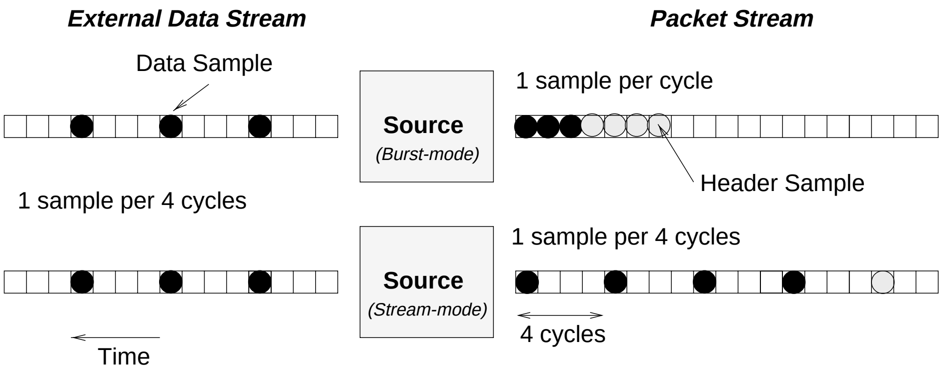


Figure 2.10. Two source functional units that operate on two external streams with a rate of 1 sample per 4 cycles.

a source functional unit operates in burst-mode. In general, the burst mode can be used effectively in both packet-switching mode and in sample-switching mode. The stream mode can only be used effectively in sample-switching mode.

A sink functional unit, like a source functional unit, operates in either stream mode or in burst mode. In the stream mode, it continuously consumes samples at a particular rate. In burst mode, it consumes samples in bursts.

Different Arbitrage Schemes of the Global Controller

Routers interact with the global controller to obtain new header information. A router starts an interaction with the controller by issuing a *request* to the global controller. The global controller *grants* this request and only then does a router actually exchanges new header information with the global controller.

Within stream-based dataflow architectures, there are multiple routers but only one global controller. It is very likely that routers try to issue requests concurrently to interact with the global controller. A global controller can serve only a certain capacity of requests, i.e., it has an instantaneous *service capacity*. The service capacity is in practice one request at a time. Consequently, the global controller needs to arbitrate between different requests. Within the class of stream-based dataflow architectures, the global controller can use different arbitrage schemes. Three examples of such an arbitrage scheme are first-come-first-served, round robin, and time division multiplex.

In the *First-Come-First-Served* (FCFS) arbitrage scheme, requests of routers are stored in a FIFO buffer. The global controller deals with the requests in the FIFO buffer in the order in which they arrive. In the *Round Robin* arbitrage scheme, the requests of routers are stored at a fixed position in an array. The global controller grants the requests in the array in a circular order as they appear. Finally, the *Time Division Multiplex* (TDM) arbitrage scheme is a FCFS arbitrage scheme with additional functionality with which to control which router can communicate samples over the communication structure at which time slot.

When the global controller grants a request, it takes a certain amount of time to actually transport new header information to a router. This is the *service time* of the global controller.

2.2 The Class of Stream-based Dataflow Architectures

In the previous section, we have discussed the structure and behavior of stream-based dataflow architectures. Recall that we have not been dealing with any stream-based dataflow architecture in particular, but with a hypothetical architecture in which some choices, both structural and behavioral, are left open, the so-called *architectural choices*. Instead of discussing any particular stream-based dataflow architecture, we consider a *class of stream-based dataflow architectures* that is characterized by the available architectural choices.

Table 2.1 shows the parameters corresponding with the architectural choices for the stream-based dataflow architecture as well as the range of values a parameter may take. For example, the number of processing elements is represented by the parameter P_0. From the range given (i.e. $\{\ 2\ldots100\}$) for P_0, we may construct any architecture instance consisting of 2 up to 100 processing elements. Another example is parameter P_1, which indicates the number of channels present in the communication structure. As a final example, the parameter P_2 represents the two protocols a communication structure can employ. The two protocols are represented by the enumerated set $\{$TDM, FCFS$\}$.

Architectural Choices		*Parameter*	
		Name	Range
Architecture	Number of processing elements	P_0	$\{2\ldots100\}$
Communication Structure	Number of Channels	P_1	$\{1\ldots100\}$
	Type	P_2	{TDM, FCFS}
	Time Slot Length	P_3	{ 1... 100 }
Sink/Source Function Units	Type	P_4	{ Stream, Burst }
	Packet Length	P_5	$\{1\ldots1000000\}$
Global Controller	Type	P_6	{FCFS, Round Robin, TDM}
	Service Capacity	P_7	$\{1\ldots3\}$
	Service Response Time	P_8	$\{1\ldots10\}$
Per processing element PE_x			
Functional Units	Number of Functional Elements	P_x	{ 1... 10 }
	Type	P_{x+1}	{Sample-Switching, Packet-Switching}
Buffers	Type	P_{x+2}	{Handshake, Bounded, Unbounded}
Input Buffers	Capacity Buffer	P_{x+3}	$\{1\ldots1000\}$
Output Buffers	Capacity Buffer	P_{x+4}	$\{1\ldots1000\}$
Routers	Type	P_{x+5}	{Share, Single}
Per Function Element FE_y			
Functional Elements	The Function it executed	P_y	{ any valid function }
	Latency	P_{y+1}	{ 1... 10 }
	Throughput rate	P_{y+2}	{ 1... 10 }
	Number of input arguments	P_{y+3}	{ 1... 3 }
	Number of output arguments	P_{y+4}	{ 1... 3 }
Per Function f_n			
Functions	Function Parameters	$P_n\ldots P_{n+z}$	depending on the function

Table 2.1. The architectural choices present within the class of stream-based dataflow architectures.

2.2.1 Architecture Template

To describe the class of stream-based dataflow architectures, we introduce the notion of an architecture template, which we define as

Definition 2.8. ARCHITECTURE TEMPLATE
An *architecture template* is a specification of a class of architectures in a parameterized form. □

An architecture template represents a class of architectures by expressing the available architectural choices in terms of a parameter set P. A particular architecture within the class of architectures is named an architecture instance, which is defined as

Definition 2.9. ARCHITECTURE INSTANCE
An *architecture instance*, or *design*, is the result of assigning values to all architectural choices, i.e., parameters of the architecture template. □

2.2.2 Design Space

The set of architectural choices in Table 2.1 describes many different architecture instances of the stream-based dataflow architecture template. All these different instances together define the *design space* D of the class of stream-based dataflow architectures. For an architecture template AT with parameters $p_0 \dots p_n$, the design space D is

$$D = P_0 \times P_1 \times \cdots \times P_n \tag{2.3}$$

We can select a point I in D by selecting for each parameter P_0 to P_n a particular value p such that

$$I = (p_0, p_1, \dots, p_n) \in D \tag{2.4}$$

For this point I we can derive an architecture instance AI of AT

$$AT(I) \rightarrow AI \tag{2.5}$$

2.3 The Designer's Problem

So far, we have considered the stream-based dataflow architecture template from the point of view of structure and behavior. Now we turn to the designer who needs to design architectures, and in particular stream-based dataflow architectures, that can execute a set of applications. The designer has to produce one or more architectures that satisfy *design objectives* such as real-time constraints, throughput of the architecture, resource efficiency, and programmability. This list can easily be extended with, for example, power consumption and silicon area. We will confine ourselves to the design objectives related to real-time constraints, throughput constraints, and utilization.

Definition 2.10. FEASIBLE DESIGN
A *feasible design* satisfies all stated design objectives. □

Thus the design problem a designer faces consists of selecting parameter values for an architecture template such that a feasible design is found. This design task becomes increasingly difficult as architectures become more programmable. A design should satisfy the design objectives not only for a single application, but for a given set of applications.

A designer's approach to constructing programmable applications-specific architectures, and in particular, stream-based dataflow architectures, is what we consider in this thesis. The problem statement for which this thesis will provide a systematic methodology is:

Central Problem Statement of this thesis
Given an architecture template AT and its design space D, a set of applications, and the design objectives, provide a method to find parameter values of the architecture template such that a feasible design results.

We discuss a feasible design and not an optimal design. The complexity of these architectures is such that we simply do not know what an optimum might be. Nevertheless, of all possible feasible designs, a designer is interested in the design best satisfying the stated design objectives. When one feasible design has a better utilization than that of other feasible designs, it is considered a better design.

Design Objectives
Real-time Constraints
Throughput Constraint
Utilization
Silicon Area Cost
Power Consumption
Programmability to Support a Set of Applications

Table 2.2. Possible design objectives when finding an architecture for a set of applications.

2.3.1 Exploring the Design Space of Architectures

We observe that a design approach is not yet available for programmable architectures like stream-based dataflow architectures. As we will show, design approaches currently in use seem unable to help designers in evaluating the consequences of architectural choices with respect to the design objectives. The level of detail introduced in current design approaches narrows down the design space that the designer can explore. As a result, a designer cannot make the trade-off between the many architectural choices shown in Table 2.1.

Because the designer cannot make the necessary trade-offs, the resulting design is either *over-designed* or *under-designed*. In the first case, a design under-utilizes its resources and it is thus a more expensive design than necessary; in the second case, the design cannot satisfy the imposed design objectives.

2.3.2 Problems in Current Design Approaches

Designers typically start by sketching rough outlines of architectures on paper, which we call *paper architectures*. Because it is difficult to evaluate particular architectural choices at this stage, designers tend to make qualitative statements about these paper architectures that lack a sound basis on which to validate them. This makes designing feasible architectures an art rather than solid engineering.

Once designers have selected a particular paper design, they tend to fill in the detail of a design in a hardware description language like VHDL or Verilog. This design approach is sometimes referred to as *Golden Point Design* [Richards, 1994]. In this design approach, designers very quickly become preoccupied by details of the design. They do not thoroughly evaluate the consequences of decisions at a high enough level. For example, the high level decision to use a particular kind of buffer of a certain size presumably has a far greater impact on the overall performance of a design than do the details of how a buffer communicates with a processing element.

As the description of the architecture becomes more detailed, it also becomes harder to change the structure or behavior of the architecture. This makes it more difficult to try out other architectural choices. Since simulation becomes an increasingly important tool when more detail is added, another consequence of a more detailed architecture is that the simulation speed drops dramatically, especially in the context of high performance video and radar applications. Consequently, only a few alternatives can be evaluated in a reasonable amount of time. An associated problem with adding more and more details is that many problems arise that were initially unforeseen. To solve these problems, designers take ad-hoc decisions that can seriously affect the ability of architectures to satisfy their stated design objectives.

Although designers develop programmable architectures, the development of the architecture is often unrelated to the development of the set of applications. This further impedes making trade-offs on behalf of supporting a set of applications. It is therefore very much the question if the final architecture is sufficiently programmable to execute the complete set of applications while satisfying the design objectives.

Another design approach is to transform digital signal processing applications in various steps of refinement into a particular architecture. This approach is used, for example, in the *systolic array* design community [H.T. Kung and Leiserson, 1978; H.T. Kung, 1982] and the *wavefront* design community [S.Y. Kung et al., 1982; S.Y. Kung, 1988]. Using all kinds of transformations like folding and partitioning, the application is transformed into an architecture (see for example, figure 6.41 in S.Y. Kung [1988]). For a set of applications, this refinement method would lead to different optimal architectures for different applications. It lacks the ability to deal effectively with making trade-offs in favor of the set of applications.

Finally, the hardware/software codesign design approach relies on an architecture template to design architectures. One application is partitioned into hardware and software parts that fit onto this architecture template [De Micheli and Sami, 1996]. Nevertheless, the architecture template typically used in hardware/software codesign is too restrictive: it often consists of one or more co-processors, a bus, and a programmable element like a CPU or a DSP. The architecture template is unable to handle the demanding requirements of high-performance signal processing.

The Hardware/Software Codesign design approach lacks the ability to deal effectively with making trade-offs in favor of the set of applications. Only recently has the problem of executing a set of multi-media applications been addressed in the hardware/software codesign community (by Kalavade and Subrahmanyam [1997]), albeit only for cyclo-static DSP applications.

2.4 Related Work on Dataflow Architectures

Work on dataflow architectures started in the early 1970s. It is Jack Dennis who is regarded as the originator of the concepts of dataflow architectures [Dennis and Misunas, 1975]. People working on dataflow architectures had great visions of plentiful computing power provided by powerful parallel architectures based on simple computing principles [Veen, 1986]. Dataflow architectures, especially dynamic dataflow architecture (see Section 2.1.3), were researched intensively during the 1980s and the early 1990s. During the 1980s, the now famous Manchester Dataflow Architecture was conceived and implemented [Gurd et al., 1985]. It typifies the dynamic dataflow architectures of that time. The Manchester Dataflow Architecture was a general-purpose computer that used fine-grained, homogeneous functional units (i.e., each functional unit executes the same set of functions). This makes it possible for the architecture to do load balancing of the functional units, giving speed-up that was nearly linear.

2.4.1 Implementation Problems of Dataflow Architectures

In general, dynamic dataflow architectures, although very elegant conceptually, suffer from severe implementation problems. It was found to be impossible, as we will explain, to implement the dataflow model of computation efficiently in hardware [Veen, 1986]. Dynamic dataflow architectures exploit fine-grained functional elements, giving these architectures the most flexibility. This result, however, in enormous communication requirements leading to special, bulky communication structures. Furthermore, dynamic dataflow architectures are able to execute recursion and data-dependent conditionals, causing parallelism to arise at run-time. This run-time parallelism disturbs the ordering of the tokens inside the architecture. To reorder these tokens at run-time, dynamic dataflow architectures require *matching units*. These matching units are special hardware units that utilize *tags* and a certain amount of associative memory to perform tag matching at run-time [2]. It is, however, very difficult to assess in advance how large this memory should be. In addition, matching units need to have some overflow mechanisms, which can be very bulky [Gurd et al., 1985]. Another issue making the implementation of dynamic dataflow architectures difficult is the granularity of the data flowing around the machine. This data is typically a single integer value, due to the fine-grained functions employed. Tagging these fine-grained tokens results in an enormous overhead. In addition, it is very difficult to generate unique tags for all the tokens flowing through the architecture at the same time. Of all the implementation problems mentioned here, it is the implementation of matching units that has proven in the end to be the Achilles heel of the implementation of dynamic dataflow architectures [Jagannathan, 1995].

2.4.2 Other Dataflow Architectures

As research continued, it became increasingly clear that pure dynamic dataflow architectures, in particular, fine-grained dataflow architectures, are not a viable option in the long term to become general-purpose computing architectures [Jagannathan, 1995]. Instead, research continued to focus more on hybrid data-driven/control-driven architectures called *Multithreading Architectures* or dataflow architectures that are specialized for a particular application domain, called *Domain-specific Dataflow Architectures*.

Multithreading architectures combine dataflow and von Neumann models of computation [Dennis and Gao, 1994]. These architectures hide the effects of both memory latency and synchronization waits by switching coarse-grained functions on a RISC-style processor. Examples of such hybrid architectures are the Monsoon processor [Papadopoulos and Culler, 1990] and the Sparcle processor of the Alewife project [Agarwal et al., 1995].

Domain-specific architectures are specialized for a particular application domain. These architectures try to exploit characteristics of a particular domain, for example real-time *video signal processing* (VSP). VSP applications can be described in a very natural way using streams and dataflow models of computation [Lee, 1992-1993; Jagannathan, 1995]. Dataflow architectures implement these dataflow models of computation directly into hardware. We discuss domain-specific dataflow architectures in more detail in the next section. We thereby make a distinction between static domain-specific dataflow architectures and dynamic domain-specific dataflow architectures.

[2] Architectures using these tags are called *tagged-token dataflow architectures.* The Manchester Dataflow Architecture is such a tagged-token dataflow architecture.

An extensive (historical) overview of dataflow architectures that have been developed is presented by Silc et al. [1998], who discuss the first dataflow architectures up to the latest developments in dataflow architectures.

Static Domain-specific Dataflow Architectures

Building domain-specific dataflow architectures has already proven to be a productive route. For high-performance video algorithms described in a static dataflow model of computation, both programmable architectures like the VSP [Vissers et al., 1995] and PADDI [Chen and Rabaey, 1990; Yeung, 1995] and dedicated architectures [Lippens et al., 1991] exist. The systolic/wavefront array community has also successfully exploited static dataflow concepts [H.T. Kung, 1982; S.Y. Kung et al., 1982].

Dynamic Domain-specific Dataflow Architectures

Nevertheless, with the increased focus on multi-media applications, new problems are being introduced that require dynamic execution of applications. Just consider applications related to MPEG standards for video or AC3 standard for audio. The concepts of Quality of Server (QoS) will also lead to the introduction of new applications that need to be scaled dynamically based on the availability of resources.

Architecture	Application Domain
Prophid	High performance video applications in consumer appliances [Leijten et al., 1997]
Jacobium	Array signal processing applications [Rijpkema et al., 1997]
Pleiades	Multi-standard mobile communication devices [Abnous and Rabaey, 1996]
Cheops	Television of Tomorrow [Michael Bove Jr. and Watlington, 1995]

Table 2.3. Architectures that have in common the use streams, medium to coarse-grained processing elements, and which are (weakly) programmable to support a set of applications.

New architectures like stream-based dataflow architectures are emerging to support sets of these new dynamic applications. Examples of such architectures are given in Table 2.3. These architectures all have in common that they the use streams, contain medium to coarse-grained processing elements, and are (weakly) programmable to support a set of applications.

2.4.3 Implementing Stream-based Dataflow Architectures

Let us focus more on problems encountered in the stream-based dataflow architecture, given the problems observed in implementing dynamic dataflow machines, i.e., the granularity of functions, overhead from tagging tokens and generating unique tags, and the use of matching units.

Granularity of Functions

We studied which grain size is optimal for a set of applications and presented this study in [Lieverse et al., 1997]. In this study, we explored the efficiency of a stream-based dataflow architecture quantitatively in terms of silicon use as a function of the grain size of the function implemented on functional elements. The architecture had to execute a set of 20 industrially relevant applications initially developed to execute on VSP2 architectures.

This study clearly showed that for a set of applications, the use of coarse-grained functions resulted in an implementation that was 2 to 5 times more efficient in silicon than a fine-grained dataflow architecture like the VSP2. At the same time, the architecture using the coarse-grained functions is programmable enough to execute the set of applications. The quantitative relationship found by Lieverse et al. is shown in Figure 2.11. It shows the ratio of silicon area needed by stream-based dataflow architectures to that needed by the VSP2 implementations to execute the set of applications, as a function of processing elements and cluster size. The cluster size is, like the grain size, expressed in terms of RISC-like instructions and is comparable to the grain size.

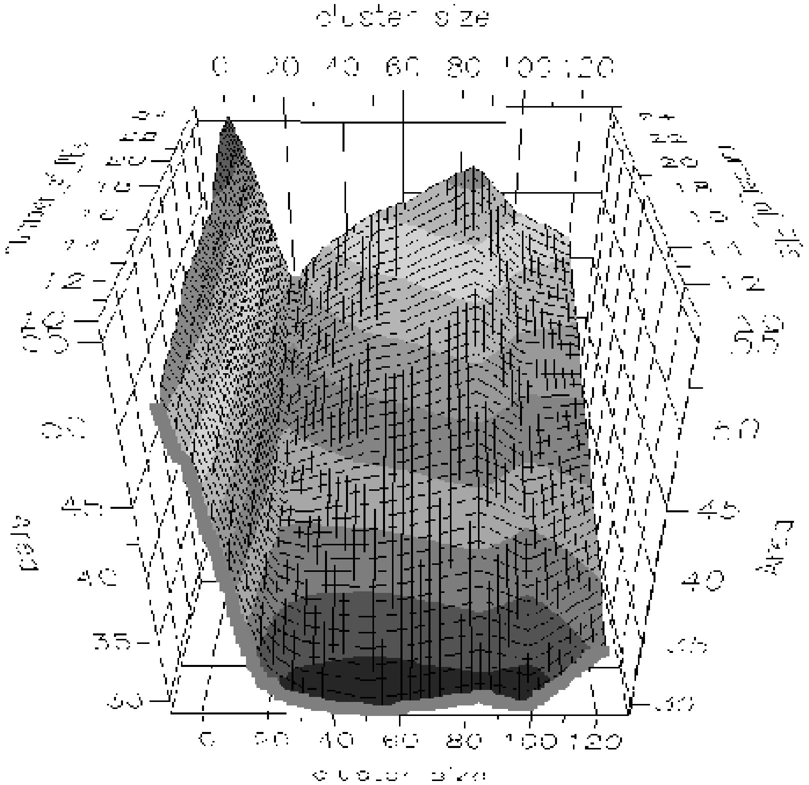


Figure 2.11. The quantitative relationship between grain size of the functional elements and silicon area of the stream-based dataflow architecture for a set of 20 industrially relevant video-applications. The figure shows the ratio of silicon area needed by stream-based dataflow architectures to that needed by the VSP2 implementations to execute the set of applications, as a function of processing elements and cluster size. The cluster size is, like the grain size, expressed in terms of RISC-like instructions and is comparable to the grain size.

Tag Overhead and Generating Unique Tags

The processing elements in stream-based dataflow architectures communicate packet streams. Packets have a data part of length L and a header part. The stream-based dataflow architecture uses four header samples to represent the header part. The header tags the data samples present in the data part. By making a packet longer, we can spread the overhead introduced by the four header samples over more samples: the longer the data part, the smaller the overhead per sample introduced by the header.

When a stream with a rate of x samples/sec is broken down into a stream of packets with length L, the overhead introduced by the four header samples results in an effective rate of the samples as

$$x * \frac{1}{1+4/L} * 100\% = \text{Effective Rate} \tag{2.6}$$

In general, the longer the packet, the smaller the overhead of a header. A packet length of 100 to 1000 is, for example, very common in video applications. A single video line takes approximately 800 samples. When using packets of 1000, the overhead is less than 0.4%. However, very long packets negatively affect the switching of packet streams on a functional unit, which leads to large buffers: other packet streams have to wait longer before being processed by a functional unit. As packets become longer, it becomes more difficult to reuse a functional unit. This reduces the programmability of functional units.

Stream-based dataflow architectures do not suffer that much from the unique tag problem. These architectures require fewer unique tags because of the use of packets and static routing of packet streams. Furthermore, knowledge about the applications can be exploited to generate unique tags. In video, for example, there are strong relations between video frames.

Matching Units

Dynamic dataflow architectures require matching units to restore order in tokens streams. In principle, stream-based dataflow architectures also have to reorder packets at run-time. We illustrate this with the example shown in Figure 2.12. It shows a functional unit in packet-switching mode, with two input ports containing three functional elements (FE_0, FE_1, and FE_2). Two input buffers connect to these two input ports. The content of these two buffers is shown over time, with each gray area representing a packet. The number beneath each packet indicates to which functional element it belongs (i.e. it is the value of $\mathsf{H_f}$ in the header). We do not show the output buffers in the figure. The example is the same as the example shown in Figure 2.8 except that it contains one additional functional element, i.e. FE_2 requires two input arguments.

The order of the packets in the OpcodeBuffer first causes the execution of FE_0, followed by the execution of FE_1, which also consumes packets from buffer B_1. The packet read from the Opcodebuffer after the execution of FE_1 again causes FE_1 to execute. Input buffer B_1, however, contains a packet belonging to FE_2 and this situation causes a *deadlock*: the functional unit will never be able to execute functional element FE_1. In such a case, a matching unit has to reorder packets in both buffers such that the packets presented to the functional unit belong to either FE_1 or FE_2. This procedure prevents deadlock from occurring.

The sharing of input buffers between various functional elements causes the reordering problem in Figure 2.12. To solve this problem, the input buffers must not be shared, as is already required in sample-switching mode. Although this is a solution for the reordering problem, it introduces as many inputs buffers as there are input arguments for the functions of the function repertoire of a functional unit, a result that might be undesirable.

Although sample-switching might solve the reordering problem through the introduction of more buffers, it also leads to many more switches of the functional elements than occurs in packet-switching mode. From work done on Multithreading architectures [Dennis and Gao, 1994], we already know that this fast switching makes implementations of sample-switching functional units difficult. Conversely,

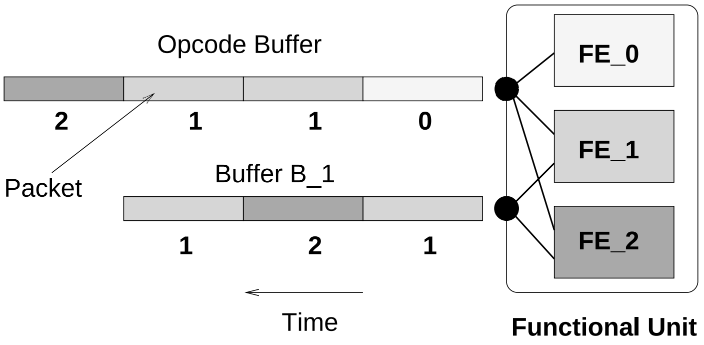


Figure 2.12. A functional unit in packet-switching mode that deadlocks due to a matching problem.

in packet-switching mode fewer switches occur. This makes the implementation of packet-switching functional units simpler.

As indicated before, the dynamic behavior of functions implemented on functional elements manifests itself in two ways: functional units dynamically produce either more packets or packets of variable length. Both ways can disrupt the order of packets in a stream and hence do not solve the reordering problem. This means further research is needed on methods guaranteeing deadlock-free execution while still obtaining efficient implementations (this will be discussed briefly in Section 9.3).

2.5 Conclusions

In this chapter, we discussed the structure and behavior of stream-based dataflow architectures. We showed that all choices available in both structure and behavior allow us to describe not just one architecture, but rather, a class of architectures. We also showed that this class of architectures could be described by means of an architecture template from which architecture instances can be derived. The problem designers face, given an architecture template with its large design space, is to find a feasible design. However, no good general design approach exists to support designers in exploring the design space of programmable architectures. As a consequence, designers cannot make the proper trade-offs leading to a good feasible design that is able to execute a set of applications. Finally, we placed the stream-based dataflow architecture in perspective regarding other work done on dataflow architectures.

Bibliography

Arthur Abnous and Jan Rabaey. Ultra-low-power domain-specific multimedia processors. In *VLSI Signal Processing, IX*, pages 461–470, 1996.

D.A. Adams. *A computational model with data flow sequencing*. PhD thesis, Stanford University, Dept. of Computer Science, 1968. TR/CS-117.

Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Ken Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceeding of the ISCA'95*, 1995.

D.C. Chen and J.M. Rabaey. Paddi: Programmable arithmetic devices for digital signal processing. In *Proceedings of VLSI Signal Processing, IV*, pages 240 – 249, 1990.

Giovanni De Micheli and Mariagiovanna Sami. *Hardware/Software Co-Design*, volume 310 of *Series E: Applied Sciences*. NATO ASI Series, 1996.

Jack B. Dennis and Guang R. Gao. Multithreaded architectures: principles, projects and issues. In Robert A. Iannucci, editor, *Multithreaded Computer Architecture, A Summary of the State of the Art*, chapter 1, pages 1–71. Kluwer Academic Publishers, 1994.

J.B. Dennis. First version of a data flow procedure language. In *Lecture Notes in Computer Science*, volume 19. Springer-Verlag, 1974.

J.B. Dennis and D.P. Misunas. A preliminary architecture for a basic data flow architecture. In *Proceedings of the 2nd Annual Symposium on Computer Architectures*, pages 126 – 132. IEEE Press, 1975.

J.R. Gurd, C.C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, 28(1):34 – 52, 1985.

John L. Hennessy and David A. Patterson. *Computer Architectures: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., second edition, 1996.

H.T. Kung. Why systolic architectures? *IEEE Computer*, 15(1), 1982.

H.T. Kung and C.E. Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Symposium*, pages 256 – 282. SIAM, 1978.

R. Jagannathan. Dataflow models. In E.Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*. McGraw-Hill, 1995.

Asawaree Kalavade and P.A. Subrahmanyam. Hardware/software partioning for multi-function systems. In *Proc. of ICCAD'97*, pages 516 – 521, 1997.

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

Edward A. Lee. Design methodology for DSP. Technical Report 92-084, University of California at Berkeley, 1992-1993.

Jeroen Leijten. *Real-Time Constrained Reconfigurable Communication between Embedded Processors*. PhD thesis, Eindhoven University of Technology, 1998.

Jeroen A.J. Leijten, Jef L. van Meerbergen, Adwin H. Timmer, and Jochen A.G. Jess. Prophid, a data-driven multi-processor architecture for high-performance DSP. In *Proc. ED&TC*, 1997.

P. Lieverse, E.F. Deprettere, A.C.J. Kienhuis, and E.A. de Kock. A clustering approach to explore grain-sizes in the definition of weakly programmable processing elements. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 107 – 120, De Montfort University, Leicester, UK, 1997.

P.E.R. Lippens, J.L. van Meerbergen, A. van der Werf, W.F.J. Verhaegh, B.T. McSweeney, J.O. Huisken, and O.P. McArdle. PHIDEO: A silicon compiler for high speed algorithms. In *Proc. EDAC*, pages 436 – 441, 1991.

V. Michael Bove Jr. and John A Watlington. Cheops: A reconfigurable data-flow system for video processing. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(2), 1995.

G.M. Papadopoulos and D.E. Culler. Monsoon: an explicit token-store architecture. In *Proceedings of of the 17th ISCA*, pages 82 – 91, 1990.

Mark A. Richards. The rapid prototyping of application specific signal processors (RASSP) program: Overview and status. In *5th International Workshop on Rapid System Prototyping*, pages 1–6. IEEE Computer Society Press, 1994.

Edwin Rijpkema, Gerben Hekstra, Ed Deprettere, and Ju Ma. A strategy for determining a Jacobi specific dataflow processor. In *Proceedings of 11th Int. Conference of Applications-specific Systems, Architectures and Processors (ASAP'97)*, pages 53 – 64, Zurich, Switzerland, 1997.

J. Silc, B. Robic, and T. Ungerer. Asynchrony in parallel computing: From dataflow to multithreading. *Journal of Parallel and Distributed Computing Practices*, 1998.

S.Y. Kung. *VLSI Array Processors*. Prentice Hall Information and System Sciences Series, 1988.

S.Y. Kung, K.S. Arun, R.J. Gal-Ezer, and D.V. Bhaskar Rao. Wavefront array processor: language, architecture, and applications. *IEEE Transactions on Computers, Special Issue on Parallel and Distributed Computers*, C-31(11):1054 – 1066, 1982.

Arthur H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4):366–396, 1986.

K.A. Vissers, G. Essink, P.H.J. van Gerwen, P.J.M. Janssen, O. Popp, E. Riddersma, and J.M. Veendrick. *Algorithms and Parallel VLSI Architectures III*, chapter Architecture and programming of two generations video signal processors, pages 167 – 178. Elsevier, 1995.

K.W. Yeung. *A Data-Driven Multiprocessor Architecture for High Throughput Digital Signal Processing*. PhD thesis, University of California at Berkeley, 1995.

Chapter 3

Solution Approach

Contents

THE architecture template of stream-based dataflow architectures offers many options to designers for instantiating an architecture, as was explained in Chapter 2. So, how can designers explore the design space of the architecture template to come to an architecture instance that satisfies his requirements? In this chapter, we discuss an approach in which designers can use performance models to make architectural choices. By evaluating the performance models of architecture instances, designers acquire performance numbers that provide them with quantitative data. They make justifiable decisions based on this quantitative data.

The idea of using performance numbers to compare architecture instances leads to the Y-chart approach presented in Section 3.1 in which the performance of architectures is analyzed for a given set of applications. Using this approach, we explain in Section 3.2 how we can explore the design space of the architecture template. The Y-chart approach consists of five components having their own particular requirements. In Section 3.3, we consider the requirements for performance analysis an mapping. In this thesis, we development a Y-chart environment for the class of stream-based dataflow architectures discussed in Chapter 2. In Section 3.4, we show each component of the Y-chart and briefly discuss what the issues are within each component. We finish this chapter in Section 3.5 by looking at general-purpose processors. After all, designers have already been constructing programmable architectures for decades. As it turns out, designers use a benchmark design approach which very much resembles the Y-chart approach.

3.1 The Evaluation of Alternative Architectures

We noticed in the previous chapter that the problem designers face when designing architectures like stream-based dataflow architectures is the many architectural choices involved (see Table 2.1 for an example). In the context of the architecture template, on what basis should designers decide that one architectural choice is better than another? We somehow have to provide designers with a basis on which they can compare architectural choices in an objective way.

3.1.1 Quantitative Data

The ranking of architectural alternatives should be based on evaluation of performance models of architecture instances. A performance model expresses how performance metrics like utilization and throughput relate to design parameters of the architecture instance. The evaluation of performance models results in performance numbers that provide designers with *quantitative data*. This data serves as the basis on which a particular architectural choice is preferred above another architectural choice in an objective and fair manner.

3.1.2 The Y-chart Approach

We propose a general scheme with which to obtain the quantitative data, as shown in Figure 3.1. This scheme, which we refer to as the Y-chart, provides an outline for an environment in which designers can exercise architectural design and was presented for the first time in [Kienhuis et al., 1997]. In this environment, the performance of architectures is analyzed for a given set of applications. This performance analysis provides the quantitative data that designers use to make decisions and to motivate particular choices. One should not confuse the Y-chart presented here with Gajski and Kuhn's Y-chart [Gajski, 1987], which presents the three views and levels of abstraction in circuit design [1]. We used the term "Y-chart" for the scheme shown in Figure 3.1 for the first time in [Kienhuis et al., 1998].

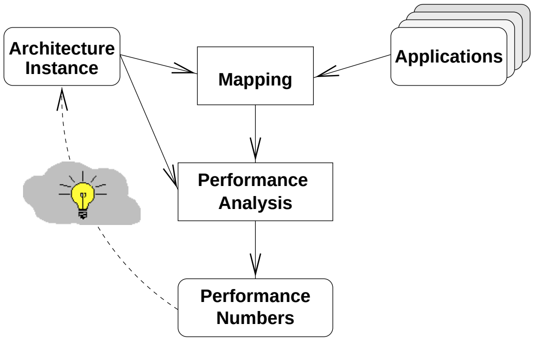


Figure 3.1. The Y-chart approach.

We define the Y-chart approach as

Definition 3.1. Y-CHART APPROACH

[1] In Gajski and Kuhn's Y-chart, each axis represents a view of a model: *behavioral*, *structural*, or *physical* view. Moving down an axis represents moving down in level of abstraction, from the *architectural* level to the *logical* level to, finally, the *geometrical* level.

The *Y-chart Approach* is a methodology to provide designers with quantitative data obtained by analyzing the performance of architectures for a given set of applications. □

The Y-chart approach involves the following. Designers describe a particular architecture instance (*Architecture Instance* box) and use performance analysis (*Performance Analysis* box) to construct a performance model of this architecture instance. This performance model is evaluated for the mapped set of applications (*Mapping* box and stack of *Applications* boxes). This yields performance numbers (*Performance Numbers* box) that designers interpret so that they can propose improvements, i.e., other parameter values, resulting in another architecture instance (this interpretation process is indicated in Figure 3.1 by the lightbulb). This procedure can be repeated in an iterative way until a satisfactory architecture for the complete set of applications is found. The fact that the performance numbers are given not merely for one application, but for the whole set of applications is pivotal to obtaining architecture instances that are able to execute a set of applications and obey set-wide design objectives.

It is important to notice that the Y-chart approach clearly identifies three core issues that play a role in finding feasible programmable application-specific architectures, i.e., architecture, mapping, and applications. Be it individually or combined, all three issues have a profound influence on the performance of a design. Besides designing a better architecture, a better performance can also be achieved for a programmable architecture by changing the way the applications are described, or the way a mapping is performed. These processes can also be represented by means of lightbulbs and instead of pointing an arrow with a lightbulb only to the architecture, we also point arrows with lightbulbs back to the applications and the mapping, as shown in Figure 3.2. Nevertheless, we focus in this thesis mainly, but not exclusively, on the loop describing the architecture design. Therefore the emphasis is on the process represented by the arrow pointing back to the architecture instance box.

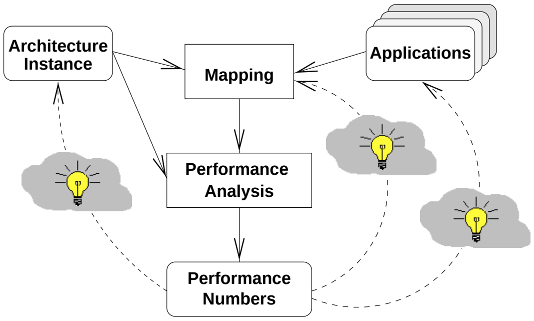


Figure 3.2. The Y-chart with lightbulbs indicating the three areas that influence performance of programmable architectures.

3.2 Design Space Exploration Using the Y-chart Approach

The Y-chart approach provides a scheme allowing designers to compare architectural instances based on quantitative data. Using the architecture template of the stream-based dataflow architecture, as introduced in Chapter 2, we can produce a set of architecture instances: we systematically select for all parameters p in the parameter set P of the architecture template AT distinct values within the conceded range of values of each parameter. Consequently, we obtain a (large) finite set $\mathcal{I}$ of points I.

$$\mathcal{I} = \{I_0, I_1, \ldots, I_n\} \tag{3.1}$$

Each point I leads to an architecture instance as given in Equation 2.5. Using the Y-chart, we map on each and every architecture instance the whole set of applications and measure the performance using particular performance metrics, a process we repeat until we have evaluated all architecture instances resulting from the set $\mathcal{I}$. Because the design space of the architecture template AT is defined by the set of parameters of AT (see Equation 2.3), in the process described above we explore the design space D of AT.

Definition 3.2. EXPLORATION

The *exploration* of the design space D of the architecture template AT is the systematic selection of a value for all parameters $P_j \in D$ such that a finite set of points $\mathcal{I} = \{I_0, I_1, \ldots I_n\}$ is obtained. Each point I leads to an architecture instance for which performance numbers are obtained using the Y-chart approach. □

When we plot the obtained parameter numbers for each architecture instance versus the set of systematically changed parameter values, we obtain graphs such as shown in Figure 1.5. Designers can use these graphs to balance architectural choices to find a feasible design.

Some remarks are in order in relation to Figure 1.5. Whereas the figure shows only one parameter, the architecture template contains many parameters. Finding the right trade-off is a multi-dimensional problem. The more parameters involved, the more difficult it will be. Note also that the curve shown in the graph is smooth. In general, designers cannot assume that curves are smooth because the interaction between architecture and applications can be very capricious. Finally, the curve in the figure shows a continuous line, whereas the performance numbers are found only for distinct parameter values. Simple curve fitting might give the wrong impression.

3.3 Requirements of the Y-chart Approach

For designers to use the Y-chart approach in the design of architectures, they must be able to describe both architecture instances (from an architecture template) and sets of applications. Designers must also be able to derive mappings for these sets of applications onto different architecture instances. The final requirement that this approach imposes on designers is that they can make models of the described architecture instance which they can then analyze to obtain performance numbers.

Each box in the schematic of the Y-chart approach denotes a discipline in its own right, having its own particular requirements. These requirements are discussed later in this thesis, with a separate chapter dedicated to each discipline and its requirements. However, the five sets of requirements are not mutually independent: the requirements for ‘performance analysis’ and ‘mapping’ stipulate the requirements for ‘architecture’, ‘applications’, and ‘performance numbers’. In this section, we consider the requirements for performance analysis and mapping in conjunction with a design process.

We have not said anything about the level at which a Y-chart should be constructed or at what level architecture instances should be modeled and applications should be specified. As we will show, there exists a trade-off between the effort required for modeling architecture instances, the effort required for evaluation of the performance of such architecture instances, and the accuracy of the acquired performance. A designer can exploit this trade-off to narrow down the design space of an architecture template in a few steps. When modeling and evaluation is relatively inexpensive, the design space can be explored extensively. By the time modeling and evaluation become expensive, the design space has already been reduced considerably and contains the design points of interest.

The Y-chart approach requires that designers deal with the mapping problem at the beginning of a design. Therefore, ideally designers should develop a mapping strategy concurrently with the development of the architecture instance and the set of applications. We formulate a mapping approach with which such mapping strategy can be developed for the stream-based dataflow architecture.

3.3.1 Performance Analysis

Performance analysis always involves three issues: a *modeling effort*, an *evaluation effort* and the *accuracy* of the obtained results [Lavenberg, 1983; van Gemund, 1996]. We address each of these in more detail in Chapter 4. Performance analysis can take place at different levels of detail, depending on the trade-offs that are made between these three issues. Very accurate performance numbers can be achieved, but at the expense of a lot of detailed modeling and long evaluation times. On the other hand, performance numbers can be achieved in a short time with modest effort for modeling but at the expense of loss of accuracy. We place the important relations between these three issues in perspective in what we call the *Abstraction Pyramid*.

The Abstraction Pyramid

The abstraction pyramid (see Figure 3.3) describes the modeling of architectures at different levels of abstraction in relation to the three issues in performance modeling. At the top of the pyramid is a designer's initial rough idea (shown as a lightbulb) for an architecture in the form of a 'paper architecture'. The designer wants to realize this architecture in silicon. The bottom of the pyramid represents all possible feasible realizations; it thus represents the complete design space of the designer's paper architecture. A discussion of the five main elements of the abstraction pyramid follows.

Cost of Modeling Moving down in the pyramid from top to bottom, a designer defines an increasing expenditure of detail of an architecture using some modeling formalism. This process proceeds at the cost of an increasing amount of effort, as indicated on the *cost of modeling* axis at the right-hand side of the pyramid. As a designer describes architectures in more detail, the number of architectural choices (i.e. the number of parameters in the architecture template) increases, expanding the basis of the pyramid. Each new architectural choice, albeit at a lower level of detail, thus further broadens the design space of the architecture.

Opportunity to Change As the designer moves down and includes more detail using the modeling formalism, the architecture becomes increasingly more specific. Details added at a given level interfere with detail added at a higher level of abstraction. Due to this intertwining of detail, the more detailed models become, the more difficult it is to make changes in architectures. Thus, the opportunity to explore other architectures diminishes. This is indicated on the *opportunity* axis at the left-hand side of the pyramid.

Level of Detail Lines intersecting the abstraction pyramid horizontally at different heights represent architecture instances modeled at various levels of detail. At the highest level of abstraction, architectures are modeled using *back-of-the-envelope models*. Models become more detailed as the abstraction pyramid is descended. The *back-of-the-envelope models* is followed by *estimation models*, *abstract executable models*, *cycle-accurate models*, and, finally, by *synthesizable VHDL models*. This represents the lowest level at which designers can model architectures.

We use the term *back-of-the-envelope model* for simple mathematical relationships describing performance metrics of an architecture instance under simple assumptions related to utilization and data rates. Estimation models are more elaborated and sophisticated mathematical relationships to describe performance metrics. Neither model describes the correct functional behavior

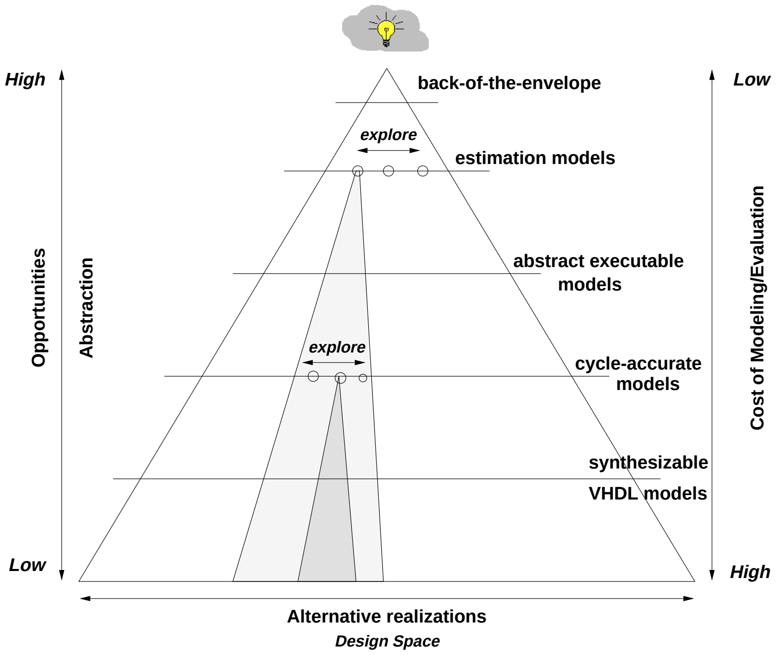


Figure 3.3. The abstraction pyramid represents the trade-off between modeling effort, evaluation speed, and accuracy, the three elements involved in a performance analysis.

or timing. The term *abstract executable model* describes the correct functional behavior first, without describing the behavior related to time. The term *cycle-accurate model* describes the correct functional behavior and timing of an architecture instance in which a cycle is a multiple (including a multiple of one), of a clock cycle. Finally, the term *synthesizable VHDL model* describes an architecture instance in such detail, in both behavior and timing, that the model can be realized in silicon.

Accuracy In the abstraction pyramid, accuracy is represented by the gray triangles. Because the accuracy of cycle-accurate models is higher than the accuracy of estimation models, the base of the triangle belonging to the cycle-accurate models is smaller than the base of the triangle belonging to the estimation models. Thus the broader the base, the less specific the statement a designer can make in general about the final realization of an architecture.

Cost of Evaluation Techniques to evaluate architectures to obtain performance numbers range from back-of-the-envelope models where analytical equations are solved symbolically, using, for example, *Mathematical* or *Matlab*, up to the point of simulating the behavior in synthesizable VHDL models accurately with respect to clock cycles. In *simulation*, the processes that would happen inside a real architecture instance are imitated. Solving equations only takes a few seconds, whereas simulating detailed VHDL models takes hours if not days. The axis at the right-hand side represents both cost of modeling and *cost of evaluation*.

Exploration

The abstraction pyramid shows the relationship between the effort required for the modeling of architecture instances at different levels of abstraction, the effort required to evaluate such an architecture model and the accuracy of the results obtained from evaluating the architecture models. When exploring the design space of an architecture template, designers may make different trade-offs between these three issues at different times.

The higher the abstraction, that is, the higher up in the abstraction pyramid, the faster a model of an architecture instance can be constructed, evaluated and changed. Conversely, the lower the abstraction, i.e. the deeper down in the abstraction pyramid, the slower a model can be constructed, evaluated and changed. The accuracy of the performance numbers obtained is less in the former case and higher in the latter case.

The trade-off designers make is as follows. Higher up in the pyramid they can explore a larger part of the design space in a given time. Although it is less accurate, it helps them to narrow down the design space. Moving down in the pyramid, the design space which they can consider becomes smaller. The designer can explore with increased accuracy only at the expense of taking longer to construct, evaluate, and change models of architecture instances.

The process of narrowing down on the design space is illustrated in the abstraction pyramid. Three circles are drawn at the level of estimation models and three are at the level of cycle-accurate models. Each circle represents the evaluation of an architecture instance. An exploration at a particular abstraction level is thus represented as a set of circles on a particular line intersecting the abstraction pyramid.

Stacks of Y-chart Environments

Due to the level-dependent trade-off between modeling, evaluation , and accuracy, designers should use different models at different levels of abstraction when exploring the design space of architectures. At each level, they should study the appropriate (part of the) design space in order to narrow it down. The Y-chart approach used at these different levels is, however, invariant: it still consists of the same elements, as shown in Figure 3.1. This leads to the following definition of a *Y-chart environment*:

Definition 3.3. Y-CHART ENVIRONMENT

A *Y-chart Environment* is a realization of the Y-chart approach for a specific design project at a particular level of abstraction. □

The different levels represented in the abstraction pyramid thus indicate that more than one Y-chart environment is needed in a design process for architectures. Instead, different Y-chart environments are needed at different levels of abstraction, forming a stack as illustrated in Figure 3.4. This figure shows three possible Y-chart environments: one each at a high, a medium, and a low level of abstraction.

In the abstraction pyramid, more than these three levels of abstraction are given. However, designers will not model architectures at all possible levels, because of limited resources (i.e. personnel and time), but will instead resort to separate models for just a few levels – say three levels – that we indicate as high, medium and low. We discuss these three levels in more detail. For each level we indicate which model from the abstraction pyramid in Figure 3.3 fits which level as well as what is typically examined at each level.

High Level Early in the design, designers make use of very abstract, high-level models – the so-called back-of-the-envelope models and estimation models – to model architecture instances. This allows them to construct many architecture instances very quickly. Designers typically use generic tools like *Matlab* or *Mathematica* to evaluate the performance of these models by solving analytic equations. These tools can compute complex equations (symbolically) within a few seconds. The

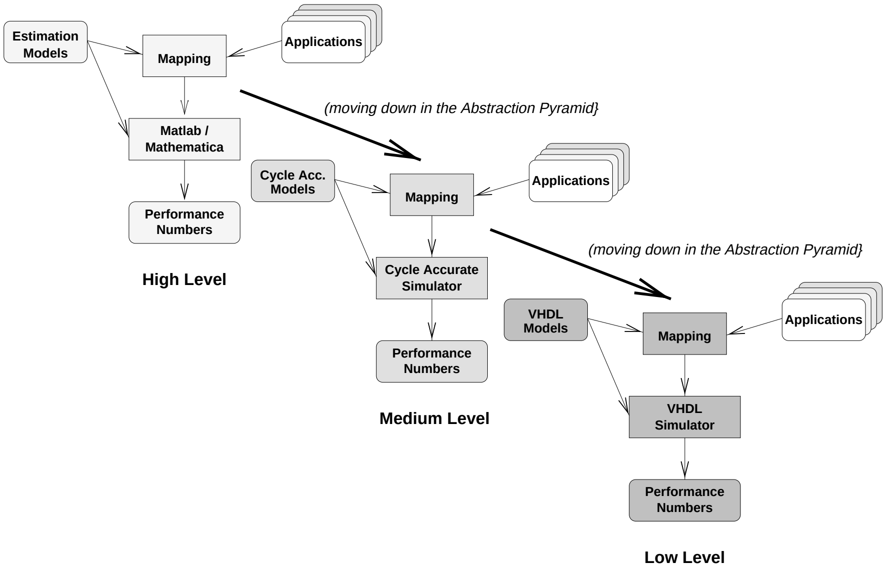


Figure 3.4. A stack of Y-chart environments, with a different model at each level of abstraction.

resulting performance numbers typically represent rough estimates for throughput, latency, and utilization. The tools evaluate the performance metrics either numerically or symbolically. At this stage, different architecture instances are compared based on the *relative* accuracy of performance numbers.

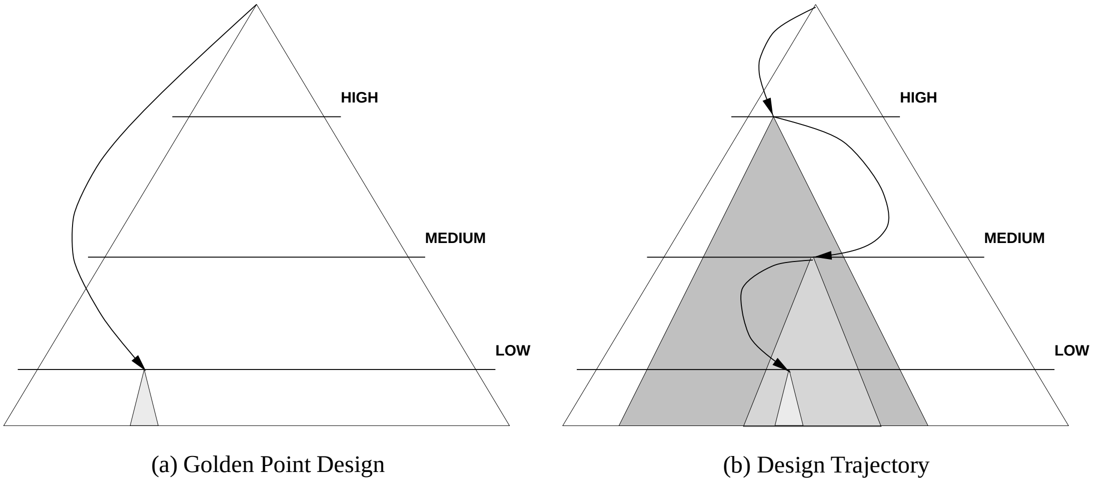
Medium Level As the design space for the architecture template narrows, designers use models that are more detailed. A designer uses abstract-executable models and cycle-accurate models at the medium level in the abstraction pyramid to describe architecture instances as required by the Y-chart approach. At a medium level of abstraction, designers can compare the performances of moderately different architectures. Both the construction of these models and the evaluation time will take longer than for the back-of-the-envelope and estimation models, since they include more detail. Models at this level require architecture simulators that typically require from minutes to hours to carry out a simulation. These simulators most likely employ discrete-event mechanisms [Ulrich, 1969; Breuer and Friedman, 1976]. The performance numbers at this level typically represent values for throughput, latency, and utilization rates for individual elements of architecture instances. As the models become more accurate, the accuracy of the performance numbers also becomes higher. At this stage, the performances of architecture instances are in the same range. The relative accuracy of performance numbers is no longer sufficient; therefore different architecture instances are compared based on *absolute* accuracy of performance numbers.

Low Level Finally, as the design space narrows down further, a designer wants to be able to compare the performance of slightly different architecture instances accurately to within a few percent. As shown in the abstraction pyramid, the designer uses detailed VHDL models to describe architecture instances as required by the Y-chart, taking significant amounts of time and resources. Designers can carry out the simulations using standard VHDL simulators. Simulation time required for these architecture instances can be as much as several days. The obtained performance numbers are accurate enough that a designer can compare differences in the performance of architecture instances to within a few percent. The performance numbers extracted at this level are not, however, high-level metrics. They do not give the utilization of the resources of the architecture instance or the throughput of a complete processing element; instead, they produce detailed waveforms. Special tools are needed to translate this detailed information back to high-level metrics.

Design Trajectory

The abstraction pyramid presents trade-offs between modeling, evaluation, and accuracy that result in a stack of Y-chart environments being used. This stack leads to a *design trajectory* in which designers can model architectures at various levels of detail. The Y-chart approach and the stack of Y-chart environments thus structure the design process of programmable application-specific architectures.

Within the design trajectory, designers perform design space exploration at each level and narrow down the design space containing feasible designs. Therefore the design space is explored when modeling and evaluation are relatively inexpensive, i.e., higher up in the abstraction pyramid. By the time modeling and evaluation become expensive, i.e., down in the abstraction pyramid, the design space is already reduced considerably and contains the interesting design points. The design trajectory in which the design space is gradually reduced differs from the golden point design approach discussed in Section 2.3.



(a) Golden Point Design

(b) Design Trajectory

Figure 3.5. (a), the golden point design approach. (b), the design approach in which designers use Y-chart environments.

In Figure 3.5(a), we show the design approach that is refer to as the *golden point design* [Richards, 1994]. Here a design is selected (the golden point) and modeled directly at a low level in the pyramid. Because hardly any exploration took place, it is first of all very much the question whether the selected point results in a feasible design. Secondly, due to the low level of detail already involved, it becomes very difficult to explore other parts of the design space, thus leading to suboptimal design. Thirdly, it is very likely that designers will be confronted with unpleasant surprises at late stages in the design process. This can lead to costly rework and slipping time schedules. In Figure 3.5(b), the design approach is shown in which designers use Y-charts at different levels of abstraction. In this approach, designers explore the design space of architectures at different levels of abstraction and gradually narrow down the design space containing potential feasible designs.

3.3.2 Mapping

Mapping pertains to conditioning a programmable architecture instance such that it executes a particular application. It leads to a program that causes execution of one application on the programmable architecture. Mapping involves assigning application functions to processing elements in architecture instances that can execute these functions. It also involves mapping the communication that takes place in applications onto communication structures in architecture instances.

We consider applications that have real-time requirements. When we map these applications onto an architecture instance, we can look at mapping "in the small" and "in the large". In mapping in the *large*, we condition an architecture instance such that it executes applications under real-time constraints and we also ensure that the applications do not deadlock. In mapping in the *small*, we only condition the architecture such that it executes applications. In this case, there is no guarantee whatsoever that real-time constraints are satisfied or that an application will not deadlock. Mapping in the small is already a difficult problem and it is the problem that we address when we talk about "mapping". Thus henceforth we mean "mapping in the small" when we say mapping [2].

We discuss the basic idea that we used to make the mapping of applications as simple as possible, i.e., that applications and architecture do – somehow – naturally fit. When mapping is simple to realize, designers will experiment more with trying out different routing programs and will be able to develop a mapping strategy concurrently with the architecture template.

Mapping Applications onto Architecture Instances

We assume that a natural fit exists if the model of computation used to specify applications matches the model of architecture used to specify architectures and if the data types used in the models are similar. This idea is shown in Figure 3.6. We first explain what a model of computation and model of architecture are and then we show how this idea results in a mapping approach.

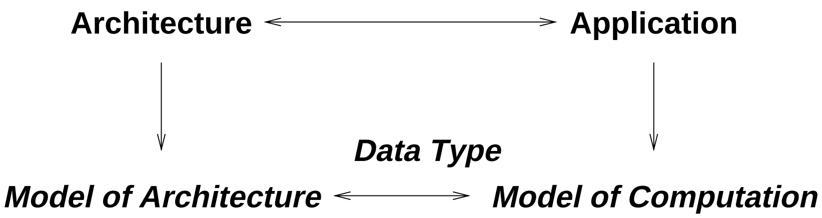


Figure 3.6. A smooth mapping from an application to an architecture only takes place if the model of computation matches with the model of architecture and when a similar kind of data type is used in both models.

[2] Mapping is sometimes also referred to as *compilation*

Model of Computation

Applications are specified using some kind of formalism that has an underlying model of computation. We define a model of computation, inspired by [Lee and et al., 1994], as

Definition 3.4. MODEL OF COMPUTATION

A *Model of Computation* is a formal representation of the operational semantics of networks of functional blocks describing computations. □

So, the operation semantics of a model of computation governs how the functional blocks interact with one another realizing computations. Many different models of computation already exist that have specific properties. Different models have proven to be very effective in describing applications in various application domains [Chang et al., 1997]. Some examples of well-known models of computation are *Dataflow Models*, *Process Models*, *Finite State models*, and *Imperative Models*.

The dataflow model of computation and some forms of process models are well suited to describing digital signal processing applications [Chang et al., 1997]. Therefore, we restrict ourselves in this thesis to the use of these models. We discuss dataflow models and process models in more detail in Chapter 6.

Model of Architecture

In analogy with a model of computation, we define the concept of model of architecture as

Definition 3.5. MODEL OF ARCHITECTURE

A *Model of Architecture* is a formal representation of the operational semantics of networks of functional blocks describing architectures. □

In this case, the functional blocks describe the behavior of architectural elements and the operational semantics of a model of architecture governs how these functional blocks interact with one another. We described the model of architecture of stream-based dataflow architectures having particular properties: these architecture are programmable and exploit parallelism and streams to satisfy tough real-time constraints, computation requirements, and bandwidth requirements.

To describe the functional blocks, e.g. the basic elements of the stream-based dataflow architectures as well as their operational semantics, we develop a modeling approach in Chapter 4 based on a high-level performance modeling approach. In Chapter 5, we use this modeling approach to describe instances of stream-based dataflow architectures.

Data Types

In both applications and architectures, data that is exchanged is organized in a particular way and has particular properties. These properties are described by a *data type*. Examples of simple data types are integers, floats, or reals. More complex data types are streams of integers or matrices.

To realize a smooth mapping, the types used in the applications should match with the types used in the architecture. If architectures use only streams of integers, the applications should also use only streams of integers. Suppose an application uses only matrices whereas an architecture instance on which we want to map the application uses only streams. Because the types do not match, we can already say that we cannot map the application directly onto the architecture instance. We first have to express the matrices in terms of streams of integers. A stream of integers, however, has very different properties from a matrix (e.g. a matrix is randomly accessible), having a profound influence on how the application should execute on the architecture (we look at this issues in more detail in Chapter 6). Therefore, to obtain a smooth mapping of applications onto architectures the data in both the applications and the architecture should use *exactly* the same kind of data types.

This does not imply that applications cannot be specified using other data types than the ones used in particular architectures. To realize a smooth mapping, however, a designer has to specify explicitly how the data types used in the application are transformed into data types used in the application. Thus, if an architecture uses only streams and an application uses matrices, a designer has to express explicitly how to transform these matrices into a stream of integers. We remark here that such a transformation can seriously affect in a negative way the performance of the application on an architecture instance.

Natural Fit

Given an application that is described using a model of computation and an architecture instance that is described using a model of architecture, when we say the application fits *naturally* onto the architecture instance, we mean that:

1. The architecture instance provides at least primitives similar to those used in the application. For example, the functions used in the application should also be found in the architecture instance.

2. The operational semantics of the architecture instance at least matches the operational semantics of the application. For example, when functional elements implement functions operating on streams in a data-driven manner, then the functions in the application should behave in exactly the same way.

3. The data types used in the application should match the data types used in the architecture instance. For example, when an architecture instance transports only streams of samples, an application should also use only streams of samples.

For applications that need to be mapped onto instances of the stream-based dataflow architectures, this natural fit has as a consequence that functions used in the applications map to functions implemented by functional elements and that the edges representing communication map to combinations of input and output buffers, routers, and the communication structure. The last consequence of a natural fit is that applications operate on streams in a data-driven manner because stream-based dataflow architectures only operate on streams in a data-driven manner.

Mapping Approach

The mapping approach we use in this thesis is to have a model of computation and a model of architecture that fit naturally as illustrated in Figure 3.6. A consequence of this mapping approach is that we have to introduce a new model of computation, which we call the Stream-Based Functions Model (SBF). We will present this model in Chapter 6. This model of computation matches the way functional elements operate within the model of architecture of stream-based dataflow architectures. We demonstrate the mapping approach in detail in Chapter 7.

3.4 Development of a Y-chart Environment

So far, we have only given a rough sketch of the generic Y-chart approach. We now focus on the development of one particular Y-chart environment for stream-based dataflow architectures as discussed in Chapter 2. We will develop the Y-chart approach shown in Figure 3.7 at the *medium level* of detail (as shown in Figure 3.4).

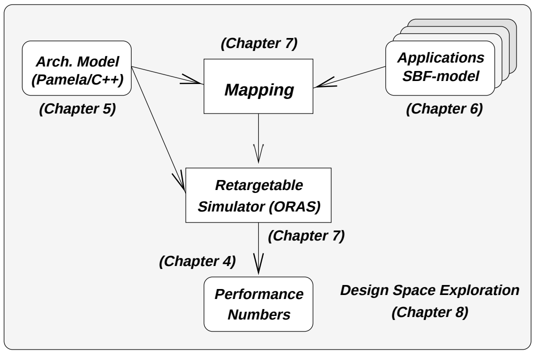


Figure 3.7. The Y-chart environment as we will build it in this thesis. The six main components (including the design space exploration) of the Y-chart environment are labeled with the number of the chapter containing the detailed discussion of the components.

The Y-chart environment uses simulation for the performance evaluation. We use simulation because it is the only technique available that handles dynamics involved in applications and architecture instances. We come back to this important point in Chapter 4.

The presentation of the Y-chart environment for the class of stream-based dataflow architectures is as follows:

Chapter 4: Performance Analysis & Performance Numbers In this chapter, we explain what performance analysis entails and we look into the aspects that determine the performance of architectures. This chapter lays the foundation for performance modeling at a high level of abstraction, which is described in the chapters that follow.

Chapter 5: Architecture Modeling In this chapter, we look at how we can model the architectural elements of stream-based dataflow architectures using the building blocks discussed in Chapter 4. We discuss the description of the architecture template of the stream-based dataflow architectures using composition rules. We also discuss in this chapter how stream-based dataflow architectures are programmed to execute a particular application.

Chapter 6: Application Modeling In this chapter, we discuss how we should model applications that operate on streams. We propose a new model of computation, called *Stream-based Functions* (SBF), with which to describe these applications. This model combines Kahn Process Networks [Kahn, 1974] with a specialization of the Applicative State Transition (AST) Model as proposed by Backus [1978]. This specialization is inspired by the AST model proposed by Annevelink [1988]. The SBF model is well suited for describing digital signal processing applications at different levels of granularity, ranging from fine-grained to coarse-grained. We also explain in this chapter how we implement this model of computation using C++ and multithreading, resulting in the simulator SBFsim.

Chapter 7: Retargetable Simulator & Mapping In this chapter, we combine the concepts presented in the previous chapters together to construct the *Object oriented Retargetable Architecture Simulator* (ORAS). We obtain the performance numbers of the architecture instance and we show that the SBF model matches the functional elements of the stream-based dataflow architecture.

Chapter 8: Design Space Exploration In this chapter, we explain how we have embedded the Y-chart environment in a design management system called *Nelsis* [ten Bosch et al., 1991]. The resulting design space exploration environment automatically constructs and systematically evaluates architecture instances, leading to the exploration of the design space of stream-based dataflow architectures.

Chapter 9: Two Design Cases In this chapter, we look at how we can use the Y-chart environment in two design cases for different application domains.

3.5 Related Work

Programmable architectures have already been being developed for decades in the domain of general-purpose processor (GPP) design. In this domain, designers from Silicon Graphics, DEC, and Intel – to name a few – develop complex architectures called *instruction-set processors* or *microprocessors*. These processors execute a word-processor application as easily as a spreadsheet application or even simulate some complex physical phenomenon. Therefore, designers working in this domain know what programmability implies in terms of (complex) trade-offs between hardware, software, and compilers.

A benchmark method is used in the design of general-purpose processors that can be cast into a particular Y-chart environment. In this section, we first show that the design of GPPs fits into our Y-chart approach. This is followed by a discussion on related work on programmable application-specific architectures.

3.5.1 Design of General-Purpose Processors

In the beginning of the 1980s, revolutionary GPPs emerged that were called RISC microprocessors [Patterson, 1985]. These processors were developed in a revolutionary way; namely, designers used extensive quantitative analysis of a suite of *benchmarks*, which is a set of applications. As a result, these architectures were smaller, faster, cheaper and easier to program than conventional architectures of that time. With the advent of the RISC microprocessors, the design of GPPs in general began to swing away from focusing purely on hardware design. Designers started to focus more on the quantitative analysis of difficulties encountered in architecture, mapping (or compiling), and the way benchmarks are written. Currently this quantitative approach is the de-facto development technique for the design of general-purpose processors [Bose and Conte, 1998] (An excellent book on the quantitative design approach of RISC based processors is “Computer Architectures: A Quantitative Approach” by Hennessy and Patterson [1996]).

We can cast the general-purpose processor architectures design approach in terms of our Y-chart approach as presented in this chapter. For example, Hennessy and Heinrich [1996] developed the MIPS R4000 microprocessor using the Y-chart depicted in Figure 3.8. The benchmarks are specified in the C programming language. These benchmarks are known as the SPECmark programs [SPECmarks, 1989]. Using a C-compiler, tuned especially to reflect the R4000 architecture, and a special architecture simulator called *Pixie*, they evaluated the performance of the R4000. The performance numbers produced by Pixie were interpreted using the *Pixstat* program. The dashed box represents the fact that the architecture in the Y-chart of Figure 3.8 is not specified as a separate entity, but that it is hard coded into the GNU GCC compiler and architecture simulator Pixie.

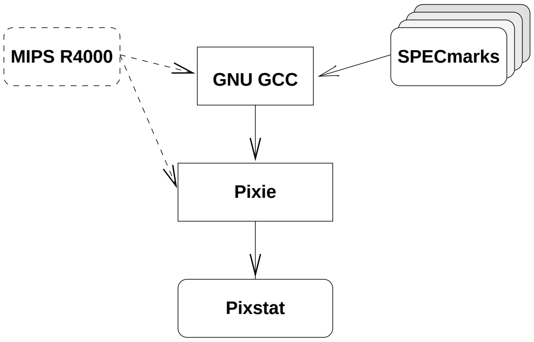


Figure 3.8. The Y-chart used in the construction of the MIPS R4000.

According to Hennessy and Heinrich, the design space of the MIPS R4000 is extremely large and evaluating alternatives is costly for three reasons: construction of accurate performance models, long simulation runs, and tuning of the compiler to include architectural changes. Therefore, different trade-offs were considered between the effort required for modeling, evaluation speed and accuracy during the design trajectory. As a result, Hennessy and Heinrich used four different simulators at increasing levels of detail in the design of the MIPS R4000, as shown in Table 3.1. One can clearly see that the simulation speed drops dramatically as more detail is added. Hennessy and Heinrich consider the top two levels of simulation to be the most critical levels in the design of processors, because they allowed them to explore a large part of the design space of the MIPS R4000, which helped them to make the best trade-offs.

Simulator	Level of Accuracy	Sim. Speed
Pixie	Instruction Set	$> 10^6$ cycles/sec
Sable	System Level	$> 10^3$ cycles/sec
RTL (C-code)	Synchronous Register Transfer	> 10 cycles/sec
Gate	Gate/Switch	< 1 cycles/sec

Table 3.1. Different Levels of Simulation used in the MIPS R4000 design.

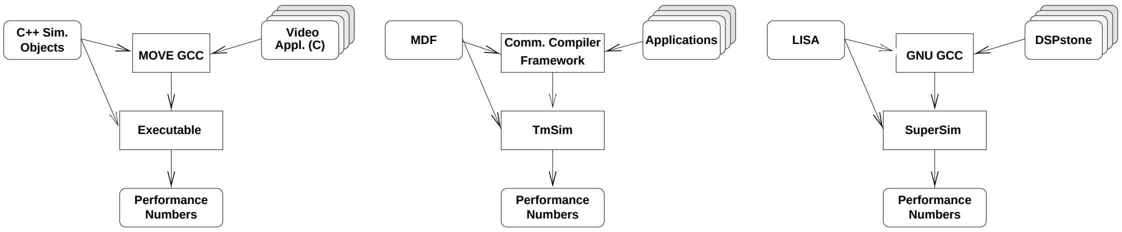
The benchmark approach leads to highly tuned architectures. By changing the suite of benchmark programs, an architecture can be made very general or the opposite, very specific. If the benchmark suite contains very different programs like a word processor application, a spreadsheet application and a compiler application, an architecture results that is optimized for a broad range of applications. Such a benchmark suite was used, for example, in the design of the R4000 by Hennessy and Heinrich. If the benchmark suite contains only video applications, a more dedicated processor architecture results that is optimized for video applications. We show next three processor designs in which the selection of benchmarks results in more application-specific processor architectures. Furthermore, the three designs show how the Y-chart approach fits into the design approach used to construct these processors.

Camposano and Wilberg [1996] used the approach for designing application-specific VLIW (Very Long Instruction Word) architectures for low-speed video algorithms like JPEG, H.262 and MPEG1.

Camposano and Wilberg use a Y-chart, as shown in Figure 3.9(a). The video applications written in C are compiled into generic RISC-instructions using the GCC/MOVE developed by Corporaal and Mulder [1991]. They annotated the RISC instructions to describe the correct behavior of the VLIW instance under design, resulting in new C-code. They compiled this C-code again using a standard C-compiler into an executable. When executed, the executable generated performance numbers describing the performance of a VLIW instance. In this project, Camposano and Wilberg used two levels of simulators, i.e., the compiled simulator and a VHDL simulator.

Sijstermans et al. [1998] used the quantitative approach for the design of the *TriMedia* programmable multi-media processor TM1000 [Rathnam and Slavenburg, 1996]. They used a Y-chart, as shown in Figure 3.9(b). They compiled a set of applications that were written in C into object-code, using a commercial compiler framework. The architecture simulator *tmsim* can simulate this object-code clock-cycle accurately. Both the compiler and the simulator are *retargetable* for a class of TriMedia architectures that they describe using a Machine Description File (MDF).

Živojnović et al. [1996] used the quantitative approach for DSP processors. They used a Y-chart, as shown in Figure 3.9(c). As a benchmark, they used a special set of C functions called DSPstone [Živojnović et al., 1994]. They mapped the benchmarks onto the retargetable simulator called *SuperSim* using a retargetable version of the GNU GCC-compiler. They described a class of DSP-architectures using the special language *LISA*.



(a) Camposano and Wilberg [1996]

(b) Sijstermans et al. [1998]

(c) Živojnović et al. [1996]

Figure 3.9. Y-chart environments used in various design projects.

The cases presented use architecture simulators developed at different levels of detail with different accuracy and execution speed. To quantify these differences, we investigated different architecture simulators for microprocessors. As given in Table 3.2, we found the average number of instructions per second to be 200,000 for an *instruction set* simulator of a MIPS R3000 [Larus, 1994], 40,000 for a *clock-cycle accurate* simulator of a TriMedia [Sijstermans et al., 1998] and 500 for an *RTL accurate* simulator of the DLX microprocessor in VHDL [Hennessy and Patterson, 1996]. The simulation speed numbers found compare with the simulation speed numbers reported by Hennessy and Heinrich in Table 3.1.

To put these simulation speeds into perspective, we assume that we want to simulate one video frame of 720×576 pixels by a simple video algorithm of 300 RISC-like instructions per pixel. The execution of the algorithm requires respectively 10 minutes, 54 minutes and more than a whole day. The numbers found emphasize again the need for different levels of simulators, because speed is a critical characteristic in the context of design space exploration in a Y-chart environment.

In the cases presented, designers make refinements to well-known architectures commonly re-

Simulator	Architecture	Language	Accuracy	Sim. Speed Instr./sec	1 Video Frame
SPIM	MIPS 3000	C	instruction	200.000	10 min.
tmsim	TriMedia	C	clock cycle	40.000	54 min.
DLX	DLX	VHDL	RTL	500	1.2 days

Table 3.2. Different Levels of Architecture Simulators.

ferred to as *load-store architectures*. Good detailed models exist for these architectures. Good compiler frameworks like GNU GCC [Stallman, 1988] and SUIF [Wilson et al., 1994] also exist for them. Consequently, designers of microprocessors can use applications written in the C-language; for example, the SPECmarks, JPEG, and MPEG [Rao and Hwang, 1997] as well as GSM and many other applications. To map these applications, designers can resort to tuning compiler frameworks to include architectural changes. The model of computation underlying the C-language fits naturally with the model of architecture of load-store architectures.

3.5.2 Design of Application-Specific Architectures

Within the domain of programmable application-specific architectures, the Y-chart approach is a new approach. In the design of emerging programmable architectures, quantifying architectural choices in architectures is by no means current practice. By casting GPP design projects into particular Y-chart environments, we showed that the use of the Y-chart approach is common practice in GPP designs. In that respect, the design of programmable application-specific architectures is moving in the direction of GPP design. Nevertheless, making the Y-chart approach available in the design of programmable application-specific architectures differs significantly from designing GPP architectures, for the following four reasons:

1. No well-defined architecture models exist for the emerging programmable application-specific architectures.
2. There does not exist a de-facto language with which to describe the high-performance digital signal processing applications naturally.
3. Because a de-facto standard is lacking, no large collection of benchmarks is quickly available.
4. Because a de-facto standard and well-established architecture models are lacking, no good and open mapping (compiler) frameworks exist that can be tailored to suit new architectures.

To construct good feasible programmable architectures requires that the design space of these architectures can be explored. Therefore Y-chart environments are needed at different levels of abstraction. In this thesis, we show that we can construct a Y-chart environment for stream-based dataflow architectures. In the chapter to come, we show how to construct a Y-chart environment for stream-based dataflow architecture at the cycle-accurate level of abstraction (medium level in Figure 3.4). In doing so, we indicate how the architecture modeling takes place, as well as how this modeling process is seriously affected by the mapping of applications and by the way the set of applications are specified.

3.6 Conclusions

The Y-chart approach is the main idea presented in this chapter. It is a methodology in which designers use quantitative data providing them with a sound basis on which to make decisions and motivate particular design choices. The realization of the Y-chart approach for a specific design project at a particular level of detail leads to a Y-chart environment. Within such an environment, we can perform a design space exploration by systematically changing design choices.

The Y-chart approach uses performance analysis to obtain quantitative data. There is an important trade-off within performance analysis between the effort required for modeling, evaluation speed, and accuracy, which we illustrated in the abstraction pyramid. As a consequence of this trade-off, designers should use a stack of Y-chart environments, each at different levels of detail. Using this stack, designers narrow down the design space of an architecture template in a stepwise fashion. By the time both modeling an architecture instance and evaluating this model become expensive, the design space is already reduced considerably and contains the interesting (feasible) designs.

The Y-chart approach takes into account the three core issues that play a role in finding good programmable application-specific architectures. These are the architecture, the mapping, and the set of applications. Therefore attention for mapping is required from the very beginning. We formulate an approach that should result in a smooth mapping of an application onto architecture instances.

Bibliography

Jurgen Annevelink. *HiFi, A Design Method for Implementing Signal Processing Algorithms on VLSI Processor Arrays*. PhD thesis, Delft University of Technology, 1988.

John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613 – 641, 1978.

Pradip Bose and Thomas M. Conte. Performance analysis and its impact on design. *IEEE Computer*, 31(5):41 – 49, 1998.

M.A. Breuer and A.D. Friedman. *Diagnosis & Reliable Design of Digital Systems*. Computer Science Press, Woodland Hill, California, 1976.

R. Camposano and J. Wilberg. Embedded system design. *Design Automation for Embedded Systems*, 1(1):5 – 50, 1996.

Wan-Teh Chang, Soonhoi Ha, and Edward A. Lee. Heterogeneous simulation - mixing discrete-event models with dataflow. *VLSI Signal Processing*, 15(1/2):127 – 144, 1997.

H. Corporaal and H. Mulder. Move: A framework for high-performance processor design. In *Proceedings of Supercomputing*, pages 692 – 701, Albuquerque, 1991.

D. Gajski. *Silicon Compilers*. Addison-Wesley, 1987.

John Hennessy and Mark Heinrich. Hardware/software codesign of processors: Concepts and examples. In Giovanni De Micheli and Mariagiovanna Sami, editors, *Hardware/Software Codesign*, volume 310 of *Series E: Applied Sciences*, pages 29 – 44. NATO ASI Series, 1996.

John L. Hennessy and David A. Patterson. *Computer Architectures: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., second edition, 1996.

Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

B. Kienhuis, E. Deprettere, K.A. Vissers, and P. van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Proceedings of 11th Int. Conference of Applications-specific Systems, Architectures and Processors (ASAP'97)*, pages 338 – 349, Zurich, Switzerland, 1997.

Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf. The construction of a retargetable simulator for an architecture template. In *Proceedings of 6th Int. Workshop on Hardware/Software Codesign*, Seattle, Washington, 1998.

James Larus. SPIM, a MIPS r2000/3000 simulator. Available from the University of Wisconsin, 1994.

Stephen S. Lavenberg. *Computer Performance Modeling Handbook*. Acadamic Press, 1983.

Edward A. Lee and et al. An overview of the Ptolemy project. Technical report, University of California at Berkeley, 1994.

D.A. Patterson. Reduced instruction set computers. *Comm. ACM*, 28(1):8 – 21, 1985.

R.K. Rao and J.J. Hwang. *Techniques and Standards for Image, Video and Audio Coding*. Prentice Hall, 1997.

S. Rathnam and G. Slavenburg. An architectural overview of the programmable multimedia processor tm1. In *Proc. Compcon*. IEEE CS press, 1996.

Mark A. Richards. The rapid prototyping of application specific signal processors (RASSP) program: Overview and status. In *5th International Workshop on Rapid System Prototyping*, pages 1–6. IEEE Computer Society Press, 1994.

F Sijstermans, E.J. Pol, B. Riemens, K Vissers, S. Rathnam, and G. Slavenburg. Design space exploration for future trimedia CPUs. In *ICASSP'98*, 1998.

SPECmarks. Spec benchmark suite release 1.0, 1989.

R.M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Cambridge, MA, 1988.

K.O. ten Bosch, P. Bingley, and P. van der Wolf. Design flow management in the NELSIS CAD framework. In *Proc. 28th ACM/IEEE Design Automation Conference*, pages 711–716, San Francisco, 1991.

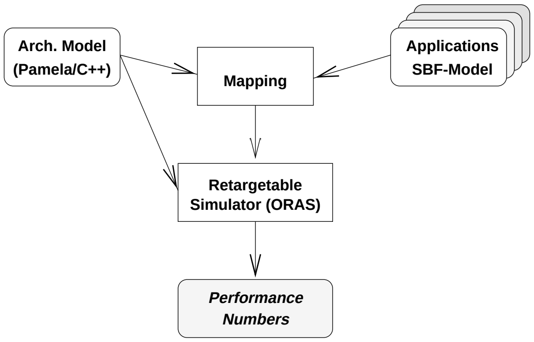
E.G. Ulrich. Exclusive simulation of activity in digital networks. *Communications of the ACM*, 12 (2):102 – 110, 1969.

Arjan. J.C. van Gemund. *Performance Modeling of Parallel Systems*. PhD thesis, Laboratory of Computer Architecture and Digital Techniques, Delft University of Technology, 1996.

V. Živojnović, J. Martinez, C. Schläger, and H. Meyr. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of ICSPAT'94*, Dallas, 1994.

Vojin Živojnović, Stefan Pees, Christian Schläger, Markus Willems, Rainer Schoenen, and Heinrich Meyr. DSP Processor/Compiler Co-Design: A Quantitative Approach. In *Proc. ISSS*, 1996.

R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, Hall M., Lam M., and J. Hennessy. SUIF: A parallelizing and optimizing research compiler. Technical report, Stanford University, 1994. CSL-TR-94-620.



Chapter 4

Performance Analysis

Contents

PERFORMANCE analysis is at the heart of the Y-chart approach. The Y-chart approach uses performance analysis to render performance numbers with which design choices can be quantified. The purpose of this chapter is to present the basic elements required for performance analysis, thus laying the foundation for the work described in succeeding chapters. These elements are used in succeeding chapters to further develop the Y-chart for stream-based dataflow architectures.

In Section 4.1, we explain the two steps involved in performance analysis: construction of a performance model and the evaluation of this model to obtain performance numbers. Section 4.2 presents a particular performance analysis method called the PAMELA method. Using an example of a simple system, we show how to use the PAMELA method to construct and simulate a performance model to obtain the desired performance numbers. The objective of the original PAMELA method differs from our objective with the Y-chart approach, as we will explain in Section 4.3. To make the PAMELA method more suitable for our purpose, we will modify it such that it becomes an object oriented modeling approach consisting of two parts, a structure part and a behavior part, as we will explain in Section 4.4. The object oriented modeling approach leads us to the development of building blocks. Using these, we can construct a performance model of a system very effectively. To simulate a performance model, we use a process scheduler which is part of PAMELA's RTL. In Section 4.5, we explain how this process scheduler simulates a performance model and how the PAMELA primitives interact with the RTL process scheduler. Performance analysis is a vast and rich field of research, mainly because performance analysis always involves modeling, evaluation, and accuracy. Different trade-offs among these three issues exist, with each leading to a different means of doing performance analysis. In Section 4.6, we look at related work on performance analysis.

4.1 Performance Analysis

The problem we consider in this chapter is how in general to determine the performance of a system for particular performance metrics like parallelism, utilization, and throughput.

4.1.1 A System

A *system* is a collection of *resources* that process a workload, whereby a *workload* is a collection of *tasks*. A system can be, for example, a chain of machines that process material according to a manufacturing protocol. These machines define the resources that process material which defines the workload of the machines. The machines process the material as specified by the manufacturing protocol. A stream-based dataflow architecture together with its set of applications is also a system. The architecture describes resources like functional elements, routers, and the communication structure. These resources process the applications that define the workloads. The resources of the architecture process these workloads according to the routing programs that are down-loaded onto the global controller.

The process to determine the performance of a system is what we call *performance analysis*. Performance analysis involves two steps: the first step is to make a performance model of a system and the second step is to evaluate this performance model to obtain performance numbers for particular performance metrics. The transformation from a system to a performance model is called *performance modeling* and the evaluation of this performance model is called *performance evaluation*. A schematic representing the steps involved in performance analysis is given in Figure 4.1.

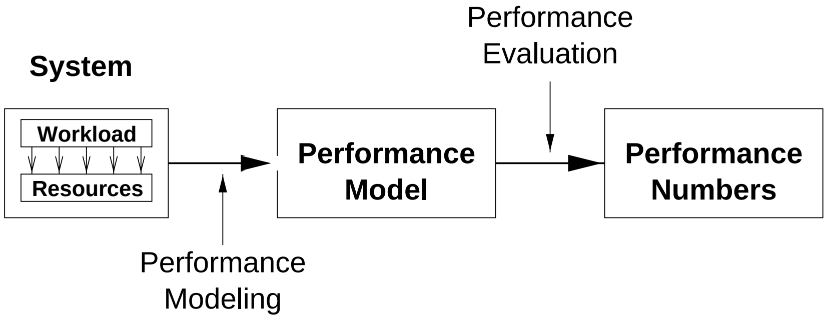


Figure 4.1. Performance analysis of a system.

4.1.2 Performance Modeling

In the performance-modeling step, we construct a *performance model* that is an abstraction of the real system it represents. This model captures only those aspects of the real system that determine the performance of that system. To be able to model a system, we need to know what determines its performance. As argued by van Gemund [1996], only four aspects determine the performance of *any* system. These four aspects define the four components of what we call the performance modeling basis.

Definition 4.1. PERFORMANCE MODELING BASIS

The *Performance Modeling Basis* (PMB) consists of the four different components that determine the performance of a system, namely *condition synchronization*, *mutual exclusion*, *execution time* and *conditional control flow*. □

These components are as follows:

Condition Synchronization (CS): relates to data dependencies between tasks or to limited resources.

Mutual Exclusion (ME): relates to the fact that two or more tasks cannot operate overlapped in time or that a resource cannot be used by more than one task at a time.

Execution Time (ET): relates to how long a task takes to execute or how long a resource is used.

Conditional Control Flow (CCF): relates to conditional selections at run-time between either tasks or resources.

Performance modeling thus means that we model a system by describing it in terms of the four components of the PMB. The first two aspects, condition synchronization and mutual exclusion, relate to synchronization between resources or tasks [Andrews and Schneider, 1983]. The other two aspects, execution time and conditional control flow, relate to the behavior of a single task or resource.

In the abstraction pyramid in Figure 3.3, we showed that performance modeling can happen at different levels of abstraction. When the four components of the PMB are described in less detail, the description becomes more abstract (i.e. higher in the pyramid), and when they are described in more detail, a description is less abstract (i.e. lower in the pyramid). In Chapter 5, we show how the four components of the PMB can describe architecture instances of the stream-based dataflow architecture.

4.1.3 Performance Evaluation

The performance evaluation step evaluates a performance model to obtain performance numbers for specific performance metrics. Examples of performance metrics are the total time required by a system to process a workload, the level of parallelism achieved in a system, the utilization of resources, and the amount of data exchanged between tasks.

A performance model can be evaluated in many ways. It can be done in a symbolic way (after transforming the performance model to a time-domain representation) or via simulation. The back-of-the-envelope and estimations models presented in the abstraction pyramid are evaluated in a symbolic way. The abstract executable models, the cycle-accurate models and synthesizable models are evaluated using simulation.

4.1.4 Accuracy

In the end, performance analysis yields performance numbers of a certain *accuracy*. If the derived value for a particular performance metric is equal to T' whereas the real performance of a system is equal to T, we say that the accuracy increases if the difference between T and T' becomes smaller. Thus, a higher accuracy indicates that the obtained performance numbers are closer to the real performance numbers. However, accuracy comes at a certain price. More accurate performance numbers are obtained at the expense of having to model a system in more detail and, typically, in having to evaluate it longer.

4.1.5 Trade-off

Performance analysis involves three issues: performance modeling, performance evaluation, and the accuracy of the obtained results. A trade-off must be found between these three aspects, as we already illustrated when discussing the abstraction pyramid in Figure 3.3.

The level of detail at which we model a system in the performance modeling step influences the accuracy, as does the kind of evaluation technique used in the performance evaluation step. Nonetheless, the accuracy of the modeling step and evaluation step cannot be considered separately. The level of detail at which we describe a performance model influences the level of accuracy obtainable in the evaluation step.

4.2 The PAMELA Method

In this thesis, we make use of a high-level performance analysis method to analyze the performance of architecture instances of stream-based dataflow architectures. This method is called the PAMELA method and was developed by van Gemund [1996] at Delft University of Technology.

The PAMELA method (PerformAnce ModEling LAnuage) includes the two techniques required to do performance analysis. It contains a technique for performance modeling and it contains a technique for performance evaluation of a performance model. The modeling technique provides a simple language in which to express performance models that can be evaluated either analytically or via simulation. We will consider in this thesis only the simulation technique.

We present the PAMELA method on the basis of an example of a simple system. Using this example, we explain how to create a performance model for this system using the PAMELA modeling technique. Next, we evaluate this model using simulation and show how we can obtain performance numbers. While working through the example, the main principles of the PAMELA method are explained.

We first explain what the PAMELA performance modeling technique encompasses and introduce a few simple primitives. These primitives allow us to describe a system using the four components of the PMB. Using these primitives and a pseudo-C language, we show how to construct a performance model for a simple system. After the modeling step, we evaluate the performance mode of the system through simulation. Based on the results of the simulation, we show how we can acquire performance numbers for specific performance metrics like total execution time, utilization, and parallelism. For a more in-depth discussion on the PAMELA method we refer to the work of van Gemund [1996].

4.2.1 A Simple System

To illustrate the PAMELA method, we model and evaluate the system given in Figure 4.2. This simple system is composed of two resources and a workload that is described by a *task graph*, which is a collection of interconnected tasks. Each node in the task graph performs a particular task (i.e., task A to task H). The directed, solid lines (or edges) between the tasks represent *data dependencies* that govern the order in which tasks have to execute. The dashed lines indicate the resource a task must possess in order to execute.

The two resources of the system are resource1 and resource2. Tasks A, G, and H require resource1 to execute and tasks B,C,D,E and F require resource2 to execute. A task does not have exclusive access to a resource, but has to share the resource with other tasks. The execution of a task takes a certain amount of time, as given between brackets in each node. For example, task A takes three time units to execute and task F requires one time unit to execute. It also means that task A keeps resource1 in its possession for three time units and task F keeps resource2 in its possession for one time unit.

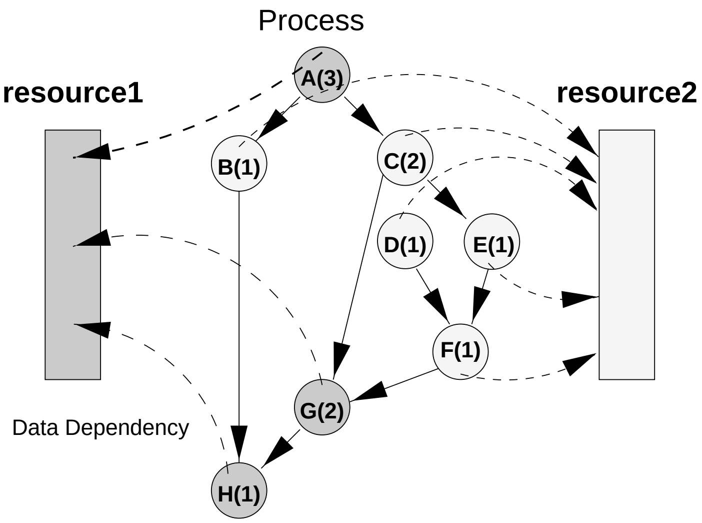


Figure 4.2. A simple system, composed of two resources (resource1 and resource2) and a workload described by a collection of tasks (A to H). The directed lines represent data dependencies governing the order in which tasks execute. The dashed lines indicate the resource a task must possess in order to execute.

4.2.2 PAMELA Modeling Technique

The PAMELA modeling technique provides a *language* to describe performance models of a system like the one given in Figure 4.2. The language is composed of three primitives: processes, semaphores and delays.

A *process* executes a sequence of statements composed of imperative language constructs like the conditional statements *if* and *while*, variables, and functions. A *semaphore* is a mechanism for enforcing process synchronization [Dijkstra, 1968]. Finally, a *delay* indicates how much progress a process makes in time, expressed as integer values [1] representing *time units*.

[1] In the original PAMELA language, the delay is a real value. However, we confine the delay value to be an integer value

A semaphore is a primitive that has an integer value. Processes manipulate this value using the **P** and **V** operators [2]. The **P** operator decrements the value whereas the **V** operator increments the value of a semaphore. When this value becomes zero using a **P** operator, the semaphore causes the process executing the **P** operator to block. A *blocked process* stops executing its sequence of statements. A process remains blocked until another process executes the **V** operator, giving the semaphore a nonzero value. This lifts the blocking condition and the blocked process can continue its sequence of statements.

Processes make progress autonomously in time using the **delay** operator and we refer to processes as *active* primitives. Processes are hindered in their autonomous progress by semaphores. We refer to semaphores as *passive* primitives.

Modeling a System Using the PAMELA Modeling Technique

Using the primitives of the PAMELA language, we now describe the system given in Figure 4.2. We do not use the original PAMELA; instead we use a *pseudo C* language combined with the PAMELA primitives. In the system, we only describe task G and resources and dependencies related to this task. After we have indicated how to describe a complete performance model, we explain how the three PAMELA primitives describe the four components of the PMB, as required to make a performance model of a system.

Use of Symbols We describe programs in a pseudo C language. Those familiar with the C programming language should have no difficulties in understanding the code. The program sections we present in this chapter and succeeding chapters contain only the minimal statements necessary to represent a particular section of program. Initialization and declarations of variables are usually not presented. In referencing elements of programs, we use the following convention. Variables or functions to which we refer in a program are written as `x` and `read`, representing a variable and a function, respectively. Both special operators like **V** or **P** and special program statements that we will introduce in this chapter are written using the bold face font. Finally, when we refer to names in figures, we write these names as A, for example, or resource1.

Modeling Approaches A system consists of resources and tasks. However, the PAMELA language provides processes, the active primitives and semaphores, and the passive primitives. This leads to two possible modeling approaches [Kreutzer, 1986]: material-oriented modeling and machine-oriented modeling. In the *Material-oriented Modeling* approach, the tasks of a workload are modeled mainly using active primitives. In the *Machine-oriented Modeling* approach, resources are mainly modeled using active primitives. Which approach we should use very much depends on what we want to achieve with the constructed model. We come back to this important issue in Section 4.3, where we motivate why we want to adapt the original PAMELA method to make it more appropriate for our purpose.

For the moment, we model the system given in Figure 4.2 using the material-oriented modeling approach. We thus model each task as a process. We use semaphores to model data dependencies and resources. The delay primitive is used to indicate how long it takes to execute a task.

because we express delays in terms of number of cycles, and these are always represented as integer values (to be explained further in Chapter 5).

[2] **P** is the first letter of the Dutch word “passeren,” which means “to pass”; **V** is the first letter of “vrijgeven,” the Dutch word for “to release” [Dijkstra, 1981].

Modeling Data Dependencies and Resources Resources and data dependencies are modeled using semaphores in the material-oriented modeling approach. When we look at task G, we need to model its data dependencies with task H, task F, and task C. We also have to model the relationship with resource1. These three data dependencies and the resource are described as semaphores in Program 4.1.

Program 4.1. MODELING DATA DEPENDENCIES AND RESOURCES

```
/* Data Dependencies */
semaphore dependency_CG 0;
semaphore dependency_FG 0;
semaphore dependency_GH 0;

/* Resources */
semaphore resource1    1;
```

In Program 4.1, we first show the three semaphores describing the data dependencies of task G with tasks C, F, and H. The value of each semaphore is initialized to zero. As a consequence, when a process uses the **P** operator on one of these semaphores, it will block. Suppose the process describing task G executes the **P** operator on semaphore dependency_CG. As explained, the process will block. It remains blocked until the process describing task C increments the semaphore using the **V** operator. This causes the value of semaphore dependency_CG to become nonzero. Using the semaphore, task C will always execute before task G, and the semaphore thus implements a data dependency.

After describing the semaphores implementing the data dependencies, we show the semaphore implementing resource1. This semaphore resource1 is initialized, in contrast to the data dependencies, to a nonzero value, and in particular, to the value of one. As such, a process using the **P** operator on this semaphore will not block. However, the next process using the **P** on the same semaphore will block because the semaphore has acquired a zero value. As a consequence, only one process can have exclusive access to resource1. When the first process uses the **V** operator, giving the semaphore a nonzero value, it releases the resource so that the blocked process can obtain exclusive access to the resource.

Modeling a Task Task G in the system of Figure 4.2 has a data dependency with task C and task F. Consequently, Task G can only execute after both dependencies with task C and task F have been satisfied, i.e., after these tasks have already executed. Before task G executes, it first needs to have resource1 in its possession. After this is the case, then task G executes, taking 2 time units. This is followed by releasing the resource and indicating to task H that it is allowed to execute. This sequence of events as explained so far is modeled – like any other task – using a process. For task G, this process description is given in Program 4.2. It describes a sequence of statements enclosed by the word **process** and curly brackets. The statements between the curly brackets are executed one after the other. When the sequence comes to the end, i.e., at the last closing curly brackets, the sequence starts all over again with the first statement after the word **process** and the opening curly bracket.

In the process description in Program 4.2, we see that the sequence of statements consists of only **P**, **V** and **delay** operators. When the process starts, it executes the **P** operator on the semaphore that implements the data dependency with task C. Since this semaphore dependency_CG is initialized to a value of zero, process G blocks. It remains blocked until the process modeling task C executes a **V** operator on semaphore dependency_CG. After the dependency with task C is satisfied, process G executes the **P** operator on semaphore dependency_FG, which models the data dependency with task

F.

Program 4.2. MODELING OF TASK G AS A PAMELA PROCESS

```
process   G {
    /* Check dependency with Process C and F */
    P(dependency_CG);
    P(dependency_FG);

    /* Claim the resource */
    P(resource1);

    /* The execution of a task takes 2 time units */
    delay( 2 );

    /* release the resource */
    V(resource1);

    /* Satisfy dependency of Process H */
    V(dependency_GH);
}
```

After both dependencies have been satisfied, process G tries to take possession of resource1 by executing the **P** operator on semaphore resource1. If this resource is available, the process continues. If not, the process blocks until the resource becomes available again. If task G has acquired the resource, other processes modeling task A and task H that try to acquire the same resource will block, giving process G exclusive access to the resource. When process G possesses resource1, it can execute in 2 time units, as modeled by the **delay** operator.

After process G takes two time units to execute, it releases the resource that it also held in its possession for 2 time units, by executing the **V** operator on semaphore resource1. This is followed by executing the **V** operator on semaphore dependency_GH to indicate that the process modeling task H can continue to execute.

Although we only showed task G, we can completely describe the system of Figure 4.2. To do this, we have to describe each task in the system in a similar way as done in Program 4.2 and each data dependency and resource in a similar way as done in Program 4.1.

Performance Modeling Basis

The three PAMELA primitives are able to describe the four components of the PMB. We now describe the components *Condition Synchronization*, *Mutual Exclusion*, and *Execution Time*. To describe *Conditional Control Flow*, we have to extend the process description as given for task G in Program 4.2 to obtain a functional performance model.

Condition Synchronization Semaphores implement condition synchronization, the component which relates to data dependencies between tasks or to limited resources. A semaphore describes a data dependency by initializing the semaphore to zero; as long as the data dependency condition is not satisfied, the semaphore remains zero and blocks a process executing the **P** operator on that semaphore. When the data dependency condition is satisfied using the **V** operator, the blocked process proceeds. The first three semaphores in Program 4.1 describe condition synchronization due to data dependencies.

Mutual Exclusion Semaphores also implement mutual exclusion, the component which determines that two or more tasks cannot operate overlapped in time or that a resource cannot be used by more than one task at a time. If a semaphore is initialized to the value of one, only one task is allowed to access a resource at a time. The first task acquiring the resource by using the **P** operator makes the semaphore zero, and all tasks trying to acquire the same resource will block. The semaphore `resource1` in Program 4.1 describes mutual exclusion of the resource1.

Task G in the system shown in Figure 4.2 requires only resource1 in order to execute. In general, a task could require possession of *multiple resources* in order to execute. This multiple resource possession is modeled correctly by **P** and **V** operators.

Execution Time The delay primitive models the execution time, the component which determines how long a task takes to execute or how long a resource is used. The integer value associated with the **delay** operator either indicates how many time units a task takes to execute or how long a resource is used. In Program 4.2, the **delay** operator shows that it takes task G two time units to execute. Furthermore, since resource1 was claimed before and released after executing the **delay** operator, it is thus used by task G for two time units. Note that we do not actually perform a task.

Conditional Control Flow The sequence of imperative statements describes the *control flow* of a process. A process has a conditional control flow when the sequence of statements consists of conditional statements like *if* and *while*. Conditional statements can affect control flow, based on the evaluation of conditions at run-time.

In Program 4.3, we see part of a process description of a task. It describes a process that executes based on the value of variable `x`. If the variable `x` has a value larger than 256, it takes the process six time units to execute, otherwise it takes the process two time units to execute.

Program 4.3. MODELING CONDITIONAL CONTROL FLOW

```
process  {

     /* Execution time of a Task */
    if ( x > 256 ) {
       delay(6);
    } else {
       delay(2);
    }
}
```

In the description of process `G` in Program 4.2 no variables are used; only relationships between tasks and resources have been modeled. Nonetheless, as shown in Program 4.3, conditional control flow requires data with which to resolve conditions at run-time. Put in other words, we need to have a *functional performance model*, which models besides the relationships between tasks and resources, also the correct functional behavior of a system. When only relationships between tasks and resources have been modeled, the model is also called an *uninterpreted performance model*, whereas a functional performance model is also called an *interpreted performance model* [RASSP Taxonomy Working Group (RTWG), 1998].

To describe the correct functional behavior of the system in Figure 4.2, we have to extend all the descriptions of the processes modeling tasks in a way such as shown for task G in Program 4.4. This program shows the process description given in Program 4.2, but also uses global and local variables and a function. To illustrate conditional control flow, we also extended the process description of task G as shown in Program 4.3.

Program 4.4. A Functional Process Description in PAMELA

```
/* Declaration of Global Variables */
integer C_data;
integer F_data;
integer G_data;

process  G {
    /* Check dependencies with Processes C and F */
    P(dependency_CG);
    integer arg1 = C_data;

    P(dependency_FG);
    integer arg2 = G_data;

    /* Claim the resource */
    P(resource1);

    /* The execution of a task takes two or six time units */
    if ( arg1 > 256 || arg2 > 256 )
        delay( 6 );
     else
        delay( 2 );


    /* Perform a function on the data */
    integer arg3 = add(arg1,arg2);

    /* release the resource */
    V(resource1);

    /* Store data in the global variable of Process G */
    G_data = arg3;

    /* Satisfy dependency of Process H */
    V(dependency_GH);
}
```

The functional process description of task G starts with declaring the global variables C_data, F_data, and G_data as integers. These variables are used to communicate data along data dependencies within the system given in Figure 4.2. Because the variables are declared globally, other processes can access this data. The description of process G has nearly the same sequence of PAMELA primitives as Program 4.2. The first two data dependencies are followed by statements in which local variables i.e., arg1 and arg2, are assigned the value of the global variables C_data and F_data. Only after the data dependencies are satisfied do the global variables C_data and F_data carry useful data.

After process G has resource1 in its possession, it evaluates the condition of the if-statement. If variable arg1 or arg2 has a value larger than 256, executing the task takes six time units instead of two, as already shown in Program 4.3. After the correct **delay** operator has been executed, the process executes a function, in this case add, which adds the values of variable arg1 and variable arg2 together. The result of the addition is stored in the local variable arg3. After resource1 has been released, process G assigns variable arg3 to variable G_data. It also indicates to the process modeling task H that the dependency is satisfied and that variable G_data carries useful data.

Summary The three primitives of the PAMELA language can describe the four components of the PMB. Three out of the four components require only that the relationships between tasks and resources be described. These three components are *Condition Synchronization*, *Mutual Exclusion*, and *Execution Time*. The component *Conditional Control Flow* requires additionally that a functional description be used to describe tasks. This implies that data is explicitly transported from process to process and that functions can operate on this data.

As we shall discuss in Section 4.3, when a performance model uses the first three components to model a system (resulting in an uninterpreted performance model), it can be evaluated using analytical techniques. When, however, conditional control flow is involved (leading to an interpreted performance model), simulation is the only available technique.

4.2.3 PAMELA Evaluation Technique

After we have described all tasks in the system of Figure 4.2 in the same way as we did for task G in Program 4.2, we end up with a performance model for that system. To evaluate this performance model, we use the evaluation technique of the PAMELA method to obtain performance numbers for particular performance metrics. Although the PAMELA method provides two evaluation techniques (i.e. an analytical and a simulation technique), throughout this thesis we use only the simulation technique which we will discuss later on in more detail.

Simulation

Simulation techniques are based on imitating processes that run in the system. Using a simulation engine, these processes are replayed as they would happen in the real system, albeit at a high higher level of abstraction. Such a simulation engine implements this abstract level by replacing the continuous time axis of the real system by a virtual time axis, leading to a *discrete event* simulation. In Chapter 5, we show how we can relate this virtual time to the notion of clock cycles in architectures. Each *event* indicates a point in time at which the simulation engine needs to reconsider the consequences of the currently occurring event for the further execution of processes. In the PAMELA performance models, events are only caused by the **P**, **V** and **delay** operators. Thus to simulate a PAMELA performance model, the simulation engine must step through all generated events and at each event, decide what to do next.

When we simulate the performance model constructed for the system in Figure 4.2, we obtain the results shown in Table 4.1. This table shows which process is active at which time instance. The set-up of Table 4.1 very much resembles a *Gantt Chart*.

Process	Res1	Res2	delay	dep.	1	2	3	4	5	6	7	8	9	10	11	12
A	*		3	-	□	□	□									
B		*	1	A						□						
C		*	2	A				□	□							
D		*	1	-	□											
E		*	1	C							□					
F		*	1	DE								□				
G	*		1	CF									□	□		
H	*		1	BG											□	

Table 4.1. The execution of the processes used to model the system of Figure 4.2. It shows which process is active at which time instance. In the first column all processes (A through H) of the performance model are given. This is followed by the two columns describing which process uses (indicated with an '*') resource1 or resource2. This is followed by two columns expressing the time required by a task to execute and with which tasks a task has data dependencies. Next the virtual time axis is given from the discrete time interval t=1 to t=12. An empty place means a process is waiting, an '□' indicates the process is running a collection of statements.

To show how the simulation engine comes to the execution trace given in Table 4.1, we next discuss how the collection of processes executes in time.

The simulation of the performance model starts at t=0. Processes A and D do not depend on other processes and can use the **P** operator to claim the required resources. Process A acquires resource1 and process D acquires resource2. It takes process D one time unit (modeled using the **delay** operator) to finish. At t=1, it releases (via a **V** operator) resource2 and satisfies the data dependency with process F using a **V** operator. Process F must wait until the data dependency relation with Process E is also satisfied, and it remains blocked until then. Process A terminates after 3 time units and releasing resource1 at t=3, thereby satisfying the data dependencies with processes B and C.

Processes B and C both require the same resource2 to execute at t=4. However, only one process can access the resource at a time. Consequently, the simulation engine must perform *resource conflict arbitration* to decide which process may access the resource first. It uses a non-deterministic resource conflict arbitration scheme.

Thus the process that claims the resource first obtains the resource first. Suppose process C acquires the resource first. Its execution takes two time units and it releases resource2 at t=5. While process C executes, process B blocks until resource2 becomes available. At t=5, process B can acquire resource2 and executes. At the same time, process C satisfies the data dependencies with processes G and E. Nevertheless, process G remains blocked until process E satisfies the dependency with process F. At t=5, both processes B and E need resource2. If process E would need resource1 instead of resource2, it could have executed in parallel with process B. Suppose process B claims the resource before process E. It executes first, taking 1 time unit. At t=6, it terminates, satisfying the data dependency with process H, and releases resource2. Process E acquires resource2 and executes, taking 1 time unit. It terminates at t=7, releasing the resource and satisfying the data dependency with process F. For process F, the data dependency with process D was already satisfied at t=1. Therefore process F claims resource2 and executes, taking 1 time unit. It terminates at t=8, releasing resource2 and satisfying the dependency with process G. Since no other process holds resource1, process G executes, taking 2 time units. At t=10, process G terminates, releases resource1, and satisfies the data dependency with process H. The dependency with process B was already satisfied for process H at t=6. Therefore process H can execute after it claims resource1. The process executes, taking 1 time unit, and terminates at t=11.

In conclusion, processing the workload described by the task-graph in Figure 4.2 with two resources requires in total 11 time units. This gives the *total execution time* $T_{end} = 11$ for the system of Figure 4.2. The total execution time is a performance metric and the performance number obtained through simulation equals 11 time units.

Performance Numbers

The workload of the system consists of 8 tasks with a combined execution time of 12. The combined execution time is found by adding up the execution time of each task, as given by the number between brackets in Figure 4.2. If the executions of the 8 tasks were independent of each other, the system would require 3 time units to process the workload and $T_{end} = 3$. However, condition synchronization influences the order in which processes must execute. Due to this ordering, the system would require a total execution time of at least $T_{end} = 10$. Moreover, mutual exclusion also influences the ordering of the processes. Because a resource can be accessed by only one process at a time, additional delays are introduced. The processes B and C as well as the processes B and E require the same resource2. These processes will be sequentialized in time, causing them to wait. As a result, the total execution time increases and becomes $T_{end} = 11$.

We can use the total execution time T_{end} to derive performance numbers for other performance metrics. One such performance metric is parallelism, which we define as

$$\text{Parallelism} = \frac{\text{Combined Execution Time of the Workload}}{T_{end}}. \tag{4.1}$$

If the workload of 8 tasks with a combined execution time of 12 is processed in parallel, $T_{end} = 3$. If condition synchronization is taken into account, $T_{end} = 10$. Finally, if mutual exclusion is also taken into account, then $T_{end} = 11$. Thus the amount of parallelism achieved in the system in these three different situations is respectively $\frac{12}{3}$, $\frac{12}{10}$, and $\frac{12}{11}$, as shown in table 4.2.

Performance Model takes into account	Parallelism
Execution Time	$\frac{12}{3}$ = 4
+ Condition Synchronization	$\frac{12}{10}$ = 1.2
+ Mutual Exclusion	$\frac{12}{11}$=1.09

Table 4.2. Parallelism in the System.

Another performance metric is the *utilization* of the resources, which is

$$\text{Utilization} = \frac{\text{Time a Resource is Used}}{T_{end}} \times 100\%. \tag{4.2}$$

If a resource is used for 4 time units during the total execution time of 11, it achieves a utilization of $\frac{4}{11} \times 100\% = 36.3\%$. The utilizations of resource1 and resource2 are given in Table 4.3. In both cases, the utilization of the resources happens to be 54.5%.

Resource	Utilization Rate	Used by Process
Resource1	(3+2+1)/11×100% = 54.5%	A+G+H
Resource2	(2+1+1+1+1)/11×100% = 54.5%	B+C+D+E+F

Table 4.3. Utilization of the Resources.

Other performance metrics besides total execution time, parallelism, and utilization exist. We present more of these in Chapter 7.

4.3 Objectives in Using the PAMELA Method

We already mentioned that two modeling approaches exist: material-oriented modeling and machine-oriented modeling. The choice between these two modeling approaches touches upon the fundamental issue of ease of *modeling* as opposed to ease of *evaluation*. As we will explain, the original PAMELA work focused on ease of evaluation, whereas we are focusing on ease of modeling. As a consequence, we will adapt the PAMELA method to better suit our needs in modeling systems.

The PAMELA method was developed originally by van Gemund to quickly yield parameterized performance models of parallel programs that run on shared-memory as well as distributed-memory (vector) computers. The PAMELA method was primarily developed to be part of a compiler for such computers. The compiler could analyze these parameterized performance models, guiding the compiler in the mapping of the programs. Thus instead of simulating the PAMELA model, an analytical method had to be employed. The analytical method transforms a PAMELA model automatically into an explicit symbolic performance model. This symbolic performance model yields fast and crude performance predictions that the compilers uses to steer the mapping. Consequently, van Gemund focused primarily on deriving symbolic performance models at run-time from a PAMELA model, because he wanted to construct PAMELA performance models such that they could be analyzed easily.

In the construction of the PAMELA method, van Gemund traded off ease of modeling for ease of evaluation [van Gemund, 1996]. He used a material-oriented modeling approach and constructed only uninterpreted performance models. Consequently, these models do not contain conditional control flow [3].

Within this thesis, however, we want to explore the design space of an architecture template and we require a modeling approach that permits us to construct many different architecture instances quickly. Therefore, we trade off ease of modeling for ease of evaluation. We will use a machine-oriented modeling approach to model architecture instances that deliver cycle-accurate performance numbers. The architecture instances are interpreted performance models of which the performance can only be evaluated via simulation. The PAMELA method provides a simulation engine referred to as the *Run-Time Library*, or RTL. We discuss the RTL in detail in Section 4.4.4.

To illustrate the two different objectives that are possible (ease of modeling versus ease of evaluation), in Figure 4.3 we show the two steps involved in performance analysis, namely performance

[3] van Gemund modeled condition control flow by means of distributions, and as a consequence conditional statements do not need to be evaluated. This only works when conditional statement can be captured by means of distributions and such distributions are known

modeling and performance evaluation. Since van Gemund desires symbolic expressions, he constructed a model of the system using a material-oriented modeling approach. The model obtained can be analyzed analytically, yielding symbolic performance models. He rendered performance numbers via a transformation of the PAMELA model into the time domain. Notice that a model of the system obtained via the material-oriented modeling approach can be evaluated using simulation, but that a model of the system obtained via the machine-oriented modeling approach cannot be reduced to a symbolic expression.

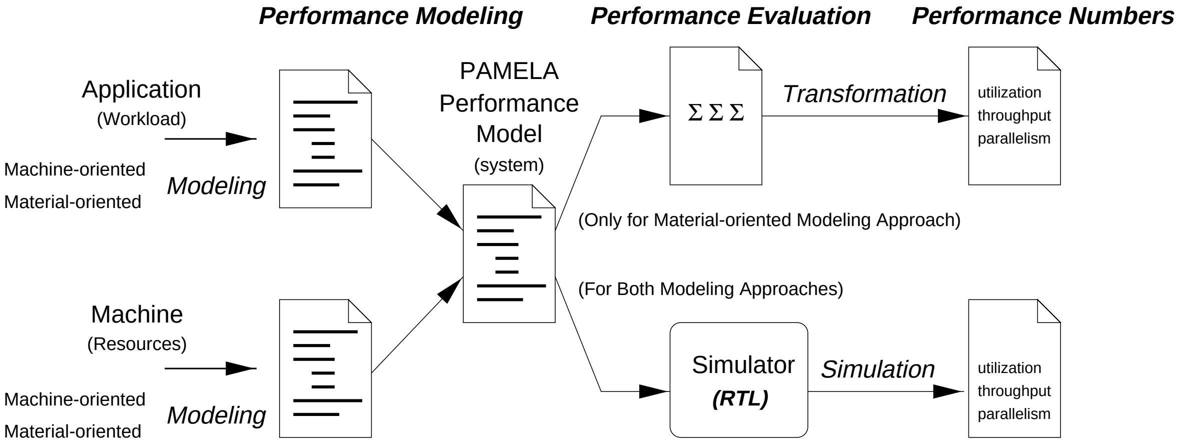


Figure 4.3. The two different objectives possible in performance analysis: ease of modeling versus ease of evaluation.

The programming paradigm used to construct a performance model of a system in a material-oriented model differs from that for a machine-oriented model [van Gemund, 1996]. A *procedural oriented* programming paradigm describes the material-oriented modeling approach well; therefore the original PAMELA language employed a procedural oriented language. On the other hand, an *object oriented* programming paradigm is a good method with which to describe the machine-oriented modeling approach. We will show in the remaining part of this chapter how we can use the PAMELA primitives and the RTL in an object oriented modeling approach. We implement a message-passing interface on top of the PAMELA primitives by embedding the PAMELA primitives in the object oriented programming language C++ to ease the construction of the performance model of complex systems.

4.4 An Object Oriented Modeling Approach using PAMELA

To ease the construction of complex systems while still being able to derive performance numbers via simulation, we embedded the PAMELA primitives in the object oriented programming language C++. We first explain what an object is, and then we explanation that the objects we use consist of a structure part and a behavior part. We use this kind of object to model the system of Figure 4.2 again.

Following this example, we look again at task G of the system and explain in detail how this task is represented as a C++ object. Then we describe how a process is created for this C++ object. This includes the description of the PAMELA run-time library (RTL), which implements the three PAMELA primitives as functions in C.

4.4.1 The Object

The basic element of an object oriented programming language is the object. An *object* consists of a *data* part as well as a *methods* part that operates on the data part [Goldberg and Robson, 1983]. The data part and the method part of an object, and thus the whole object, is defined using a *class description*. Individual objects can be *instantiated* from such class description, which means that an object is created taking up a certain amount of space. When an object is instantiated, a special method, the *constructor*, is called. This constructor is called only once for each object at the time when the object is instantiated.

The strength of objects is that they are well suited for use in constructing a (complex) *structure* [Mellor and Johnson, 1997; Monroe et al., 1997; Kumar et al., 1994; Post et al., 1998] that is the decomposition of a system into parts and relationships between those parts. In an object oriented programming language, a structure is hence constructed by interconnecting objects whereby each object can be composed of other interconnected objects. Within a structure, objects connect to other objects and only exchange information along these connections. An object exchanges information with other objects by invoking methods of those other objects. This way of exchanging information is referred to as "message passing" [Goldberg and Robson, 1983]. The available methods of each object thus defines for other objects which messages it can exchange; in other words it defines an *interface*. Objects exchanging messages with another object remain completely unaware of what happens inside this object, since it hides unnecessary information.

Besides describing the structure, we also have to describe the behavior of each object. By *behavior* we mean the function that the object describes in time. To describe the behavior of an object, we specify a process that describes the input/output relations of that object in time. As we will see in Chapter 5, not every object is given a process. Recall that there is a distinction between active primitives, which are processes, and passive primitives, which are semaphores. An object that uses a process is called an *active object* and one that does not is called a *passive object*. A passive object uses **P** and **V** operators in its methods, but no processes.

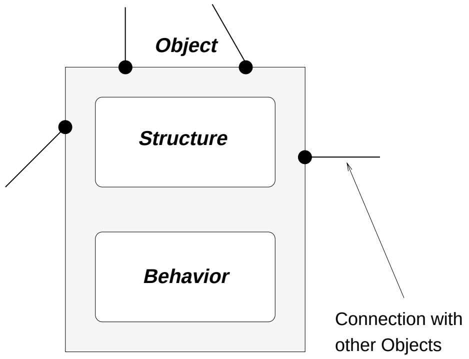


Figure 4.4. The objects we construct have a structure part and a behavior part.

We use the object oriented programming language C++ [Stroustrup, 1991] to create an object. Although other object oriented programming languages like objective-C [The Stepstone Corporation, 1989] or Java [Arnold and Gosling, 1996] could have been used, we use C++ for reasons of performance. The constructed objects in C++ have a structure part and a behavior part, as shown in Figure 4.4. The figure also shows the object connects with four objects, as indicated by the black circles. The object can only exchange information with the four objects to which it connects.

4.4.2 Modeling a System as a Network of Objects

To show how we can use an object as depicted in Figure 4.4, we will model the system of Figure 4.2. Because we haven't yet explained how this system relates to architectures like stream-based dataflow architectures, we cannot yet explain machine-oriented modeling adequately; therefore, for didactic reasons, we model the simple system using a material-oriented modeling approach. In Chapter 5, however, we show several examples of objects used in a machine-oriented modeling approach.

The performance model of the system in Figure 4.2 obtained using objects is shown in Figure 4.5. To come to this model, we replaced each task by an active object having a structure part and a process part. We also replaced the two resources by a passive object having only a structure part. The model shown in Figure 4.5 still models the system using a material-oriented modeling approach.

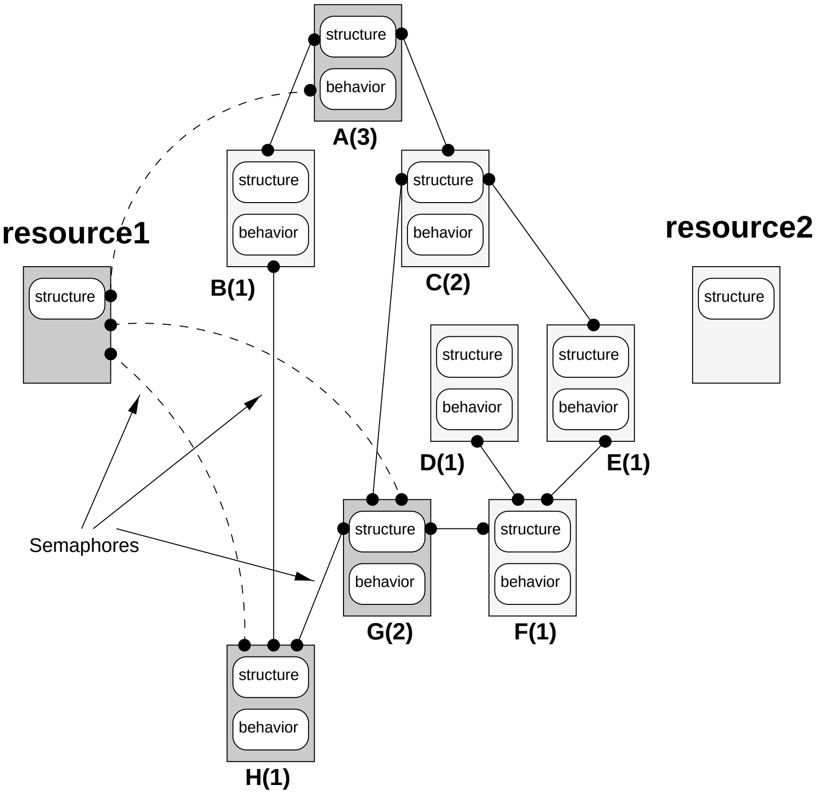


Figure 4.5. The system described using objects that have a *structure* and a *behavior* part. For simplicity, the links between the objects that use resource2 are not shown, but do really exist.

We implement all the links between different objects (for example, between object B and object H) directly via a semaphore. The dashed line from object resource1 to the objects modeling tasks A, G, and H is one and the same semaphore. Notice that although it is not shown in Figure 4.5, a semaphore also exists between object resource2 and the objects modeling tasks B, C, D, E and F.

We now further elaborate on how the object modeling task G is described as a C++ object of type *Task* and how a process from the run-time library is added to this object. We do not describe the object modeling the resources, because enough examples will be given in the next chapter.

4.4.3 Describing the Structure of an Object in C++

Before we can construct an object *Task* modeling a task in the system of Figure 4.5, we need to describe a class definition. In Program 4.5, we give the class description for the object `Task`. In C++, a class definition always starts with the keyword **Class**.

Program 4.5. CLASS DEFINITION OF THE `Task` OBJECT

```
class Task {
       /* Data Elements */
        semaphore in1;
        semaphore in2;
        semaphore out;
        semaphore resource;
        int delay;

       /* Methods */
      /* Constructor */
      Task( semaphore, semaphore, semaphore, semaphore, int )

      /* init Process */
      init_process(void);
}
```

Class Definition

Within this class description, an object is defined that consists of five data elements [4] of which four are semaphores and one is a single integer value. Three semaphores represent dependencies (`in1`, `in2`, and `out`) and one semaphore represents a resource (`resource`). The integer delay value `delay` indicates the time required by a task to execute.

Within the same class description, two methods are defined for a `Task` object. Each method has an *implementation*, which describes the function a method performs. One method is the constructor [5] of the `Task` object while the other method is `init_process`. The constructor method is called when a `Task` object is instantiated from the class description. The constructor has a number of arguments that need to be provided when an object is created. In this case, the arguments include four semaphores and an integer value. The constructor is used to set up the structure of an object, while the `init_process` method is used to create the behavior of an object. We describe first what happens when the constructor is called, followed by what happens when the `init_process` method is called.

Calling the Constructor

To instantiate an object representing task `G` in the system of Figure 4.5, we call upon the C++ function `new`, as given in Program 4.6. In creating the `Task` object, we pass on the semaphores already described in Program 4.1 as arguments of the constructor. We also pass on the integer value 2, to indicate task `G` takes two time units to execute.

[4] All the elements in the data part of the class definition are variables in C/C++. Therefore, semaphore `in1` is a variable, as is the integer `delay`

[5] In C++, methods are described using the `::` notation. It should be read as `class::method`, which describes the implementation of a particular method of a class. The constructor method of a class always has the same name as the class. Thus `Task::Task` describes the implementation of the constructor of class `Task`.

Program 4.6. Instantiate an Object representing task G

```
new Task(dependency_CG, dependency_FG, dependency_GH, resource1, 2)
```

Executing the `new` function causes the execution of the constructor, as shown in Program 4.7. In the constructor, the arguments `dep1`, `dep2`, `dep3`, `res` and, finally, `task_delay` are assigned to the variables declared in the data part of the `Task` object in Program 4.5. The value of variable `dep1` is thus assigned to variable `in1`. Because object `Task` is created as shown in Program 4.6, argument `dep1` equals the semaphore `dependency_CG` that describes the dependency between task C and task G. Therefore variable `in1` equals semaphore `dependency_CG`.

Program 4.7. The Constructor of `Task` Object

```
Task::Task(semaphore dep1, semaphore dep2, semaphore dep3, semaphore res, int task_delay)
{  // Constructor of a Process
   in1      = dep1;
   in2      = dep2;
   out      = dep3;
   resource = res;
   delay    = task_delay;
}
```

4.4.4 Describing the Behavior of an Object Using PAMELA

We have shown how an object captures the structural elements in its data part, and in particular, for task G. Now we will look at how we can add a behavior part to the structure part of an object after we present the run-time library PAMELA.

Run-time Library (RTL)

The PAMELA language describes performance models using three primitives: *processes*, *semaphores* and *delays*. These three primitives have been implemented as C functions in the PAMELA *Run-Time Library* (RTL) [Nijweide, 1995]. The RTL is a simple, non-preemptive *multithreading* package. It provides the notion of parallelism in the sequential programming language C [Kernighan and Ritchie, 1988]. The RTL implements a *process scheduler*, interleaving the parallel processes on a single CPU. It differs from many other multithreading packages [Open Software Foundation, 1995; Proven, 1995; Mueller, 1993], in that it uses the notion of *virtual* time.

The RTL implements processes as *lightweight processes*. Lightweight processes are implemented in the same address space [Finger, 1995], which makes communication of variables between processes very simple and efficient. Each process has its own *thread-of-execution*, and a thread has its own *context* of variables (state). Whenever the process scheduler switches a process on the CPU, it restores the context of the thread. This is called *context switching*. The processes are called lightweight processes because the threads have a small or ‘light’ context [6]. This introduces only a small overhead and, consequently, the process scheduler can switch the context of processes quickly.

[6] This is in contrast to *heavyweight* processes used, for example, in the UNIX operating system. These processes have their own address space and require pipes to communicate variables with other processes. As such, these processes have a big or ‘heavy’ context, and context switching on a UNIX system requires more time than a context switch of a lightweight process.

The complete RTL and its C functions are described in detail in [Nijweide, 1995]. We now discuss the function used to construct a process, the functions that implement the semaphore operators **P** and **V** and the function which implements the **delay** operator.

Process: The RTL describes a process as an ordinary C function called the *process function*. The RTL creates an instance of a process function in memory. A process scheduler switches these instances of the process functions in time to give the appearance of their executing in parallel. A process is created via the **pam_fork** statement. This statement requires three arguments: a name for identification purposes, a C function that is the process function, and an additional argument. This additional argument is used by a process to initialize itself when it is created.

Semaphore operators: The RTL implements the **P** and **V** operators as **pam_P** and **pam_V** statements respectively. These statements operate on semaphores created via the **pam_alloc** statement. This statement requires two arguments, namely a name for identification purposes and an initialization value.

Delay operator: The RTL implements the **delay** operator as a **pam_delay** statement. This statement manipulates the time of a process. The operator requires a non-negative integer value.

The RTL provides more functions than just those mentioned above. We will wait, however, to introduce these functions until they are used.

Creating a Process

All the tasks in Figure 4.2 are modeled using a process. We therefore need to create a process for each object that models a task in the system-modeling example in Figure 4.5. Assume that we have instantiated an object for each task and stored links to other objects in the data part of these objects. In this example we do this using only semaphores.

A process is created for each object by calling the `init_process` method, as shown for the `Task` object in Program 4.8. This method causes the execution of the **pam_fork** statement. As explained when discussing the RTL-library, this invokes the creation of a process with the name “`task`” that executes the process function `task_function`. The argument `this` is passed on to this process. In the C++ programming language the `this` pointer represents the complete data part of an object. Therefore, the **pam_fork** statement passes on the complete data structure of the object to the process function `task_function`.

In the case of the `Task` object that models task G, the data part of this object contains the semaphores describing the data dependencies with tasks C, F and H. It also contains the semaphore describing the resource1.

Program 4.8. CREATING A PROCESS FOR AN OBJECT

```
Task::init_process( void ) { pam_fork ("task",task_function,this); }
```

A Process

The RTL creates a process from the process function by creating a unique instance in memory using the **pam_fork** statement. In the case of the `Task` object, the RTL creates a process based on the process function `task_function`, as shown in Program 4.9.

A process function always consists of two parts. In the first part, the process decodes the `this` pointer and initializes its local variables. The `this` pointer received is the data structure from the object creating the process. In the second part, the process function runs as the actual process.

Program 4.9. Modeling a Process via a Process Function

```
void task_function(void)
{
   /* Initialize the Process */
   Process p = (Process) pam_args (pam_me ());
   semaphore in1        = p.in1;
   semaphore in2        = p.in2;
   semaphore out        = p.out;
   semaphore resource   = p.resource;
   int       task_delay = p.delay;

   /* Run the Process */
   while(1) {

    /* Check dependencies with other Processes */
    pam_P ( in1 );
    pam_P ( in2 );

    /* Claim the resource */
    pam_P ( resource );

    /* The execution of a task takes a parameterized amount of time */
    delay( task_delay );

    /* release the resource */
    pam_V ( resource );

    /* Satisfy dependency of a Process */
    pam_V ( out );

   }
}
```

Initialize the Process In the first part, the process function decodes the data structure that was passed on by the constructor via the `this` pointer. To actually acquire the `this` pointer from the object creating the process, the process uses two RTL statements, **pam_args** and **pam_me**. Statement **pam_me** identifies the process and the **pam_args** statement retrieves the argument for that process. The data structure of the object is accessible via the variable `p` in Program 4.9 and `p.in1` is one and the same semaphore as `in1` in Program 4.5. This semaphore is assigned to the local semaphore `in1` in the process function as part of the initialization of the process. The initialization step finishes after all variables have been decoded to a local variable of the process function.

Execute the Process In the second part, the process function runs the actual process. The process starts with the `while(1)` statement, which creates an endless loop. The curly bracket after the `while` statement matches one-to-one with the curly bracket after the word **process** in Program 4.2. Furthermore, the statements in between the curly brackets of the `while` statement match one-to-one with the statements given between the curly brackets related to the **process** described in Program 4.2. Henceforth, in this thesis, the word **process** always relates to the second part of a process function.

In the sequence of statements describing the **process**, the delay statement **delay** does not have a fixed value. Instead it receives a *parameterized* value from the Task object. The delay value is passed on as an argument of the constructor in the same way as was done for the semaphores.

4.4.5 Building Blocks

A C++ object describes both a structure part and a behavior part. Its structure and methods are described via the class definition. Its behavior is described using a RTL process. The result of constructing a Task object for task G is given in Figure 4.6.

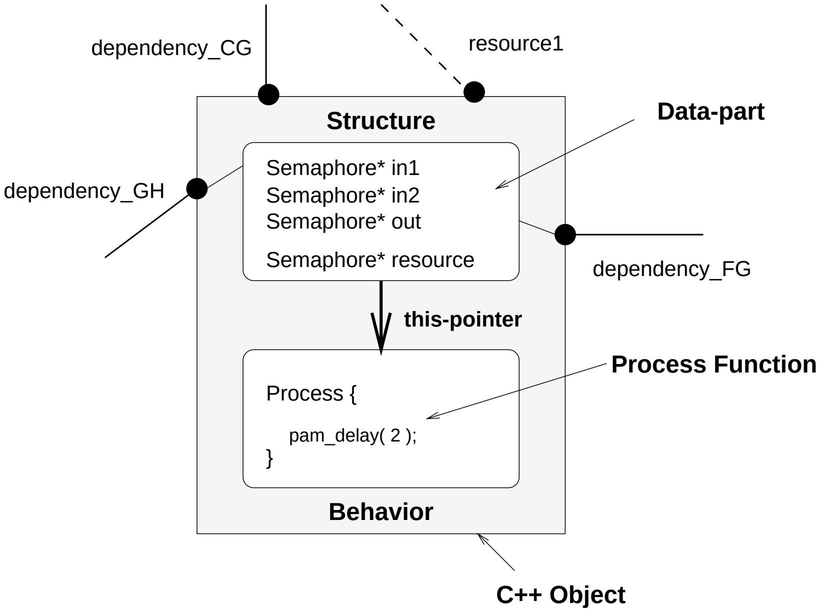


Figure 4.6. The construction of a PAMELA Process inside a C++ object.

In the structure part of the object we see again the three semaphores representing data dependencies with other objects and one semaphore representing resource1. In the way task G is modeled, it is the semaphores that allow objects to exchange information with other objects.

Localized References

The object shown in Figure 4.6 encapsulates a process, as well as localizes the references to other objects (or semaphores) used by that process. This process can only use the structural information defined for an object in the class description. Therefore, all references to other objects are expressed in variables of the class description of the object and thus *localized* to that object.

The process encapsulated by the object in Figure 4.6 can only operate on the semaphores in1, in2, out and resource. The process does not know to which semaphore it actually connects because this is determined when the object is instantiated. The semaphores expressing the data dependencies dependency_CG, dependency_FG, and dependency_GH in Program 4.6 bind to the semaphores in1, in2 and out respectively in Program 4.2. This also applies to the semaphore resource1, which binds to resource in the process.

A consequence of localizing the references is that the definition for object Task in Program 4.5 can also be used to define task F in the system-modeling example in Figure 4.5. Task F also has two input semaphores, one output semaphore, and one semaphore for a resource. By instantiating the object as was done in Program 4.10, we obtain an object that models task F.

Program 4.10. INSTANTIATE AN OBJECT REPRESENTING TASK F

```
new Task(dependency_DF, dependency_EF, dependency_FG, resource2, 1)
```

Note that we change the parameter value that describes the execution time of task F in Program 4.10. Instead of taking two time units as we did for task G, we now set the parameter value to 1, which causes task F to execute in one time unit.

By localizing the references of a process, we obtain objects representing *Building Blocks*. These building blocks represent modular elements that are self-contained and parameterized. We define a building block as

Definition 4.2. BUILDING BLOCK

A *building block* is a self-contained, parameterized object with a structure part and a behavior part. □

That a building block is *self-contained* means that it can only exchange information with other building blocks to which references are present in the structural part. The exchange of information takes place using the "message passing" mechanism. Consequently, the internal structure of a building block remains invisible to the other building blocks in a system, a key requirement in obtaining modular elements.

There is a distinction between active objects and passive objects; a similar distinction exists for building blocks. An *active building block* uses one or more **processes** to describe its behavior part. A *passive building block* uses only semaphores and **pam_P** and **pam_V** statements to describe its behavioral part. The **pam_P** and **pam_V** statements are incorporated into methods of the building block and these methods describe the behavioral part of the building block. We show several examples of active and passive building blocks in Chapter 5, in which we use building blocks extensively to construct the performance models of stream-based dataflow architectures by combining building blocks.

4.5 Simulating Performance Models with the RTL

We use simulation to obtain the performance number of a performance model for a system. PAMELA's RTL provides not only processes, semaphores and delay statements, but also a process scheduler. We now explain how this process scheduler simulates a performance model and how the PAMELA primitives interact with the RTL process scheduler. We discuss the RTL in more detail, showing that the RTL has a very simple structure. This simple structure makes it possible to execute faster than with other kinds of discrete event simulators.

A process is said to be *runnable* if it is not in a blocking state. A process can attain a blocking state only when executing a **pam_P** statement, and a process is taken out of this state via a **pam_V** statement. Each process has its own notion of time, represented by its own *time stamp*. The value of this time stamp changes directly using a **pam_delay** statement or indirectly using a **pam_P** statement. The time stamp is used by the process scheduler to order processes in time.

Although only one process can execute at a time, RTL implements a *process scheduler* that schedules one process after the other, giving the impression that processes execute in parallel. The process scheduler uses two different queues to schedule processes. One queue, the *Runnable Processes Queue*

(RPQ), contains all runnable processes, while the other queue, the *Blocked Processes Queue* (BPQ), contains per semaphore a queue with blocked processes. Each semaphore has a queue, because more than one process can block on one and the same semaphore. A schematic of the process scheduler with the two queues is given in Figure 4.7. This figure shows a “snapshot in time” of the filling of the two queues and the running process. In the figure, a circle represents a process. The number in each circle in the RPQ represents the value of the time stamp of each process. The letter in the circles in the queues of the BPQ represents the semaphore on which a process blocks.

One process runs at t=5, executing a sequence of statements. RPQ contains 6 processes that are runnable. Two processes can run at t=5, two processes can run at t=6, one process can run at t=10 and, finally, one process can run at t=12. After the running process terminates for reasons we explain later, the next runnable process to become the running process would be the first process in the RPQ. The runnable processes in RPQ are ordered according to their time stamp. BPQ contains three queues for three different semaphores: **A**, **B** and **C**. The queue of semaphore **A** contains three blocked processes, the queue of semaphore **B** contains one blocked process, and the queue of semaphore **C** contains two blocked processes. These processes can block because, for example, a resource is not yet available. A process is placed into the BPQ only via a **pam_P** statement. It is taken out of the BPQ only via a **pam_V** statement.

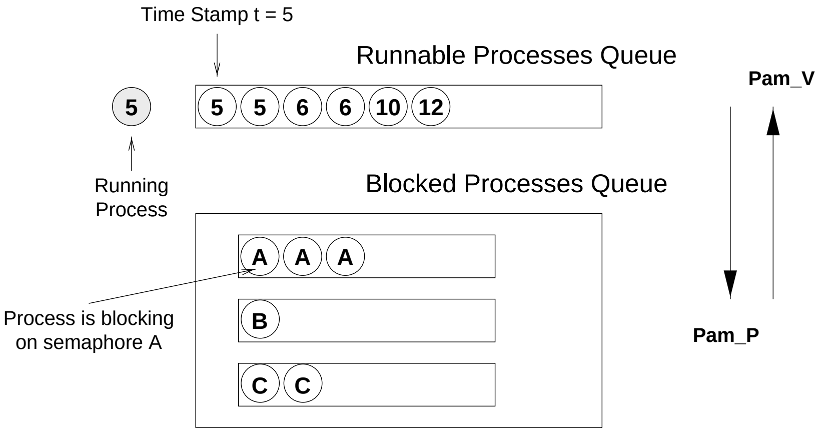


Figure 4.7. The process scheduler of the PAMELA run-time library uses two queues to schedule processes.

If the running process executes a PAMELA primitive (e.g., a **pam_P**, a **pam_V**, or a **pam_delay**), then an event takes place. At that moment, the process that is running returns control to the process scheduler, which reconsiders the consequences for further execution of processes. Depending on the PAMELA construct executed, the process scheduler reacts as follows:

pam_P: If the semaphore is zero, then the running process must block on this semaphore. In this case, the process scheduler places the process in the appropriate semaphore queue in the BPQ and makes the next runnable process from the RPQ the running process. Otherwise, the running process decrements the semaphore and the process scheduler allows the process that is running to continue.

pam_V: This leads to two different cases. In the first case, the value of this semaphore (e.g., semaphore A) changes from zero to one. This causes the first blocked process in the appropriate semaphore queue in the BPQ to become runnable. The process scheduler makes this runnable process the new running process. Before it does this, it puts the running process, which has a time stamp t=5, back into the RPQ. The new running process is given the same time stamp as the running process (t=5). In the second case, the running process increments a semaphore and the value of the semaphore becomes larger than one. In this case, the process scheduler allows the process that is running to continue.

pam_delay: This statement adds a certain amount of time to the time stamp of the running process and the process scheduler puts this process back into the RPQ. This process gets ordered in the RPQ according to its new value of the time stamp. The process scheduler makes the next runnable process from the RPQ the running process.

The process scheduler of the RTL is a *non-preemptive* scheduler, which means that the process scheduler cannot preempt a running process in order to regain control. It must wait to regain control until the running process reaches a PAMELA construct that causes an event. Only then does the process scheduler get the control back from a process.

Note that the way processes get scheduled by the process scheduler results in a feasible schedule, which is not necessarily the fastest schedule. Processes are activated as soon as possible because the RTL implements a simple list-scheduler; however, this does not always result in the fastest schedule [De Micheli, 1994]. For an example, look at Table 4.1, which shows the scheduling of processes used to model the system of Figure 4.2. If task B were not scheduled at t=6, but instead slightly later, it could execute in parallel with task G without resource conflict. As a consequence, the resulting total executing time in this case would be $T_{end} = 10$ instead of $T_{end} = 11$.

4.6 Related Work

Traditionally, *Queuing Theory* [see, for example, Lazowska et al., 1984] has been used in performance analysis of computer designs. It can be used to describe a system in terms of servers and independent clients. These clients access the servers following a negative exponential distribution in time. The strength of queuing theory is that it derives a system's steady state behavior analytically. This makes queuing theory very useful for determining the capacity of servers. Queuing theory is, however, too restrictive for our purpose because of its model. This model is not able to describe, for example, condition synchronization between various clients or conditional control flow.

Petri Nets [Petri, 1962; Jensen, 1991] were originally developed to study concurrency and to analyze issues like *liveness*, *deadlock* and *starvation* [Ben-Ari, 1982], using analytical techniques. To use petri nets for performance analysis requires the use of *Timed Petri Nets* [Van der Aalst, 1992]. These petri nets can be analyzed analytically in the time domain only under the assumption that transition takes place using a negative exponential delay. If the system constructed is ergodic, a Markov chain can be constructed that can be analyzed analytically. However, instead of using analytical techniques, simulation can be used to derive the performance of a system [ASPT, 1994].

In contrast to queuing theory and petri nets, *simulation tools* lack an analytical basis. To render performance numbers, simulation tools use simulation instead and typically employ event-based simulation techniques. Simulation tools are popular because they provide simple compositional constructs to build large and complex models and they describe interpreted performance models well. Simulation tools use a textual or graphical representation to define a system. Many different simulation tools exist that are tailored to specific application domains. In the signal processing domain the following tools (to name a few) are available: the Ptolemy system [in particular, the DE-domain, Buck et al., 1992], the Maisie environment [Bagradia, 1995] and the BoNES environment [Cadence, 1998].

Another tool that can be used for performance analysis is VHDL or Verilog. Although these languages were developed originally to describe and simulate hardware at a much lower level, i.e., Register Transfer Level (RTL), a VHDL simulator is basically a parallel simulator using event simulation. In contrast to the previously discussed simulation tools, these languages are able to simulate systems very accurately. Nonetheless, VHDL is also used for performance analysis on a higher level of abstraction, as done, for example, by Rose and Shackleton [1997].

Standard programming languages like C [Kernighan and Ritchie, 1988] or C++ [Stroustrup, 1991] are used instead of simulation tools to construct performance models. These provide a high degree of freedom in constructing models of systems, but at the same time, they lack a systematic approach or well defined simulation model. Furthermore, standard programming languages lack the notion of a simulation engine to handle parallel simulations.

The object oriented modeling approach that uses PAMELA to describe the behavior of each building block as presented in this chapter is a very interesting approach for the following reason. The use of C++ gives us a large amount of freedom to model different architecture instances, as we will show in Chapter 5, where we discuss the modeling of stream-based dataflow architectures. The PAMELA primitives provide high-level modeling primitives like semaphores, processes, and delays. This results, as we will also see in Chapter 5, in compact descriptions when modeling the building blocks of architecture instances. VHDL, for example, does not provide these high level primitives.

The PAMELA method gives accurate performance values using simulation. Other modeling approaches can provide symbolic representations for performance numbers, but these representations are not accurate enough since not all four components of the Performance Modeling Basis are included. For example, queuing theory cannot describe condition synchronization, whereas the analytical evaluation technique of the PAMELA method cannot handle conditional control flow.

The RTL provides a very efficient simulation engine because of its simple structure. We we will quantify the simulation speed in Chapter 7, when we discuss the construction of a retargetable simulator for stream-based dataflow architectures. The execution model of the RTL is more restrictive than the discrete simulation engines used in most simulation tools, but this results in fast simulation.

4.7 Conclusions

The Y-chart approach uses performance analysis to render performance numbers. In this chapter, we presented the Performance Modeling Basis (PMB), which describes the four components determining the performance of a system. We presented the PAMELA method with which to carry out performance analysis. The method describes the four components of the PMB using only three high-level primitives: semaphores, processes and delays.

Our objective in light of Y-chart approach is to model architecture instances easily. To satisfy this objective, we had to modify the original PAMELA method from a procedural modeling approach to make it to an object oriented modeling approach. This led us to the definition of building blocks, which have a structure part and a behavior part. A system is easily constructed by simply combining these building blocks. The structure parts of building blocks use C++ whereas the behavior parts use primitives of the PAMELA run-time library (RTL). The RTL is a simple multithreading package that implements semaphores, processes, and delays. The RTL also provides a process scheduler, allowing us to perform parallel simulations of a system constructed using building blocks.

Bibliography

G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):4 – 43, 1983.

Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

ASPT. *ExSpect 5.0 User Manual*. Eindhoven University of Technology, Eindhoven, the Netherlands, 1994.

Rajive Bagradia. *Maisie User Manual*. UCLA Parallel Computing Lab, 1995. Release 2.2.

M. Ben-Ari. *Principles of Concurrent Programming*. Prentice-Hall, 1982.

Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, 1992. Special issue on Simulation Software Development.

Cadence. Bones 4.0. http://www.cadence.com, 1998.

Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill International Editions, 1994.

E.W. Dijkstra. Cooperating sequential processes. In F. Geunys, editor, *Programming Languages*, pages 43 – 112. Academic Press, 1968.

E.W. Dijkstra. Personal Communication, 1981.

Jonathan Finger. Lightweight tasks in C. *Dr. Dobb's Journal*, pages 48 – 50, 1995.

Adele Goldberg and David Robson. *Smalltalk-80, The Language and Its Implementation*. Addison-Wesley Publishing Company, Xerox Palo Alto Research Center, 1983.

K. Jensen. Coloured petri nets: A high level language for system design and analysis. In K. Jensen and G. Rozenberg, editors, *High-level Petri Nets: Theory and Application*, pages 44 – 122. Springer-Verlag, 1991.

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentine Hall Software Series, second edition, 1988.

Wolfgang Kreutzer. *System simulation, programming styles and languages*. International Computer Science Series. Addison-Wesley, 1986.

Sanjaya Kumar, Jamer H. Aylor, Barry W. Johmson, and Wm. A. Wulf. Object-oriented techniques in hardware design. *Computer*, 27(6):64–70, 1994.

E.D. Lazowska, J. Zahorja, G. Graham, and K. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice-Hall, 1984.

Stephen J. Mellor and Ralph Johnson. Why explore object methods, patterns, and architectures? *IEEE Software*, 14(1):27 – 30, 1997. Guest Editor's Introduction.

Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. Architectural styles, design patterns, and objects. *IEEE Software*, 14(1):43 – 52, 1997. Theme Issue.

Frank Mueller. A library implementatio of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 253–261, San Diego, CA, 1993. `http://www.informatik.hu-berlin.de/ mueller/`.

Marc Nijweide. The pamela run-time library, version 1.3. Technical Report 1-68340-27(1995)06, Laboratory of Computer Architecture and Digital Techniques, Delft University of Technology, 1995. `http://dutepp0.et.tudelft.nl/ gemund/publications.html`.

Open Software Foundation. *OSF/DCE Application Development Guide*, chapter Threads, pages 2–1 to 2–24. Prentice Hall, 1995. release 1.1.

C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.

Guido Post, Andrea Muller, and Rainer Schoenen. Object-oriented design of ATM switch hardware in a telecommunication network simulation environment. In *Proceedings of GI/ITG/GMM Workshop Methoden und Bescheibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, Paderborn, 1998.

Christopher Proven. POSIX threads, 1995. WWW ref only. http://www.mit.edu:8001/people/proven/pthreads.html.

RASSP Taxonomy Working Group (RTWG). RASSP VHDL modeling terminology and taxonomy. `http://rassp.scra.org/documents/taxonomy/taxonomy.html`, 1998. Revision 2.4.

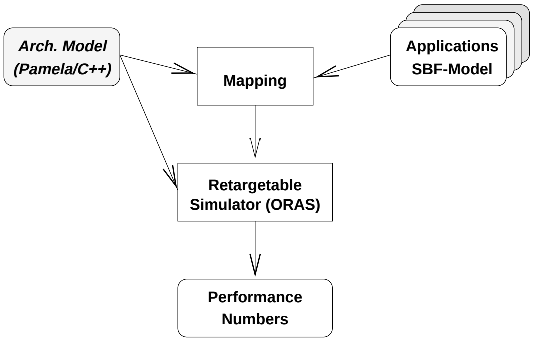
Fred Rose and John Shackleton. Performance modeling of system architectures. *VLSI Signal Processing*, 15(1/2):97 – 110, 1997.

Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.

The Stepstone Corporation. *Objective-C Compiler v4.0*. The Stepstone Corporation, 1989.

W.M.P. Van der Aalst. *Interval Timed Coloured Petri Nets and their Analysis*. PhD thesis, Dept. of Computer Science and Mathematics, Eindhoven University of Technology, 1992.

Arjan. J.C. van Gemund. *Performance Modeling of Parallel Systems*. PhD thesis, Laboratory of Computer Architecture and Digital Techniques, Delft University of Technology, 1996.



Chapter 5

Architectures

Contents

THE subject of this chapter is the modeling of architecture instances of the architecture template of stream-based dataflow architectures that yield cycle-accurate performance numbers. In the Y-chart approach, we must be able to derive architecture instances from an architecture template that we can evaluate to obtain performance numbers. As motivated using the abstraction pyramid shown in Figure 3.3, we want to model those architecture instances at a higher level of abstraction, i.e., we represent only the details relevant to obtaining the performance of an architecture instance, suppressing irrelevant details. As motivated in Chapter 4, the Performance Modeling Basis (PMB) describes exactly those details relevant to describing the performance of a system. The PMB can be expressed using only processes, semaphores and delays. In this chapter, we use these primitives together with the concept of building blocks that was also introduced in Chapter 4 to describe architecture instances cycle-accurately. This way we model architectures at a level that is more abstract than that offered by standard hardware description languages.

We start off in Section 5.1 with explaining what we consider an architecture to be; until now, we have given only pictorial representations of architectures. We also explain what we consider a cycle-accurate model to be. Modeling an architecture in the context of the Y-chart approach implies that we need to derive an instance from the architecture template of stream-based dataflow architectures. In Section 5.2, we explain the four issues involved in this modeling process. In Section 5.3, we explain how executable architecture instances are constructed from building blocks. We construct a building block for each architectural element of the architecture template of stream-based dataflow architectures and we construct executable models of architecture instances by combining these building blocks. Stream-based dataflow architectures are composed of many different architectural elements. In Section 5.4, we model each architectural element as a building block and describe in detail its structure and its behavior. We also show how only the three PAMELA primitives (semaphores, processes and delays) introduced in Chapter 4 are used to model the various architectural elements. We describe an architecture template in Section 5.5, using composition rules. Stream-based dataflow architectures are programmable architectures. In Section 5.6, we explain the programming model of stream-based dataflow architectures. We show how instances of the stream-based dataflow architecture are programmed for a particular application, leading to the routing program that is down-loaded onto the global controller. We conclude this chapter by presenting a comprehensive example in which we program a particular architecture instance for a particular application.

5.1 Architectures

So far, in our references to architectures, and especially stream-based dataflow architectures, we have depended on our rather intuitive understanding of what an architecture might be. Since this chapter is about modeling architectures, we begin by explaining what we consider an architecture to be.

5.1.1 Pictorial Representation

Architectures are normally represented in a pictorial way, using block diagrams. These *block diagrams* describe a network consisting of boxes and lines connecting the boxes to each other. The boxes are given names related to the kind of functions they are supposed to perform. We have been presenting the stream-based dataflow architecture in exactly this way, which is shown again in Figure 5.1. In this figure (which is the same as Figure 2.2, but shown again here for clarification), we gave names to the boxes to indicate their function in the architecture. We gave boxes the name *routers* to indicate that they should transport packets, *buffers* to indicate that they should temporarily store data and *functional units* to indicate that they should select the correct *functional element* at run-time, and so forth.

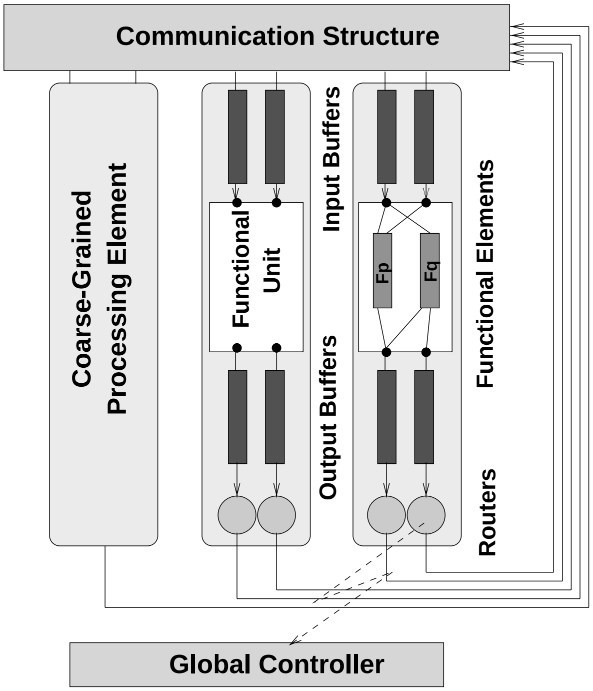


Figure 5.1. A pictorial representation of a stream-based dataflow architecture.

This way of representing an architecture is intuitive an it may lack precision. Yet when it comes to reasoning about architectures, then it is mandatory that the architectures be given well-defined structure and behavior. This does not exclude pictorial representation, but it does require that all constituent parts be well defined and have unique behavior.

5.1.2 Architectures

Architectures can be represented intuitively using a block-diagram representation as discussed previously. We define an architecture as

Definition 5.1. ARCHITECTURES

An *architecture* is a network or graph $G(V, E)$, where V defines a set of architectural elements and E defines a set of connections along which architectural elements can communicate with one another corresponding to a particular model of architecture. □

An architecture is thus made up of a network of interconnected architectural elements. The architectural elements in the network can only communicate with those architectural elements to which it connects. We define an architectural elements as

Definition 5.2. ARCHITECTURAL ELEMENT

An *architectural element* is either an architecture or an entity having a particular structure and behavior. □

Examples of architectural elements are routers, functional elements, and global controllers. Each of these architectural elements describes a particular behavior. A router routes packets, a functional element executes a particular function, and the global controller governs the flow of packets. Using these architectural elements, we can describe, for example, the stream-based dataflow architecture shown in Figure 5.1 as a network of architectural elements, giving the architecture a distinct structure and behavior in time.

Alternative Behaviors

Stream-based dataflow architectures describe a class of architectures within which architectural choices are present. A communication structure, for example, can have either a FCFS or TDM behavior. Similarly, functional units can have a packet-switching or sample-switching behavior, or a buffer can have a handshake, a bounded FIFO, or an unbounded FIFO buffer behavior.

Given these choices, we look at architectural elements as having a particular *property* that is realized using one or more behaviors. The property of a communication structure is that it communicates packets from routers to input buffers, the property of a functional unit is that it selects the correct functional element at run-time, and the property of a buffer is that it stores data temporarily. This leads to the notion of architectural element types, which we define as

Definition 5.3. ARCHITECTURAL ELEMENT TYPE
An *architectural element type* describes the aggregated properties of an architectural element without implementing these properties using a particular behavior. □

Therefore, if we use architectural element type 'Buffer' in an architecture, we indicate that it has to store data temporarily, without specifying which behavior implements this property. We can still choose between a handshake behavior, a bounded FIFO behavior, and an unbounded FIFO behavior.

5.1.3 Cycle-accurate Model

An architectural element describes the behavior of a piece of hardware that it models. It describes the function of the hardware in time, in which time is divided into cycles that we define as

Definition 5.4. CYCLE
A *cycle* is multiple of a clock cycle as used in synchronous hardware. □

The duration of a behavior of an architectural element can be expressed in cycles. Since an architectural element models a piece of hardware, it can only describe the behavior of the hardware it represents functionally correctly at the level of cycles. As such, an architectural element is a cycle-accurate model of the hardware. We define a cycle-accurate model to be

Definition 5.5. CYCLE-ACCURATE MODEL
A *cycle-accurate model* is a model that describes the behavior of the actual hardware that it represents correctly at time instances separated by cycles. □

We use cycle-accurate models of hardware, e.g. the architectural elements, because it simplifies the description of the behavior of architectural elements significantly. We rely in this chapter on the PMB presented in Chapter 4 to describe the architectural element cycle-accurately. The four components of the PMB can be expressed in terms of high-level primitives like processes, semaphores, and delays. These high-level primitives permit us to specify the behavior of architectural elements in less detail, as we will show in Section 5.4. Due to the abstract descriptions, a consequence is that the simulation speed increases and that it become easier to modify the behavior of architectural elements.

5.2 Modeling Architectures

Modeling an architecture in the context of the Y-chart implies that we derive architecture instances from the architecture template; for example, the architecture template of stream-based dataflow architectures described in Chapter 2. The problem of constructing an executable performance model of an architecture instance involves the following four issues:

1. We must be able to describe an architecture template.
2. We must be able to define particular architecture instances of an architecture template.
3. We must be able to construct an architecture instance such that we can obtain performance numbers.
4. We must be able to program the architecture instance such that it can execute applications.

The four issues are shown in Figure 5.2. This figure shows the architecture template from which an architecture instance is derived. Which instance that is, must be specified in some way, most likely in a textual format. In the modeling process, architectural elements are interconnected according to the specification of an architecture instance, resulting in an executable performance model. Finally, the architecture instance needs to be programmed to execute a particular application.

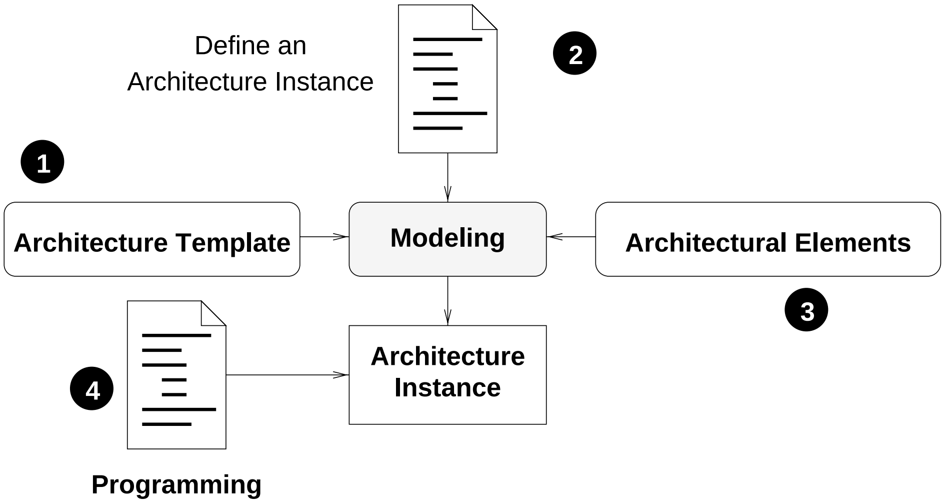


Figure 5.2. The four issues involved in performance modeling in the context of the Y-chart.

In the remaining part of this chapter, we describe the four elements of Figure 5.2 in separate sections.

5.3 Modeling Architectures using the PMB

To model an architecture instance at a high level of abstraction, we express architectural elements in terms of the PMB. This implies that we have to identify the four components of the PMB in stream-based dataflow architectures. Functional units and functional elements, for example, are driven by the availability of data and we can implement this behavior using condition synchronization. Routers communicate exclusively with one global controller. We implement this kind of interaction using mutual exclusion. Functional elements execute functions to process data samples taking a certain amount of time, which we implement using execution time. Packets have different lengths and the length of a packet determines the amount of time a functional element requires to operate on a particular packet. We implement this behavior using conditional control flow.

5.3.1 Machine-oriented Modeling Approach

We use the machine-oriented modeling approach to describe architecture instances of the stream-based dataflow architecture, i.e., we view an architecture as a machine that processes material. The machine is described by the architectural elements like buffers, functional units, and routers. The material is described by the stream that flows through the architecture. As streams (the material) flow through the architectural elements (making up the machine) they are processed by the architectural elements. How the architectural elements operate together to process streams is controlled by the routing program that is down-loaded onto the global controller. Thus, the architectural elements define the resources and the streams define the workload.

In the machine-oriented modeling approach, we model resources mainly using active primitives [Kreutzer, 1986]. We describe functional elements, functional units, and routers using one or more processes. We describe the remaining resources, i.e., the communication structure, the global controller, and buffers, using semaphores. Although we could have modeled these resources as active elements, we use semaphores because these lead to a more efficient simulation. In contrast to the material-oriented modeling used in Chapter 4, here data dependencies and tasks representing a workload are distributed over various architectural elements.

5.3.2 Building Blocks

We introduced building blocks in Chapter 4 to describe a system in terms of the PMB. Building blocks describe architectural elements well, because architectural elements are also self-contained entities having a particular structure and behavior. Therefore, if we describe architectural elements as building blocks, we can describe architecture instances by simply combining building blocks (see, for example, Figure 4.5).

Describing a FIFO Buffer

A building block is an object and an object consists of a data part and methods. Methods define an interface that other objects use to communicate with an object. If we consider the architectural element ‘FIFO’, we know that we want to write data to the buffer to store it and, later, read this data from the same buffer using a FIFO behavior. To access the FIFO buffer we define a *read* and a *write* method. The implementations of the read and write methods of object FIFO are given in Program 5.1.

Because we model a buffer as a passive element, we use only **pam_P** and **pam_V** statements that operate on the semaphores `data` and `room`. The FIFO buffer has a particular capacity and we initialize the semaphore `room` with the value `cap` and initialize the semaphore `data` with the value zero. To see how a FIFO buffer behaves, we need a process that writes to the buffer and a process that reads from the buffer, i.e., the classic Producer/Consumer example [Ben-Ari, 1982].

Program 5.1. THE READ AND WRITE METHODS OF A FIFO BUFFER

```
FIFO::read
{
  pam_P (data);                  // Is there data available?
  aSample  = queue[readfifo];   // read from buffer
  readfifo = (++readfifo)%cap;
  pam_delay (1);                   // Reading takes one cycle
  pam_V (room);                  // Tell there is room again
}
FIFO::write( aSample )
{
  pam_P (room);                  // Is there Room on the FIFO?
  queue[writefifo] = aSample;   // Write in buffer
  writefifo = (++writefifo)%cap;
  pam_delay (1);                   // Writing takes one cycle
  pam_V (data);                  // Tell there is data available
}
```

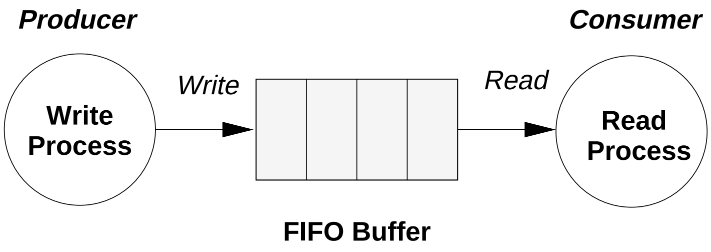


Figure 5.3. A simple Producer/Consumer system.

Producer/Consumer Example

A simple producer/consumer system is given in Figure 5.3. A write **process** tries to write samples into a FIFO buffer that has a capacity of 4 positions. Concurrently, a read **process** tries to read samples stored in the FIFO buffer. The FIFO buffer is modeled as an object with a `read` and a `write` method, as given in Program 5.1. The write **process** accesses the FIFO buffer using the `write` method and the read **process** accesses the FIFO using the `read` method.

If the write **process** wants to write a sample into the FIFO buffers by using the `write` method, there needs to be room available in the buffer. This condition is checked using the **pam_P(**`room)` statement. If no room is available, the write **process** blocks until data becomes available. When room is available, sample `aSample` is written in the circular buffer `queue` and then a **pam_V** is executed on `data`.

Concurrently with the write process, the read **process** tries to read samples from the FIFO buffer with the `read` method. To read a sample, there needs to be data available. This condition is checked using the **pam_P(**`data)` statement. If no data is available, the read **process** blocks and remains blocked until data becomes available. Semaphore `data` initially causes the blocking of the read **process**, but after the write **process** executes **pam_V(**`data)`, indicating that data is available, the read **process** unblocks, taking a sample from the `queue`. This is followed by executing **pam_V(**`room)` to indicate that room is again available to store a new sample.

Cycle-Accuracy and Cycle Calibration

Reading or writing a sample into the FIFO buffer takes time and must be taken into account when modeling hardware. In both the methods `read` and `write` we therefore included the **pam_delay** (1) statements, which causes the reading and writing of samples to take time. The read **process** trying to read a sample at t=10 will execute the **pam_delay** statement in the `read` method and, as a consequence, the time stamp of the read **process** changes from t=10 to t=11. The same applies, after the necessary changes, for writing a sample into the FIFO buffer.

In a real hardware implementation, this reading and writing might require x clock cycles. If we take 1 time unit, which is modeled with the **pam_delay** (1) statement, equal to these x clock cycles, we relate the virtual PAMELA time to the notion of clock cycles. Now each time unit is a cycle representing x clock cycles. We calibrate a cycle by taking the number of clock cycles required to read a sample from a buffer or to write a sample to a buffer.

5.3.3 Architectural Element Types

We have shown that we can construct a FIFO buffer to which a sample can be written or from which a sample can be read. In the architecture template of the stream-based dataflow architectures, however, we indicated that buffers can have various behaviors, of which the FIFO behavior is only one choice. In the example given in Program 5.1, the behaviors of the `read` and `write` methods are fixed, while we want them to be a choice.

Object Oriented Techniques

To implement such options, we rely on object oriented programming principles like polymorphism and inheritance. An object is instantiated from a class definition. We can describe a hierarchy of class descriptions using *inheritance*, which specifies that a derived class inherits the data part and methods of its parent class. These derived classes thus have at least the same methods as their parent class, but they can re-implement methods using *late binding* or *polymorphism*. This technique allows one and the same method to have different implementations describing different behaviors. Finally, there exists in object oriented programming *abstract classes* which only define methods but no implementations. In this case, derived classes only inherit the method description and it is the responsibility of the derived class to implement these methods. We use abstract class definitions to provide a uniform interface for derived classes.

The Buffer Type

We use abstract classes and inheritance to describe architectural element types. For each type we define an abstract class and derive new classes implementing the abstract methods. In this way, we attain architectural element types with a uniform interface having different behaviors. Using abstract classes, we defined a uniform interface that other types can use to communicate with this type, without their knowing which behavior a type actually uses.

Architectural element type *buffer* has several possible implementations (e.g. handshake, bounded, and unbounded FIFO). Therefore we describe type *buffer* as an abstract class defining the methods *read* and *write*. Since class *buffer* is abstract it does not provide an implementation for these methods. We derive new class descriptions from class *buffer* that represent a handshake buffer behavior, a bounded FIFO buffer behavior, and an unbounded FIFO buffer behavior. These derived classes have

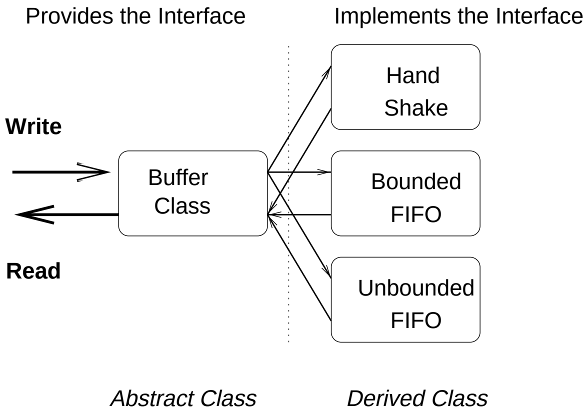


Figure 5.4. The buffer class defines the interface (e.g., the read and write methods) that other classes implement.

the same two methods *read* and *write* from the buffer class via inheritance, but implement them differently, resulting in a different behavior. The handshake buffer, the bounded FIFO buffer, and the unbounded FIFO buffer implement the two methods using a single buffer position, a circular buffer, and a linked list, respectively.

The separation between the abstract class *buffer* and the three derived classes is illustrated in Figure 5.4. Architectural element type *buffer* can store data temporarily using the uniform interface which consists of a read and a write method. Nevertheless, type *buffer* can store data using three different behaviors.

For architectural elements that can have more than one behavior (see Table 2.1), we construct abstract classes as shown in Figure 5.5. For these abstract classes, we derived classes that provide the implementation of the methods of the abstract classes. In Figure 5.5, we see again the architectural element type *buffer* with its three implementations as described in Figure 5.4: handshake, bounded FIFO and unbounded FIFO. One architectural element type, the processing element, does not have an implementation. It represents a hierarchical architectural element that serves as a *container* for other element types.

Revised Producer/consumer System

We have revised the way we model the producer/consumer system shown in Figure 5.3. In Figure 5.6, we show the producer/consumer system modeled using building blocks. It shows three building blocks of which two are processing elements (A and B) that exchange samples with each other via the FIFO buffer (FIFO Buffer). Processing element A is the producer and processing element B is the consumer.

The representation of the active building blocks A and B was already explained in Chapter 4. Note that the structural part of A and B contains references to the architectural element type Buffer and not to a FIFO buffer. In the **process** description of processing element A, the **process** writes a sample to the Out buffer. In **process** description of Processing Element B, the **process** reads a sample from the In buffer. In contrast to the **process** descriptions given in Chapter 4, these processes start all over again when they have written a sample to a buffer or when they have read a sample from a buffer. Thus, these processes read and write samples *continuously*; in other words, they operate on streams.

The representation of the passive building block FIFObuffer does not encapsulate a process, but instead uses methods to describe the behavior of the building block. The methods read and write

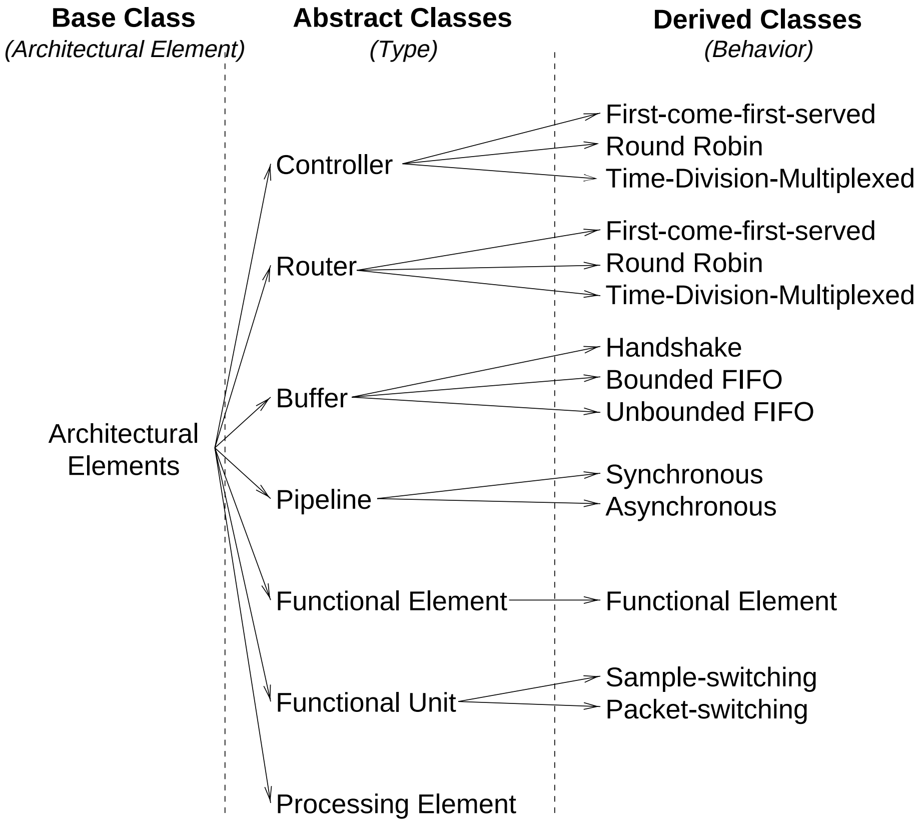


Figure 5.5. The class hierarchy of architectural elements and different behavior implemented by derived classes.

(of which the implementation is given in Program 5.1), operate on the two semaphores room and data. Strictly speaking, the structure part of the FIFO buffer is void, since this building block does not contain structural information. However, for clarification, we included the two semaphores stored in the data part of the object FIFO.

Notice that the processes are unaware of whether the buffers they read to or write from by calling the methods read and write are handshake buffers, bounded FIFO buffers, or unbounded FIFO buffers. The read and write methods of the FIFO buffer are selected via polymorphism. Note also that the processes are shielded from the knowledge of which buffer they are reading from or writing to in the system. Neither inBuffer nor outBuffer relates to a specific element in the system. Only when a processing element is instantiated as part of total system does the inBuffer of PE B relate to buffer In and the outBuffer of PE A to buffer Out.

5.4 Modeling the Architectural Elements as Building Blocks

In this section, we explain how the architectural elements of stream-based dataflow architectures are described as building blocks. In Table 5.1, we show the architectural element types that we consider, along with the section in which each type is discussed.

As Table 5.1 shows, we make a clear distinction between active building blocks and passive building blocks. If we describe an active building block, we describe its **process**. If we describe a passive

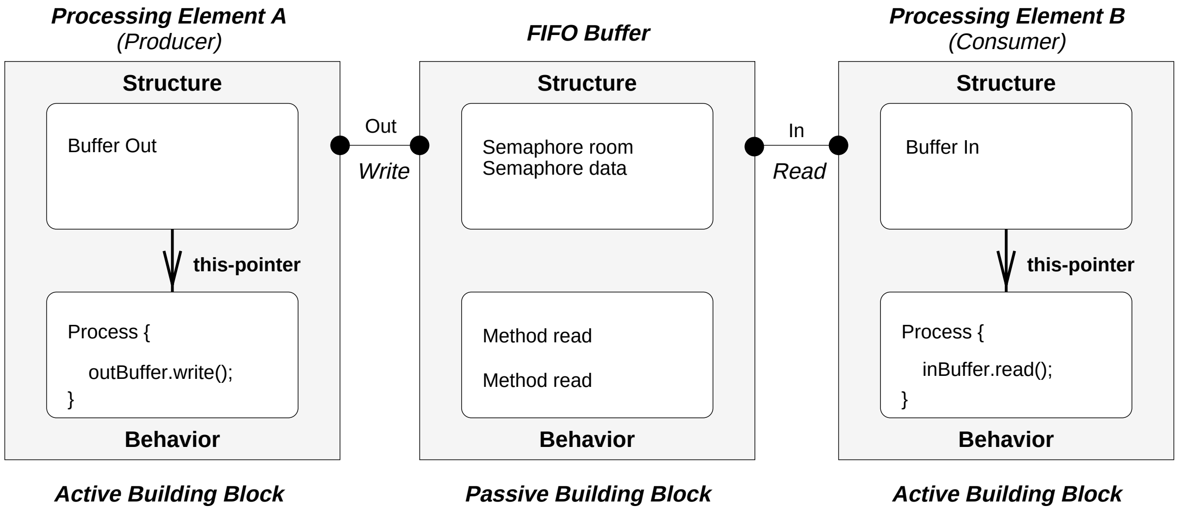


Figure 5.6. The producer/consumer system, revised to included building blocks to model the system.

element, we describe its **methods** using only **pam_P**, **pam_V**, and **pam_delay** statements. Although a type can have more than one behavior, we explain only one behavior. We make an exception for functional units and functional elements of which we discuss both the packet-switching and sample-switching of functional units and functional elements. We also make an exception for type pipeline; for this type, we discuss both the synchronous and asynchronous pipeline behavior.

In Table 5.1, we make the distinction between *workload modeling* and *resource modeling*. In the workload modeling part, we explain how we represent streams that flow through architecture instances. In the resource modeling part, we explain architectural element types. In discussing types, we start by describing two types that serve only as containers for other types.

In describing the structure and behavior of architectural elements, we use the same pseudo C language as used in Chapter 4. Although pointers are used in the real code, we avoid the use of pointers in the program descriptions, to make the code more readable. Furthermore, we used constructs of the *Standard Template Library* (STL) [Musser and Saini, 1996] in the code. This library provides high-level programming constructs like *vectors* and *sets*. These constructs appear as `vector<>` and `set<>`. The object on which the constructs operate is given between the <> brackets. Sometimes a vector is used where the use of a set would be more intuitive. However, a vector can be implemented more efficiently than a set, leading to faster execution. In general, we write C code in such a way that we obtain the fastest executable code. In Chapter 7, we show that the speed at which we can evaluate the performance of an architecture instance is a very important quality.

5.4.1 Packets

Streams that flow through the architecture are each partitioned into packets, forming a stream of packets. Packets consist of a header part and a variable data part. The data part contains samples and the header part contains header fields. We construct objects for both data types. We created the class `Sample` for the sample object, and `Header` for the header object. To be able to store both types in a buffer, we have to create an abstract class `Token` from which we derive class `Samples` as well as class `Header`. The class `Token` thus represents an arbitrary data structure that we treated as a monolithic

Architectural elements Type	Section	Behavior	Passive/Active
Workload Modeling			
Packet	5.4.1	Sample	
		Header	
Resource Modeling			
Architecture	5.4.2	*Container*	
Processing Element	5.4.3	*Container*	
Communication Structure	5.4.4	SwitchMatrix	Passive
Global Controller	5.4.5	FCFS	Passive
Buffer	5.4.6	Bounded FIFO	Passive
Router	5.4.7	FCFS	Active
Functional Unit	5.4.8	Sample-Switching FU	Active
		Packet-Switching FU	Active
Functional Element	5.4.9	Function Execution	Passive
		Packet-switching Read	Active
		Sample-switching Read	Active
		Write	Active
Pipeline	5.4.10	Synchronous	Passive
		Asynchronous	Passive
Ports	5.4.11	Write Ports	Passive

Table 5.1. Outline of discussion of various architectural elements.

entity.

Sample

A *sample* has a value `value` and carries a time stamp `time_stamp`. The structure of a sample is given in Structure 5.1.

Structure 5.1. THE STRUCTURE OF A SAMPLE

```
int       time_stamp;
double    value;
```

Object sample carries a true value that is operated upon by the function implemented by a functional element. Because an individual sample can hold a value, we can simulate the functional behavior of an architecture correctly. This allows us to construct both uninterpreted and interpreted performance models.

Header

A *header* consists of four fields: the base field `base`, the source field `source`, the function field `function` and the length field `length`. The structure of a header is given in Structure 5.2.

Although header consists of four fields, we modeled it as one data structure. The `base` (H_b) and `source` (H_s) fields are involved in the routing of packets through an architecture instance. The

Structure 5.2. The Structure of a Header

```
int      base;
int      source;
int      function;
int      length;
```

`function` field (H_f) stipulates which function of a functional unit should process the data part of a packet and, finally, the `length` field (H_l) indicates the number of samples contained in the data part of the packet.

5.4.2 Architecture

Type *architecture* consists of the following types: a communication structure `communication`, a global controller `controller`, and a set of processing elements `processingElements`. The structure of element *architecture* is given in Structure 5.3.

Structure 5.3. The Structure of Element Architecture

```
GlobalController          controller;           /* structure */
CommunicationStructure    communication;
set<ProcessingElement>    processingElements;
```

Type *architecture* captures the top-level structure of a stream-based dataflow architecture (see, for example, Figure 2.1). It does not describe a behavior, because it serves only as a container for other types.

5.4.3 Processing Element

Type *processing element* (PE) consists of a vector of input buffers `inBuffers`, a vector of output buffers `outBuffers`, a vector of routers `routers` and one functional unit `functionalunit`. The structure of element *processing element* is given in Structure 5.4.

Structure 5.4. The Structure of Element Processing Element

```
vector<Buffer>     inBuffers;         /* structure */
FunctionalUnit     functionalUnit;
vector<Buffer>     outBuffers;
vector<Router>     routers;
```

Type *processing element* does not describe a behavior, because it only serves as a container for other types. Two special kinds of PEs do exist: a *source PE* and a *sink PE*. The source PE produces packet-streams and the sink PE consumes packet-streams.

5.4.4 Communication Structure

Type *communication structure* contains the parameter `capacity` and the semaphore `channels`. The structure of element *communication structure* is given in Structure 5.5.

Structure 5.5. THE STRUCTURE OF ELEMENT COMMUNICATION STRUCTURE

```
  int           capacity; /* parameter */
  semaphore     channels; /* semaphore */
```

Type *communication structure* provides for the communication of streams of packets between processing elements. It has a number of channels available, the total of which equals `capacity`. The semaphore `channels` is initialized with the value of `capacity`. The semaphore describes condition synchronization. If we assign a value of 1 to `capacity`, we model a bus structure consisting of one channel. If an architecture instance contains M output buffers and we assign this value M to `capacity`, we model a switch matrix.

We model a switch matrix behavior of type *communication structure* as a passive element. Its methods are given in Program 5.2. A router **process** claims a channel on the communication structure using the **method** `claimChannel`. It checks the availability of channels by executing **pam_P(**`channels`**)**. If no channels are available, the router **process** blocks until a channel becomes available. A router **process** releases the channel by calling the **method** `releaseChannel`, which executes **pam_V(**`channels`**)**.

Program 5.2. METHODS OF THE COMMUNICATION STRUCTURE

```
method  claimChannel
{
  pam_P ( channels );
}
method  releaseChannel
{
  pam_V ( channels );
}
```

5.4.5 Global Controller

Type *global controller* contains the `routing_program`, the semaphore `requests` and the two parameters: `service_time` and `capacity`. The structure of element *global controller* is given in Structure 5.6.

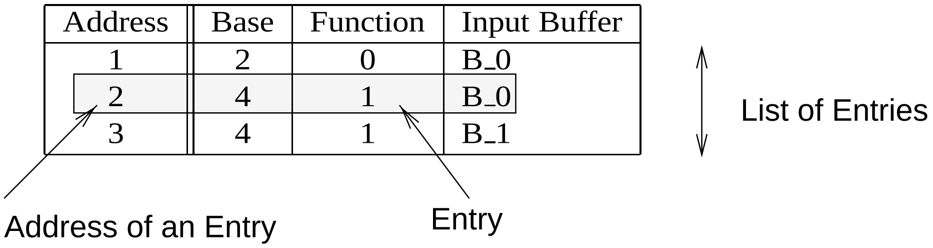
Structure 5.6. STRUCTURE OF ELEMENT GLOBAL CONTROLLER

```
  Program    routing_program;      /* parameters */
  int        service_time;
  int        capacity;
  semaphore  requests;             /* semaphore */
```

Type *global controller* steers the flow of packet streams through an architecture instance. It includes a *routing program* containing the information that routers use to route packet streams through an architecture instance. For that purpose, routers interact with the global controller, which provides new header information (how this is done is explained later when we discuss routers). The router uses this information to update the header information of the packet that it is currently processing. The global controller also provides a reference to an input buffer. A router must send the packet that it is currently processing to this referenced input buffer.

The global controller can handle a certain amount of requests from routers in parallel, as indicated by the parameter `capacity`. The semaphore `request` is initialized using this parameter value and describes mutual exclusion. When a router posts a request to the global controller, it takes the global controller a certain amount of time (in cycles) to serve that request. The parameter `service_time` indicates how many cycles such a request takes, describing execution time in terms of the PMB.

The global controller contains a routing program. This is a list of entries in which each entry contains three fields: a *base value*, a *function value*, and a *reference* to an input buffer. These three fields are part of the programming model of stream-based dataflow architectures (which we explain in detail in Section 5.6). The global controller can address the entries in the list. It uses the base field of a received header to do this. An example of a program list is given in Table 5.2, where the first column defines the address of an entry and the three values next to the address describe the content of the entry: the three fields mentioned previously. Suppose a header arrives at the global controller. The base field of this header carries the value 2. The global controller must therefore look up the entry with address 2, which is represented in Table 5.2 by the gray box. This entry contains the value 4 for the base field, the value 1 for the function field and references to input buffer `B_0`.



Address	Base	Function	Input Buffer
1	2	0	B_0
2	4	1	B_0
3	4	1	B_1

List of Entries
Address of an Entry
Entry

Table 5.2. Example of a routing program of the global controller.

We model the FCFS behavior of type *global controller* as a passive element. One of its methods is given in Program 5.3. The global controller also implements methods similar to the ones described in Program 5.2 that are called `claimController` and `releaseController`. These methods operate on semaphore `requests`. After a router **process** claims the global controller using **method** `claimRouter`, it calls the **method** `newHeader` to access the global controller.

The global controller decodes the received header `aHeader`, to gain access to the base field of the header. This gives the address `address` of the required entry in the routing program. Before the global controller assigns new values to the header, it gives the value of the base field to the source field of the header, as part of the programming model. Next, the base and function fields receive a new value based on values stored in the entry in the routing program. This takes `service_time` cycles and is modeled by delaying the router **process** interacting with the global controller for `service_time` cycles using the **pam_delay** statement. After this delay, the global controller returns a reference to a new input buffer part of an architecture instance. The router **process** will send the packet that it is currently processing to this input buffer.

5.4.6 Buffer

Type *buffer* contains the parameter `capacity`, and two semaphores `room` and `data`. The structure of element *buffer* is given in Structure 5.7.

Type *buffer* interconnects either the input of a functional unit to the communication structure or the output of a functional unit to a router. A buffer can store a particular number of samples, as given by the parameter `capacity`. The buffer initializes the semaphore `room` with the parameter

Program 5.3. Global Controller Methods

```
method  newHeader( aHeader )

  // Get The Address from the Base
  int address = aHeader.getBase();

  // Assign the Base field to the Source Field
  aHeader.setSource( aHeader.getBase() );

  // Get New information from the Program
  aHeader.setBase(     program[address].getBase() );
  aHeader.setFunction( program[address].getFunction() );

  pam_delay ( service_time  );

  // Determine new input Buffer
  input_Buffer = program[address].getBuffer();
  return input_Buffer;
```

Structure 5.7. The Structure of a Buffer

```
  int            capacity;      /* parameter  */
  semaphore      room;          /* semaphores */
  semaphore      data;
```

`capacity` and initializes the semaphore `data` with a zero value. Both semaphores describe condition synchronization.

We model the bounded FIFO behavior of type *buffer* as a passive element. Its methods are given in Program 5.4. This program describes the implementation of the **method** `read` and the **method** `write` of a FIFO buffer (These two methods were also discussed in Section 5.3).

The bounded FIFO buffer uses a `queue` to store tokens (`vector<Token>`). When a router **process** reads a `Token` [1] from a FIFO buffer using the **method** `read`, the read **method** first executes **pam_P(**`data)`. If no `Token` is available, the router blocks, implementing a blocking read. It remains blocked until new data becomes available. Otherwise, the read **method** continues by reading a `Token` from the queue `queue`. After the read **method** reads the `Token`, it executes **pam_P(**`room)` to indicate that new room is available on the FIFO buffer.

When a router **process** tries to write a `Token` into the FIFO buffer, it uses the **method** `write`. The **process** executes **pam_P(**`room)`. If no room is available in the FIFO buffer, the router **process** blocks, implementing a blocking write. The router **process** remains blocked until new room becomes available. Otherwise, the **process** continues by storing the `Token` in the queue. After the `Token` is stored, the **process** executes **pam_V** on semaphore `data`, using the write method, to indicate that new data is available on the FIFO buffer.

Before a router **process** writes to an input buffer, it must first have exclusive access to this buffer. For that purpose, the input buffers are extended with a semaphore that implements mutual exclusion. This semaphore is claimed and released in a similar way as used in the communication structure and global controller (described in Program 5.2).

[1] We say `Token`, because it can be either a `Sample` object or a `Header` object

Program 5.4. The methods of a FIFO Buffer

```
method  read
{
  pam_P (data);                // Is there data available?
  aToken = queue[readfifo];     // Read from the buffer
  readfifo = (++readfifo)%cap;
  pam_V (room);                 // Tell there is room again
}
method  write( aToken )
{
  pam_P (room);                  // Is there Room on the FIFO?
  queue[writefifo] = aToken;    // Write in the buffer
  writefifo = (++writefifo)%cap;
  pam_V (data);                  // Tell there is data available
}
```

5.4.7 Router

Type *Router* contains a reference to a global controller `controller`, a reference to the communication structure `communication` and a buffer `bufferIn`. Its structure is given in Structure 5.8.

Structure 5.8. Router

```
 Controller              controller;     /* structure */
 CommunicationStructure  communication;
 Buffer                  bufferIn;
```

Type *router* is responsible for the routing of packets produced by a functional unit to its new destination in an architecture instance, via the communication structure. A router interacts with the global controller for each packet to provide arriving packets with new header information.

We model the FCFS behavior of type *router* as an active element. Its **process** is given in Program 5.5. It starts by reading a header from the buffer `bufferIn`, which connects to the router at its read side and a functional unit at its write side. It takes the router **process** 4 cycles to read in a header (i.e., one cycle for each header item). Then the router **process** claims the global controller `controller` to get new header information and a reference to a buffer `outBuffer`. The router **process** sends the packet-stream from the `bufferIn` to the `outBuffer`. Before it can do this, it must claim a channel on the communication structure `communictionStructure`. In addition, the router **process** checks whether the `outBuffer` is available [2]. When both a channel and the `outBuffer` are available, the router **process** reads in a sample from buffer `bufferIn` and writes out the sample to buffer `outBuffer`, taking in total one cycle. The router **process** repeats this routine until it has read and written a `numberOfSamples` samples to the new input buffers, describing the conditional control flow of the PMB. The router **process** releases the `outBuffer` and the channel on the communication structure and starts all over again with reading a new header.

[2] The names `inBuffer` and `outBuffer` buffer are given as seen from the router, because of the localized references technique. This sometimes leads to confusing names because the output buffer of a router is the input buffer connected to a functional unit.

Program 5.5. A First-Come-First-Served Router Process

```
process  FCFS_router {
    // Read Header
    header = bufferIn.read();
    pam_delay ( 4 );

    controller.claimController(); {
      // And decode to which new buffer the data should go
      bufferOut = controller.newHeader( header );
      // Assign the output Buffer to the outputPort
      outputPort( bufferOut );
    }
    controller.releaseController();

    communicationStructure.claimChannel(); {
    bufferOut.claimBuffer();
    {
        // Send out the New Header
        bufferOut.write( header );
        pam_delay ( 4 );
        // Determine how many Samples are in a Packet
        numberOfSamples = header.getLength();
        // Read the rest of the packet!
        for numberOfSamples do {
           aSample = bufferIn.read();
           pam_delay ( 1 );
           bufferOut.write( aSample );
        }
     }
     // Packet is processed, we can continue
     bufferOut.releaseBuffer();
    }
    communicationStructure.releaseChannel();
  }
}
```

5.4.8 Functional Unit

Type *Functional Unit* (FU) consists of a vector of input ports `inputPorts`, a vector of output ports `outputPorts` and a vector of functional elements `repertoire`. The structure of element *Functional Unit* is given in Structure 5.9.

Structure 5.9. STRUCTURE OF A FUNCTIONAL UNIT

```
vector<ReadPort> inputPorts;            /* structure */
vector<WritePort> outputPorts;
vector<FunctionalElement> repertoire;
```

Type *FU* contains of a number of functional elements (FE) which defines the repertoire $\mathcal{F}$ (see Equation 2.1). At run-time, the FU selects the correct FE to execute. Besides activating the correct FE, it also strips off headers from packets at input ports and prepares new headers on output ports. Two special kinds of FUs exist: a *source FU* and a *sink FU*. A source FU does not have input ports. A sink FU does not have output ports. A source FU is exclusively part of a source PE, and a sink FU, of a sink PE.

We model the behavior of type *FE* as an active element. The process describing the behavior is referred to as the local controller of the FU. The local controller can describe two different behaviors: a packet-switching behavior where the local controller switches between FEs in between packets, and a sample-switching behavior where the local controller switches between FEs in between samples.

We are now going to discuss the behavior of the local controller for the packet-switching **process** described in Program 5.6 and the sample-switching **process** described in Program 5.7.

Packet-Switching

The packet-switching **process** switches between FEs on the boundaries of packets. In this mode, the FEs of the repertoire can share input and output buffers (or ports, because a buffer connects to a port) of a FU. A special port, the *opcode port* (which connects to the opcode buffer), is reserved for the run-time selection of the FE. The header arriving at the opcode port is used by the local controller to activate a specific FE. The opcode port is always the first input port of a FU (i.e. `inputPorts[0]`). We have explicitly named this port (and buffer) to emphasize its special role.

A model of a FU is depicted in Figure 5.7. The FU has two input ports and two output ports (represented by black circles in the figure). The repertoire of the FU consists of two FEs: FE_0 and FE_1. The FEs connect via their ports to the ports of the FU. This is called the *binding* of the FE to the FU. In this mode, FEs always share the opcode port of the FU. In the example of Figure 5.7, the two FEs also share an output port of the FU.

The **process** representing the local controller of the FU is represented as the white disc in Figure 5.7. Although not shown explicitly in the figure, the **process** connects to all input and output ports to strip off headers from read ports and prepares new headers on write ports. The **process** activates a FE using the semaphores Execute and Done.

The packet-switching **process** starts by reading a header from the `OpcodePort`. It gets the correct function field `func` from the header via **method** `getFunction`. Next, the **process** selects from the vector `repertoire` the `func`-th FE. The **process** strips off the header from the input ports of this FE and places new headers on the output ports of this FE. Next, the **process** activates the FE from the repertoire by executing **pam_P(**`fe.execute`). This semaphore is connected to the functional element `fe`. This is directly followed by executing **pam_V(**`Done`), which causes the **process** to

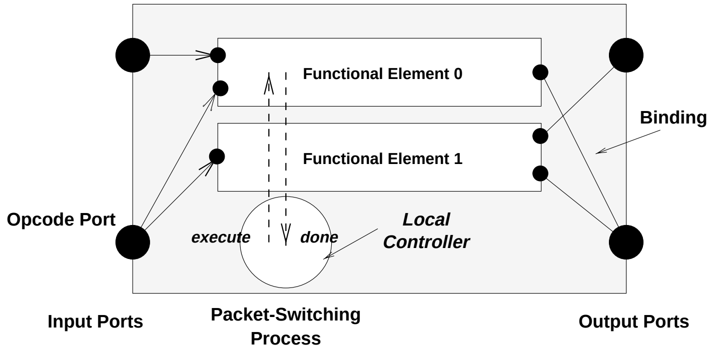


Figure 5.7. Model of a Packet-Switching FU.

block: the **process** has given control to the functional element `fe`. In packet-switching mode, the functional element will read `length` samples from the opcode port before it returns control to the FU. The functional element returns control by executing **pam_V(**`Done`). This causes the blocked **process** to continue, which will start all over again by reading a header from the opcode port. The semaphores `Execute` and `Done` describe a dependency between the FU and a functional element and thus describe the condition synchronization of the PMB.

Program 5.6. The Packet-Switching Process

```
process  packet_controller {
    // Read a Header from the Opcode Buffer
    header            = opCodePort.readHeader();

    // Determine the correct function of the repertoire
    func              = header.getFunction();
    // Resolve which function from the repertoire we want to use
    fe = repertoire[func];

    // How many Inputs and Outputs do we have for the function
    inputPorts  = fe.getInputBufferVector();
    outputPorts = fe.getOutputBufferVector();

    // Read Header from the Buffers, except the Opcode Buffer
    foreach inputport ∈ inputPorts\{opCodePort} {
        inputport.stripHeader();
    }

    // Prepare a Header on the Outputs that are involved
    foreach outputport ∈ outputPorts {
        outputport.pushHeader( header );
    }

    fe.setLength( length );

    pam_V ( fe.execute ); // hand over control to the functional Element

    pam_P ( fe.done); // Wait until the Functional Element terminates
}
```

Sample-Switching

The sample-switching **process** switches between FEs on the boundaries of samples. In this mode, FEs cannot share buffers and each FE requires its own set of input and output buffers. Each buffer relates exclusively to a particular FE. In sample-switching mode, the function field is not used to select the correct FE from the repertoire. The sending of a packet to a particular buffer already determines which FE will operate on the data part of the packet. Nonetheless, each FE requires its own opcode port. The header read from this port is needed to create new headers on the output ports of a FE and to determine when to read the next header, because `length` samples are read from the opcode buffer. In Figure 5.8, a sample-switching FU is shown that has a repertoire of two functional elements: FE_0 and FE_1. Each FE has its own set of input and output buffers. Each FE also has its own opcode port.

The **process** representing the local controller of the FU is represented as the white disc in Figure 5.8. It describes a Round Robin scheduler that decides which FE to activate using the semaphores Execute and Done.

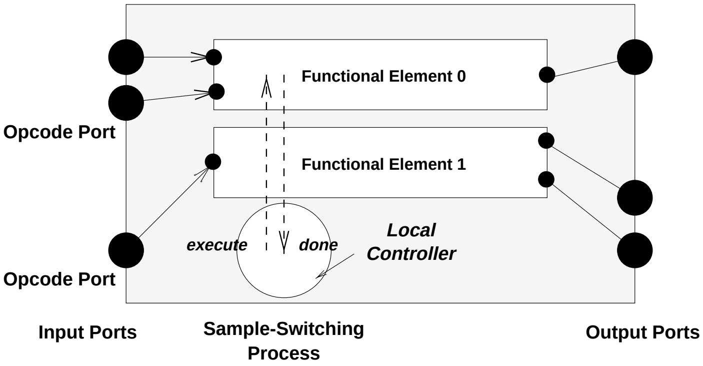


Figure 5.8. Model of a sample-switching FU.

The **process** checks whether all FEs from the repertoire `repertoire` are runnable in a Round Robin fashion (What is required for an FE to be runnable, will be explained later when we discuss the FEs). The **process** checks whether a FE is runnable by testing the semaphore `fe.execute` using the **pam_T** statement of the Run-Time Library. A semaphore is either *blocking* or *non-blocking*. If semaphore `fe.execute` of FE `fe` tests as being blocked (i.e. `REQUEST`), it represents a request to the Round Robin scheduler that it wants to execute. The **process** activates the FE (i.e. **pam_V(**`fe.execute`)) and waits until the FE finishes (i.e. **pam_P(**`fe.done`)).

When none of the FEs in the repertoire has posted a request at a particular time instance T, the **process** tries again to check the FEs one cycle later (i.e., at time $T + 1$) by advancing its time using the **pam_delay** (1) statement.

5.4.9 Functional Element

Type *Functional Element* (FE) consists of a vector of input ports `inputPorts`, a vector of output ports `outputPorts`, a pipeline `pipeline` and a function `function`. It also contains the parameters `latency` and `initiation_period` and the semaphores `execute` and `done`. The structure of Element *Functional Element* is given in Structure 5.10.

Type *functional element* executes function `fe_function` on the data part of packets. It reads this data from the input ports with a particular throughput, given by the parameter `initiation`

Program 5.7. SAMPLE SWITCHING FU PROCESS

```
process   FUsample_controller {
    activate = false;
    // Check requests to execute from all FEs in the repertoire
    foreach fe ∈ repertoire {
      if ( pam_T ( fe.execute ) == REQUEST ) {
       activate = true;
       pam_V ( fe.execute ); // Grant the Request
       pam_P ( fe.done );    // Wait until FE terminates
      }
    }
    if (activate == false) {
      // No FE posted a request, advance
      // time by one cycle and try again
      pam_delay (1);
    }
}
```

Structure 5.10. THE STRUCTURE OF A FUNCTIONAL ELEMENT

```
  vector<ReadPort>    inputPorts;       /* Structure */
  vector<WritePort>   outputPorts;
  Pipeline            pipeline;
  Function            fe_function;

  int                 latency;          /* Parameters */
  int                 initiation_period;

  semaphore           execute;          /* Semaphores */
  semaphore           done;
```

period [3]. Functions in the domain of signal processing are often pipelined. Therefore, samples reside in the FE for a certain amount of time depending on the depth of the pipeline. The parameter *latency* indicates how many stages a pipeline contains. We indicate each stage as being a *slot* in the pipeline.

Type functional element consists of three parts, as shown in Figure 5.9. It consists of a *read* part, a *pipeline* and a *write* part. We model the read and write parts as active elements and the pipeline as a passive element. A FE has a read **process** and a write **process** which are both given as white circles in Figure 5.9. The pipeline is shown in between the two processes. The semaphore pair execute and done relate only to the read **process** and describe the same semaphores as given in Figure 5.7 and Figure 5.8.

In describing the functional element, we first look at how function fe_funtion executes, followed by a discussion on the read **process** and write **process** of a functional element. The pipeline is discussed later in Section 5.4.10.

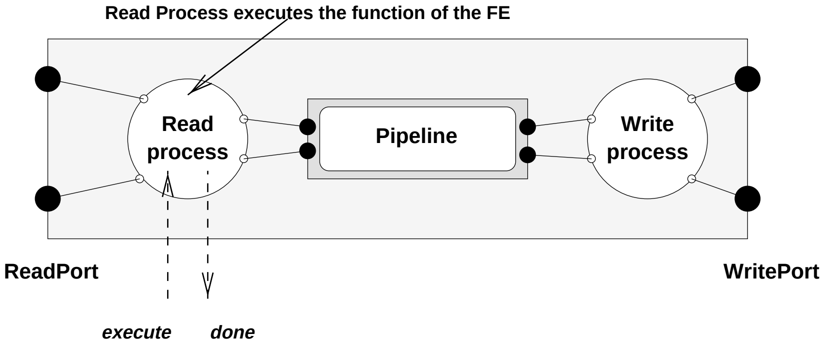


Figure 5.9. Model of a functional element.

Function Execution

The FE executes a signal processing function. For that purpose, the read **process** calls the **method** executeFunction22 as given in Program 5.8. This causes the function fe_function to execute, consuming two input samples and producing two output samples.

When the read **process** executes **method** executeFunction22, it starts to read the samples sample0 and sample1 from respectively input ports portIn[0] and portIn[1] of the FE. The samples read by calling the **method** getSample belong to the data part of packets because the local controller of the FU has already removed the headers on both ports. After reading the samples, the function fe_function executes, consuming the two input samples while producing the two new samples new_sample0 and new_sample1.

The function fe_function is not pipelined, since it does not operate concurrently on different sets of samples. Nevertheless, we want to describe pipelined functions. Hence, we have to model the pipeline behavior explicitly, as we explain next.

Modeling Pipeline Behavior Suppose we want to describe the behavior of a function that is pipelined latency slots [4] deep. If we put a new sample in the pipeline at each new cycle (e.g. let initiation_period equal one), the first sample should leave the pipeline latency cycles later at the earliest. Using this

[3] Recall that the initiation period is the reciprocal of throughput

[4] Also referred to as the *depth* of a pipeline

Program 5.8. Execute a 2 input, 2 output Function

```
method  executeFunction22 {
    sample0 = portIn[0].getSample();
    sample1 = portIn[1].getSample();

    // Execute the FE-function
    (fe_function)(sample0, sample1, &new_sample0, &new_sample1);

    // Calculate when these Samples are ready to leave the FE
    leave_time = pam_time () + (latency * initiation_period );

    // Pass the Execution to the sample
    new_sample0.setTime( leave_time );
    new_sample1.setTime( leave_time );

    // Put results in the Pipeline
    pipeline_portIn[0].putSample( new_sample0 );
    pipeline_portIn[1].putSample( new_sample1 );
}
```

assumption, we model a pipeline as follows. The read **process** executes the function `fe_function` instantaneously. Let the function produce new samples at, say, time instance t. The read **process** calculates for these samples a time t_l, which is the time instance at which these samples are allowed to leave the FE. The time t_l is calculated as

$$t_l = t + latency. \tag{5.1}$$

The time t_l is stored in the time stamp (see Structure 5.1) of the samples produced by the function `fe_function` and the read **process** puts the samples in the pipeline.

The pipeline is basically nothing more than a FIFO queue. When the read **process** writes samples in a slot, all slots in the queue move one place. At the end of the queue, the write **process** wants to read a slot with samples. However, the pipeline does not give the slot to the write **process** before the time t' of the write **process** equals or is greater than the time stamp t_l of the samples stored in the slot read by the write **process**.

$$t' \geq t_l \tag{5.2}$$

When the time of the write **process** satisfies Equation 5.2, the pipeline allows the write **process** to read the slot. As a consequence, the write **process** write the samples contained in the slot to output ports at time t_l or later. Thus the samples leave the FE at the correct moment in time.

The initiation period value `initiation_period` influences the *residence time* of samples in the pipeline. If the initiation period of a FE increases, for example from one sample per cycle to one sample per 2 cycles, the residence time of the sample in the pipeline increases. The pipeline of the FE is filled with new slots at a lower pace. The time the samples can leave the FE instead becomes equal to

$$t' \geq t + (initiation\_period * latency) \tag{5.3}$$

Notice that we have completely *uncoupled* the function's execution from the time it takes to execute. This provides a very flexible way to change the behavior of the function, without needing to rewrite the function. We only have to change the parameters `latency` and `initiation_period` to obtain a function that behaves as if it were pipelined more deeply.

We defined a specific method in Program 5.8 to execute a function with 2 input arguments and 2 output arguments. It would be more convenient to have a generic method that calls any function with n inputs and m outputs. However, a C function call always requires a fixed number of input arguments while producing a fixed number of output results. Therefore, a generic solution is not possible and we need a specific method for all different combinations of numbers of input arguments and output results.

Read and Write Processes of a Functional Element

Type FE can have two possible behaviors: a packet-switching behavior or a sample-switching behavior. The packet-switching and sample-switching behaviors differ in how they describe the read **process**, but the write **process** is the same for both behaviors.

Packet-Switching Read Process of a FE A FE in packet-switching mode has a read **process** as given in Program 5.9. When this **process** starts, it immediately blocks on `execute` using a **pam_P**. The FU in which the FE resides can activate the **process** by executing **pam_V(**`execute)`. See, for example, the end of Program 5.6 which describes the packet-switching **process** of a FU. Here the **pam_V** is executed on semaphore `fe.execute`, which is the same semaphore as `execute`. When the **process** unblocks, it gets into a for-loop to guarantee that it reads `length` samples from the opcode port. In the loop, the FE tries to get a slot on the pipeline `pipeline`. It blocks if no slot is available. If a slot is available, the read **process** executes the **method** `executeFunction22` as shown in Program 5.8. This causes the execution of function `fe_function` and the function results are stored in the pipeline. Following this, the read **process** indicates to the pipeline `pipeline` that new data is available by executing **method** `readSlot` of the pipeline. Next, the read **process** is delayed `initiation_period` cycles using the **pam_delay** statement. After the read **process** has read `length` data samples from the opcode port, it executes **pam_V(**`done)`. The read **process** has processed a complete packet from the opcode buffer and gives back control to the local controller. This describes the conditional control flow of the PMB. After the control is given back, the read **process** blocks again on the **pam_P**`execute` until the FU re-activates the FE.

Sample-Switching Read Process of a FE A FE in sample-switching mode has a read **process** as given in Program 5.10. Recall that FEs do not share buffers in sample-switching mode. Therefore we model the header processing within the read **process**, which is differently from the packet-switching mode, where the header processing is done by the local controller of the FU.

The read **process** starts by reading a header from the opcode buffer. This is followed by removing headers from other input ports `inputports` and producing new headers on the output ports `outputports`. After the header processing, the read **process** gets in a for-loop to make sure it reads at least `length` samples from the opcode port. The read **process** tries to get a slot on the pipeline `pipeline`. If a slot is available, the read **process** executes the **method** `executeFunction22` as shown in Program 5.8. This causes the execution of function `fe_function` and the function results are stored in the pipeline. Then the read **process** posts to the Round Robin scheduler of the FU in which it resides that it wants to execute. It executes **pam_P** on semaphore `execute`, which is the same semaphore as `fe.execute` in Program 5.7, and describes the sample-switching **process** of a FU. The read **process** blocks on the semaphore and remains blocked until the Round Robin scheduler grants the read **process** permission to proceed by executing **pam_V(**`execute)`. What follows after that was explained when discussing the read **process** in packet-switching mode. The only difference is that the read **process** already executes **pam_P(**`done)` already after it has processed one function

Program 5.9. READ PROCESS FUNCTION OF A PACKET-SWITCHING FE

```
process  execute_packet {
  // Wait until the local controller of the Functional Unit reads the
  // Headers from the input ports
  pam_P ( execute );

  // Read at least the complete data part of the Opcode port
  for i=0 to length step 1
  {

    // Get a write slot on the Pipeline
    pipeline.getReadSlot();

    // Call the Function
    Execute_Function22();

    // Arguments are stored in Slot, and stable
    pipeline.readSlot();

    // Model the throughput of this Functional Element
    pam_delay ( initiation_period );

  }
  // Complete Packet is processed, indicate to the FU this FE is done

  pam_V ( done );
}
```

execution. Therefore, the Round Robin scheduler can execute another read **process** of another FE in the next cycle. Finally, the read **process** continues until it has read `length` samples. Then the read **process** has processed a complete packet from the opcode buffer and the read **process** starts to read a new header from the opcode port.

Write Process of a FE The write **process** is the same for packet-switching and sample-switching FEs. It is given in Program 5.11. The write **process** starts by obtaining a slot from the pipeline `pipeline` using the **method** `getWriteSlot`. If no slot is available, the write **process** blocks. If a slot is available, the write **process** receives the samples from the pipeline slot and puts them into the appropriate output port, using the **method** `putSample`. The write **process** reads the slot only when its time matches the time stamp of the samples. How this takes place is explained later when we discuss the pipeline. After the write **process** has written all the samples of the pipeline slot into the output ports, it releases the pipeline slot using the `writeSlot` method. This frees up a slot in the pipeline for use by the read **process** of the FE.

5.4.10 Pipeline

Type *Pipeline* consists of a vector of input ports `inputPorts` and a vector of output ports `outputPorts`. It also contains the two semaphores `room` and `data`. The structure of element *pipeline* is described in Structure 5.11.

Type *pipeline* models the pipeline in a FE. A read **process** puts samples onto the read ports `inputPorts`. A write **process** puts samples onto the write ports `outputPorts`. There are always as many input ports as output ports. Besides ports, the pipeline accommodates a queue `queue` of slots (`vector<slot>`). This is a FIFO queue, implemented as a circular buffer. A slot on the queue con-

Program 5.10. Read Process Function of a Sample-Switching FE

```
process  execute_sample {

    // Read a Header from the first input Buffer
    header = opCodePort.readHeader();

    // Read Header from the Buffers
    // Start at 1, because the Opcode buffer is already read
    foreach inputport ∈ inputPorts\{opCodePort} {
        inputport.stripHeader();
    }

    // Prepare a Header on the Outputs that are involved
    // The binding is already resolved in the FE
    foreach outputport ∈ outputPorts {
        outputport.pushHeader( header );
    }

    length = header.getLength();
    for i = 0 to length step 1{

       // Get a write slot on the Pipeline
       pipeline.getReadSlot();

       // Call the FE Function
       Execute_Function22();

       // Tell the Functional Unit we have the samples
       // and want to execute the function
       pam_P ( execute );

       // Arguments are stored in Slot, and stable
       pipeline.readSlot();

       // Model the throughput of this Functional Element
       pam_delay ( initiation_period );

       // Tell the Functional Unit we are done with the samples
       pam_V ( done );

     }
}
```

Program 5.11. Write Process Function of a FE

```
process  FE_writeprocess {
    // First we get the arguments from the Pipeline
    pipeline.getWriteSlot();

    j = 0;
    // Read a sample from the pipeline slot to the
    // Appropriate output port
    foreach outputport ∈ outputPorts {
        aSample = pipeline_portOut[j].getSample();
        outputport.putSample( aSample);
        j = j + 1;
    }
    // Give the slot free
    pipeline.readSlot();
}
```

Structure 5.11. The Structure of a Pipeline

```
vector<ReadPort>         inputPorts;  /* Structure */
vector<WritePort>        outputPorts;

semaphore                room;        /* Semaphores */
semaphore                data;
```

sists of a vector of samples (`vector<Sample>`). The number of samples stored in the slot depends on the number of output ports of the FE. If a FE has three output ports, then a slot contains three sample positions. The latency parameter `latency` of the FE function stipulates the number of slots contained in the queue. Thus, if the function has a latency of 7, then 7 slots are accommodated in the queue.

A pipeline is shown in Figure 5.10. It has two input ports, two output ports and a queue containing seven slots. Each slot can store two samples. The semaphores `room` and `data` provide the synchronization between the read **process** and write **process** connected to the pipeline. The semaphores describe condition synchronization.

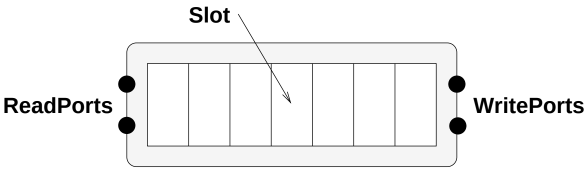


Figure 5.10. Model of a pipeline. It has two input ports, two output ports and a queue containing seven slots. Each slot can store two samples.

Type *pipeline* has two behaviors: a *synchronous* and an *asynchronous* pipeline behavior.

A synchronous pipeline allows a write **process** to take a slot only if all of the slots in the pipeline are full. If the write **process** reads a slot, it cannot read a new slot until the read **process** writes a new slot.

An asynchronous pipeline has its read **process** uncoupled from the write **process**. A write **process** can read slots from the pipeline while a read **process** is blocking and a read **process** can write slots while a write **process** is blocking. If the write **process** reads slots, empty slots appear in the asynchronous pipeline that are also referred to as *bubbles*. A read **process** compresses these bubbles when it writes slots while the write **process** blocks.

We model the behaviors of type *pipeline* as a passive element. Program 5.13 goes with the read **method** of a pipeline. Program 5.12 goes with the write **method** of a pipeline.

The Write Methods of Pipelines

The write methods of a pipeline are given in Program 5.12. When a write **process** demands a slot from the pipeline using the **method** `getWriteSlot`, the write **process** checks the semaphore `data` using the **pam_P** statement to verify whether a full slot is available in the queue `queue`. If so, then the write **process** receives the slot from the queue. It takes out a singe sample from the slot, namely the one at slot position 0. The write **process** checks the sample's time stamp, which indicates when the sample should leave the FE. The write **process** calculates the difference `delay` between the time of the time stamp and its own current time as given by **pam_time**. If `delay` is less than zero, the current time is ahead of the time stamp. In this case, the sample must have experienced some delay because of congestion. Nevertheless, it is ready to leave the FE. If `delay` is greater than zero, then the write **process** delays itself `delay` cycles, using the **pam_delay** statement. After the elapse of this delay time, the samples of the slot leave the FE at the time instance calculated by the read **process**.

After the write **process** has received a slot and has written all the samples of the slot into output ports, it signals that the slot is available again, using the `readSlot` method. This method executes **pam_V(**`room`**)**, which indicates to the read **process** that a free slot is again available.

Program 5.12. THE WRITE METHODS OF PIPELINES

```
method  getWriteSlot
{
  // Is a slot with data available?
  pam_P (data);
  slot= queue[readfifo];
  readfifo = (++readfifo)%cap;

  // Get a sample from the slot
  aSample = slot[0];

  // Check time stamp against the current time
  delay = aSample.getTime() - pam_time () ;

  // Slow down the sample if needed
  if (delay > 0) {
    pam_delay ( delay );
  }
}
method  writeSlot
{
  --full_slots;
  // There are free slots available
  pam_V (room);
}
```

The Read Methods of Pipelines

The read methods of a pipeline are given in Program 5.13. The read **process** claims a slot on the pipeline using the **method** `getWriteSlot`. The read **process** determines whether a slot is available by checking the semaphore `room` using a **pam_P**. If a slot is available, then the pipeline receives the slot from the queue. Moreover, the read **process** initializes each slot position to `NULL`. This becomes important when multi-rate functions are used. In that case the function does not always produce samples (this will be explained in Chapter 6, when we discuss stream-based functions). The 'NULL' allows the write **process** to distinguish between a new written sample and empty slot positions, so that it knows whether it has to write a sample to an output port or not.

When a read **process** has written to a slot, it uses the **method** `readSlot` to signal that the slot contains data. We model the distinction between synchronous and asynchronous pipelines in the way we define the **method** `readSlot` in Program 5.13. The first `readSlot` **method** is for a synchronous pipeline. This **method** indicates that data is available only when all slots of a pipeline are occupied. If a pipeline contains `cap` slots, data is only available when `cap` slots are full. If the pipeline is full (i.e. `full_slots` equals `cap`), the read **process** indicates that data is available by executing

pam_V(`data`**)**. Hence, a write **process** can read a slot from the pipeline only when the read **process** completely fills the pipeline.

The second `readSlot` **method** in Program 5.13 is for an asynchronous pipeline. In this case, the read **process** writes slots which the write **process** reads as soon as they are written using the **method** `getReadSlot`. The write **process**, however, is delayed until its time stamp has the correct value. In the case of an asynchronous pipeline, the read **process** signals immediately that data is available after it executes the `readSlot` method, by executing **pam_V(**`data`**)**.

Program 5.13. THE READ METHODS OF A PIPELINE

```
method  getReadSlot
{
  // Is there a slot available on the Pipeline?
  pam_P (room);
  slot = queue[writefifo];
  writefifo = (++writefifo)%cap;
  foreach sample ∈ slot {
       sample = NULL;
  }
}
method  ReadSlot (Synchronous Pipeline)
{
  full_slots++;
  // Only when the pipeline is completely full, data becomes available
  if (full_slots == cap)
    pam_V (data);
}
method  ReadSlot (Asynchronous Pipeline)
{
  full_slots++;
  pam_V (data);
}
```

5.4.11 Ports

Ports act as interfaces between different architectural elements. For example, they handle the headers in the FUs and FEs. Ports of FUs and FEs interface to input or output buffers. Pipeline ports interface between the read **process** and write **process** and the pipeline. Ports come in different variants. We now take a closer look at one of them, the write port which connects to output buffers. This port processes the new headers on the output ports. It also determines the time it takes to write a sample to an output buffer.

Write Port

The structure of a write port of a FU (or a FE, because, for that matter, in both cases the write ports are the same) is given in Structure 5.12. The structure description should actually contain only the buffer `outBuffer`, but for an explanation of how ports behave, we show other elements as well.

The write port connects to the buffer `outBuffer`. It processes the headers that precede streams of samples produced by FEs. Both the packet-switching **process** of a FU and the sample-switching read **process** of a FE push headers onto the queue `headerQueue` in the write port. The write port keeps track of the status of a packet using the variables `length` and `state`. Finally, the write port has a variable `Offset` which plays an important role in the programming model of the architecture, as we explain in Section 5.6.

Structure 5.12. THE STRUCTURE OF A WRITEPORT

```
Buffer         outBuffer;        /* structure */
deque<Header>  headerQueue;

int           length;
int           state;

int           Offset;
```

The write port requires a header queue, because FEs are pipelined. It can happen that a packet-switching **process** of a FU or sample-switching read **process** of a FE tries to push a header onto the write port using **method** `pushHeader` before the processing of the previous packet has completed. A read **process** can already start to process a new packet before the write **process** has finished processing the old packet. Nevertheless, a write port knows when a packet is done, using the two state variables previously mentioned. The variable `length` indicates the length of the packet. The variable `state` keeps track of the progress made with the filling of a packet of length `length`.

The methods of a write port are given in Program 5.14. The write **process** of a FE writes samples to a write port using the **method** `putSample` as shown in Program 5.11. Because the function of a FE does not necessarily produce results on its outputs, it may be that `aSample` represents a `NULL` instead of a sample. If a `NULL` is present, it is ignored because it indicates that the function did not produce a result. Otherwise the write **process** checks whether `aSample` is the first sample of a packet. If so, it writes a header, followed by sample `aSample`.

Writing a sample to this output port and thus to the output buffer `outBuffer` requires one cycle for the write **process** and is modeled with a **pam_delay** statement. When a write **process** of a FE writes a sample at time instance t, it can write the next sample no sooner than at time instance $t + 1$.

When the write **process** must write a header, it pops the first header from the `headerQueue`. It modifies the base field of the header by adding an offset (`Offset`) to the base field. Within the **method** `writeHeader`, the write **process** obtains the `length` of the header and, finally, the write **process** writes the header to the output buffer. This takes four cycles, as modeled by the **pam_delay**. The write **process** thus experiences an additional delay of 4 cycles when writing a header.

5.5 Describing an Architecture Template

The building blocks discussed in the previous section need to be combined to describe an architecture instance of the architecture template of stream-based dataflow architectures. In this section, we look at how we can describe an architecture template and how we can specify a given architecture instance of the architecture template. Moreover, we describe how building blocks are combined to realize an executable architecture instance.

5.5.1 Composition Rules

We describe an architecture template in terms of *composition rules* that state which architectural element types are allowed to connect with each other and to what extent. We use the *Backus-Naur Form* (BNF) [Backus and Naur, 1959] to describe these composition rules.

At the top level, stream-based dataflow architectures consist of a global controller, a communication structure and a list of processing elements as explained in Chapter 2. The composition rule

Program 5.14. WRITE PORT METHODS

```
method  pushHeader( aHeader )
{
  headerQueue.push( aHeader );
}
method  putSample ( aSample )
{
  // Write the data if valid
  if (aSample != NULL)
    // Sample is valid
    if ( state == 0)
      // First Sample, write the Header First
      writeHeader();

    // Write the Sample
    pam_delay ( 1 );
    output_buffer.write( aSample );
    state = ++(state)%length;

}
method  writeHeader
{
  // Each time we write a Header, we have to take it off the Queue
  header = headerQueue.pop();

  // Get the correct offset of the output port
  header.setBase( header.getBase() + Offset );

  // Get the length of the packet
  length = header.getLength();

  // While writing the Header, give back control to caller
  pam_delay ( 4 );
  output_buffer.write( header );
}
```

expressing this top-level structure is expressed in BNF as follows:

```
Architecture := Global_Controller Communication_Structure List_Of_Pes
```

Next, we further specify what a list of processing elements (PEs) implies. A list of processing elements should contain at least one or more processing elements. We express this requirement in a recursive way as follows:

```
List_of_Pes := PE
           |  List_of_Pes PE
```

In this rule the "|" symbol indicates a choice: we can select this or ("|") that. Thus, we select either a single PE (PE) or a list of PEs (List_of_Pes) that could again consist of a single PE or a list of PEs, and so forth. Different variants exist of a processing element. They can either be a regular PE, a source PE, or a sink PE. The composition rule describing the existence of these three types is stated in BNF as follows:

```
PE := Regular_PE
   |  Source_PE
   |  Sink_PE
```

Recall that the "|" symbol indicates a choice. If we look further at these three variants, then we see that each variant has its own set of requirements in terms of the functional unit type, the use of input and output buffers and the use of routers. The composition rule expressing these requirements for each of the three PE variants is as follows:

```
Regular_PE := Input_Buffers FU Output_Buffers Routers
Source_PE  := Source_FU Output_Buffers Routers
Sink_PE    := Input_Buffers Sink_FU
```

We can continue to construct composition rules until we have completely described the architecture template of stream-based dataflow architectures. We have done just that; the BNF rules describing the complete architecture template of stream-based dataflow architectures are given in Appendix A.

5.5.2 Architecture Description Language

The composition rules used to describe the architecture template can also be used to define the *grammar* of a language. We use the BNF rules of Appendix A to define the grammar of the *Architecture Description Language*. We can use this language to describe architecture instances of the architecture template.

To compare a textual description of an architecture instance with the grammar, we used the tools *Flex* [Paxson, 1990] and *Bison* [Donnelly and Stallman, 1992]. Flex breaks up the textual description into keywords and other identifiers. Bison matches these keywords and identifiers against the grammar of the language (i.e., the set of BNF rules given in Appendix A) and activates the approriate action if a feasible composition rule is found.

Creating a Building Block for an Architectural Element

We now consider the composition rules for architectural element type *buffer* and show how and when a parser creates a building block for this type. The composition rule expressing the three different behaviors of type *buffer* is:

```
Input_Buffer := Bounded_FIFO
              | Unbouned_FIFO
              | HandShake
```

Now, if we look futher at the composition rule related to `Bounded_FIFO`, we find a composition rule that looks as follows:

```
Bounded_FIFO := INPUTBUFFER '{' TYPE ':' FCFS '(' NUM ')' ';' '}'
              {
                // Associated Action in C++
                Buffer* buffer = new FIFO( $7 );
              }
```

In this composition rule, we introduce *keywords* like `INPUTBUFFER`, `TYPE` and `FCFS`. The parser uses these keywords to decide where to break up a textual description defining a particular architecture instance. It then tries to match these keywords with the defined set of BNF rules. If the parser finds a valid rule, like the one given for `Bounded_FIFO`, it activates the action associated with that rule, which is given between the curly brackets. We associated the action of creating a building block for the FIFO buffer with the composition rule for the `Bounded_FIFO`, by instantiating an object of class `FIFO` using the `new` statement.

In the instantiation of this object or building block, the parser passes along a parameter. In the case of the FIFO buffer, the parser passes on the parameter representing the capacity of the FIFO buffer. The parameter is the seventh position in the composition rule of `Bounded_FIFO`, which corresponds with the `NUM` field describing a numerial value. This value is represented by `$7` and is passed on to the constructor of the FIFO buffer and bound to the parameter `capacity` (shown in Structure 5.7).

Specify an Architecture Instance

Using the architecture description language, we can specify one particular architecture instance of the stream-based dataflow architecture. As an example, in Figure 5.11 we show the architecture description of the architecture instance shown in Figure 5.12. We include the same terms (preceded by the symbol "//") as used in Figure 5.12 in this architecture description.

In the first lines, we define a communication structure (`Communication`) and a global controller (`Controller`), followed by a list of processing elements (`ProcessingElement`). The list consists of three processing elements: a sink PE, a regular PE, and a source PE. For the regular PE we define the input and output buffers (`InputBuffer`, `OutputBuffer` respectively), the routers (`Router`) and the functional unit (`FunctionalUnit`). We also give the regular PE the name `Filter`. The functional unit of PE `filter` consists of two functional elements (`FunctionalElement`), named `LowPass` and `HighPass`. The ports of the functional element bind to the ports of the functional unit in a particular way as described in the binding part (`Input` and `Output` part).

```
Architecture Dataflow {
   Communication { Type:  FCFS ( 10);  }
   Controller { Type:  Fcfs ( 1,  5 );  }

   ProcessingElement Source(0,1) {                                                     // B_3
      OutputBuffer { Type:  BoundedFifo (  100 );  }
      Router { Type:  Fcfs ;  }                                                        // R_0
      SourceUnit {
         Type:  Burst (packets= 120,base= 45 );
         FunctionalElement input(0,1) {                                        // FE_{source}
            Function { Type:  GeneratorSource(file= in ); }
            Binding {
               Output ( 0->0 );
             }  }  }  }

  ProcessingElement Filter(2,2) {
      InputBuffer { Type:  BoundedFifo(  100 );  }                               // B_0 & B_1
      OutputBuffer { Type:  BoundedFifo(  100 );  }                              // B_4 & B_5
      Router { Type:  Fcfs; }                                                    // R_1 & R_2
      FunctionalUnit {
         Type:  Packet;
         FunctionalElement LowPass(1,2) {                                      // FE_{source}
            Function { Type:  LowPass(initiation_period= 1,latency= 18);  }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
               Output ( 1->1 );
             } }
         FunctionalElement HighPass(2,1) {                                     // FE_{source}
            Function { Type:  HighPass(initiation_period= 1,latency= 10);  }
            Binding {
               Input ( 0->0 );
               Input ( 1->1 );
               Output ( 0->1 );
             }  }  }  }

   ProcessingElement Sink(1,0) {
      InputBuffer { Type:  BoundedFifo(  100);  }                                      // B_2
      FunctionalUnit {
         Type:  Burst(packets= 120);
         FunctionalElement Sink(1,0) {                                           // FE_{sink}
            Function { Type:  GeneratorSink (file= out);  }
            Binding {
               Input ( 0->0 );
             }  }  }  }
}
```

Figure 5.11. An Example of an architecture description.

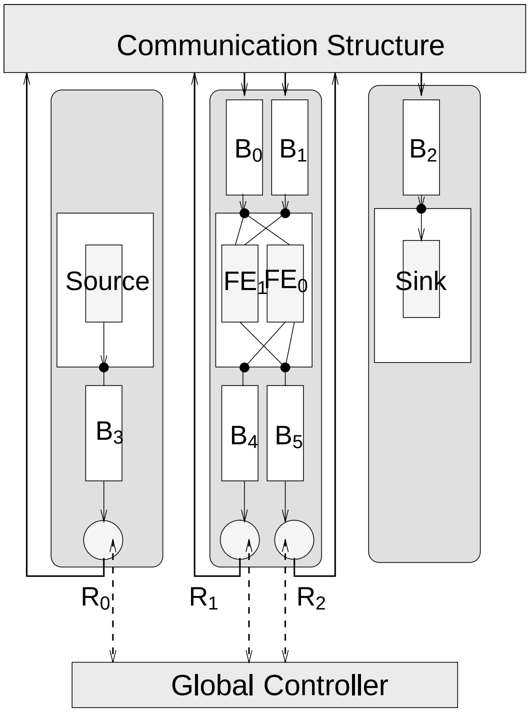


Figure 5.12. A particular stream-based dataflow architecture instance.

Levels of Parameterization

We defined in Chapter 2 the architecture template of stream-based dataflow architectures as a list of parameters as shown in Table 2.1. The architecture description, however, uses a language to make the construction of an architecture instance easier and more intuitive, while defining the same list of parameters, albeit implicitly. Parameters now appear in three different forms: *structural*, *behavioral*, and *functional*. We now describe these three forms of parameterization and illustrate how they surface in the architecture description shown in Figure 5.11.

Structural Parameters: A variable number of architectural elements can be used to describe an architecture instance. The number of processing elements used and the number of functional elements used in a functional unit are both examples of structural parameters. In the architecture description, for example, we have selected three processing elements. For one of these, the regular processing element, we selected a functional unit containing two functional elements.

Behavioral Parameters: An architectural element type has one or more behaviors. For example, the communication structure has two different behaviors (FCFS or TDM), and the input and output buffers have three different behaviors (handshake, bounded FIFO, and unbounded FIFO). A behavioral parameter selects a behavior from the list of available behaviors. In the architecture description, the keyword *Type* indicates these behavioral parameters. The behaviors selected for each type are emphasized by writting them in italics. For example, the communication structure is of type *FCFS* and the input buffers are of type *BoundedFifo*.

Functional Parameters: Each behavior of an architectural element can have its own particular parameter. For example, a FIFO buffer requires a parameter value indicating its capacity (for all buffers the capacity is set to `100` samples) and the global controller requires a value for the number of requests it can handle in parallel (`1`) from routers and how many cycles a request takes (`5`).

The functions installed onto the functional element can also have parameters. In the architecture description, the functions `GeneratorSource` and `GeneratorSink` in the source and sink PE, respectively, have a parameter indicating from which file the `GeneratorSource` should read (`file=in`) or to which file the `GeneratorSink` should write (`file=out`). The functions `LowPass` and `HighPass` have parameters like `latency` and `initiation_period`, but could also have additional parameters that express, for example, an array of filter coefficients.

5.6 Programming an Architecture Instance

Stream-based dataflow architectures are programmable architectures. We need to execute applications on architecture instances, and therefore we want to program the architecture instance. Programming an architecture instance means that we down-load a routing program onto the global controller of the architecture instance. The global controller is responsible for the correct routing of packets through an architecture instance. For that purpose, routers interact with the global controller to obtain new header information that they store in the header of the packet they process. Which information to store in such a header is determined in the *programming model* used in stream-based dataflow architectures. Before we explain this model [5], we first have to say very briefly something about how we model applications as a network and what mapping of an application onto an architecture instance implies. After we discuss modeling of applications and mapping, we introduce the programming model.

5.6.1 Application Network

Applications are modeled as a *network* or a direct graph $G(V, E)$. In these graphs, the nodes V execute functions and the edges E represent FIFO buffers. The functions consume streams of samples from the buffers and produce streams of samples on buffers. The buffers interconnect the nodes in a point-to-point fashion. An example of an application modeled as a network is given in Figure 5.13. The nodes designate the function $\mathsf{f_A}, \mathsf{f_B}, \ldots, \mathsf{f_Q}$, and $\mathsf{f_{source}}$ and $\mathsf{f_{sink}}$. The $\mathsf{f_{source}}$ produces a stream of samples and the $\mathsf{f_{sink}}$ consumes the streams. How each function consumes and produces samples, or alternatively, what the model of computation is, will be discussed in detail in Chapter 6. Only structural information of the network is relevant for the purpose of explaining the programming model.

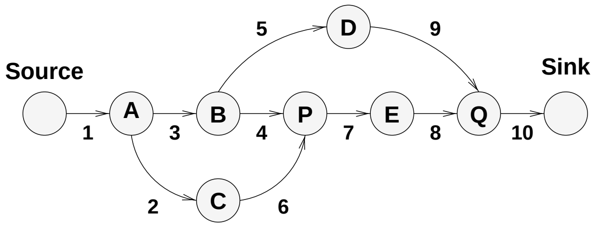


Figure 5.13. An application network.

[5] This programming model is based on a proposal of Leijten et al. [1997]

5.6.2 Mapping

To understand the programming model, we need to know how application networks such as the one given in Figure 5.13 map onto an architecture instance. If we look at application networks, we see that they contain three elements: edges, nodes, and streams. Somehow these three elements need to find a place on the architecture instance. Although we come back to this issue of mapping in Chapter 7, we will say already that we describe application networks in such a way that the three elements map directly onto architectural elements, as illustrated in Figure 5.14.

In this figure, we see that an edge (Edge) maps onto a combination of an output buffer (OutputBuffer), a router (Router), and an input buffer (InputBuffer). A node (Node) maps one-to-one onto a functional element (FE) of a functional unit (functionalunit). A node can only map to the FE when the FE implements the same function as the node. If a function of the application is not implemented by one of the FEs, the application cannot be mapped. Finally, the stream (Stream) consisting of individual samples (indicated as light gray circles) maps to a stream of packets, where a single packet (packet) has a header (header) which consists of four fields (indicated as dark gray circles) and a data part (data). The four header fields are: the base field H_b, the source field H_s, the function field H_f, and the length field H_l (see the picture of a header in Figure 2.3). When we refer to a header, we represent it as $< H_b, H_s, H_f, H_l >$.

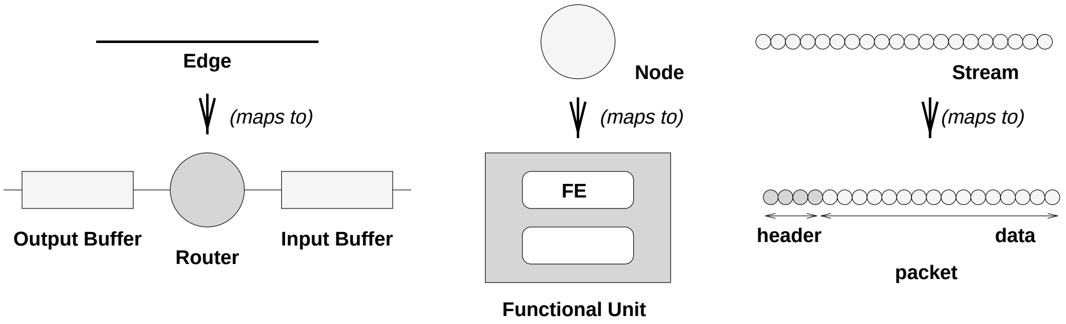


Figure 5.14. Mapping of an application onto an architecture.

5.6.3 Programming Model

In the programming model, we must number the edges in application networks in a particular way. Another part of the programming model relates to how routers and fucntional units change the content of headers at run-time, which we explain below.

Numbering the Edges in the Application Network

We number all edges in an application network in such a way that each edge has a unique number. Moreover, we number the edges that leave a node in such way that we obtain consecutively numbered output edges. To illustrate what we mean by “consecutively numbers output edges”, we show in Figure 5.15 part of an application network consisting of three nodes: P, Q and R. All the edges that leave these nodes have a unique number, and they are numbered consecutively from left to right. The edges leaving node P are numbered 2,3; The edges leaving node Q are numbered 4,5,6; and the edges leaving node R are numbered 9,10,11. In this numbering, the left-most edge carries the lowest number. This number is called the *base* of the node and plays an important role in the programming model. In Figure 5.15, the base of node P is 2, the base of node Q is 4, and the base of node R is 9.

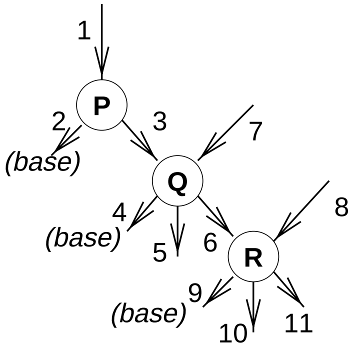


Figure 5.15. Edges leaving a node must have consecutive numbers.

The edges in the application network in Figure 5.13 are numbered in the correct way: all edges have a unique number. Furthermore, the edges leaving node A are numbered consecutively (i.e. 2,3), as are the edges leaving node B (i.e. 4,5). The base of node A is equal to 2 and the base of node B is equal to 3. If a node has only one output edge, the base equals the number of that edge. Thus, the base of node P is equal to 7.

Header Changes at a Router

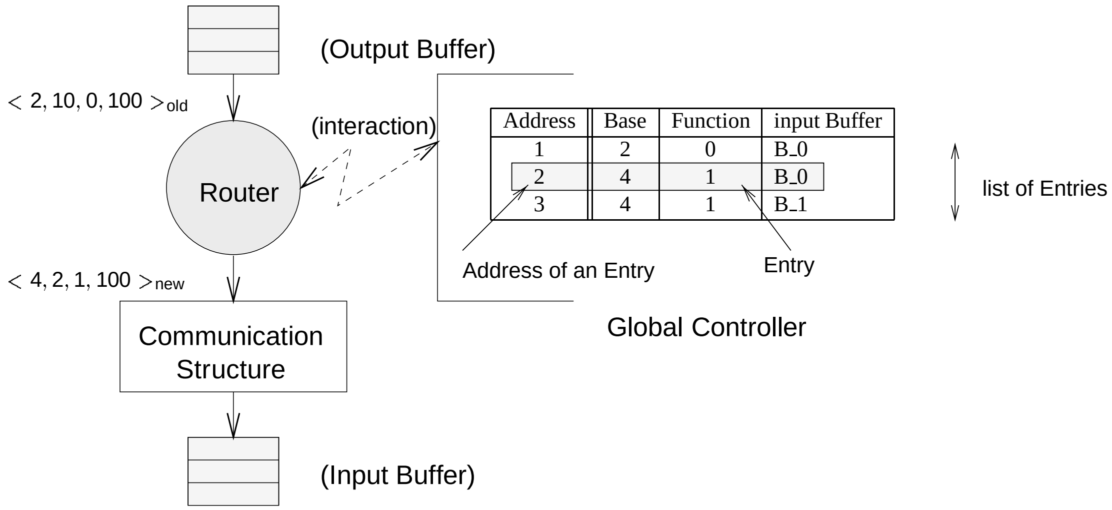
A router interacts with the global controller to obtain new content for the header of the packet that it is currently processing. When a packet arrives at a router, the header field of that packet changes as given in Equation 5.4.

$$< \mathsf{H_b}, \mathsf{H_s}, \mathsf{H_f}, \mathsf{H_l} >_{\mathsf{old}} \overset{\mathsf{Router}}{\rightarrow} < \mathsf{H'_b}, \mathsf{H_b}, \mathsf{H'_f}, \mathsf{H_l} >_{\mathsf{new}} \tag{5.4}$$

The length of the packet remains unchanged and thus field $\mathsf{H_l}$ remains unchanged. As part of the programming model, the base field $\mathsf{H_b}$ of the old header is copied into the source field ($\mathsf{H_s}$) of the new header. Therefore in Equation 5.4 we write the old value of $\mathsf{H_b}$ at the position of $\mathsf{H_s}$. The two fields in the new header that receive new content are the base field $\mathsf{H'_b}$ and the function field $\mathsf{H'_f}$. This new content is provided by the global controller.

When the global controller receives a header, it uses the base field $\mathsf{H_b}$ to determine the address of the entry containing the new header information (see Section 5.4.5). Each entry consists of a base field, a function field, and a reference to an input buffer. The content of the base field is assigned to $\mathsf{H'_b}$ and the content of the function field is assigned to $\mathsf{H'_f}$.

As part of the routing program, the global controller provides a reference to the correct input buffer. The correct programming model, however, is that routers interact with the communication structure to obtain a channel to a specific input buffer based on field $\mathsf{H_s}$ in a header. Nonetheless, we use a slightly different scheme: we avoid another decode step in the communication structure, by allowing the global controller to give a straightforward pointer to an input buffer. As a consequence, a router now only needs to claim a channel on the communication structure and it needs to claim the input buffer exclusively to model the existence of a path from the router to the input buffer. The scheme used correctly describes the programming model, but it very much simplifies the model of the communication structure, which leads to a faster simulation.



Address	Base	Function	input Buffer
1	2	0	B_0
2	4	1	B_0
3	4	1	B_1

Figure 5.16. A router with an input buffer and an output buffer. Since the router does not connect directly to the input buffers, but via the communication structure, the communication structure is also shown. To get the correct content for the new header, the router interacts (as indicated by the dashed line) with the global controller which contains a routing program.

The changes taking place in the header are shown again in Figure 5.16. The figure shows a router with an input buffer and an output buffer. Since the router does not connect to the input buffers directly, but instead via the communication structure, we show the communication structure explicitly in the figure. To obtain the correct content for the new header, the router interacts (as indicated by the dashed line) with the global controller. The global controller contains a routing program, which is the one already shown in Table 5.2 and discussed in Section 5.4.5.

When a packet arrives with the header $< 2, 10, 0, 100 >$, the base field ($H_b = 2$) is assigned to the source field H_s in the new header. The length of the packet ($H_l = 100$) is copied without modification into the new header. To get the new content for the base field and function field, the router interacts with the global controller. To decide which entry is relevant for the packet, the global controller looks at the base field H_b. This provides the correct address of the entry applicable for this header. In Figure 5.16, the base field equals 2, which addresses the entry containing a base field equal to 4 and a function field equal to 1. Therefore, in the new header the new base field becomes $H'_b = 4$ and the new function field $H'_f = 1$. Finally, the new header is $< 4, 2, 1, 100 >$.

Header Changes at a Functional Unit

When a packet is produced by a functional element having more that one output, the router must be able to distinguish between the packets, i.e., which packet is produced on which output. A router does not generally know to which output port of a functional element it connects. A functional element may share an output buffer with other functional elements. Furthermore, the output ports of a functional element can bind to any particular output port of a functional unit. Packets produced at different outputs still need to go to different locations in an architecture instance. A router considers a packet to be different from other packets when it has a different value for the base field H_b.

Functional units use a particular scheme to decode which output port produces a particular packet. In a functional unit, the header read from the opcode buffer is used to produce new headers on the output ports (see Section 5.4.8, where we explain the modeling of functional elements). The header read from the opcode buffer contains the base field $\mathsf{H_b}$. An *Offset* is added to this base field of which the value depends on the output port used (see Section 5.4.11 where we describe the write port). A functional unit thus changes the old header content into a new header content as given in Equation 5.5.

$$< \mathsf{H_b}, \mathsf{H_s}, \mathsf{H_f}, \mathsf{H_l} >_{\mathsf{old}} \overset{\mathsf{FU}}{\rightarrow} < \mathsf{H_b} + \mathsf{Offset}, \mathsf{H_s}, \mathsf{H_f}, \mathsf{H_l} >_{\mathsf{new}} \tag{5.5}$$

All the output ports of the functional unit have consecutively ordered offset values. Thus, if the functional unit has N output ports, then the left-most output port has an offset of 0 and the right-most output port has an offset equal to $N - 1$.

The concept of adding an offset is illustrated in Figure 5.17. This figure shows a FU with one input port and two output ports. The output ports are numbered consecutively from left to right: the left output port has an Offset of 0 and the right output port has an Offset of 1. Now when a header arrives at the input port (the OpcodeBuffer) with $\mathsf{H_b}$ equal to 4, the local controller of the FU produces two new headers at the two output ports, with one having a new base field of 4 and the other of 5.

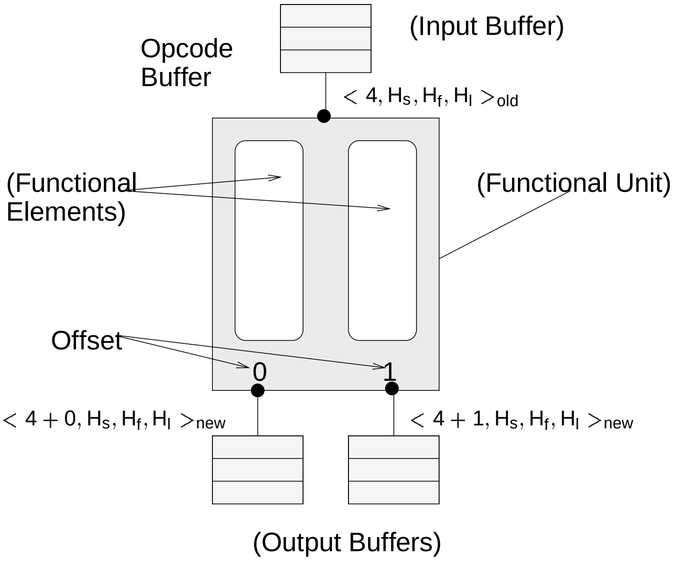


Figure 5.17. Base number of a functional unit. This figure shows a FU with one input and two output ports. The output ports are numbered consecutively from left to right: the left output port has a Offset of 0 and the right output port has an Offset of 1.

Program an Architecture Instance in 3 Steps

We explained what happens with a header at routers and at the output ports of functional units. We also explained how we number edges in an application network. We now explain how these three elements combine to program an application. Recall that the output edges of a node and the output ports of a functional unit are both numbered consecutively. We now show how the edge numbers relate to the base field ($\mathsf{H_b}$) of a header.

Suppose a functional unit reads a header from the opcode buffer. The header has a base field containing the base of a node in an application network. For example, the base field $\mathsf{H_b}$ carries the value 4, which is the base of node $\mathsf{f_B}$ in the application network given in Figure 5.13. The functional unit produces new headers on the output ports by adding an offset to the base field. The new base fields now represent the edge numbers of the edges leaving the node. Therefore, for the application network in Figure 5.13, the new headers contain base fields equal to 4 and 5, which are the two output edges of node $\mathsf{f_B}$.

The packet with a header containing $\mathsf{H_b} = 4$ traverses edge number 4 and the output packet containing $\mathsf{H_b} = 5$ traverses edge number 5. For one packet a router reads the entry at address 4, which contains routing information to route the packet to a functional element executing the function $\mathsf{f_P}$. For the other packet, a router reads the entry at address 5, which contains routing information to route the packet to a functional element executing the function $\mathsf{f_D}$.

Now assume that a functional unit contains a functional element which executes the functionality of node f in an application network. If the base field of the header on the opcode buffer of a functional unit contains the base value of node f, then the new headers produced on the output ports of the functional unit contain base fields representing the edge numbers of the edges leaving node f. This is due to the consecutively numbered edges leaving a node as well as the output ports having consecutive numbered offsets. At routers, the new base fields cause entries to be read from the routing program at addresses which equal the edge numbers leaving node f.

Address old $\mathsf{H_b}$	*Base* new $\mathsf{H_b}$	*Function* $\mathsf{H_f}$	*Input Buffer* Name
1	2		
2	6	1	$\mathsf{B_0}$
3	4	1	$\mathsf{B_0}$
⋮	⋮	⋮	⋮
10	x		

Table 5.3. The routing program stored in the global controller.

A routing program for an application network is set up in three steps:

1. We set up a table, as shown in Table 5.2. There is an entry in the table for each edge in a network. We create a table with the addresses 1 to 10, as shown in Table 5.3, for the application network in Figure 5.13, which contains edges numbered from 1 to 10.

2. For each address – which corresponds to an edge – we determine to which node it points in the application network. We fill in the base of this node at the base address. Thus, in the application network in Figure 5.13, address 1 points to node $\mathsf{f_A}$, which has a base equal to 2. Address 2 points to node $\mathsf{f_C}$, which has a base equal to 6. We fill in the base field for each address in Table 5.3. Address 10 in this table points to the $\mathsf{f_{Sink}}$ function. This function does not have a base and in this case we can fill in any arbitrary value. We therefore wrote down a “x” to indicate a “don’t care” situation.

3. We fill in specific information about the architecture instance, namely the input buffer field and the function field. Each address in Table 5.3 points to a node in the application network executing a particular function f. In the architecture instance, we identify a functional unit containing a functional element that executes the same function f. The position of this functional element in the repertoire of the functional unit determines the value of the function field. The selected functional element reads from a different input buffer for each input argument. The address

we consider corresponds with an edge in the application network as well as to one of the input buffers of the functional element. The name of this buffer determines the content of the buffer field. We repeat the procedure to find the function field and input buffer field until all entries in Table 5.3 are filled.

Note that in the last step, we locate a functional element that executes the correct function f. Hence, executing an application on an architecture requires that there be at least a functional element available for each function in the application network. It is always possible to construct an architecture that can always execute the application. To do this for each node in the application network, we construct a FU with one FE executing the node's function. The application network then maps *one-to-one* to the architecture. If an architecture instance is given and a function of the application network can be executed by more than one FE, then we can arbitrarily choose a FE. However, one choice can give better performance than another choice.

5.6.4 Example

To further illustrate how an application maps onto an architecture instance, we now go though an example in which we map a particular application onto a particular architecture instance. The architecture instance is given in Figure 5.12 (this is the architecture instance described in the architecture description in Figure 5.11) and the application network is given in Figure 5.18.

The application network is depicted in Figure 5.18. The application has four function nodes: $\mathsf{f_{Source}}$, $\mathsf{f_{Sink}}$, $\mathsf{f_0}$ and $\mathsf{f_1}$. The function $\mathsf{f_{Source}}$ generates a stream of samples (represented as the black dots) on edge 1. These samples are taken in by function $\mathsf{f_0}$, resulting in two streams: one on edge 2 and one on edge 3. The samples on these edges are taken in by function $\mathsf{f_1}$, resulting in one stream on edge 4. Finally, the samples on this edge are taken in by the function $\mathsf{f_{Sink}}$ where the stream ends.

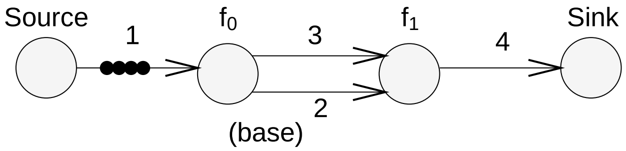


Figure 5.18. The Application.

The Mapping

We map the application function $\mathsf{f_0}$ to the architecture instance's functional element $\mathsf{FE_0}$. Similarly, we map the application function $\mathsf{f_1}$ to the architecture instance's functional element $\mathsf{FE_1}$. Hence, $\mathsf{FE_0}$ and $\mathsf{FE_1}$ execute the functions $\mathsf{f_0}$ and $\mathsf{f_1}$, respectively. The functions $\mathsf{f_{Source}}$ and $\mathsf{F_{Sink}}$ map to the source FE and sink FE. The edges in the application map onto combinations of an output buffer, a router and an input buffer, as shown in Table 5.4. Thus, edge 2 maps onto output buffer $\mathsf{B_4}$, router $\mathsf{R_1}$ and input buffer $\mathsf{B_0}$.

Routing Program

The global controller uses a routing program as given in Table 5.5. This table causes the architecture instance to execute the application network given in Figure 5.18.

Edge	Output Buffer → Router → Input Buffer
1	$B_3 \rightarrow R_0 \rightarrow B_0$
2	$B_4 \rightarrow R_1 \rightarrow B_0$
3	$B_5 \rightarrow R_2 \rightarrow B_1$
4	$B_5 \rightarrow R_2 \rightarrow B_2$

Table 5.4. Mapping of Edges to buffer/router/buffer combinations.

Address	*Base*	*Function*	*Input Buffer*
old H_b	new H_b	H_f	Name
1	2	0	B_0
2	4	1	B_0
3	4	1	B_1
4			
5	x	0	B_2

Table 5.5. The routing program stored in the global controller.

We derived this table using the 3-step approach given in the previous section. First of all, we need to set up an entry for each edge in the application network. The edges are numbered from 1 to 4 and we create the list of entries as given in Table 5.3 for these four edges. Secondly, we fill in the base fields. For example, address 1 represents edge 1 in the application network and points to the node executing function f_0. This node has a base value of 2 and we thus fill in a 2 at the base field on address 1. We repeat this procedure for all addresses in Table 5.5. Thirdly, we fill in the function field and buffer field for each entry in the table. We look again at address 1, which represents edge 1. The function f_0 maps to FE_0, the first entry in the function repertoire of the regular PE. Consequently, the function field of the entry at address 1 is assigned the value 0. Furthermore, edge 1 maps to the input buffer of FE_0, which is B_0. Thus, the buffer field of the entry at address 1 is assigned the name B_0.

Table 5.3 differs from Table 5.5 in the sense that the entry at address 4 is empty, while there is an entry at address 5 although edge 5 does not exist in the application network. Entry 4 is empty because the output port of FE_1 does not connect to output buffer B_4, but connects instead to output buffer B_5. The port connecting to buffer B_5 adds an offset of 1 to the base field. This interferes with the consecutive number of the output ports and, as a consequence, address 4 never appears. In this case, we start to renumber the addresses – but also the edge numbers in the application network – from address 4 onward by adding a 1 to each address. Address 4 thus becomes address 5.

The result of the interaction between the routers and the global controller is changes in the headers at both routers and at the output ports of functional units while they flow through the architecture. The changes taking place in the headers are given for the application in Table 5.6.

Resulting Behavior in Time

The routing program given in Table 5.5 causes the interleaving of two streams in time on the same FU, as shown in Figure 5.19. In the figure, we can recognize the four buffers of the FU: the two input buffers and two output buffers. The packets belonging to f_0 are light gray and the packets belonging to f_1 are dark gray. Field H_f in the header of the opcode buffer (B_0) determines which FE is activated. A 0 indicates that H_f equals 0 and a 1 indicates that H_f equals 1. The first packet in the opcode buffer

	At the Router	*At the Output Port of a FU*
Edge	Before $\xrightarrow{\text{Router}}$ After	Before $\xrightarrow{\text{FU}}$ After
1	$<1,0,0,x> \xrightarrow{R_0} <2,1,0,x>$	$<2,1,0,x> \xrightarrow{FU_0} <2,1,0,x>$ (B_4)
		$<2,1,0,x> \xrightarrow{FU_0} <3,1,0,x>$ (B_5)
2	$<2,1,0,x> \xrightarrow{R_1} <4,2,1,x>$	
3	$<3,1,0,x> \xrightarrow{R_2} <4,3,1,x>$	$<4,2,1,x> \xrightarrow{FU_1} <5,2,1,x>$ (B_5)
4	$<5,0,0,x> \xrightarrow{R_2} <x,5,0,x>$	

Table 5.6. Header changes in the architecture.

activates FE_0, the next packet activates FE_1, and so forth.

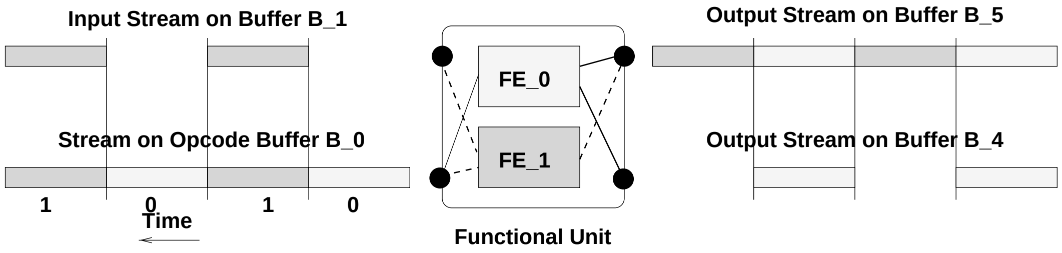

Input Stream on Buffer B_1
Stream on Opcode Buffer B_0
1
0
1
0
Time
FE_0
FE_1
Functional Unit
Output Stream on Buffer B_5
Output Stream on Buffer B_4

Figure 5.19. Interleaving streams on the functional unit.

When FE_0 is activated, it reads a packet from B_0 and produces packets on B_4 and B_5. The packet written into B_4 will be written into B_1 via router R_1. The packet written into B_5 will be written into B_1 via router R_2. Consequently, different packets will appear in B_0. Buffer B_0 thus contains packets belonging to both functions f_0 and f_1. The FU reads a new packet from the opcode buffer B_0, which activates FE_1. Because FE_1 requires two input arguments, the FU strips the header from the packet on B_1. FE_1 produces a packet on B_5 that follows the packet previously produced by FE_0. Router R_2 can, however, separate these two packet streams at run-time such that some packets go to B_0 and other packets go to B_2. Notice that when the source FU is involved with filling input buffer B_0 via router R_0, packets produced by FE_0 have to wait in B_4 until router R_1 notices that B_0 is available.

5.7 Conclusions

In the Y-chart approach we need to be able to derive executable architecture instances from the architecture template of stream-based dataflow architectures. To do so, we used the Performance Modeling Basis (PMB), which allows us to describe architecture instances at a higher abstract level than possible with standard hardware description languages, while being cycle-accurate.

We indicated that constructing an executable architecture instance from an architecture template yielding cycle-accurate performance numbers involves four issues: describing an architecture template, specifying an architecture instance, creating an executable model for the architecture instance, and programming an architecture instance.

We used the building block approach presented in Chapter 4 to construct architecture instances based on the PMB. We modeled all the architectural elements of stream-based dataflow architecture as a building block at the abstract, cycle-accurate level. Furthermore, we showed the usefulness of building block and object oriented techniques like polymorphism and inheritance. These techniques were key to describing in a flexible way architectural elements having more than one behavior.

We showed that we use composition rules expressed in Backus-Naur Form (BNF) to define an architecture template. We modeled the architecture template of stream-based dataflow architectures in Appendix A using BNF rules. We showed that BNF rules can also be used to define the grammar of an architecture description language. We used this language to describe a specific architecture instance of the architecture template. We also showed how composition rules and building blocks combine such that a parser can automatically construct an executable architecture instance from an architecture description.

To provide an architecture instance with a workload, we need to program the architecture instances. We explained the programming model of stream-based dataflow architectures. This involved our explaining how we model applications as networks and how we map these networks onto an architecture instance. We also discussed a comprehensive example showing how we program a particular architecture instance for a particular application.

The generality of the architecture modeling approach presented in this chapter will be discussed in Chapter 7, where we use this approach to construct a retargetable architecture simulator for stream-based dataflow architectures.

Bibliography

John Backus and Peter Naur. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *Proceedings of the International Conference on Information Processing (ICIP)*, Paris, 1959.

M. Ben-Ari. *Principles of Concurrent Programming*. Prentice-Hall, 1982.

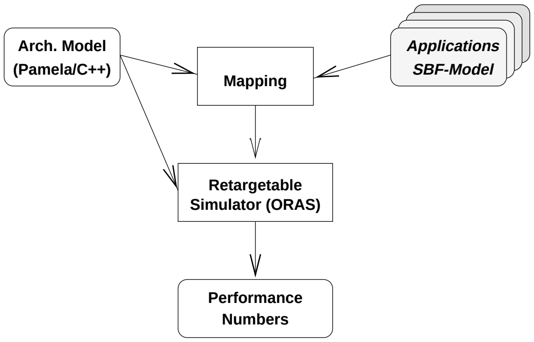
Charles Donnelly and Richard Stallman. *BISON, The YACC-compatible Parser Generator*. The Free Software Foundation, 1992. release 1.20.

Wolfgang Kreutzer. *System simulation, programming styles and languages*. International Computer Science Series. Addison-Wesley, 1986.

Jeroen A.J. Leijten, Jef L. van Meerbergen, Adwin H. Timmer, and Jochen A.G. Jess. Prophid, a data-driven multi-processor architecture for high-performance DSP. In *Proc. ED&TC*, 1997.

David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ programming with standard template library*. Addison-Wesley Professional Computing Series, 1996.

Vern Paxson. *Flex - fast lexical analyzer generator*, 1990. Online manual pages of Flex.



Chapter 6

Applications

Contents

THE digital signal processing (DSP) applications on which we focus in this thesis appear in domains like video, audio and graphics. Examples of such applications are compression and decompression applications, encryption, and all kinds of quality improvements. A natural way of describing these applications is a network of nodes that execute functions operating on streams of data. The nodes interconnect with each other via buffers over which data is communicated and abstracted by means of tokens. The operation semantics of these networks is described by their model of computation. In this chapter, we will introduce a new model of computation, which we call Stream-Based Functions (SBF). We will use this SBF model to describe the sets of applications in the context of the Y-chart environment we develop in this thesis. The main reason why we developed the SBF model was to make possible a smooth mapping of applications onto architecture instances. This mapping process is discussed in Chapter 7, along with a discussion of the development of a retargetable simulator.

In Section 6.1, we consider stream-based applications and give an example of one – the Picture in Picture application which is used in modern high-end TV-sets. When stream-based applications are described as a network, the functions in the network exchange tokens between each other. An important question is then what the token represents, and we investigate this problem in Section 6.2. In Sections 6.3–6.3, we present stream-based dataflow functions model of computation. Many models of computation already exist, like dataflow models, process models, and mixed dataflow/process models. In Section 6.7, we put the SBF model in perspective with respect to these models. We have constructed the simulator SBFsim, for simulating networks of SBF objects. In Section 6.8, we explain how we constructed this simulator using the programming language C++ and a multithreading package. We also show that we can reuse the building block approach (presented in Chapter 4) within the simulator SBFsim.

6.1 Stream-Based Applications

We model DSP applications by means of a network, as already discussed in Chapter 5. We focused there only on the structural information of the network. In this chapter, we mainly focus on the model of computation of such networks and the internals of each node in a network.

In a network describing a DSP application, the nodes describe functions that have a complexity ranging from fine-grained to coarse-grained. Furthermore, the nodes execute concurrently, exchanging streams of tokens with each other via buffers. The buffers temporarily store the streams of tokens without changing the ordering of the tokens. We refer to applications that can be described in this way as being *stream-based applications*, and we define them as:

Definition 6.1. STREAM-BASED APPLICATION

A *stream-based application* is an application that can be described as a network or directed graph $G(V, E)$, where V defines a set of nodes executing functions on streams of data and where E defines a set of buffers through which nodes exchange streams of data corresponding to a particular model of computation. □

A stream-based application describes a *static network*, which means that nodes are not created or destroyed during execution. Nodes within the network may produce and consume a variable amount of tokens while executing. As a consequence, the stream-based application may describe a *dynamic application*, i.e., an application containing data-dependent conditionals that can only be resolved at run-time.

Picture in Picture

An example of a stream-based video application is shown in Figure 6.1. It describes the *Picture in Picture* (PiP) algorithm, which reduces a picture to half its size in both horizontal and vertical directions and places the reduced picture onto a full screen picture showing two images on a TV screen [Janssen et al., 1997]. The Picture in Picture application is used in modern high-end televisions.

In the PiP example, a Source produces an infinite stream of video samples that are filtered by an N-tap Finite Impulse Response (FIR) filter. Then the stream is passed through a Sample-rate Converter (SRC) that performs a down-sampling of a factor two. Next, video images are transposed (Transpose), i.e., samples are re-ordered in such a way that two consecutive samples belong to two different video lines. The stream then passes again through an N-tap FIR filter and a SRC, this time to perform a vertical down-sampling of a factor two. The second transpose function performs a re-ordering on the stream that results such that consecutive samples now belong to the same video line. Finally, the sink consumes the samples.

In the example, the sample-rate converters have a fixed factor of two. In this case, each node knows exactly how many tokens to consume and produce while executing. The PiP algorithm thus

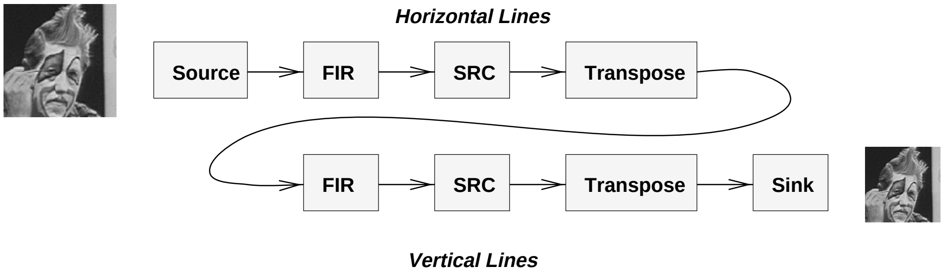


Figure 6.1. The Picture in Picture (PiP) algorithm, which reduces a picture to half its size in both the horizontal and the vertical directions.

describes a static algorithm. If the scale factor of the sample-rate converters can vary during execution, the nodes have to consume and produce a variable amount of tokens based on the value of the scale factor. In that case, the PiP application describes a dynamic application.

6.2 Imperative Programming Languages

People use imperative programming languages like C or Matlab to describe stream-based applications. If we want to describe the PiP application in Figure 6.1 using such a language, an important question is what the tokens being exchanged between the functions of the application represent, i.e. do they represent a complete *video image* or a single *video pixel*?

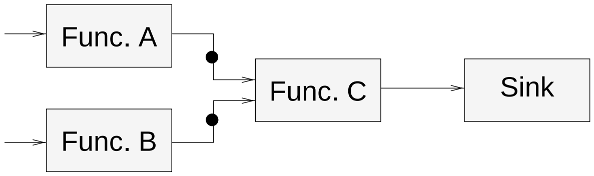


Figure 6.2. A simple video application consists of four functions (A, B, C, and Sink) operating on video streams. The black circles represent data.

Suppose we have the stream-based application shown in Figure 6.2. In this figure, two functions (A and B) have already produced data (represented by the black circles). When describing this application using an imperative language, the tokens will in all likelihood be chosen to represent a complete video image, because this makes it easier to describe the application. There are two reasons for this. The first one is that a simple *sequential ordering* of functions can be used to describe the application, i.e., in the video application given, we can first execute function A, then function B followed by function C and finally function Sink. The second reason is that the representation of a complete video image can be treated as a single *matrix*, with as consequence that functions can merely use indices to randomly access individual video pixels.

The filter function *Neighborhood* is given as a Matlab program in Program 6.1 as an illustration of a matrix representation of a complete video image. Matlab differs from C in that it uses only matrices and vectors as data types and it uses a different syntax. Other than that, the two languages are very similar. The function shown accepts the matrix `matrix` and produces the matrix `new` and can be used for both function nodes A and B in Figure 6.2. The filter determines a new value for each pixel (i.e.

`(i,j)`) using its four neighbors. The function accesses these neighborhood pixels using `for`-loops as iterators (e.g., `i`,`j`) and indices expressed in these iterators (e.g., `i+1`, `j+1`). To keep the function simple, we did not include the exceptions that occur at the fringes of the matrix. Describing the filter `neighborhood` in this way is *only* possible because a complete video image is available as the matrix. When the filter function terminates, the matrix `new` represents the video image for the next function that will execute in the sequence of function executions (e.g. function `C`).

Program 6.1. NEIGHBORHOOD FILTERING

```
matrix new = Function NeighborHood( matrix )
for i = 2 to N-1 step 1
  for j = 2 to N-1 step 1
      new(i,j) = Average( (i,j), (i-1,j),
                (i+1,j), (i,j-1), (i,j+1) );
  end
end
return matrix A_new
```

It is relatively simple to describe stream-based applications using imperative languages. The above function described the functionality correctly, and in that respect, there is nothing to find fault with in the way the application is described, including the fact that a token represents a complete video image.

However, the application needs to be implemented onto an architecture instance. In fact, the application in Program 6.1 *is* already mapped onto a particular architecture, namely the computer executing the application. The architecture of a standard computer architecture assumes that a large background memory is available which is accessed by only one statement at a time. For example, the matrices of Program 6.1 are mapped into this background memory and use indices in the matrices to address the background memory.

The computer architecture differs strongly from the stream-based dataflow architectures. Stream-based dataflow architectures do not assume a large background memory. Furthermore, video-images are only available as streams of video samples. Stream-based dataflow architectures would become impractical if they had to buffer complete images between the functions executing on the architectures. These architectures would consist mainly of memory and use only a few small computational units. Also, the buffering of complete images would prevent much of the parallelism present in applications from being utilized. For example, in the case of the `NeighborHood` function, all pixels not within close range of pixels `(i,j)`, could in principle operate in parallel with pixels `(i,j)`. However, in an imperative language only one statement executes at a time, so this will not happen.

To fully utilize the potential of stream-based dataflow architectures, we thus have to rewrite applications such that they use streams and express the amount of parallelism present in the application when operating on these streams. To achieve this goal, we can no longer assume that a video image is available as a matrix ot that functions can use indices to access neighborhood pixels, as was done in Program 6.1. Suppose the `NeighborHood` function in Program 6.1 were to read video samples from a stream in which the video samples are ordered according to the lexicographical ordering of the iterators `i` and `j`. Then some neighborhood samples would have been read from this stream too early (e.g. via indices `i-1`, `j-1`), whereas other samples would still need to be read (e.g. via indices `j+1`, `i+1`).

If the type of data used in a stream-based application does not match the type of data used in a stream-based dataflow architecture, then a smooth mapping of the application onto an architecture becomes difficult. We already indicated this problem in Section 3.3.2 when we discussed mapping.

We cannot assume that an application which uses matrices would map easily onto an architecture that uses neither background memory nor random accesses on this memory, but instead only involves streams.

In the case of stream-based dataflow architectures, we therefore need to rewrite stream-based applications such that they are consistent with the assumption that we *only* have streams of video pixels instead of other data types. This dramatically affects how stream-based applications can be described, and introduces the following three problems:

Parallelism: Functions will become active concurrently. A simple sequential ordering of the original functions is no longer satisfactory.

Decomposition: Functions will operate on streams of samples. The original functions have to be decomposed into functions operating on streams of samples only.

Consumption Patterns: Functions will operate on a one-dimensional data structure instead of multi-dimensional data structures like matrices. This requires a transformation from multi-dimensional data structures to one-dimensional data structures, thus causing functions to consume and to produce tokens in different patterns.

If we rewrite the application in Figure 6.2 using streams of samples instead of images, the four functions A, B, C and Sink no longer follow a sequential ordering; they operate concurrently on different streams. Function C consumes tokens concurrently with the functions A and B, which produce tokens. In Figure 6.3 we show again the application as given in Figure 6.2, but now with the two tokens (i.e. the black circles) in the figure representing a single video pixel instead of a complete image.

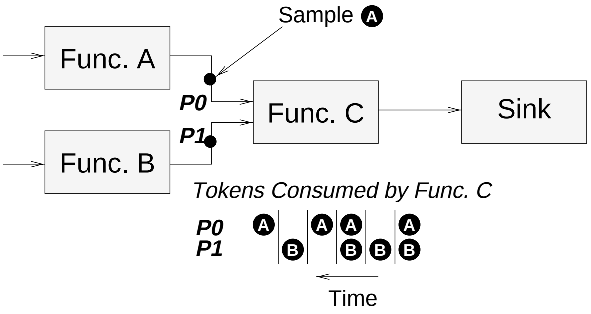


Figure 6.3. A simple video application (revised). We rewrote the application using streams instead of matrices. As a consequence, all four functions become active concurrently. Functions A, B, C and Sink no longer follow a sequential ordering but operate concurrently on different streams. Function C, for example, consumes tokens produced by the functions A and B in a particular pattern.

By removing the `for-next` loops in Program 6.1, we obtain the function `Average`, which operates only on single samples. Function C in Figure 6.3 reads samples from the functions A and B in a particular pattern as shown below the box labeled Func.C. The two input ports of function C, ports P0 and P1, consume the tokens produced by function A or function B in the way given in the table.

Given a stream-based application using streams instead of images, we want to rewrite it using only streams of samples. The question we need to answer is which model of computation best describes the applications, taking into account issues like parallelism, decomposition of functions, and different consumption/production patterns.

6.3 Stream-Based Functions

We propose a new model of computation called *Stream-Based Functions* (SBF) with which stream-based applications can be described. The essential concepts in this model are *Stream-Based Function Objects* and *Channels*. Stream-based applications are described as a *network* of SBF objects communicating concurrently with each other using channels. These channels interconnect SBF objects and buffer the tokens communicated by SBF objects. The buffers are unbounded FIFO queues that can contain an infinite sequence of tokens, i.e. a stream.

A network of SBF objects describes a special kind of network: *Kahn Process Networks* [Kahn, 1974]. In Kahn Process Networks, a *process* maps one or more input streams into one or more output streams by executing a sequence of statements. Some statements perform a read on a channel and some statements perform a write on a channel. When reading from a channel containing no tokens, the process stalls completely until tokens become available again, implementing a *blocking-read*. A write on a channel can always proceed, implementing a *non-blocking write*.

An example of a process network is given in Figure 6.4. It shows a number of processes connected to each other via channels over which the processes exchange streams of tokens. We assume that these tokens represent a scalar variable. The Source process produces a stream that is taken in by the filter_A process. This process produces a stream that is further processed by the filter_B process. This second filter produces two streams: one back to the filter_A process and the other to the Sink process.

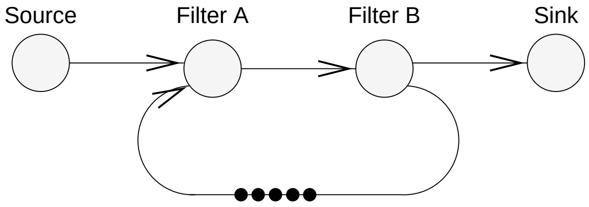


Figure 6.4. An example of a Kahn Process Network. A number of processes connected to each other via channels over which the processes exchange streams of tokens represented as black circles.

The sequence of statements inside a Kahn process consists of a mix of control statements, read and write statements, and function-call statements. A Kahn process does not structure these statements in any particular way. The SBF objects, on the other hand, structure these statements according to the 'Applicative State Transition' (AST) Model described by Backus [1978]. An SBF object does not implement the full AST model, but a specialization that is inspired by the AST model proposed by Annevelink [1988]. As a consequence, an SBF object contains three components: a *set of functions*, a *controller*, and *state*. An enabled function consumes a number of tokens from input channels, performs its function, and writes tokens to output channels. By repeatedly enabling functions, an SBF object operates on streams. The controller enables the functions of the set in a particular sequence and uses data stored in the state of the SBF object to determine which function it must enable next.

Kahn process networks have the property that they execute in a *deterministic order* [Kahn, 1974; Kahn and MacQueen, 1977], which means that the computation of the result of the network is independent of the schedule of the processes in the Kahn process network. Therefore, for a stream with the same input data presented to the network, the computed output stream is always the same, irrespective of the schedule used to compute this result: it could have been a completely sequential schedule or a complete parallel schedule or any schedule in between. A Kahn process network executes deterministically by using a blocking read to prohibit a process from testing the availability of data. Hence, a

Kahn process is either waiting for data or performing computations. An SBF object only structures a Kahn process and does not impose new restrictions affecting the behavior of the process. Thus, networks of SBF objects and Kahn process networks describe the same applications.

6.4 The SBF Object

An SBF object has an inside view and an outside view. Inside an SBF object the following three components are present: a *set of functions,* a *controller,* and a *state.* Outside an SBF object ports are present: read and write *ports*. These ports connect to channels, allowing SBF objects to communicate streams with each other.

An SBF object is shown in Figure 6.5. The object contains a set of functions $\{f_a, f_b\}$, a controller, and state. The SBF object also has two read ports and one write port that connect to the unbounded FIFO buffers *Buffer0*, *Buffer1*, and *Buffer2*, respectively. These buffers implement the channels between different SBF objects.

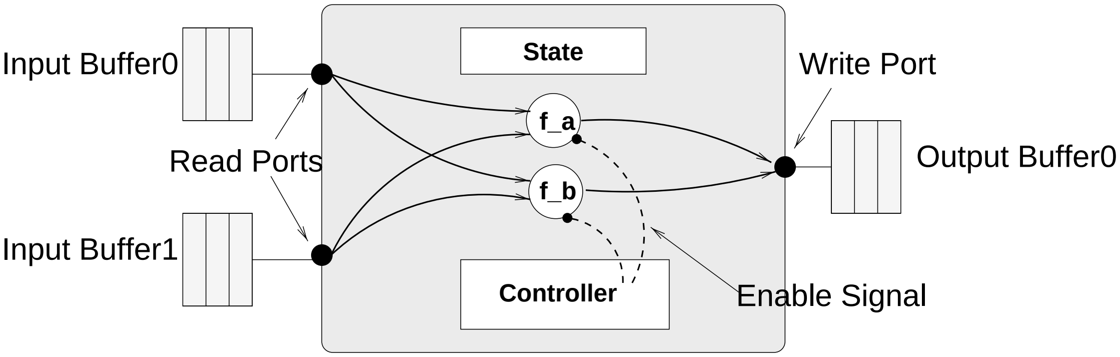


Figure 6.5. An SBF object.

The set of functions P determines the functionality of an SBF object. It must not be empty and may contain the following functions:

$$P = \{f_{init}, f_a, f_b, \ldots, f_x\}. \tag{6.1}$$

These functions evaluate within an SBF object in a *sequential order* such as,

$$f_{init}, f_a, f_b, f_a, f_b, f_a, \ldots. \tag{6.2}$$

The *controller* governs the order of function evaluations. It keeps track of the evaluation order using the variable c, called the *current state*. Each time the current state c changes, a *transition* takes place. The current state is stored in the control space C. Other variables are stored in the data space D. The control space and data space together define the state space S of an SBF object

$$S = C \times D, \text{ such that } C \cap D = \emptyset. \tag{6.3}$$

As explained in Chapter 4, an *object* consists of a data part and a methods part operating on the data part [Goldberg and Robson, 1983]. SBF objects have a similar structure in the sense that the state corresponds with the data of an object and the set of functions corresponds with methods of an object. The control of an SBF object can be seen as a special method of the object.

6.4.1 Functions

For each function f from the set P, there is a function call

$$f(x_0, \ldots, x_m, D) = (y_0, \ldots, y_n, D'), \tag{6.4}$$

such that the function f modifies the current data space D into the new data space D' and the input data $x_0, \ldots, x_m$ into the output data $y_0, \ldots, y_n$. A function call can either have no input data or no output data, in which case it describes a *source* or a *sink* function respectively. A source function only produces tokens and a sink function only consumes tokens. A function reads its input data from read ports and writes its output data to write ports. It is statically determined on which read port or write port a function operates.

When the controller *enables* a function, it reads input data from the input ports, performs the function and writes output data to output ports. The function reads all its input data using blocking reads. State variables are immediately available. A function can read only one token from an input argument. Once a function has obtained all input data, it will not obtain any new inputs until it writes the output data to the write ports and installs new state variables. During the evaluation of a function, the state S cannot change and does not change. Consequently, a function operates without any side effects. When the enabled function has written all its output data, we say the function has *fired*.

A function reads data from a read port using a blocking read. Therefore, if data is available, the function reads the data; otherwise it blocks. This causes a function to read its input data $x_0, \ldots, x_m$ in a sequential order from the input ports. It first reads token x_0, followed by token x_1 and so forth until token x_m has been read. The blocking read prohibits the testing of a read port on the availability of tokens. As a consequence, a function cannot have another behavior based on the results of testing a condition; thus it evaluates unconditionally: it is either waiting on a port or it is performing computations. A function writes the output data $y_0, \ldots, y_n$ to the appropriate write port using a non-blocking write, also in a sequential order.

6.4.2 Controller

The controller governs the enabling sequence of functions. It keeps track of this sequence in the current state variable c. The controller moves from the current state c to another state c' whenever a transition occurs. All obtainable states are described by the control space C. The control space is traversed as determined by the *transition function* ω, which associates the next state c' with each current state c.

$$\omega : C \times D \rightarrow C, \;\; \omega(c, d) = c' \tag{6.5}$$

To determine the next state c', the transition function observes the control space C and the data space D. The controller cannot change the data space; it can only observe it. Although the controller is not connected to any read or write port, the transition function describes dynamic behavior. When deciding what the next state will be, it observes the data space D affected only by the function of P.

The transition function describes a path through the control space by going from the current state to a new state. If both the data space and control space are observed to determine the next state, the transition function describes infinitely many paths. At each new state, which path is started depends on the content of D.

In a more restricted case, when the transition function observes only the control space C to determine the next state, it always describes a single path to traverse the control space. Furthermore, if this path is finite, the transition function determines a new state c which has previously been the current state, thus describing a *cycle*. This cycle brings the state of the controller back into a previous state in a fixed number of transitions.

At each state, a specific function needs to be evaluated as determined by the *binding function* μ. This binding function associates a function f from the set P with a particular control state c. Only one function can be associated with a state.

$$\mu : C \rightarrow P, \;\; \mu(c) = f \tag{6.6}$$

Using the transition function ω and the binding function μ, the controller describes a sequence in which the functions of set P are enabled. A function becomes enabled and fires. This causes a transition to take place, and using ω and μ a new function is determined that is enabled the next time, leading up to the sequence described as follows:

$$f_{init} \overset{\mu(\omega(S))}{\longrightarrow} f_a \overset{\mu(\omega(S))}{\longrightarrow} f_b \overset{\mu(\omega(S))}{\longrightarrow} \ldots f_x \overset{\mu(\omega(S))}{\longrightarrow} \ldots \tag{6.7}$$

Thus an SBF object operates on streams by repeatedly enabling a function from the set of functions P, which reads from a buffer and writes tokens to a buffer. We remark that only the functions use a blocking read. A transition, or alternatively the evaluation of the ω and μ functions, takes place instantaneously. This behavior described by an SBF object is referred to as a *Fire-and-Exit behavior*. When we explain how we implement an SBF object, we explain why SBF objects describe such behavior.

The sequence given in Equation 6.7 results in a deterministic behavior of the SBF object. The functions of set P all evaluate unconditionally because of the blocking read. They also observe only one read port at a time. For the same input sequence of tokens, functions evaluate in the same output sequence. At the same time, the functions alter the data space D in a deterministic way, after all the functions have executed unconditionally. The transition function, which observes both the control space and the data space to resolve a new current state, also changes deterministically. Consequently, the complete SBF object describes a deterministic sequence.

When an SBF object is created, the controller needs to start at a particular state c. Within an SBF object, a special *initialization* function f_{init} is available that initializes the state C. This way the variable c is set to a particular value. Within an SBF object, the first evaluated function is the initialization function. It evaluates only once.

6.5 Example of an SBF Object

An example of an SBF object is given in Figure 6.6. The SBF object contains a set of functions $\{\mathsf{f_a}, \mathsf{f_b}, \mathsf{f_c}\}$, a controller, and a state containing the variables c (the current state, being an element of C) and x (a variable, being an element of D).

Function $\mathsf{f_a}$ reads input data from the two read ports and writes output data to the write port. Function $\mathsf{f_b}$ reads the input data from the read port connected to `Buffer0` and writes the output data to the write port. Function f_c does not have input data; it only writes output data to the write port. All functions can change the data space D. However, only function $\mathsf{f_a}$ changes variable x in D and only function $\mathsf{f_b}$ reads variable x in D.

If function $\mathsf{f_a}$ is enabled, it reads first one token from Buffer0 and then one token from Buffer1. If Buffer0 does not contain any tokens, the complete SBF object blocks until a token becomes available in Buffer0 even though data might already be available on Buffer1. When both tokens are available,

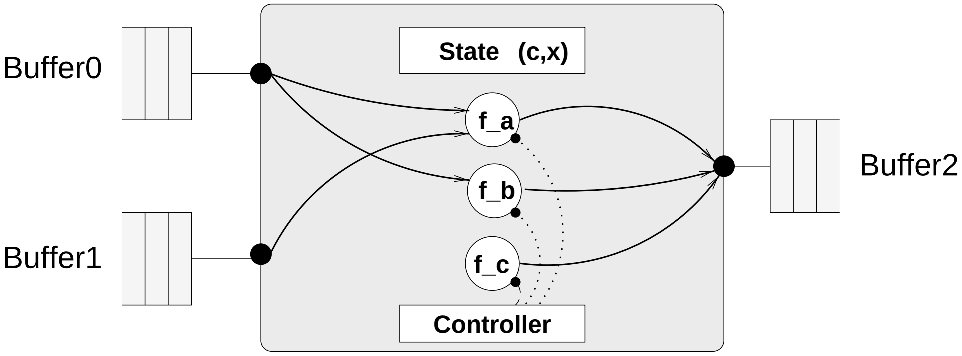


Figure 6.6. An SBF object containing a set of functions $\{\mathsf{f_a}, \mathsf{f_b}, \mathsf{f_c}\}$, a controller, and a state containing the variables c and x.

the function $\mathsf{f_a}$ evaluates. The resulting token is written to Buffer2. The function also installs the state variable x in D. When $\mathsf{f_a}$ has fired again, a transition takes place and the next function enabled is again $\mathsf{f_a}$. When $\mathsf{f_a}$ has fired, a transition takes place and the next function enabled is function $\mathsf{f_b}$. It reads a token from Buffer0 first and then it reads the state variable x and evaluates. The result is written to Buffer2. Thus $\mathsf{f_b}$ has fired and a transition takes place and the next function enabled is function $\mathsf{f_c}$. It immediately evaluates because it requires no input. Function $\mathsf{f_c}$ writes the resulting token to Buffer2. Function $\mathsf{f_c}$ has fired and a transition takes place and the next function enabled is function $\mathsf{f_a}$. This function was already enabled and the SBF object now describes a path in the sequence of enabling functions. The sequence of enabling can thus repeat indefinitely.

In the example, the controller enables the functions in the sequence f_a, f_a, f_b, f_c. This sequence is obtained by defining the binding function as

$$\mu(c) = \begin{cases} f_a, & \text{if } c = 0 \\ f_a, & \text{if } c = 1 \\ f_b, & \text{if } c = 2 \\ f_c, & \text{if } c = 3, \end{cases} \tag{6.8}$$

and the transition function as

$$\omega(c) = c + 1 \pmod 3. \tag{6.9}$$

The transition function determines a new state based only on c, describing a single path through the state space $C = \{c_0, c_1, c_2, c_3\}$. After four transitions, the transition function always returns to its start state c_0, thus it is a cyclo-static scheduler [Bilsen et al., 1995].

The sequence of functions results in a particular consumption/production pattern of tokens. These patterns are shown in Table 6.1 for the SBF object in our example. It shows the four states, the functions executed at these states, and the three buffers from which the functions read tokens (R) or to which they write tokens (W). For function $\mathsf{f_a}$, a token is read from both Buffer0 and Buffer1 and a token is written to Buffer2. This is then repeated once and followed by the execution of function $\mathsf{f_b}$ consuming one token from Buffer0 and producing one token on Buffer2. Next the function $\mathsf{f_c}$ is executed; it consumes nothing, but produces a token on Buffer2. The whole consumption pattern shown in Table 6.1 can be repeated an infinite number of times. We remark that a function may read only one token from a buffer at a time. A function is not allowed to read more than one token at a time or to write more than one token at a time. In order for a function to be *multi-rate*, it must be divided into a sequence of functions.

Current State	Function	Buffer0	Buffer1	Buffer2
c_0	f_a	R	R	W
c_1	f_a	R	R	W
c_2	f_b	R		W
c_3	f_c			W

Table 6.1. Token consumption/production pattern of the SBF object shown in Figure 6.6. It shows the four states of the object and the functions executed at these states. It also shows the three buffers from which the functions read tokens (R) or to which they write tokens (W).

The SBF object shown in Figure 6.6 could implement the filter_A process given in Figure 6.4. In that case, Buffer0 implements the channel between the processes Sink and filter_A. Buffer1 implements the feedback channel between the processes filter_B and filter_A. Finally, Buffer2 implements the channel between the processes filter_A and filter_B.

6.6 Networks of SBF Objects

We describe stream-based applications as networks of SBF objects in which the SBF objects represent the coarse-grained functions. We already defined the granularity of functions in Chapter 2. We define the granularity of an SBF object as

Definition 6.2. GRANULARITY OF AN SBF OBJECT

The granularity of an SBF object is the total number of RISC-like operations needed to express each function of the set of the SBF object. □

As indicated in Chapter 2, the granularity of the functions that a functional element implements is an important design parameter. And as we have seen in Section 5.6 where we discussed how we program a stream-based dataflow architecture, a node in an application network maps directly to a functional element. In the networks discussed in this chapter, these nodes are SBF objects. Given a network of SBF objects, is it possible to change the granularity of the various SBF objects?

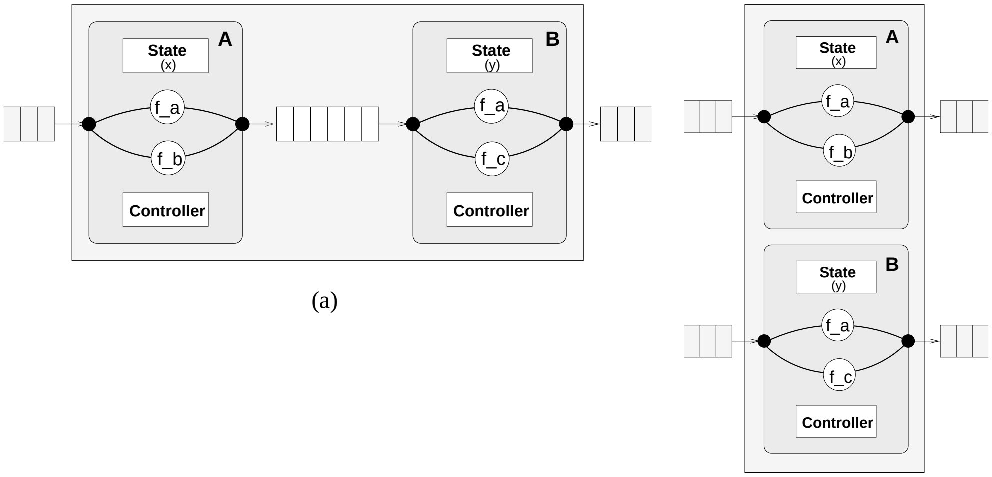
We can change the granularity of nodes in a network by combining SBF objects to construct a new SBF object (i.e. a composition) that will have a larger granularity, or we can construct SBF objects (i.e. a decomposition) that will have a smaller granularity by partitioning an SBF object. Although we could also describe a hierarchy of SBF objects, i.e., a function of the set of functions is again an SBF object, we will not further discuss this.

Before we explain how a composition and a decomposition of a SBF object works out, we first introduce the term “variant” of a function; A function in a SBF object is bound statically to input and output ports or a state variable. A *variant* of function f performs the same function f but is bound to other input and output ports or state variables.

6.6.1 Composition of SBF Objects

We construct a new SBF object by combining two or more SBF objects, using the following steps:

1. Combine the two sets of functions P and Q of both SBF objects to obtain the new set $Z = P \cup Q$. The set Z will contain variants of functions if necessary.
2. Combine the states S and S' of both SBF objects to get the new state $W = S + S'$.
3. If one or more channels (i.e., the unbounded FIFO buffers) are enclosed by the new SBF object, map the channels into the state of the new SBF object.
4. Construct a new schedule that interleaves the functions of set Z to determine a new transition function ω and a new binding function μ for the new SBF object.



(a)

(b)

Figure 6.7. Combining two SBF Objects to form a new, more coarse-grained SBF object.

As an example, in Figure 6.7 we show two cases ((a) and (b)) in which two SBF objects, respectively A and B, are combined to form a SBF object. SBF object A contains the set of functions $P = \{f_a, f_b\}$ and the state S containing the variable x. SBF object B contains the set of functions $Q = \{f_a, f_c\}$ and state S' containing the variable y. Both SBF objects have only one read port and one write port. The functions in set P read from the read port and write to the write port of SBF object A. The same applies to the functions in set Q for SBF object B.

In the case shown in Figure 6.7(a), we combine the two SBF objects such that a channel connecting both SBF objects is enclosed by the new SBF object. The new SBF object is shown in Figure 6.8(a). According to the three steps presented previously, we first combine the functions of set P and Q to form the new set $Z = \{f_a, f'_a, f_b, f_c\}$. The function f'_a is a variant of the function f_a. Secondly, we combine the state of both SBF objects to obtain the new state W containing the variables x and y. Thirdly, because the new SBF object encloses a channel present between the two SBF objects A and B, we map this channel into the state W. A consequence of mapping the channel into the state W is that functions f_a and f_b of SBF objects A and B respectively, write and read from the state instead of writing and reading from ports. The same is true for the functions f_c and f_a of SBF objects A and B respectively. Because function f_a of SBF object A writes to the state and function f_a of SBF object

B reads from the state, we need a variant of f_a in the set of function Z. Fourthly, and finally, we schedule the functions in set Z of the new SBF object.

In the case shown in Figure 6.7(b), we combine the two SBF objects in one new SBF object. The new SBF object is shown in Figure 6.8(b). According to the three steps presented previously, we first combine the functions of set P and Q to form the new set $Z = \{f_a, f'_a, f_b, f_c\}$. Again a variant of f_a is required. The function f_a reads from and writes to the top two ports. The variant f'_a reads from and writes to the bottom two ports. Secondly, we combine the state of both SBF objects to obtain the new state W containing the variables x and y. We can omit the third step because no channel is enclosed by the two SBF objects. Fourthly, and finally, we schedule the functions in set Z of the new SBF object. By combining SBF objects as we have shown, we obtain a new SBF object which is coarser in its grain size.

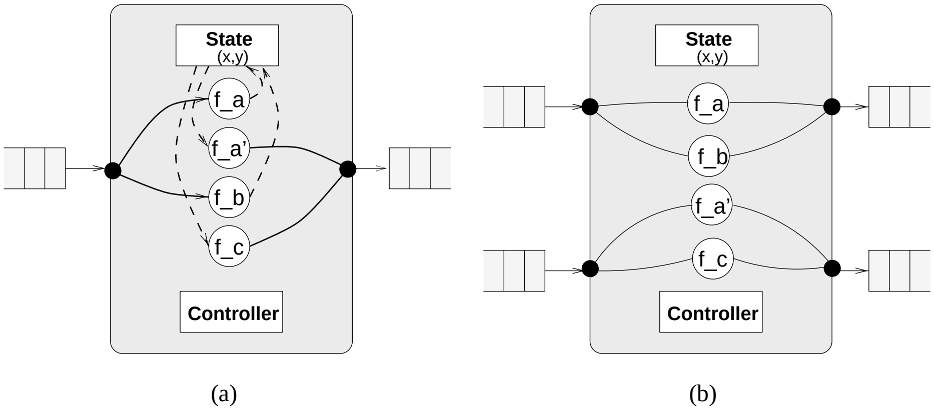


Figure 6.8. The new combined SBF objects.

Where the SBF objects A and B operate in parallel, the new SBF object executes the functions sequentially. In the fourth step, we interleave the functions such that they execute one after the other. Thus, constructing a coarser SBF object leads to less parallelism.

In the case in which we combine the SBF objects enclosing a channel as depicted in Figure 6.7(a), we mapped the channel into the state of the new SBF object. A channel is, however, an unbounded FIFO buffer. If we find a static schedule between the functions of set Z in the SBF object given in Figure 6.8(a), then we can definitely map the unbounded FIFO buffer into a bounded buffer in the state of the new SBF object [Bhattacharyya and Lee, 1994].

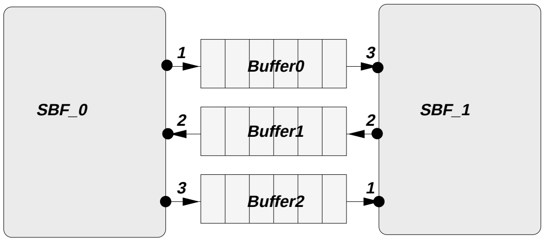


Figure 6.9. When scheduling the functions inside a SBF object, we should ensure that we do not introduce deadlock.

When scheduling the functions inside a SBF object, we should make sure that we do not introduce deadlock. In Figure 6.9 we show two SBF objects, SBF_0 and SBF_1, connected to each other via three buffers, Buffer0, Buffer1, and Buffer2. SBF_0 is scheduled in such a way that it first produces a token on Buffer0 and then reads a token from Buffer1 and, finally, produces a token on Buffer2. This sequence is indicated by the numbers 1 to 3 from top to bottom. The other SBF object, SBF_1, tries to read a token from Buffer2 first, then produces a token on Buffer1 and, finally, reads a token from Buffer0. This sequence is indicated by the numbers 1 to 3 from bottom to top.

The situation presented in Figure 6.9 will deadlock. SBF_0 is able to produce a token on Buffer0, but blocks when trying to read a token from Buffer1. On the other side, SBF_1 tries to read a token from Buffer2, but blocks because no token is ever produced on this buffer because SBF_0 blocks while reading from Buffer2. Thus, if two or more SBF objects are related, care should be taken to prevent deadlock from occurring.

6.6.2 Decomposition of SBF Objects

Decomposing an SBF object into new SBF objects that have a smaller granularity is more difficult than composing one new SBF object from smaller SBF objects. When decomposing an SBF object, we need to determine the available parallelism in the object. This implies that we have to solve the transformation from a sequential schedule into a parallel schedule, which is known to be a difficult problem [Banerjee, 1988].

We continue to decompose SBF objects until the set of functions consists of a single function: a RISC-like function. These SBF objects thus have a granularity of one, by definition. SBF objects in this case have a state space equal to $C = \{c_0\}$ and the transition function is equal to $c = 1$ and the binding function binds the RISC function to this state. The state of this SBF object is empty. When, however, a data-dependent function, for example a *compare*, is used, it is not clear what the transition and binding functions or the state look like.

6.7 Related Work

Many models of computation have been proposed over the years to describe stream-based applications. The models that are relevant to the proposed SBF model are the dataflow, process, and mixed models.

6.7.1 Dataflow Models

The dataflow model of computation describes stream-based applications is a natural way, retaining the parallelism present in an application. It describes applications as a network of dataflow *actors* that performs a particular computation. Actors connect with each other via buffers, allowing them to communicate tokens with each other. The condition under which an actor is able to perform a computation is determined by a *firing rule*. When an actor has evaluated, e.g., has consumed tokens and produced new tokens, it has *fired*. A dataflow model requires a *schedule* to compute. A scheduler determines at compile time or at run-time the ordering of actors, using a scheduling technique like *data-driven* evaluation, *demand-driven* evaluation, or *lazy* evaluation [Jagannathan, 1995]. Some well-known dataflow models are homogeneous dataflow (HDF), synchronous dataflow (SDF), cyclo-static dataflow (CSDF) and dynamic dataflow (DDF).

Homogeneous dataflow [Veen, 1986] is the simplest dataflow model. In this model, buffers hold only a single token and each actor has only one firing rule. To fire, incoming buffers connected to an actor must contain a token and all outcoming buffers must be empty. A feasible schedule always exists and can be derived at compile time. This model cannot describe multi-rate behavior since a buffer can contain at most one token. HDF can describe static applications.

The synchronous dataflow model [Lee and Messerschmitt, 1987] is capable of describing multi-rate effects. In this model buffers are bounded FIFO buffers. Each actor still has one firing rule and therefore the actors consume and produce a fixed pattern of tokens. The tokens consumed and produced by all actors in a network can be expressed using *balance equations*. If a solution to this system of balance equations exists, a feasible schedule exists and can be found at compile time. The sizes of the buffers can also be determined at compile time. SDF can describe static multi-rate applications.

The cyclo-static dataflow model [Bilsen et al., 1995] allows each actor to have one or more firing rules called upon in a known static order. This results in different consumption/production patterns of tokens, repeated infinitely many times. If a feasible schedule exists, it is found at compile time by solving a system of balance equations. CSDF can describe cyclo-static multi-rate applications.

The dynamic dataflow model [Jagannathan, 1995] allows each actor to have a set of firing rules that are not ordered in any particular way. A specific function is bound to each rule of the set. As soon as one of the rules in the set is satisfied, a valid firing is found and the function associated with that rule is executed. This model can describe data-dependent consumption/production patterns and is capable of describing dynamic applications. Since the rules are not ordered, non-deterministic effects occur. A typical example is the two-input merge function, which has two non-ordered firing-rules; the rule chosen first is unknown. The DDF model requires a run-time scheduler to evaluate. Because a schedule cannot be found at compile-time owing to the Turing halting problem [Buck, 1993], the boundedness of the buffers cannot be calculated at compile time. DDF can describe dynamic applications in a non-deterministic way.

A more restrictive DDF model is used in the Ptolemy system [Buck et al., 1992]. This model also allows each actor to have a set of firing rules, but they are ordered instead; one rule must follow the other. This model describes dynamic applications in a deterministic way. It requires a run-time scheduler to evaluate. This restrictive DDF model can describe dynamic applications in a deterministic way.

6.7.2 Process Models

Another way to describe stream-based applications is via process models. Process models describe an application as a network of processes connected with each other via buffers. The buffers allow processes to communicate tokens with each other. Each process proceeds autonomously, i.e., it is not controlled by a global scheduler. A process interacts according to a particular *protocol* with other processes in a network. Two well-known process models with different protocols are *Kahn Process Networks* (see Section 6.3) and *Communicating Sequential Processes*.

In Kahn process networks [Kahn, 1974; Kahn and MacQueen, 1977], buffers represent unbounded FIFO queues. These queues automatically buffer output of a process and allow processes, which run forever, to consume tokens from buffers at different rates. A process stores tokens on a buffer by using a non-blocking write and reads tokens from a buffer with a blocking read. The blocking read prohibits testing of ports on the availability of data; a process can only observe one port at a time. Hence a process is either waiting on a port or performing computations; therefore, Kahn process networks describe applications in a deterministic way.

In communicating sequential processes [Hoare, 1978, 1985], a buffer is a single place buffer. If one process puts a token into the buffer, it blocks until the process at the other end of the buffer removes the token from the buffer. It is possible to use unbounded FIFO buffers, but CSP must model them as a process. Within a CSP process which is intended to terminate, it is possible to test on the availability of data and to select and perform a specific function based on the test results. Hence, the CSP model can describe applications in a non-deterministic way.

6.7.3 Combined Dataflow/Process Models

A combination of dataflow models and process models can also be used to describe stream-based applications. Examples of combined dataflow/process models are the *Dataflow Process Network* model and the *Application State Transition* model. The former model combines the Kahn process network model with dynamic dataflow, whereas the latter model combines CSP with dynamic dataflow.

In the dataflow process network model (DPN) [Lee and Parks, 1995; Parks, 1995], a single Kahn process contains a function and a set of *sequential firing rules*. The process itself checks which firing rule applies, instead of some global scheduler doing this as in the dataflow models. The process checks these rules in a sequential order with blocking reads (which is why this model uses *sequential firing rules*). If a valid firing rule is found, the process fires the actor or function. At the moment the process fires the actor, all input arguments of the function are present and it evaluates instantaneously. Tokens are written to the output ports using a non-blocking write. After evaluating the function, the process checks the firing rules again until a valid firing is found.

In the Application State Transition Model (AST) [Kung, Annevelink, and Dewilde, 1984; Annevelink, 1988], a single CSP process contains a set of functions, whereby each function reads input data from specific read ports and writes data to specific write ports. When one function from the set is active, other functions are idle until it has evaluated. After a function evaluates, a control function f_c decides which function to evaluate next based on state s and installs a new state s' [Held, 1996]. Within the AST mode, the control function can read tokens from a special control port and use these tokens to decide which function it should evaluate next. The CSP process used in the AST model is specialized by forbidding the use of non-deterministic processes and by forbidding the use of sequential compositions of processes. Because this model was inspired by the concept of the applicative state transition model presented by Backus [1978], it is named accordingly.

When we compare the SBF model with other models discussed, we see that the SBF model describes a combined dataflow/process model. It combines the Kahn process network model with the dynamic dataflow model. The SBF model differs from the DPN model by having a set of functions instead of a single function present in a process. Furthermore, the SBF model uses the notion of a controller instead of a set of firing rules. As a consequence, when a function is activated within the SBF model, the function reads input data from input ports with a blocking read and writes data with a non-blocking write. The SBF model differs from the AST model in that it uses Kahn process networks instead of CSP processes. A Kahn process runs forever, which is more consistent with the notion of streams. In addition, the Kahn model describes dynamic applications naturally in a deterministic way. In contrast, a CSP process typically terminates and describes a non-deterministic process [Hoare, 1978]. Furthermore, the controller in a SBF model only observes the state of a SBF object to determine the next state. Hence the controller does not have a direct connection to the world outside the SBF object using a control port, as it does in the AST model.

All three models presented distinguish between one or more functions and a control part. In addition, the models describe a fire-and-exit behavior. Each time a function evaluates, the control part observes that a transition took place. Such a transition leads to the clear sequence of function activations as shown in Equation 6.7. However, a Kahn process is a free interleaving of control and function statements, as long as the sequence of statements uses blocking reads. In that case, a Kahn process might contain `for`-loops and other control constructs; the clear notion of a transition is then no longer present. The notion of a transition is nevertheless pivotal to successfully integrating applications onto architecture instances, as we will explain in Chapter 7 (where we discuss the mapping of SBF objects onto functional elements of instances of the stream-based dataflow architecture). Transitions define clear moments at which to switch from evaluating an application to evaluating an architecture instance on which the application is mapped.

Finally, we remark that the SBF model can act as dataflow models HDF, SDF, CSDF and deterministic DDF. In the case of HDF and SDF, the set of functions contains only one function. Furthermore, the transition function is equal to $c = 1$ and the binding function binds the only function present to the control state c, which is always 1. In the case of CSDF, the set P contains more than one function. In this case, the transition function describes a cycle-static schedule by observing only the control space C to determine the next state. In the case of deterministic DDF, the set P contains more than one function. The transition function determines the next state based on both the data space D and the control space C.

6.8 Implementation of the SBF model

We have constructed the simulator *SBFsim* for simulating arbitrary networks of SBF objects. This simulator uses the object oriented programming language C++ and a multithreading package, and it consists of approximately 3000 lines of code. To describe a stream-based application as a network of SBF objects requires that we describe both the network of SBF objects as well as the SBF objects themselves. We use two different languages, as proposed in [Lee and Parks, 1995]; one to describe the network, called the *coordination language,* and one to describe the SBF objects, called the *host language*.

6.8.1 Host Language

The SBFsim simulator uses the programming language C++ as the host language. A C++ object describes an SBF object very easily because of the similarity between a C++ object and an SBF object. The state S (i.e., both data space D and control space C) is implemented as data of an object. The functions of the set P are implemented as methods of an object. The controller of an SBF object is implemented using the C++ function-call `operator()` [Stroustrup, 1991]. This operator is a special method that overloads the functional-call operator of an object, a step which facilitates the integration of an SBF object in a process, as we show later.

To show how a C++ object describes an SBF object, we describe the sample rate converter (SRC) used in the PiP example that was given in Figure 6.1 as an SBF object. In the PiP application, the SRC performs a simple down-sampling of a stream of tokens by a particular factor (e.g., a factor of 2). The SRC is described as an SBF object by the C++ object `DownSampling`. Program 6.2 describes the set of functions and Program 6.3 describes the controller. The SBF object has two functions, `function_A` and `function_B`. They are both implemented as methods of the C++ object `DownSampling`, as shown in Program 6.2. Within the functions, tokens are read using the `get` function, and written using the `put` function. In `function_A`, a token (e.g., `aToken`) is read from a channel via read port `in`, processed by the (identity) function `f` and assigned to a new token (e.g., `aNewToken`) written to a channel via write port `out`. `Function_B` only reads a token and then destroys it immediately. Hence, it does not produce any output.

The functions within Program 6.2 have as arguments a read and a write port instead of values.

Program 6.2. THE SET OF FUNCTIONS MODELED IN C++

```
DownSampling::function_A(ReadPort in, WritePort out)
{
   // Get Token
   Token aToken = in.get();

   // Perform Function
   Token aNewToken = f(aToken);

   // Put Token
   out.put( aNewToken );
}
DownSampling::function_B(ReadPort in)
{
   // Get Token
   Token aToken = in.get();
   delete aToken; // Throw away the token
}
```

This way functions themselves can perform the synchronization with channels using the `get` and `put` functions.

The controller of the SBF object is described in Program 6.3. It implements the binding function μ and transition function ω. The current state of the controller c is modeled via the C++ object variable `state`. This variable is defined in the data part of the C++ object. The binding function decides that `function_A` is enabled if the state variable equals zeros, otherwise `function_B` is enabled. After a function ends, i.e., has fired, the transition function evaluates. It determines the new current state value `state`. In this example, the transition function increases the state variable `state` by one, modulo a factor `factor`. SBF object `DownSampling` realizes a down-sample factor of two if we assign a value of two to the variable `factor` in the controller.

Program 6.3. CONTROLLER MODELED IN C++

```
DownSampling::operator()(ReadPort in, WritePort out)
{
   // The Binding Function
   if (state == 0) {
     function_A(in, out);
   } else {
     function_B(in); // Throw away token, no output
   }
   // Transition Function
   state = (++state%factor); // State variable
}
```

Each SBF object executes concurrently with other SBF objects in a network. To have concurrency within a C++ environment, we use a simple multithreading package that provides *processes* and *semaphores*, which can describe Kahn process networks. We already discussed such a package in in Chapter 4, namely the PAMELA Run Time Library [Nijweide, 1995] (RTL). We are already familiar with this library and therefore reuse the RTL Library in SBFsim. SBFsim basically describes a Kahn process network simulator that should correctly describe the partial ordering of samples within a network. Because this does not require time, other multithreading packages would do equally well. To describe Kahn process networks, Parks [1995] uses standard POSIX threads and Bŏhm [1983] uses

UNIX processes.

Using the RTL, a process is created that we use to let an SBF object run concurrently with other SBF objects. We use semaphores to implement FIFO buffers between the processes, as shown in Program 5.1. The RTL implements a process scheduler that interleaves the execution of processes as explained in Section 4.5. This gives the notion of SBF objects that execute concurrently.

Program 6.4. EXAMPLE OF PROCESS

```
// endless loop
while(1)
{
   (SBF_object)(inPort_0, outPort_0);
}
```

An example of a RTL process is given in Program 6.4. It describes a process that never terminates, because it contains an endless loop using the `while(1)` statement. Inside the loop, an SBF object is called as a function call with two arguments, `inPort_0` and `outPort_0`. These two arguments are respectively the read port and write port connected to channels which are FIFO buffers that connect to other SBF objects within a network.

Building Blocks

The building blocks introduced in Chapter 4 are used again to describe a single SBF object. The behavioral part of a building block describes the process of an SBF object and could be the process description as given in Program 6.4. The structural part of the building block contains references to channels. For the SBF object described in Program 6.4, the structural part would thus contain references to the channels via read port `inPort_0` and write port `outPort_0`.

The description given of the example of a system in Figure 4.5 describes a network of SBF objects, if we do not take resource resource1 and resource2 into account. Each node in the network is a building block representing an SBF object and the edges between the nodes are unbounded FIFO buffers instead of semaphores. In this way, the network in Figure 4.5 describes an application as a network of SBF objects.

Fire-and-Exit Behavior

Each process defines a single thread-of-control that is passed on from the endless loop to the controller of an SBF object via the function-call operator (this is shown in Program 6.3). Assuming that this is the SBF object `DownSampling`, then the binding function in the controller program gives the thread-of-control to either `function_A` or `function_B`. If, at any moment, one of the functions terminates (i.e., the function has fired), the function returns the thread-of-control to the controller, which executes the transition function. This function updates the state of the SBF object. Finally, the thread-of-control exits the controller and returns to the endless loop in Program 6.4 and the whole process described is repeated another time.

The description in Program 6.4 shows clearly why an SBF object describes a *Fire-and-Exit behavior*. Recall the sequence that we presented in Equation 6.7, in which a function of the set fires and a transition takes place. Each time a function *fires*, the transition function executes and the thread-of-control *exits* the controller of the SBF object to return to the endless loop. Within the endless loop, we can determine whether transition took place. As we will explain in Chapter 7, we use the notion of a

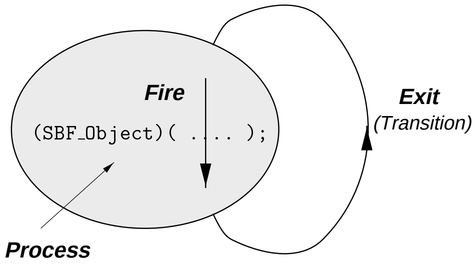


Figure 6.10. Fire-and-Exit behavior.

transition to couple time to an SBF object. Each time a transition takes place, we execute a **pam_delay** statement. Therefore, the execution of a function takes time.

Library of SBF Objects

We store SBF objects in a *library* to encourage reuse of previously defined SBF objects. Furthermore, we allow SBF objects to describe SBF objects in a parameterized form. Using the overloading of the function-call operator, an arbitrary SBF object with one input and one output can be used in Program 6.4. For that purpose, an SBF object `SBF_Object` is instantiated from the library. Notice that a process description as given in Program 6.4 describes a one-input, one-output function call. Therefore separate function descriptions are needed for other combinations of input and output arguments.

6.8.2 Coordination Language

The SBFsim simulator uses a dedicated coordination language that is implemented using a *parser* to recognize a network description language [1]. This language describes the processes used in a network and the connections between these processes. The parser also instantiates SBF objects from the library and places them onto the processes. Furthermore, it instantiates unbounded FIFOs for the channels between SBF objects.

Program 6.5. Network of SBF Objects

```
Program PictureInPicture {
    ( x1 )     = Source<Source>( NULL );
    ( x2 )     = FIR_H<FIR>( x1 );
    ( x3 )     = SRC_H<DownSampling(factor=2)>( x2 );
    ( x4 )     = Transpose_H<Transpose>( x3 );
    ( x5 )     = FIR_V<FIR>( x4 );
    ( x6 )     = SRC_V<DownSampling(factor=2)>( x5 );
    ( x7 )     = Transpose_V<Transpose>( x6 );
    ( NULL )   = Sink<Sink(stop=1000)>( x7 );
};
```

An example of a network description is given in Program 6.5. It describes the PiP application given in Figure 6.1. We create a process in the network description for each coarse-grained function in the application. We created a process, for example, for the horizontal SRC `SRC_H` that one input `x2` and one output `x3`. For each process, SBFsim creates a RTL process as described in Program 6.4.

[1] See Section 5.5, where we describe an architecture template, for more information about the use of a parser. In that section we also explain how the parser instantiates building blocks

SBFsim installs an SBF Object that is taken from a library on these processes, as described between the "$< >$" brackets. For example, SBFsim installs on process `SRC_H` the SBF object `DownSampling`, as described in Programs 6.2 and 6.3. An SBF object like `DownSampling` can be parameterized. In the network description, we indicate that the SBF objects for `SRC_H` and `SRC_V` should perform a down-sampling of a factor of 2.

Each process in Program 6.5 has input and output arguments: on the left-hand side the output arguments and on the right-hand side the input arguments. The source and sink processes have no input and output arguments, respectively, as indicated by the keyword `NULL`. The channels between the processes are `x1` to `x7`. In the example, SBFsim implements each channel as an unbounded FIFO. The sink process has the additional parameter `stop=1000`, to indicate that it should stop executing after it has read 1000 tokens. When the sink process stops, the whole network stops.

Although Kahn process networks require unbounded FIFOs, Parks [1995] discusses the construction of Kahn networks using bounded FIFOs. He argues that the deterministic properties in a Kahn process networks are retained when bounded FIFOs are used. The use of bounded FIFOs requires blocking writes to implement a write to a bounded channel. We constructed SBFsim such that it can instantiate FIFO buffers of arbitrary (parameterized) capacity.

6.9 Conclusions

Within the Y-chart environment we want to map sets of applications onto architecture instances. In this chapter, we introduced the stream-based functions model of computation to describe the set of applications. The main reason why we developed the SBF model was to make a smooth mapping of applications onto architecture instances possible. The SBF model is a new model of computation suitable for describing stream-based applications as a network of SBF objects.

We presented the SBF object and showed that it has a solid foundation in two well-known models: Kahn process networks and the AST model developed by Backus. The SBF model structures Kahn processes such that they consist of a set of functions, a controller and state. We call such a structured Kahn process an SBF object. SBF objects describe control (the controller) and dataflow (the set of functions) in one and the same SBF object. This provides a high degree of flexibility when describing (complex) consumption/production patterns of tokens from streams. These consumption/production patterns result from transforming multi-dimensional data structures into one-dimensional streams.

SBF objects can describe fine-grained to very coarse-grained functions. As such, networks of SBF objects describe applications at different levels of granularity. The granularity of the SBF objects influences the granularity of the function executed by an functional element. The granularity of these functions is an important design parameter in stream-based dataflow architectures. The granularity of SBF objects can be enlarged by combining them into a new SBF object or it can be reduced by decomposing an SBF object into a set of new SBF objects.

We placed the SBF model of computation in the context of other relevant models of computation. We considered dataflow models like homogeneous dataflow, synchronous dataflow, cyclo-static dataflow, and dynamic dataflow. We also discussed process models like Kahn process networks and communicating sequential processes. Finally, we discussed models that combined dataflow/process models like the dataflow process network model and the application state transition model. We showed that the SBF model also describes a combined dataflow/process model.

Finally, we described the construction of SBFsim, a simulator that can simulate networks of SBF objects. In the description of these networks, we made the distinction between the host language, in which we describe SBF objects, and the coordination language in which we describe the network of SBF objects. We showed that the SBF model of computation can be implemented very simply using C++ and a multithreading package. We also showed that we can reuse the building block approach presented in Chapter 4 by representing each SBF object by a building block. The coordination language used in SBFsim is built using a parser that parses a network description language.

Bibliography

Jurgen Annevelink. *HiFi, A Design Method for Implementing Signal Processing Algorithms on VLSI Processor Arrays*. PhD thesis, Delft University of Technology, 1988.

John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613 – 641, 1978.

U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988.

Shuvra S. Bhattacharyya and Edward A. Lee. Memory management for dataflow programming of multirate signal processing algorithms. *IEEE Transactions on Signal Processing*, 42(5), 1994.

G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static data flow. In *IEEE International Conference ASSP*, pages 3255 – 3258, 1995.

Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, 1992. Special issue on Simulation Software Development.

J.T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, Dept. of EECS, University of California at Berkeley, 1993. Tech. Report UCB/ERL 93/69.

A.P.W. Bŏhm. *Dataflow Computation*. Number 6 in CWI tract. Centrum voor Wiskunde en Informatica, Amsterdam, 1983. Dutch.

Adele Goldberg and David Robson. *Smalltalk-80, The Language and Its Implementation*. Addison-Wesley Publishing Company, Xerox Palo Alto Research Center, 1983.

Peter Held. *Functional Design of Dataflow Networks*. PhD thesis, Delft University of Technology, 1996.

C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666 – 677, 1978.

C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

R. Jagannathan. Dataflow models. In E.Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*. McGraw-Hill, 1995.

Johan G.W.M. Janssen, Jeroen H. Stessen, and Peter H.N. de With. An advanced sampling rate conversion algorithm for video and graphics signals. In *IEE Sixth International Conference on Image Processing and its Applications*, Dublin, 1997.

Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *Proc. of the IFIP Congress 77*, pages 993 – 998. North-Holland Publishing Company Co., 1977.

S.Y. Kung, J. Annevelink, and P. Dewilde. Hierarchical iterative flowgraph integration for VLSI array processors. In *Proceedings University of Southern California (USC) Workshop on VLSI and Signal Processing*, 1984.

Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9):1235 – 1245, 1987.

Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5): 773–799, 1995.

Marc Nijweide. The pamela run-time library, version 1.3. Technical Report 1-68340-27(1995)06, Laboratory of Computer Architecture and Digital Techniques, Delft University of Technology, 1995. http://dutepp0.et.tudelft.nl/ gemund/publications.html.

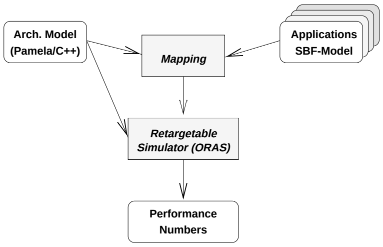
Tom Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California at Berkeley, 1995.

Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.

Arthur H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4):366–396, 1986.

Chapter 7

Construction of a Retargetable Simulator and Mapping



Contents

THE use of the concepts presented in the previous chapters, for the construction of the *Object oriented Retargetable Architecture Simulator* (ORAS) is what we discuss in this chapter. We also discuss how applications map onto executable architecture instances that have been derived from the architecture template of stream-based dataflow architectures, using ORAS.

So far, we have discussed performance modeling techniques in Chapter 4 to model systems to acquire performance numbers. In Chapter 5, we used this modeling technique to construct executable architecture instances of stream-based dataflow models using building blocks. In Chapter 6 we explained how we model applications using the stream-based function (SBF) models. In this chapter, we combine the concepts presented in the last three chapters to construct ORAS.

A retargetable simulator derives specific simulators for different architecture instances. It does this from an architecture template as we discuss in Section 7.1. The retargetable simulator ORAS is presented in Section 7.2 and is followed in Section 7.3 by an explanation of the three steps involved within ORAS. Being able to derive a simulator for an architecture instance is not all that is required to produce performance numbers; we must also map applications onto such an architecture instance. How applications, which are written as networks of SBF objects, map onto architecture instances is discussed in Section 7.4. The SBF model describes dynamic dataflow. The workload that an application represents may change, depending on the content of the stream that the application processes at run-time. To reflect the correct dynamic workload on an architecture instance without much modifications an architecture/application interface is needed, as we discuss in Section 7.5. In Section 7.6, we present an example that shows how ORAS constructs an architecture instance that executes the Picture in Picture (PiP) application (discussed in Chapter 6). We look at a one-to-one mapping and a many-to-one mapping of this application onto an architecture instance. In Section 7.7 we discuss related work on constructing high-level retargetable architecture simulators. In Section 7.8 we evaluate ORAS in the context of other architecture simulators. Finally, in Section 7.9, we reconsider the three steps used in ORAS and show how they provide a generic outline for retargetable simulators for other architecture templates.

7.1 Retargetable Architecture Simulators

To be able to execute architecture instances within the Y-chart environment, we need a retargetable architecture simulator for the architecture template of stream-based dataflow architectures. We define, inspired by [Marwedel and Goossens, 1995], a retargetable architecture simulator to be

Definition 7.1. RETARGETABLE ARCHITECTURE SIMULATOR

A *Retargetable Architecture Simulator* is a simulator that can simulate different architecture instances from an architecture template if an architecture description is provided for each architecture instance. □

In this chapter, we develop a retargetable simulator that derives a specific executable simulator for a specific architecture instance. We also discussed the development of the retargetable simulator in [Kienhuis et al., 1998]. When we say a retargetable simulator *derives* a specific executable simulator, we mean that the retargetable simulator instantiates a specific executable simulator in various steps. By executing such a derived simulator, we obtain the desired performance numbers for an architecture instance.

7.1.1 Requirements

The retargetable simulator that we develop for stream-based dataflow architectures needs to satisfy to the requirements of retargetability, execution speed, and accuracy.

Retargetability The retargetable architecture simulator must be capable of deriving simulators for all possible architecture instances of an architecture template. The parameters present in an architecture template of stream-based dataflow architectures clearly indicate the level of retargetability required. We should exploit this confinement to obtain efficient retargetable simulators. Furthermore, the derived simulator should simulate valid instances of the architecture template. We thus need to be able to verify whether an architecture instance is a valid one with respect to the architecture template of stream-based dataflow architectures.

Execution Speed The retargetable simulator will be used in Chapter 8 to evaluate the performance of architecture instances for the purpose of design space exploration. This leads to a very important requirement for the derived simulators, namely their execution speed. In the end it is the execution speed that determines the number of instances which we can evaluate within a certain amount of time. This requirement is especially important in application domains like video and radar that are very computationally intensive and involve a large amount of communication. Providing support for retargetability introduces a certain amount of overhead in a simulator. Since this overhead reduces the simulation speed of the derived simulators, it should be kept as small as possible.

Accuracy Here we study derived simulators that deliver cycle-accurate performance numbers. These simulators must also be able to execute the dynamic behavior of both the architecture and the applications. That is to say, the simulators must evaluate an interpreted performance model [1]. We provide such a model by mapping applications onto architecture instances and executing the application on the architecture instance.

7.2 The Object Oriented Retargetable Simulator (ORAS)

We develop the *Object Oriented Retargetable Simulator* (ORAS), which derives a specific simulator for an architecture instance using the following three steps:

1. ORAS constructs the structure of an architecture instance from a textual architecture description.
2. ORAS adds an execution model to the structure of the architecture instance to obtain an executable architecture instance.
3. ORAS instruments the executable architecture instance with *metric collectors* to measure and to extract performance numbers for selected performance metrics.

The three steps implemented in ORAS are shown in Figure 7.1. The first step concerns only the structure of architecture instances. In this step, ORAS exploits object oriented principles to construct the structure of architecture instances. It employs a parser that combines architectural elements, which are modeled as building blocks, to construct the structure of a stream-based dataflow architecture

[1] See Chapter 4 where we explain what interpreted and uninterpreted performance models are

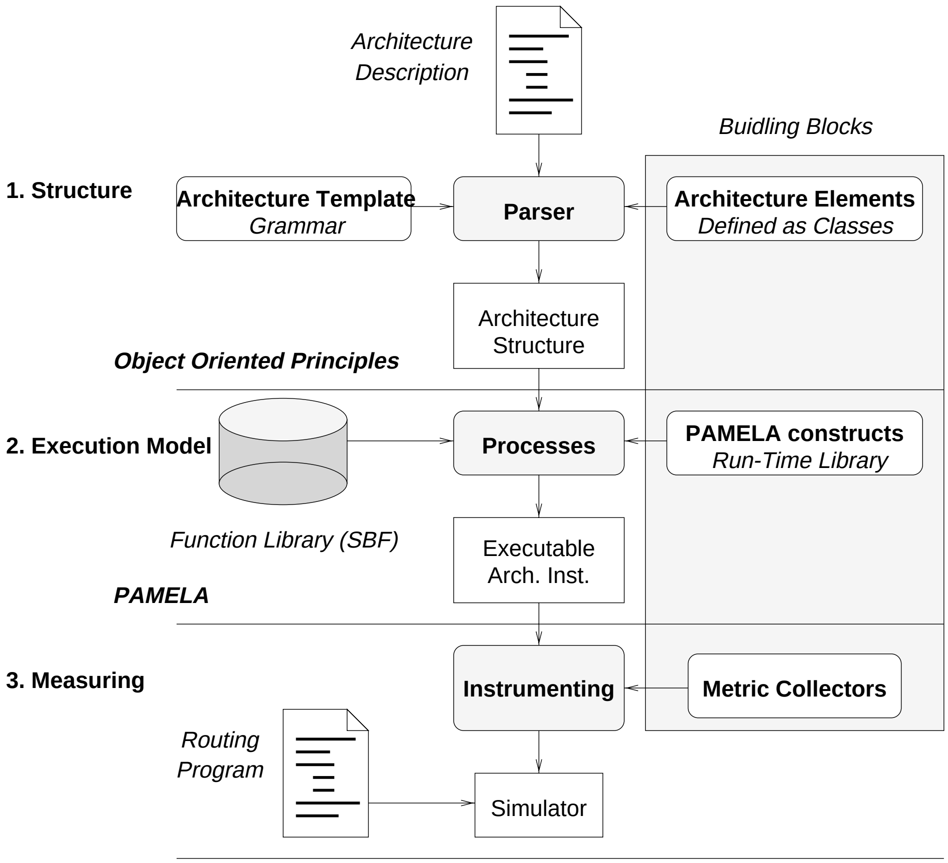


Figure 7.1. The three steps used within the Object oriented Retargetable Architecture Simulator (ORAS).

instance. The parser validates whether an architecture instance specified in a textual architecture description complies with the architecture template. In the second step, ORAS adds an execution model based on the Run-Time Library (RTL) to the structure of an architecture instance, yielding an executable architecture instance. It also instantiates SBF objects from the function library. In the third step, ORAS instruments the executable architecture instances with metric collectors to measure and extract performance numbers for particular performance metrics during a simulation. Finally, the derived simulator reads in a routing program to execute an application, thus rendering the requested performance numbers.

Figure 7.1 is very similar to Figure 5.2, which shows the four issues involved in constructing an executable model of an architecture instance from an architecture template. Figure 7.1 differs from Figure 5.2 in that the use of building blocks to model architectural elements is explicitly indicated in it. In addition, the modeling process is split into three separate steps. Each building block contains a structural part used in the first step, a behavior part used in the second step, and the building blocks are instrumented to extract performance numbers in the third step. Another difference is that SBF objects are instantiated onto functional elements from a library of SBF objects in the second step.

7.3 Development of ORAS

In our development of ORAS, we focused primarily on constructing a retargetable architecture simulator that executes fast while being cycle-accurate and capable of performing a functionally correct simulation. We now elaborate on the three steps used in ORAS.

7.3.1 Step 1: Structure

The first step taken in ORAS concerns the construction of an architecture instance that is a feasible instance of the architecture template of the stream-based dataflow architecture. The construction of such complex structures is easier if a programming language satisfies the following requirements:

Composition We defined an architecture to be a collection of interconnected architectural elements as explained in Section 5.1. Each architectural element defines the constituent elements of an architecture instance and is modeled as a building block. We combine these building blocks to form larger hierarchical structures. For setting up the structure of an architecture instance, only the structural part of a building block is relevant.

Behaviors We introduced the notion of “architectural element type” to indicate that an architectural element has properties that can be implemented using more than one behavior. We describe the concept of *types* by means of abstract classes. When we construct an architecture using only *types*, we can still select various behaviors. This is illustrated in Figure 5.4), in which we show the type *Buffer* described as an abstract class with read and write methods. Three classes are derived from this abstract class, and they implement different behaviors, namely that of a handshake buffer, a bounded FIFO buffer, and an unbounded FIFO buffer. For the stream-based dataflow architecture we set up a class hierarchy (as shown in Figure 5.5) which shows the abstract classes describing architectural element types and the derived classes describing the various behaviors of the types.

Verification The architecture instance for which ORAS derives a simulator must comply with the architecture template of stream-based dataflow architecture. We explained in Section 5.5 how we can specify an architecture template in terms of composition rules in BNF [2]. These rules also describe the grammar of an architecture description language.

ORAS uses a parser (i.e., Flex and Bison) to decompose a textual architecture description into elements that the parser matches against the grammar of the architecture template. If a valid rule is found, the parser activates the associated action; for example, instantiating an object representing a building block from the class hierarchy shown in Figure 5.5. The parser also resolves interconnections between objects, i.e., which object connects to which other object.

High-Level Constructs ORAS constructs the structure of architecture instances by combining objects. To facilitate the combination of objects, ORAS adopts high-level programming constructs like *vectors*, *lists*, and *maps*. These high-level constructs are part of the new standard in C++ libraries called the *Standard Template Library* (STL) [Musser and Saini, 1996]. Because STL [3] implements the high-level constructs using the C++ template mechanism [Stroustrup, 1991], the constructs are very efficient and introduce very little overhead. STL has proven to be a key technology for the effective and efficient development of ORAS.

[2] See appendix A for the BNF rules describing the composition rules of the architecture template of stream-based dataflow architectures

[3] Examples of STL constructs were shown when describing the various building blocks in Section 5.4

7.3.2 Step 2: Execution Model

In the second step, ORAS adds an execution model to the structure of an architecture instance, to acquire an executable architecture instance. The model it adds is the Run-Time Library presented in Section 4.4. The process descriptions of building blocks are created during this step, using **method** `init_process` as discussed in Chapter 4. The processes activated for the various types of architectural elements were discussed in Section 5.4.

ORAS instantiates the structure of all objects in the first step and in the second step passes this structure on to the processes of building blocks (Using the *this-pointer* as explain in Section 4.4). Because the structure of the architecture is already resolved in the first step, processes no longer have to decode any structural aspects of the architecture at run-time. This is a crucial step in the creation of an efficient simulator that is still retargetable. In this second step, ORAS also instantiates SBF objects (as presented in Chapter 6) onto functional elements. ORAS does this *without any modification* of the SBF object. We explain in detail in the second part of this chapter how the instantiation of SBF objects onto functional elements takes place.

7.3.3 Step 3: Metric Collectors

The building blocks in ORAS are instrumented with metric collectors so that they can harvest the performance numbers from the derived simulator. These *metric collectors*, which can be activated in the third step, are special objects that collect performance numbers during a simulation for high-level performance metrics and present the results, possibly in statistical form, at the end of a simulation.

Element Type	Performance Metric
Comm. Structure	Utilization
Controller	Utilization
Buffer	Filling Distribution
Routers	Response Time Controller
Functional Unit	Utilization, Number of Context Switches
Functional Element	Utilization, Pipeline Stalls Initiation Period, Number of Operations
Architecture	Number of Operations, Total execution time

Table 7.1. Metric collectors implemented for the instrumentation of different architectural element types.

In Table 7.1 some performance metrics are given for architectural elements of the stream-based dataflow architecture. The metric collectors gather information about the complete architecture, such as the number of executed operations or the total execution time in cycles. We can use these numbers to evaluate, for example, the performance metric “parallelism”. Other collectors may measure how long a semaphore blocks a process, thus measuring “response time” or “waiting time”.

In Program 7.1, we show the code of **method** `write` of a FIFO-buffer again, but now we include the code for the metric collector `metricCollector` to determine the filling distribution within the FIFO buffer, an instrumentation done for every FIFO buffer in the architecture. Each time a token is written into the FIFO buffer, the metric collector determines how many tokens are present in the buffer at write times. At the end of a simulation, a metric collector creates a histogram of the FIFO

Program 7.1. THE WRITE METHOD OF A BOUNDED FIFO BUFFER

```
void Fifo::write(Sample a)
{
    pam_P (room);                               // Is there Room in the FIFO?
     metricCollector.histogram(token++);          // Instrumentation
    buffer[writefifo] = a;                     // Write in buffer
    writefifo = (++writefifo)%capacity;
    pam_delay (1);                              // It takes 1 clock cycle to write
    pam_V (data);                              // Tell there is data available
}
```

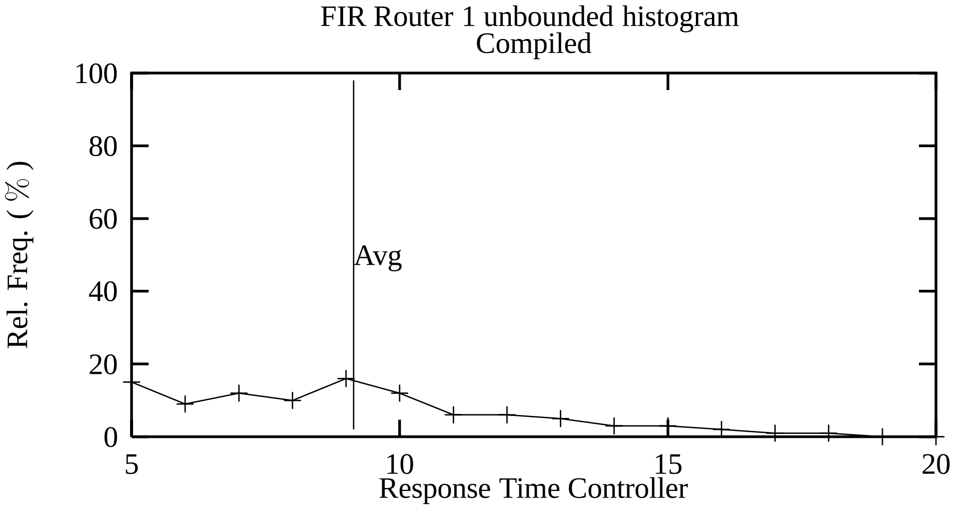


Figure 7.2. Histogram of the sample distribution of a buffer.

buffer filling, as shown in Figure 7.2.

The histogram shows in percent how many samples are in a buffer when a sample is written into the buffer. On average, there are 9 samples in the buffer when a sample is stored in the buffer. However, the maximum number of samples that can be stored at a particular time instance is 19 samples. The minimum number of samples that are stored at a particular time instance is 5 samples.

We show performance numbers in Table 7.2, produced by some of metric collectors given in Table 7.1. It shows the visit count and utilization for the functional elements “LowPass” and “Transpose”. As an example, `LowPassFilter` was visited 25830 times during a simulation; it was thus utilized for 33.19%. The table also shows the total number of operations executed by all functional elements in a simulation (83937), which, according to metric collector `parallelism`, means that on average 1.08 functional units were active per cycle.

7.4 Mapping Applications

Not only must we derive a simulator for an architecture instance, but we must also be able to map applications (e.g. the workload) onto such an architecture instance to evaluate its performance. We think that this mapping can be solved quickly only if the model of computation matches the model of architecture (we presented this mapping approach in Section 3.3.2). In this section, we show what a natural matching of models of architectures and computations actually implies. We also show that

LowPassFilter_visit_count	= 25830
LowPassFilter_utilization	= 33.19%
Transpose_visit_count	= 12911
Transpose_utilization	= 16.59%
Parallelism	= 1.08
Operations	= 83937

Table 7.2. Results produced by metric collectors.

this mapping leads to a 3-level control hierarchy.

7.4.1 Mapping Approach

We discussed the mapping of applications already briefly in Section 5.6, where we discussed the programming model of stream-based dataflow architectures. In that section, we showed that edges of the network map to a buffer/router/buffer-tuple, that a node of the network maps to a functional element installed on a functional unit, and that a stream maps to a packet stream. Later, in Chapter 6, we specified what the model of computation of such network is and how the internals of such a node look. So, instead of mapping a node onto a functional element, we map an SBF object onto a functional element, as shown in Figure 7.3. A smooth mapping of this SBF object onto a functional element is only workable when the model of computation of the SBF model matches the model of architecture of stream-based dataflow architectures.

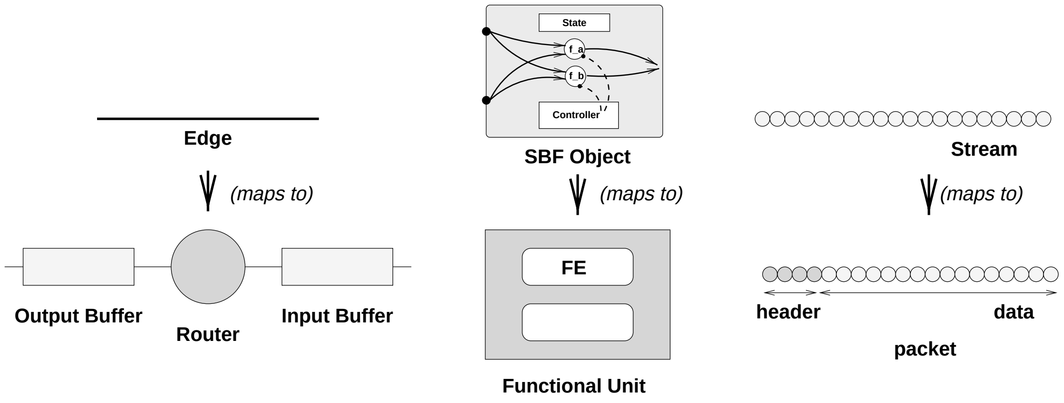


Figure 7.3. The mapping of an application described using the SBF model onto an architecture instance.

The actual digital signal processing takes place within functional elements. They execute functions that process the streams containing data samples (see, for an example, Section 5.4.9 where we modeled a functional element as a building block). When we want to map an application onto an architecture instance, it is the function of a functional element, however, that should describe the functionality represented by SBF objects. Such a simple mapping only takes place when the functional elements are indeed capable of describing the behavior of SBF objects. In stream-based dataflow architectures, functional units shield off functional elements from details of the architecture. Consequently, functional elements appear to operate in isolation on unlimited streams and they remain unaware of the fact that these streams are transported through architecture instances as packets.

As we defined it in Chapter 3, the model of architecture is the interaction between the various architectural elements. In the model of architecture of stream-based dataflow architectures, however, functional elements may be considered separately from other architectural elements. Because SBF objects map to these functional elements, it is only the behavior of functional elements that needs to be considered in the mapping. The communication taking place over buffer/router/buffer-tuples needs to adhere to a FIFO behavior, so the matching relationship between the stream-based dataflow architecture and stream-based functions model becomes the relationship as shown in Figure 7.4. The SBF objects part of the SBF model of computation should match the functional elements, which is part of the model of architecture of stream-based dataflow architectures.

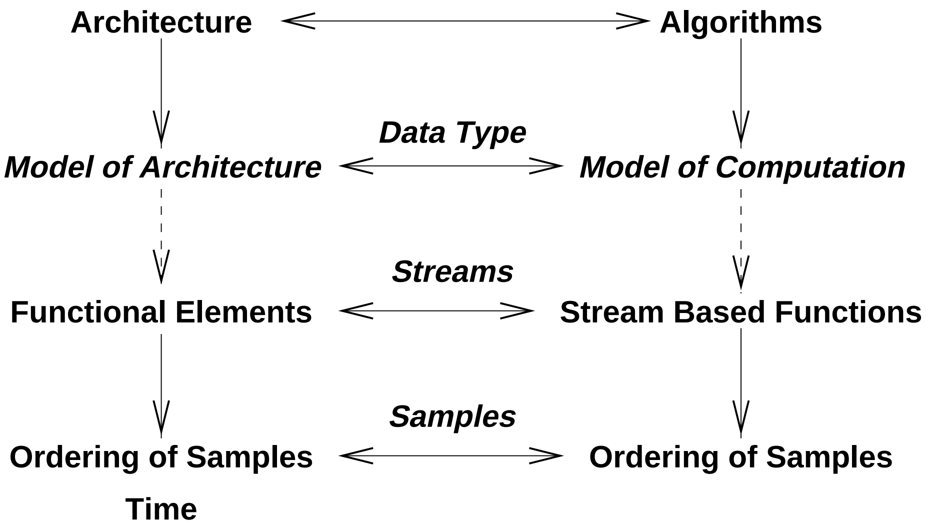


Figure 7.4. Matching the model of computation of the SBF object to the model of architecture defined by the functional elements implies that SBF objects map to functional elements.

7.4.2 Matching Models of Architecture and Computation

The mapping of an SBF object onto a functional element is illustrated in Figure 7.5. It shows a functional element with its read and write processes and a pipeline between these processes. It is similar to Figure 5.9, except that it has an SBF object with 2 input ports and 2 output ports mapped onto the read process of the functional element. This is possible because the model of computation of the SBF object matches the model of architecture determined by the behavior of the functional element.

Model of Computation A network of SBF objects is a Kahn process network and the SBF objects obey the Kahn rules. These rules assert that the underlying process of a single SBF object observes one buffer at a time and either reads a token from the buffer or blocks, implementing a blocking read. The same rules apply when writing to buffers (based on the assumptions stated by Parks [1995]).

Model of Architecture A functional element operates in a data-driven fashion, which means that its activity is determined solely by the availability of data. The read process of a functional element checks the availability of data by reading tokens from the input buffer using blocking reads. It reads from one buffer at a time. The writing of tokens happens using a blocking write.

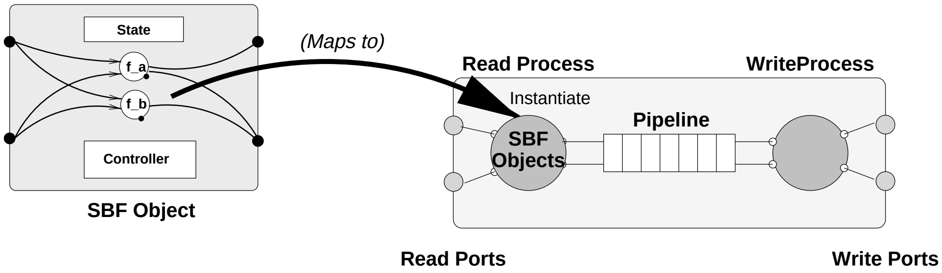


Figure 7.5. Mapping of an SBF Object onto the read process of a functional element.

Matching Both an SBF object and a read process of a functional element describe the same execution rules, i.e. the Kahn rules. Likewise, both an SBF object and a functional element operate on continuous streams of data of the same type, i.e. streams of samples. Because the behavior of both the functional element and the SBF object is the same, and moreover, because both operate on continuous stream of samples representing data samples, we conclude that SBF objects can seamlessly integrate into read processes of functional elements. How this integration takes place inside ORAS is discussed in Section 7.5.

7.4.3 Control Hierarchy

When we write applications as a network of SBF objects, we assume unlimited resources: each SBF object is a process on its own and it interconnects with other SBF objects in a point-to-point fashion using FIFO buffers. In contrast, an architecture instance on which we map applications has a limited amount of resources at its disposal. It exploits functional unit sharing to enhance efficiency, and uses a function repertoire to enhance flexibility, as explained in Section 2.1.3 where we discussed the behavior of stream-based dataflow architectures.

Because an SBF object maps onto a functional element, we map the point-to-point communications between SBF objects onto buffer/router/buffer-tuples, as shown in Figure 7.3. The mapping of the communication between SBF objects leads to a control hierarchy on the stream-based dataflow architecture that consists of three levels of controllers:

1. The global controller,
2. The controller that is local to the functional unit,
3. The controller that is part of the SBF object instantiated onto a functional element.

This control hierarchy is depicted in Figure 7.6. The first level, *Level 1*, relates to the global controller that we can program by means of a routing program (see Section 5.6, where we explain the programming model of stream-based dataflow architectures). The second level, *Level 2*, relates to the local controller of a functional unit. Based on the $\mathsf{H_f}$ field of a header arriving at the opcode-port of the functional unit, a functional element is activated from the function repertoire (i.e. $\mathcal{F} =$

$\{F_1, F_2, F_3, F_4\}$) (see Section 5.4.8, where we modeled the local controller of a functional unit). In the figure, the global controller has assigned a value to H_f that relates to functional element F_1. An SBF object is instantiated on each functional element and the third level, *Level 3*, relates to the controller being part of that SBF object (see section 6.4, where we described the modeling of SBF objects).

The controllers at level 1 and level 2 only concern the communication between SBF objects and the multiplexing of streams. However, the controller at level 3 is completely determined from the given application description. The control at level 3 is invariant with respect to the architecture instances.

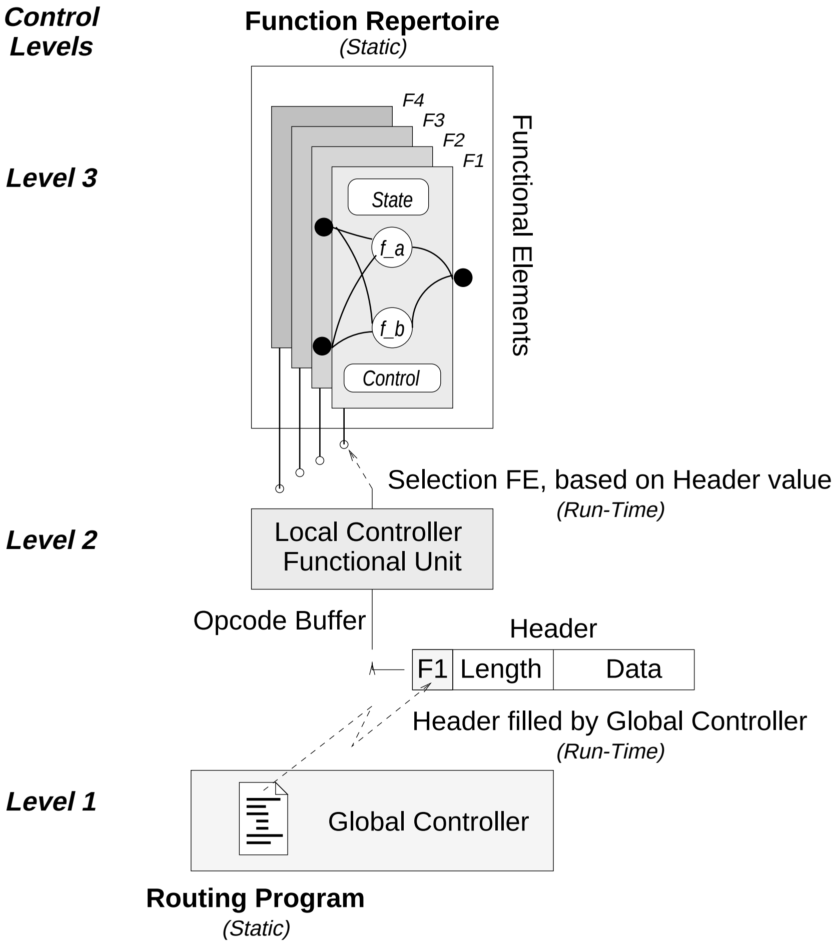


Figure 7.6. Control Hierarchy in the stream-based dataflow architecture.

Level 1 The first level of control is at the global level where the global controller resides. The global controller contains a routing program that describes the static routing of packets through an architecture instance. Routers interact with the global controller to receive this routing information (see Figure 5.16, where we showed how a router interacts with the global controller to change the content of a header). By down-loading another routing program, we can establish another flow of packets, causing the execution of another application.

Level 2 The second level of control is at the level of the functional units where the local controller resides. The local controller of the functional unit selects and activates a functional element at run-time from the function repertoire $\mathcal{F}$ of the functional unit, based on the H_f field of the header arriving on the opcode-buffer of the functional unit (in packet-switching mode). The value of $\mathsf{H_f}$ is set at run-time by routers. The function repertoire, on the other hand, is statically resolved when an architecture instance is defined. For example, in the architecture description given in Figure 5.11 we devised the regular processing element *Filter* with a functional unit with two functional elements, respectively *LowPass* and *HighPass*.

Level 3 The third level of control is at the level where SBF objects reside, which is internal to functional elements. The controller of an SBF object specifies the consumption/production pattern of an SBF object, and thus of a functional element (see, for an example, Table 6.1, where we show how the controller of an SBF object specifies the consumption/production pattern of the SBF object).

In the architecture description given in Figure 5.11 the functional element *LowPass* instantiates SBF object “Lowpass” and functional element *HighPass* instantiates SBF object “Highpass”, both from the library of SBF objects. At this level, by selecting a particular SBF object for the functional element to execute, we determine the grain size of a functional element, which equals the granularity of an SBF object.

Modeling the Function Repertoire

We describe function repertoires as consisting of different functional elements. Not every functional element is going to become a piece of hardware on its own. In practice, we combine functional elements into one piece of hardware. In the Jacobium case (which will be discussed in Section 9.3), we indicate how such a functional unit may look. Although we can construct a function repertoire of very different SBF objects, it is more logical to combine SBF objects that have more similar sets of functions (as given by Equation 6.1).

7.5 The Interface Between Application and Architecture

The SBF model describes dynamic dataflow. The workload that an application represents may change depending on the content of the stream(s) that the application processes at run-time. In the case of video applications, the workload that results depends on the sequence of images being processed. MPEG [Rao and Hwang, 1997] algorithms typify such applications. Suppose such an application describes a dynamic workload and executes on a simulator derived by ORAS, as shown in Figure 7.7. The processing of the clown video-sequence results in a workload, for which the simulator delivers particular performance numbers. If another sequence is chosen instead of the clown video sequence, then another workload results for which the simulator should deliver other performance numbers. Notice that the routing program – and thus the mapping – remains the same.

To have the simulator execute the correct dynamic workload without our having to make a large modification to the original application description, we want to evaluate SBF objects functionally correctly at run-time in the context of architecture instances. This implies that SBF objects should access and process the data which flows through an architecture instance, simulating an interpreted performance model of the architecture instance. Alternatively, SBF objects need an *Application – Architecture interface*. An SBF object interacts on this interface with the data flowing through the architecture instance. Via the interface, the SBF object can access the data from the architecture instance, process it, and put new results back into the architecture instance.

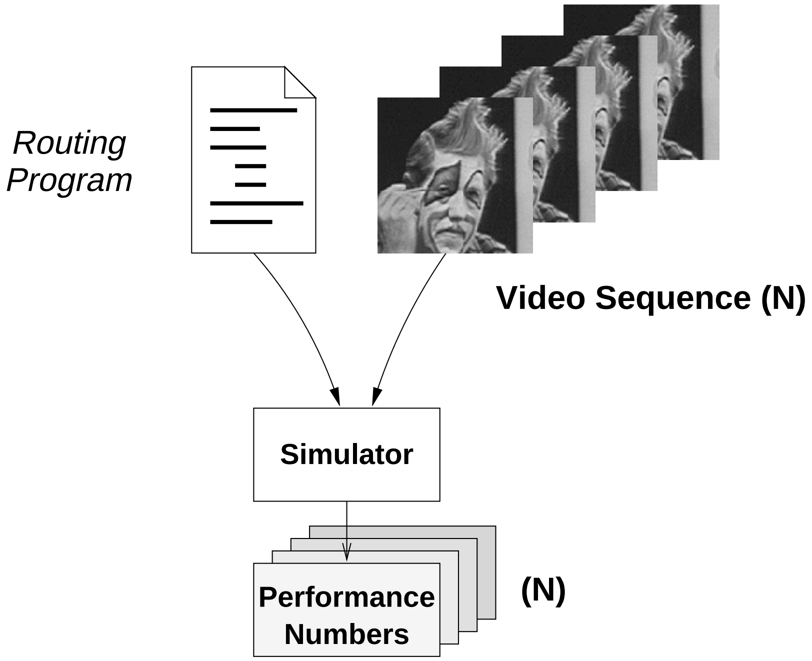


Figure 7.7. The performance numbers found by executing an application onto an architecture instance is also influenced by the type of content processed.

7.5.1 The Application – Architecture Interface

When we were discussing the 2-input, 2-output functional element in Section 5.4.9, we indicated that a functional element evaluates a function in its read **process** and executes the **method** `executeFunction22` shown in Program 5.8. This method contains the function call

```
(function)(sample0, sample1, &new_sample0,&new_sample1),
```

which instantaneously evaluates a standard C function call consuming two samples and producing two samples. The disadvantage of using this scheme is that the function call cannot handle *multi-rate* effects, i.e. a function must always consume a fixed number of arguments and always produce a fixed number of output values. One can circumvent this limitation by tagging the samples as either valid or not valid, but this would lead to many additional control statements, because checks are needed as to whether the data is valid or not.

Function Call Overloading

The SBF objects we introduced in Chapter 6 use an interface that can be seen as a regular C function call (This is shown in Program 6.4), where we integrate an SBF object into a RTL process). The interface is realized by overloading the C++ function-call operator (i.e. `operator()`) as explained in Section 6.8. The integration of an SBF object into a functional element only requires that we replace the original C function call in the read process of the functional element with the SBF object function call

```
(SBF_Object)(portIn[0],portIn[1],pipeline_portIn[0],pipeline_portIn[1]).
```

The SBF object functional call accepts references to read and write ports, respectively `portIn` and `pipeline_portIn`, whereas the original C function call accepts samples. This way, the controller of the SBF object becomes responsible for the read **process** of the functional element that reads a correct, but variable amount of tokens from `portIn` and writes a correct, but variable amount of tokens to `pipeline_port`.

We show the revised program for **method** `executeFunction22` in Program 7.2. The read and write ports used in the architecture modeling are different from the read and write ports used in the application modeling. In the architecture modeling, ports are involved in header processing, whereas in application modeling, headers do not exist. However, because of polymorphism, SBF objects remain unaware of this difference. In Program 7.2, we pass on the input ports of the functional element (i.e., `portIn[]`) and the input ports of the pipeline (i.e., `pipeline_portIn[]`) to the SBF object.

Program 7.2. EXECUTE A 2-INPUT, 2-OUTPUT SBF OBJECT

```
method  executeFunction22 {

    // Execute the SBF Object
    // Pass on pointer to input and output, do not pass samples
    (SBF_Object)(portIn[0],portIn[1],pipeline_portIn[0],pipeline_portIn[1]);

    // Calculate when these Samples are ready to leave the FE
    leave_time = pam_time () + (latency * initiation_period );

    // Pass the Execution to the sample
    pipeline_portIn[0].setTime( leave_time );
    pipeline_portIn[1].setTime( leave_time );
}
```

Passing Over the Thread-of-Control

As soon as the read **process** of a functional element passes the thread-of-control to the SBF object, the controller of the SBF object becomes responsible for the reading and writing of tokens. Because SBF objects have a fire-and-exit behavior (see Section 6.8, where we integrate an SBF object in a RTL process), the read **process** of the functional element always gets back the thread-of-control when a function has fired. Notice that when the enabled function of the SBF object tries to read from an input buffer (e.g., via portIn[0]) and blocks, the read **process** of the functional element also blocks.

Augmenting SBF Objects with Time

Because the read **process** of the functional element always gets back the thread-of-control when a function has fired, we can simply augment an SBF object with time by calling a **pam_delay** statement each time a transition (i.e., the firing of a function) takes place. We put a **pam_delay** statement in the endless loop in Program 6.4 following a call for the SBF object. In the context of the read process of a functional element, the **pam_delay** statement must be replaced by the **pam_delay** (`initiation_period`) statement shown in Program 5.9.

We use the architecture description that we gave in Figure 5.11 to illustrate how we define functional elements executing a function. In Figure 7.8, we describe one functional element on which ORAS will instantiate the SBF object `LowPass` from the library of SBF objects. This SBF object has two parameters (i.e., `latency` and `initiation_period`) resulting in a pipelined implementation of

18 stages and an initiation period of 1 sample per cycle. The initiation period of 1 is assigned to the **pam_delay** statement in Program 5.9. The pipelining is handled as explained in Section 5.4.10.

```
FunctionalElement LowPass(1,1) {
            Function { Type: LowPass(initiation_period=1,latency=18);  }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
            }
}
```

Figure 7.8. Definition of a functional element named `LowPass` in an architecture description, with one input and one output port.

7.5.2 Restricting Resources

An architecture instance is responsible for the handling of packets. An SBF object cannot do this, because it is, after all, isolated by the functional unit from the notion of packets. When an architecture instance uses an SBF object involving multi-rate effects, however, the amount of data produced by that object changes and it is the responsibility of the architecture to decide how to manage these changes.

It can choose between combining various input packets into one new packet and sending out smaller packets. Thus, suppose we have an architecture instance that uses a functional element on which an SBF object is installed that describes a down-sampler with a factor of two (see, for example, Program 6.3, describing the SBF object `DownSampling`). Now, if the SBF object processes a packet stream with a packet length of 100 samples, the local controller of the functional unit in which the functional element resides can either produce two packets of 50 samples each or combine two collections of 50 samples into a single output packet of 100 samples.

Although it is the responsibility of the architecture instance to handle the length of new packets, it is convenient to add an extra method (i.e., `getNewLength`) to SBF objects that provides control over the length of new packets. We incorporated the **method** `getNewLength` in the revised version of **method** `writeheader` of Program 5.14, as shown in Program 7.3. Before sending out a new header, the write port negotiates with an SBF object a new header length H_l using the **method** `getNewLength`.

7.6 Construction Example

As an example of how to use ORAS to construct an architecture instance, we now construct two architecture instances that are able to execute the Picture in Picture (PiP) application presented in Section 6.1. Both architecture descriptions and their routing programs are given in Appendix B.

For the PiP application we defined one architecture instance with eight functional units, as shown in Figure 7.9(a). Because it is an architecture instance, we have to specify all its parameters. For example, we have to specify that the architecture instance employs a switch matrix and a First-Come-First-Served (FCFS) strategy for the global controller. Furthermore, the packets produced by the source will contain 120 samples and a request to the global controller will take 4 cycles. The input and output are bounded FIFO buffers with a capacity of 30 and 20 samples, respectively. The functions from Figure 6.1 map *one-to-one* onto the eight functional elements, one for each functional unit.

Program 7.3. WRITE PORT METHODS

```
method  writeHeader
{
  // Each time we write a Header, we have to take it of the Queue
  header = headerQueue.pop();

  // Get the correct offset of the output port
  header.setBase( header.getBase() + Offset );

  // Get the length of the packet
  old_length = header.getLength();

  // Get a New length from the SBF Object
  length = SBF_Object.getNewLength( old_length );

  // While writing the Header, give back control to caller
  pam_delay ( 4 );
  output_buffer.write( header );
}
```

Since stream-based dataflow architectures permit sharing of functional units between different streams, they also support *many-to-one* mappings. In that case, more than one application function maps to one and the same functional unit. In Figure 7.9(b), we show an architecture instance that also can execute the Picture in Picture application but shares functional units. Both the architecture description and routing program are given in Appendix B. In this architecture instance, each functional unit has a function repertoire of two functional elements each executing identical functions (e.g., sample rate conversion). Each time a context switch occurs, a function has to operate on the correct data belonging to the correct stream. Because each function linked into these functional elements is an object (e.g., an SBF object) encapsulating data and functions within the same entity, we only have to switch between objects to make the activated function operate on the correct data (see the modeling of the functional element in packet-switching mode in Section 5.4.8).

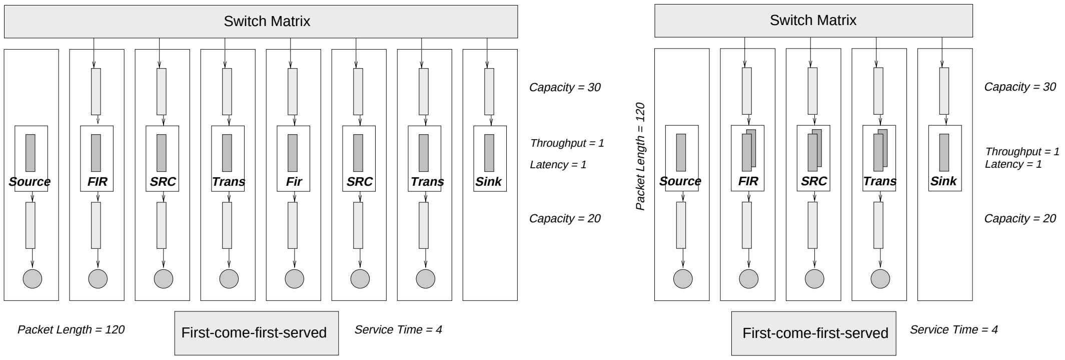
The current version of the simulator can simulate 10,000 SBF function calls (or firings) per second instantiated on arbitrary functional elements with all metric collectors active. ORAS needs 9 minutes to functionally process a full video picture of 720×576 pixels through the 8 functional units of Figure 7.9(a). The total implementation of ORAS required approximately 20,000 lines of C++ code.

7.7 Related Work

Very few high-level architecture simulators have been built for application-specific dataflow architectures. These simulators have typically been dedicated to a specific architecture, built at a low level of abstraction, and not, or hardly, made retargetable.

We already considered architecture simulators for general purpose processors in Chapter 3. Some of these simulators are retargetable, like the TriMedia simulator *TmSim* [Sijstermans et al., 1998] and the DSP simulator *Supersim* [Živojnović et al., 1996]. Both architecture simulators have in common at least the use of the C programming language, mainly for reasons of performance. Nonetheless, the C programming language does not support parallelism as required by the kind of architectures that we are considering.

Rose and Shackleton [1997] used VHDL to perform dedicated high-level architecture simulations



(a) One-to-one Mapping

(b) Many-to-one Mapping

Figure 7.9. Two architecture instances for Picture in Picture (PiP); (a) the *one-to-one* mapping of PiP and (b) the *many-to-one* mapping of PiP.

for high-performance signal processing systems within the RASSP project [Richards et al., 1997]. VHDL offers a parallel execution model and allows the instrumentation of architecture models to obtain performance numbers, but it cannot derive different architecture instances efficiently and quickly, mainly because VHDL is a specification language and not a programming language. Furthermore, the execution speed of VHDL is acceptable only when very high-level architecture models are used.

Witlox et al. [1998] used timed colored petri nets to construct a retargetable simulator for dataflow architecture, which is very similar to the stream-based dataflow architecture. The Petri nets are constructed and simulated using the ExSpect Tool [ASPT, 1994]. Although petri nets have strong modeling capabilities, their evaluation time is often slow. For example, the ExSpect simulator requires one day for processing one video frame using uninterpreted models.

The Ptolemy environment developed by Buck et al. [1992] is very useful for the specification of algorithms for high-performance applications. It is, however, less suitable for architecture explorations. Yet the Atrade tool, which was developed by Pauer and Prime [1997], makes it possible to define architecture instances in Ptolemy. From this definition, a performance model is synthesized within the discrete event (DE) domain. The tool also allows designers to define applications in the SDF domain. Since a static schedule characterizes these applications, the Atrade tool only needs to carry out an uninterpreted performance simulation and can thus very quickly obtain very accurate results. The Atrade tool implements a kind of Y-chart environment using the Ptolemy environment.

7.8 Discussion on ORAS

We have expend considerable effort on the construction of ORAS to make it retargetable and, above all, fast in execution speed. To quantify the execution speed of ORAS, we measured the execution speed of derived simulators and found that they simulate on average approximately 10,000 function calls of SBF objects (or firings) per second. To put this number in perspective, we compare this speed with the execution speed of the architecture simulators for microprocessors given in Table 3.2. Suppose each SBF object implements a simple RISC-like function; then ORAS would take 3.6 hours to process a video image application having a complexity of 300 RISC-like operations per sample (i.e., per video pixel). Given the high level of retargetability, the complexity of the architecture instances, and the fact that it is an interpreted performance model, ORAS is a fast retargetable simulator.

ORAS is not limited, however, to the use of RISC-like functions, but permits the use of arbitrary coarse-grained functions. We can still execute 10,000 function calls of SBF objects while describing a much more complex application, as shown in the Picture in Picture application example. In the PiP case, the simulation speed ends up in the range of minutes and not hours (i.e., 9 minutes for a whole frame), which clearly confirms that ORAS is a fast retargetable simulator.

7.8.1 Building Blocks

One reason why the execution speed of ORAS is high and at the same time still retargetable is that we used building blocks to construct the architecture.

The building block modeling technique developed in Chapter 5 allows us to describe every possible architecture instance in the design space of a stream-based dataflow architecture. The use of objects and high-level construction primitives of the STL library makes it very simple to describe complex architecture instances. Secondly, the building blocks separated the structure from the behavior. Because the structure of an architecture instance is resolved by the time processes are created, the processes no longer need to decode any structural aspects of the architecture at run-time.

7.8.2 High-level Primitives

Another reason why the execution speed of ORAS is high is that we used high-level primitives to model architectural elements and a very efficient simulation engine was available that can simulate very fast structures using these high-level primitives.

The PAMELA method discussed in Chapter 4 provides 3 high-level primitives: processes, semaphores, and delays. These primitives allowed us to model architectural elements at a high level of abstraction in terms of the four components of the Performance Modeling Base. As a consequence, we move up in the abstraction pyramid shown in Figure 3.3, while simulating the performance of an architecture instance cycle-accurately.

The three primitives can be simulated very efficiently using the RTL process scheduler. It requires a minimal amount of sorting, compared to other simulation engines, because of its simple structure, as we explained in Section 4.5. In particular, we compare the RTL with a VHDL simulation engine in Appendix C.

7.8.3 Interpreted Model

In addition to being fast and retargetable, ORAS is also able to obtain cycle-accurate performance numbers expressing the correct dynamic behavior of both the architecture and applications. ORAS integrates SBF objects seamlessly onto the functional elements of derived simulators, because of the architecture – application interface. This interface hardly introduces any overhead, while the derived simulator is able to execute an application functionally correctly.

7.8.4 Limitations

We conclude that with ORAS we have indeed succeeded in constructing a simulator that is fast and accurate. The limitation of ORAS is that it is built for a specific architecture. Further, the RTL works well for stream-based, data-driven kinds of architecture, but is inapplicable for control-oriented architectures like load/store architectures. The RTL is less suited to describing low-level control like polling, as we explain in Appendix C.

7.9 A Generic Approach for a Retargetable Simulator

In this chapter, we developed the retargetable simulator ORAS for stream-based dataflow architectures. In our opinion, the three-step approach shown in Figure 7.1 provides a generic outline for retargetable simulators; therefore, we discuss each step used in ORAS and indicate the changes needed when a retargetable simulator is needed for another architecture template.

7.9.1 Step 1: Structure

In the first step, we need to distinguish the various architectural elements used in the architecture template. For each architectural element, we define a class describing the structure of the architectural element. If architectural elements have alternative behaviors, we will describe the architectural elements as an architectural element type and describe this type as an abstract class. This procedure leads to a class hierarchy similar to the one shown in Figure 5.5. Next, the structure of the architecture template is described in terms of composition rules in BNF, as explained in Section 5.5. The BNF rules also provide a grammar for an architecture description language. Based on this grammar, a parser can process architecture descriptions.

To come to the first step shown in Figure 7.1, we must replace the “architectural elements” with the newly developed class hierarchy. Furthermore, we must replace the grammar of the architecture template and use the newly devised architecture description language.

7.9.2 Step 2: Execution Model

In the second step, the retargetable simulator adds an execution model or simulation engine to the structure. We can again use the RTL library as simulation engine and develop new process descriptions for each architectural element. We make a distinction between “active” and “passive” elements and define the correct methods and processes.

Given the limitation of the RTL as described in Appendix C, we could consider other simulation engines. These simulation engines furnish a more accurate execution model (i.e., lower in the abstraction pyramid) or an execution model that fits better with the architecture.

Gupta and Liao [1997] describe an execution model named *Scenic*. Their simulation engine uses lightweight threads (like the RTL does) and C++, and implements the VHDL execution model. As pointed out by Benders [1996], VHDL has severe problems with specifying *process control*, *exceptions* and *asynchronous communication* between concurrent entities while at the same time it is well suited to specifying low-level control. By integrating threads with a clock-cycle simulation engine, Gupta and Liao combined the power of threads to describe process behavior with the power of VHDL to describe low-level control. Post et al. [1998] have already shown how this simulation engine can be used together with C++ to describe ATM switches for performance evaluation. Because C++ is used, this engine fits in very well with our building block approach. Unfortunately, no information is available on the execution speed of the simulations.

Koot et al. [1997] developed another simulation engine, named *TSS*, that could also be considered. TSS is a tool developed at Philips Research for internal use. This is a very fast and modular clock-cycle-accurate simulation engine based on the C programming language. Although TSS can describe both synchronous and asynchronous communication, it is particularly fast for the synchronous case.

If we use another simulation engine, then to come to the second step in Figure 7.1, we must change the way we describe processes since each simulation engine provides other primitives. The approach, however, remains the same. We first build up the structure of an architecture instance and then add an execution engine.

7.9.3 Step 3: Metric Collectors

In the third step, we instrument the executable architecture instance with metric collectors. The metric collectors that we developed are generic and can be reused with little modification. To come to the third step in Figure 7.1, we can reuse the metric collectors.

7.9.4 Mapping

A smooth mapping of applications onto an instance is only workable when the model of computation matches the model of architecture. This implies that we need to investigate the architecture template to determine its model of architecture. Using this information, we select the appropriate model of computation to describe applications. If such a model does not exist, adjustments are needed in the architecture template or in the model of computation until a match is found. We remark that the classification of models of architectures is not an area as well developed as that of the model of computation.

7.10 Conclusions

In this chapter we discussed the construction of the Object Oriented Retargetable Architecture Simulator (ORAS). We also discussed the mapping of applications onto executable architecture instances that were derived by ORAS from the architecture template of stream-based dataflow architectures.

We showed that ORAS uses a three-step approach to derive executable architecture instances that deliver performance numbers. We showed how ORAS creates the structure of an architecture instance in the first step; it adds an execution model to the structure in the second step, making it possible to describe the behavior of an architecture instance; and activates metric collectors to harvest the performance numbers in the third step.

In Chapter 6, we developed the SFB Model and SBF objects. In this chapter, we showed that applications written as networks of SBF objects map indeed very smoothly onto architecture instances, because the model of computation matches the model of architecture. Functional elements use rules similar to the Kahn rules used in SBF objects; therefore, both SFB objects and functional elements describe the same behavior. SBF objects thus map seamlessly onto functional elements. Because an SBF object maps to a functional element, the communication taking place between SBF objects over channels map onto the buffer/router/buffer-tuples, which behave like FIFO buffers. We have showed that this communication results in a 3-level control hierarchy.

The workload that SBF objects describe can change dynamically based on the content of streams. For an architecture instance to describe the correct workload without large modifications to the original application, SBF objects need to execute functionally correctly within the context of an architecture instance. This requires an interface such that SBF objects can obtain data samples from an architecture instance, process them, and put new results back into the architecture instance. We have showed such an interface using function call overloading of the function operator in C++. The function overloading and the fact that SBF objects have a fire-and-exit behavior resulted in a simple interface allowing SBF objects to interact with functional elements.

We showed an example in which ORAS derives an architecture instance that can execute the Picture in Picture application. We showed that ORAS handles a one-to-one mapping and a many-to-one mapping. We also discussed related work on high-level retargetable architecture simulators.

We compared the execution speed of ORAS with other architecture simulators. We found that ORAS is indeed a fast simulator, while still being a retargetable one. Simulators derived by ORAS execute 10,000 SBF object function calls (firings) per second. To process a complete video picture of 720×576 pixels by the PiP application took 9 minutes. The reason why ORAS is fast and retargetable is that it uses building blocks. Architecture instances adhering to the architecture template can be constructed by simply combining building blocks, using high-level programming constructs. Furthermore, building blocks separate structure from behavior. As a consequence, when a process is activated, all references between building blocks are already fixed and the process does not have to decode any aspect of an architecture instance. In addition, the use of high-level primitives such as processes, semaphores and delay allows us to describe the performance of architecture instances at an abstract level, but still cycle-accurately. The process scheduler of the RTL can simulate these high-level primitives very efficiently. The seamless integration of SBF objects onto functional elements introduces very little overhead in the functionally correct execution of an application onto an architecture instance.

We also showed the limitations of ORAS. The main limitation is that it can only execute instances of stream-based, data-driven architectures. It is less suitable for load/store architectures. We indicated in Appendix C that the RTL is less suited for describing low-level control like polling.

Finally, we showed that the three steps used in ORAS provide a generic outline for other retargetable simulators as well. We indicated the changes required in ORAS when a retargetable simulator must be created for another architecture template.

Bibliography

ASPT. *ExSpect 5.0 User Manual*. Eindhoven University of Technology, Eindhoven, the Netherlands, 1994.

Leon Benders. *System Specification and Performance Analysis*. PhD thesis, Eindhoven University of Technology, 1996.

Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, 1992. Special issue on Simulation Software Development.

Rajesh K. Gupta and Stan Y. Liao. Using a programming language for digital system design. *IEEE Design & Test of Computers*, 14(2):72 – 80, 1997.

Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf. The construction of a retargetable simulator for an architecture template. In *Proceedings of 6th Int. Workshop on Hardware/Software Codesign*, Seattle, Washington, 1998.

Sjaak Koot, Arjan Versluys, and Pieter de Visser. *TSS User Manual*. Philips Research Laboratories, Eindhoven, 1997.

Peter Marwedel and Gert Goossens. *Code Generation for Embedded Processors*, chapter 1, pages 13 – 31. Kluwer Academic Publishers, 1995.

David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ programming with standard template library*. Addison-Wesley Professional Computing Series, 1996.

Tom Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California at Berkeley, 1995.

Eric K. Pauer and Jonathan B. Prime. An architectural trade capability using the Ptolemy kernel. On Ptolemy Site http://ptolemy.eecs.berkeley.edu, 1997.

Guido Post, Andrea Muller, and Rainer Schoenen. Object-oriented design of ATM switch hardware in a telecommunication network simulation environment. In *Proceedings of GI/ITG/GMM Workshop Methoden und Bescheibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, Paderborn, 1998.

R.K. Rao and J.J. Hwang. *Techniques and Standards for Image, Video and Audio Coding*. Prentice Hall, 1997.

Mark A. Richards, Anthony Gadient, Geoffrey A. Frank, and Randolph Harr. The RASSP program: Origin, concepts, and status: An introduction to the issue. In *Journal of VLSI Signal Processing*, volume 15, pages 7 – 28, 1997.

Fred Rose and John Shackleton. Performance modeling of system architectures. *VLSI Signal Processing*, 15(1/2):97 – 110, 1997.

F Sijstermans, E.J. Pol, B. Riemens, K Vissers, S. Rathnam, and G. Slavenburg. Design space exploration for future trimedia CPUs. In *ICASSP'98*, 1998.

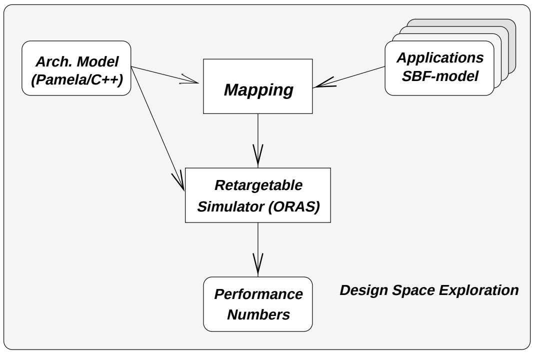
Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.

Vojin Živojnović, Stefan Pees, Christian Schläger, Markus Willems, Rainer Schoenen, and Heinrich Meyr. DSP Processor/Compiler Co-Design: A Quantitative Approach. In *Proc. ISSS*, 1996.

B.R.T.M Witlox, P. van der Wolf, E.H.L Aarts, and W.M.P. van der Aalst. Performance analysis of dataflow architectures using timed coloured petri nets. In *Proceedings of the 19th International Conference on Applications and Theory of Petri Nets*, Lisabon, 1998.

Chapter 8

Design Space Exploration



Contents

DESIGN Space exploration is the subject that we discuss in this chapter. A designer must know what the design space of architectures looks like in order to make trade-offs related to the design. The acquisition of this knowledge requires an exploration of the design space of architectures. In the previous chapter we developed ORAS, which allows us to quantify the performance of an architecture instance. We will now integrate ORAS in a generic Design Space Exploration (DSE) environment. This environment allows us to change parameters that are presented to ORAS in systematical and automated way to generate different architecture instances. In this way, the environment makes it possible for us to explore part of the design space of the architecture template of stream-based dataflow architectures automatically and systematically. At the end of an exploration, the environment presents the relationships between parameter values and performance values as a Response Surface Model (RSM). These RSMs enable designers to consider trade-offs in architectures.

We start in Section 8.1 by explaining what design space exploration is about, and we look at Response Surface Models that express the relationship between parameter values and performance metrics. In Section 8.2, we present four problems that we have to solve in order to perform design space exploration. We make use of a generic design space exploration environment developed at Philips Research to perform actual design space exploration. We describe this environment and we show in

Section 8.3 how the four problems identified are solved within this generic DSE environment. Given the generic DSE environment, we show in Section 8.4 how we integrate ORAS in this environment and we explain how to use the environment to perform design space exploration. In Section 8.5, we put the exploration environment to work for the Picture in Picture application. Finally, in Section 8.6 we discuss related work in design space exploration.

8.1 The Acquisition of Insight

Designers need to design architectures for new products that are becoming multi-functional, multi-standard, or both, and the systems that they create are becoming increasingly programmable. When designing such systems, designers are given the performance that the system is required to exhibit, i.e., the design objectives such as response times, throughput, and utilization (see the design objectives in Table 2.2). The actual performances that can be achieved by a system depends on the composition of the architectural elements which make up an architecture instance as well as the set of applications that need to execute on the architecture instance.

Given a set of parameter values, a designer derives an architecture instance from the architecture template and evaluates this architecture instance using a Y-chart environment to obtain the corresponding performance values as shown in Figure 8.1. However, doing this does not solve the designer's main problem: he or she obtains performance metric values, not design parameter values. Somehow, the designer must find the appropriate parameter values to find feasible designs. Hence, the designer needs to find the *inverse transformation* from the performance back to the parameters as represented by the "lightbulb" in Figure 8.1.

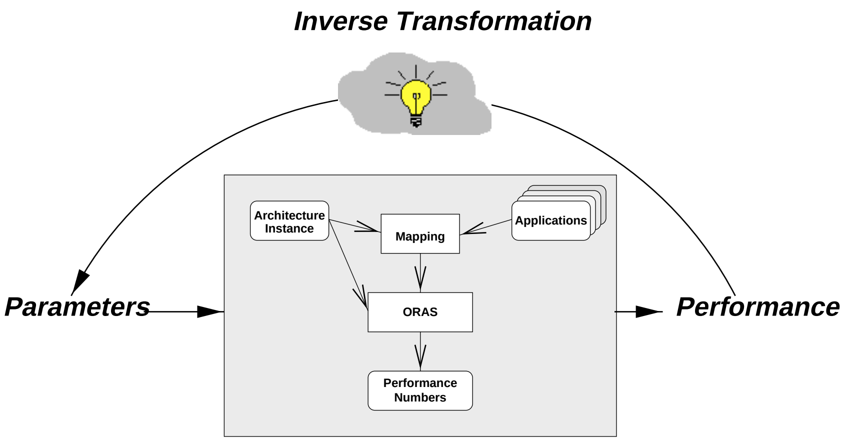


Figure 8.1. The *inverse transformation* from the performance back to the parameters, as represented by the "lightbulb".

Unfortunately, in general this "inverse transformation" can not be computed analytically and is therefore difficult to obtain. Because establishing the inverse transform is so difficult, designers cannot simple point to a desirable performance metric, thereby identifying a suitable set of parameter values, or "vary" a performance metric and see the effect on other performances.

The only thing the designer can do is propose a set of parameter values of what he or she hopes will satisfy the required performance metric of the system. The actual performance metric of the system is computed for this set of parameter values and if there is any discrepancy between computed and required performances, the designer then suggests a modified set of parameter values. Designers repeat this cycle, often many times, before they find a suitable design, a process Spence [1974] refers to as the *Acquisition of Insight*.

We already discussed this process in Chapter 3, where we explained what exploration of the design space implies. Instead of having a designer select parameter values, calculate the performance of an architecture instance, and then make the appropriate changes, we select parameter values systematically such that we obtain the finite set of points $\mathcal{I}$ = $\{I_0, I_1, \ldots I_n\}$ where each point I consists of a set of parameters values $I = (p_0, p_1, \ldots, p_x)$. Each point I leads to a different architecture instance, for which performance numbers are obtained using the Y-chart approach.

When we plot the obtained parameter numbers for each architecture instance versus the set of systematically changed parameter values, we obtain graphs as shown in Chapter 1 in Figure 1.5. This simple graph is generally called a *Response Surface Model* (RSM) [Friedman, 1991a]. However, unlike the graph shown in Figure 1.5, RSMs normally describe multi-variable relationships. Using the RSM, we are able to locate knee points, which represent the best trade-off between particular performances and parameter values.

8.2 Design Space Exploration

We want to create an environment permitting us to explore the design space of stream-based dataflow architectures in a systematic and automatic way. At the heart of such an environment, there must be a model that links parameter values to performance numbers. A Y-chart environment does exactly this: it links parameter values used to define an architecture instance to performance numbers as shown in Figure 8.1. The Y-chart environment is realized by means of ORAS, which was developed in Chapter 7. ORAS accepts an architecture description and derives a simulator from this description which, when executed, delivers the performance numbers for the architecture instance. Consequently, in a DSE environment we use ORAS to link parameter values to performance numbers.

If we want to automate the process of performing design space exploration and the construction of RSMs, we encounter the following problems:

Spanning the Design Space In the exploration of stream-based dataflow architectures, we need to select a finite set of points $\mathcal{I}$ that span the design space. The first problem is how to select this finite set of points. Furthermore, since the evaluation of a particular architecture instance takes minutes to hours (recall it took the Picture in Picture application presented in Section 7.6 nine minutes to process one video frame), we should select $\mathcal{I}$ such that it contains the minimum amount of points required to correctly span the design space.

Construction of the Response Surface Model After iterating many times in the Y-chart, we end up with a collection of performance numbers for specific performance metrics and parameter values. The problem is that we need to have some kind of model to express the relationship between them. These expressions result in RSMs.

Data Management and Consistency In each iteration, for particular parameter settings ORAS generates large amounts of performance numbers for selected performance metrics. The problem is that we need to manage this large amount of data and keep the performance numbers consistent with the selected parameter values.

Parameterize the Architecture Description For each tuple of parameter values $I = (p_0, p_1, \ldots, p_x)$, the problem is that we need to be able to derive an architecture description that ORAS accepts.

8.3 Design Space Exploration Environment

For the construction of the DSE environment, we have relied on the generic DSE environment developed by Bingley and van der Linden [1994] at Philips Research. This environment combines a design-data management tool with a set of tools for performing dynamic and statistical analysis of the collected data. The generic DSE environment assumes that there is a black box that takes in a file containing a list of parameter values and subsequently delivers a file with performance numbers for specific performance metrics, as shown in Figure 8.2.

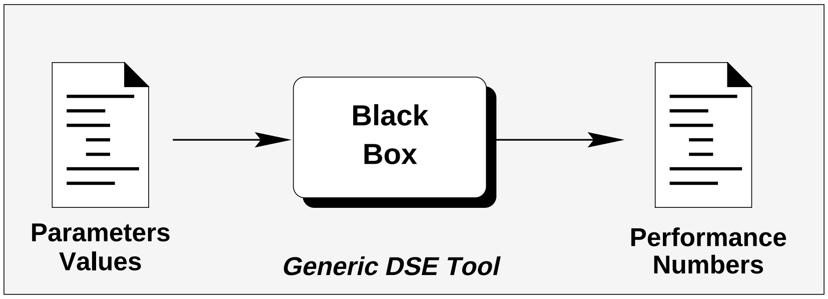


Figure 8.2. The generic DSE environment.

To solve the first two problems mentioned in the previous section, i.e., spanning the design space and the construction of the RSMs, Bingley and van der Linden use mathematical theories developed within a specific area of statistics known as the *Design of Experiments* (DOE). We now introduce the main concepts underlying DOE, without going into much detail. We explain the techniques used in the generic DSE environment and review their relevance.

8.3.1 Spanning the Design Space

We already explained in Chapter 3 that the design space can be explored by selecting distinct values within the range of values of each parameter systematically for all parameters p in the parameter set P. This results in the finite set $\mathcal{I}$, which spans (part of) the design space; we say set that $\mathcal{I}$ represents an *experiment*. Careful planning of an experiment can save a great deal of time and effort, as we will show.

If performance depends on just one parameter, the strategy used in an experiment is simple: measure the performance at several different values of the parameter and fit a curve to the results. However, if there is more than one parameter involved, then the situation is somewhat more complicated. Traditionally, what is done is based on the assumption that the parameters are independent variables, so that they can be taken into account one at a time. This approach does work, i.e., one gets an idea how the system responds to changes in the parameters, but it is not a reliable way to go about the investigation, because the parameters are not really independent of each other.

Design Of Experiments (DOE)

A better way to study the effects of parameter changes on a system is called *Design of Experiments* (DOE) or statistical experimental design. It is largely the result of work begun by an English statistician, Sir Ronald Fisher, in the 1920s and 1930s. He worked at an agricultural experimental station, and his job was to aid in the study of different fertilizers and such on the yields of different crops. He introduced and advanced the important concept of *factorial design* [Logothetis and Wynn, 1989]. The statistical techniques that he developed have been expanded, mainly due to Taguchi, and are in use in many different industries.

In DOE, an experiment consists of a number of *trial runs*. In each trial run the relationships between parameters, called *factors* in DOE parlance, and the response from a model are determined. A *reliable experiment* is obtained when the response of a model can be determined accurately for two factors even though other factors vary [Logothetis and Wynn, 1989].

Full Factorial Experiment

To obtain a reliable experiment, a model should be evaluated for distinct combinations of parameters in the set P, which determines k factors. These k factors have ranges of conceded values and by selecting l distinct values, named *levels* in DOE parlance, for each of them one achieves a *full factorial experiment*. Such a reliable experiment would thus requires l^k trial runs. The geometric interpretation of a full factorial experiment is that the k-dimensional design space is uniformly filled with l^k points.

As an example, suppose we want to set up a full factorial experiment in which 7 parameters are changed at 2 distinct levels. This would lead to

$$2^7 = 128 \text{ trial runs.} \tag{8.1}$$

The problem with a full factorial experiment is that the number of trial runs grows exponentially with the number of factors. Since each trial run corresponds with one evaluation round in the Y-chart, which takes minutes to hours of simulation time, the evaluating time of the experiment quickly becomes exorbitantly large, making the method unsuitable in practice.

Fractional Factorial Experiment

Instead, a *fractional factorial experiment* can be performed, in which statistically techniques are used to remove redundant trial runs. The result is a reliable experiment with almost the same amount of information as with a full factorial experiment, but with fewer trial runs being required [Logothetis and Wynn, 1989].

In a fractional factorial experiment, a certain fractional subset of the full factorial set of experiments is carefully selected such that orthogonality is maintained among the various factors. Orthogonality allows estimation of the average effects of factors without the risk that the results are being distorted by effects of other factors [Lochner and Matar, 1990]. Orthogonality thus allows for independent estimation of responses from factors of the entire set of trial runs, resulting in reliable experiments while dramatically reducing effort, expense, and time compared to full factorial experiments.

The fractional orthogonal subsets are derived using *orthogonal arrays* [Bush, 1952] that describe basic combinatorial structures. In orthogonal arrays, the columns are mutually orthogonal; that is, for any pair of columns, all combinations of levels occur, and they occur an equal number of times. Therefore, orthogonal arrays provide a method for selecting an intelligent subset of the parameter, significantly reducing the number of trial runs required within an experiment.

Using orthogonal array $OA_8(2^7)$ [Owen, 1991], we repeat the full factorial experiment in which we studied the 7 factors at 2-level as a fractional factorial experiment requiring only 8 trials. This is $\frac{1}{16}$th-fractional design of the full factorial design normally requiring 128 trials. This shows that a fractional factorial experiment indeed dramatically reduces the number of experiments, and thus evaluation time, allowing us to evaluate more architecture designs in less time.

Spanning the Design Space

We use orthogonal arrays to span the design space and we select l distinct values in the range of parameter values for all k parameters. This results in the experiment given by set $\mathcal{I}$, where each point I is a trial run.

Notice that we said that the set of parameters P determines k factors. The set P, however, often contains too many parameters. We can limit the number of parameters either by sampling the space coarsely, i.e., carrying out fewer trial runs in an experiment, or by reducing the number of parameters that we vary, i.e., using a subset of P. This subset determine the k factors.

8.3.2 Construction of the Response Surface Model

Given the performance numbers gathered from an experiment, we must establish the relationship between parameters and the obtained performances; thus we need some kind of model. One model, which is based on the assumption that the relationship is a function f, approximates this function as $\hat{f}$.

The function f can rely on many variables, depending on the number of factors k used in the experiment. Thus we must relate the response variable y, a performance metric, to one or more distinct values of the factors $x_0, x_1, \ldots, x_k$ that span the domain S. The result is the approximation $\hat{f}$, which expresses the relationship between parameter values and performance and which represents the RSM we want to obtain

$$y = \hat{f}(x_0, x_1, \ldots, x_k). \tag{8.2}$$

Multivariate Adaptive Regression Splines

To construct $\hat{f}$, Bingley and van der Linden used a multivariate regression modeling technique, in particular, *Multivariate Adaptive Regression Splines* (MARS) developed by Friedman [1991a]. MARS uses two techniques to construct $\hat{f}$: recursive partitioning and spline fitting. The basic rationale behind MARS is to approximate f with several simple functions, each defined over a different subsection of domain S. These simple functions are qth order splines that are described as

$$(x - t)_+^q = \begin{cases} 0, & \text{if } x - t < 0 \\ (x - t)^q, & \text{if } x - t \geq 0 \end{cases}. \tag{8.3}$$

The splines are truncated such that the function returns either the value of zero for negative arguments or the spline value $(x - t)^q$. In the DSE environment, only first order splines (i.e., $q = 1$), also known as *basic splines*, are used. A basic spline is zero for negative values, and it behaves as a straight line otherwise. The basic spline can have an offset t, called the *knot*, as show in Equation 8.3. Using a set of these knots, MARS divides the domain S up into several subsections. A picture of a basic spline is shown in Figure 8.3.

The objective of MARS is to divide domain S recursively into a good set of subsections by selecting a set of knots v dynamically. At the same time, for each subsection it also finds the coefficient of the basic spline that makes it possible to derive a good approximation of f in that subsection. The model for $\hat{f}$ resulting from using MARS [Friedman, 1991a] is

$$\hat{f}(x_0, x_1, \ldots, x_k) = a_0 + \sum_{m=1}^{M} a_m \prod_{l=1}^{L} [s_{lm} . (x_{v(l,m)} - t_{lm}]_+, \tag{8.4}$$

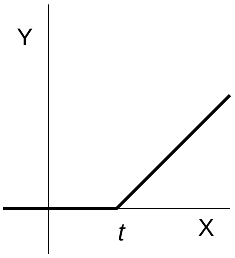


Figure 8.3. A basic spline.

which is a summation of M partial products of L functions (i.e., basic splines) each expressed in one particular factor k to account for univariate functions $f(x_i)$ (i.e., l=1), bivariate functions $f(x_i, x_j)$ (i.e. l=2), or, in general, multi-variate functions (i.e. $1 \leq l \leq L$). The model also uses the function s that is either equal to +1 or -1. Thus, the function $\hat{f}$ that is obtained is a *piecewise linear model* expressed in the parameters which were changed in the experiment, i.e., the k factors. Within the generic DSE environment the cross product is limited to be of maximal order $L_{max} = 2$, i.e., bivariate functions.

To specify a RSM, which describes the relationship between parameters and a particular performance metric, we assume that the relationship is a function f, which we approximate as $\hat{f}$. We are, however, thoroughly aware that the relationship we want to describe does not need to be a function at all, but might be some wildly varying relationship. Therefore, when inspecting approximations like $\hat{f}$, we should be careful when drawing conclusions. Nevertheless, the assumption that the relation is a function has proven to be very useful for the cases we have considered so far.

8.3.3 Data Management and Data Consistency

During DSE, we produce enormous amounts of data and we need to manage the consistencies between all this data. For each trial run of an experiment, performance numbers are created for performance metrics for a particular set of parameter values. If we had used a simple file system, we would not have preserved this kind of relationship. The generic DSE environment is therefore built on top of the *Nelsis* design data management tool developed by ten Bosch et al. [1991] at Delft University of Technology. The Nelsis system provides for the simple encapsulation and integration of design tools. It manages the large amount of data produced by the encapsulated tools and it preserves the relationships between the data offered to and produced by these tools.

Nelsis also has a visual interface as shown in Figure 8.4. The boxes in the figure represent tools and the directed lines between boxes represent relationships between tools. The boxes together with the lines define a particular *design flow* and the case shown by Figure 8.4 represents the design flow of the generic DSE environment. We explain the tool that each box represents in detail in Section 8.4.

The oval balloons connected to the lines represent design data that is exchanged between tools and managed by Nelsis. In Nelsis, a box can only execute when all its relationships with other boxes have been satisfied. Nelsis uses different colors for boxes and different thicknesses for lines to visualize the status of a box, including whether it can execute or not.

8.3.4 Parameterizing the Architecture Description

Given a set of parameter values, we need to create different textual architecture descriptions. We use the versatile scripting language *Perl* [Wall and Schwartz, 1992] for that purpose. This very powerful

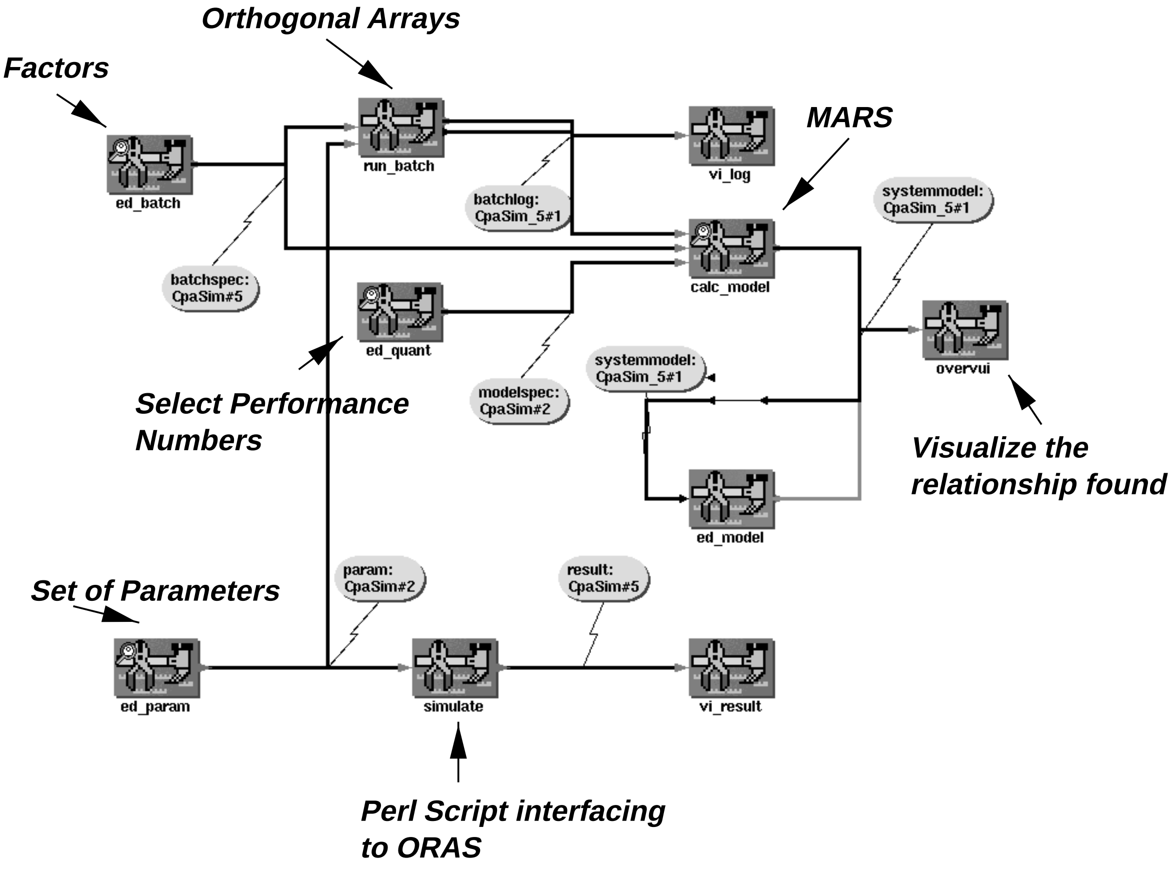


Figure 8.4. The design space exploration flow in Nelsis.

scripting language provides high-level programming constructs like *associative arrays*, making the construction of different textual architecture descriptions very simple.

Associative Arrays

Suppose that we have a set of parameter values as given in Table 8.1 (how we obtain this table will be explained in Section 8.4). In this table, each parameter has a *name* (column *Parameter Name*) and has adopted a particular *value* that is either a numerical value or a string from an enumerated set (column *Selected Value*). Perl establishes a link between the name of a parameter and its value, using the associative variable `$var` as follows;

```
$var{'name'} = value
```

When we write an architecture description we use associative arrays to retrieve values from a set of parameters as given in Table 8.1. The description of a processor element as part of the architecture description is given in Program 8.1. This description is based on the architecture description given in Figure 5.11, but we replaced all parameterized options with associative arrays `$var`. We did not replace the options for the type of buffers used. This shows that not every option needs to be parameterized; some can have default values. Because this option is fixed in Program 8.1, it does not appear in the set of parameters given in Table 8.1. The name between brackets indicates which parameter we will include in the final description of the architecture.

Parameter Name	Selected Value	Range
buffersize	50	1 – 100
router	FCFS	{ FCFS, TDM, RoundRobin }
switching	Packet	{ Packet, Sample }
function1	LowPass	{ HighPass, LowPass, Filter }
func1_initiation_period	1	1 – 10
func1_latency	18	1 – 20

Table 8.1. Set of Parameters.

Program 8.1. An Example of the integration of Perl Variables in an Architecture Description

```
  ProcessingElement Filter(2,2) {
      InputBuffer { Type: BoundedFifo(  $var { ’bufferSize’ }  );   }
      OutputBuffer { Type: BoundedFifo(  $var { ’bufferSize’ } );   }
      Router { Type:  $var { ’router’ }; }
      FunctionalUnit {
         Type:  $var { ’switching’ };
         FunctionalElement LowPass(1,1) {
            Function { Type:  $var { ’function1’ } (initiation_period= $var { ’func1_initiation_period’ },
    latency= $var { ’func1_latency’ };   }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
               }
            }
      }

}
```

An architecture description has three different levels of parameterization, as explained in Section 5.5.2. The first level represents structural parameters of an architecture instance (e.g., the number of processing elements used). The second level represents behavioral parameters of an architectural element type (e.g., select a bounded FIFO buffer behavior or unbounded FIFO buffer behavior). The third level represents functional parameters for architectural elements (e.g., the size of a FIFO buffer, throughput and latency values, or an array of filter coefficients).

In the example given in Program 8.1, we included two levels of parameterization, respectively the second level (e.g., `$var{‘router’}`, `$var{‘switching’}`, and `$var{‘function1’}`) and the third level (e.g., `$var{‘bufferSize’}`, `$var{‘func1_initiation_period’}`, and `$var{‘func1_latency’}`).

Program 8.2. Level one parameterization in Architecture Descriptions

```
for ($i=1, $i<  $var{’number_of_pes’} , $i++ )
       { include the code shown in Program 8.1 }
}
```

We cannot use only associative arrays to describe the first level of parameterization. We require control statements like `for` loops and `if-then-else` statements as well. These requirements do not pose a problem, because Perl is a fully-fledged scripting language and hence provides these statements. In Program 8.2, we show, for example, how we parameterized the number of processing elements present (`$var{'number_of_pes'}`) in an architecture description.

8.4 Integrating ORAS within the Generic DSE Environment

The design flow used in the generic DSE environment is depicted in Figure 8.4. It consists of three different parts. In one part, the environment selects parameter values representing different points in the design space of stream-based dataflow architectures. In the second part, the environment executes the Y-chart using ORAS for architecture instances represented by the parameter values selected. In the third part, the environment creates RSMs. We discuss the three parts and explain which tool is encapsulated by the various boxes shown in Figure 8.4.

8.4.1 Selecting Parameter Values

In box ed_param we define the set P of parameters in which each parameter p is given a default value. In box ed_batch, we define the k factors that we want to use in an experiment. We define a range of values for each factor. In this manner, we obtain the table given in Table 8.1. In ed_batch we also indicate how many trial runs, given by x, an experiment should consist of.

Then tool run_batch selects the appropriate orthogonal array based on the number of factors k and the number of trial runs x. It selects the appropriate orthogonal array from a standardized family of orthogonal arrays that is generated using software developed by Owen [1994]. Then, based on the selected orthogonal array, run_batch sets up an experiment by selecting values for each factor within its range. The experiment is the set $\mathcal{I} = \{I_0, I_1, \ldots I_x\}$. In this set each $I = (p_0, p_1, \ldots, p_n)$. The n parameters contain $n - k$ default values and k values selected by run_batch. Examples of selected values are given in Table 8.1.

8.4.2 Running the Y-chart in the DSE Environment

The generic DSE environment considers the simulate box to be a black box as depicted in 8.2. The DSE environment provides the simulate box with a file containing a set of parameter values representing point I. It will do this for each element of set $\mathcal{I}$. The file containing n parameter values represents a particular architecture instance for which we want to determine performance numbers using a Y-chart environment. The box simulate should produce a file containing the resulting performance numbers.

A Perl script runs inside the simulate box, as illustrated in Figure 8.5. We again see the file containing the parameter values and the file containing the resulting performance numbers. The Perl script that is run is as given in Program 8.3. It starts by generating an architecture description from the list of parameters using associate arrays shown in Program 8.1 and Program 8.2. Then, the Perl script generates a routing program (see, for example, Table 5.5) for the architecture description, for each application in the set.

If the performance of the application does not depend on the content of the stream that it processes, then the Perl script runs ORAS for each application only once. If the performance of an application does depend on the content of the stream (see, for example, Figure 7.7), then the Perl script runs ORAS for each application with different streams containing other content, e.g., another video sequence.

When all applications of the set have executed, the Perl script collects all performance numbers of the various runs and combines them into one file, performing post-processing. This is required to obtain performance numbers for not merely one application, but the whole set of applications. As

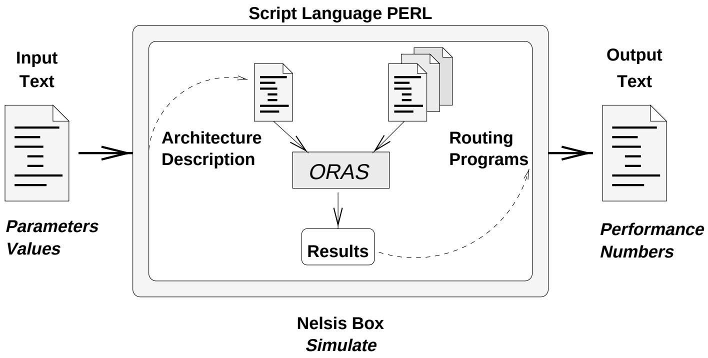


Figure 8.5. The realization of the simulate box, which interacts with the generic DSE environment using an input file containing parameter values and an output file containing performance numbers.

indicated in Chapter 3, this is essential to obtaining architecture instances that are able to execute a set of applications and obey set-wide design objectives.

Lastly, the Perl script writes out the performance numbers in the format of which an excerpt is shown in Table 7.2.

Program 8.3. PERL SCRIPT USED WITH THE GENERIC DSE ENVIRONMENT

```
read File containing parameter values representing an architecture instance
create the correct architecture description for the architecture instance
foreach application in the set
   create correct routing program
   if application is a dynamic applications,
      foreach stream with different content
         execute ORAS using the routing program and architecture description
   else
      execute ORAS using the routing program and architecture description
   endif combine found performance numbers
perform post processing if need
write File containing performance numbers
```

8.4.3 Creating Response Surface Model

All the performance numbers that are generated by the simulate box are stored in a database which is part of Nelsis. This database contains the performance numbers as well as the parameter values used to obtain these performance numbers. In box ed_quant we can select a few performance metrics, for which box calc_model constructs the approximation $\hat{f}$ using the MARS model as given in Equation 8.4. Tool calc_model extracts the parameter values $(p_0, p_1, \ldots, p_n)$ from the database as well as the performance number y measured for the selected performance metric. Calc_model uses the software developed by Friedman [1991b] to construct the approximation $\hat{f}$.

At last, the overvui box visualizes the approximation $\hat{f}$ as a 3-dimensional figure, as shown in Figure 8.6(a) and Figure 8.6(b). Notice that these 3-dimensional figures represent a k-dimensional function. The other $k - 2$ parameters are thus considered to have fixed values.

8.5 An Example of Design Space Exploration

All ingredients are now available to perform a design space exploration of the stream-based dataflow architecture. As an example, we look into a design trade-off of the stream-based dataflow architecture. Although the architecture consists of processing elements operating in parallel, it uses a single centralized controller, the global controller. This controller might potentially become a bottleneck in the architecture.

Per packet, a router issues a request to the global controller. The longer the packets, the fewer requests a router issues in a given amount of time. The number of requests that the global controller can handle in a given time depends on the service time of the controller. We now want to investigate the effect the *packet length* and the *service time* of the global controller have on the level of *parallelism* achieved in an architecture, using the DSE environment developed in this chapter.

The architecture and the Picture in Picture application that we study in this exploration were already discussed in Section 7.6. We set up an experiment which consists of 25 trial runs and we have two factors: the parameter *packet length* with the range of $\{5..200\}$ samples and the parameter *service time* with the range of $\{1..20\}$ cycles. In each trial run, we construct a different architecture instance and we measure the performance metric *achieved parallelism* and *utilization* of the global controller for that instance. We measure the performance while an architecture instance executes the Picture in Picture application for two small video frames consisting of 14,400 video samples per frame.

At the end of the experiment, we establish the RSM $\hat{f}_p$ for achieved parallelism;

$$\text{Achieved Parallelism} = \hat{f}_p(\text{packet\_length}, \text{service\_time}), \tag{8.5}$$

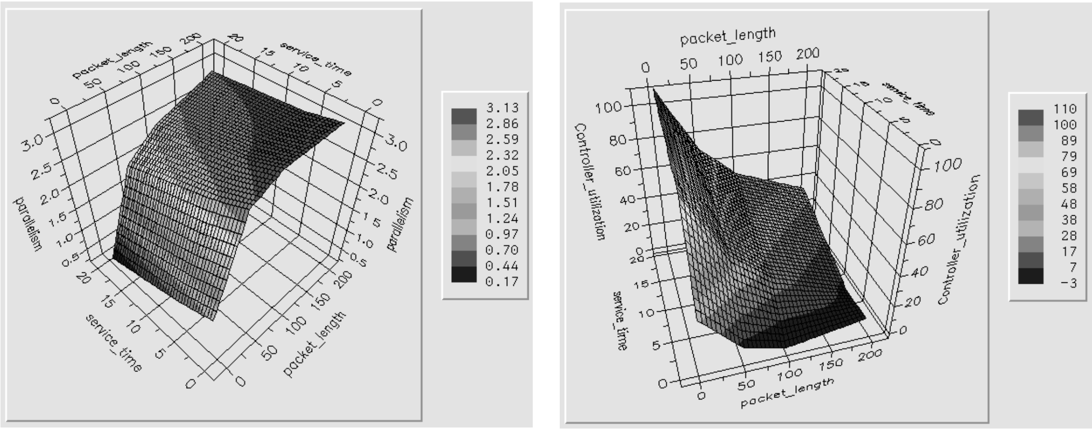
and we establish the RSM $\hat{f}_u$ for the utilization of the global controller;

$$\text{Utilization global controller} = \hat{f}_u(\text{packet\_length}, \text{service\_time}). \tag{8.6}$$

In Figure 8.6(a) we visualize the Response Surface Model $\hat{f}_p$ and in Figure 8.6(b) we visualize the Response Surface Model $\hat{f}_u$. Moreover, we show the piecewise linear approximation function $\hat{f}_p$ in Figure 8.7. This piecewise linear function consists of four univariate functions and two bivariate functions. In this function, the PP represents the truncated basic spline function given in Equation 8.3 for q equal to one.

Looking at the RSMs, we conclude that the service time of the global controller indeed has a significant influence on the level of parallelism acquired. At points where the parallelism is low, we notice that the architecture uses the global controller for almost 100%, indicating that the global controller is indeed the bottleneck in those cases.

A knee in Figure 8.6(a) indicates that a trade-off is present. From an engineering point of view, such knees are of interest, as was explained in Chapter 1. In Figure 8.6(a) a knee is present between packet length and service time. For a very fast global controller, with a service time of, say, five cycles, the knee value is close to a packet length of 80 samples. If we select a parameter value at the left-hand side of the knee (i.e., a value smaller than 80 samples), parallelism drops, whereas if we select a parameter value at the right-hand side (i.e., a value larger than 80 samples) parallelism hardly improves. Moreover, the longer packets influence the flexibility of the architecture in a negative way. A functional unit can switch less often between streams in time when longer packets are used. Consequently, a designer should select values close to the knee point of a packet length of 80 samples.



(a) Achieved Parallelism in Operations per Cycle for Packet Length versus Service Time

(b) Utilization of the global controller in Percentage for Packet Length versus Service Time

Figure 8.6. (a) the Response Surface Model for achieved parallelism and (b) the Response Surface Model for the utilization of the global controller.

```
8.786
+ 1.142*PP((packet_length/1.0E+02-.7000))
- 10.47*PP(-(packet_length/1.0E+02-.7000))
- 2.541*PP((packet_length/1.0E+02-1.350))
- 4.127*PP((service_time/1.0E+01-.5000))
+ 2.905*PP((packet_length/1.0E+02-.7000))*PP((service_time/1.0E+01-.3000))
+ 6.124*PP(-(packet_length/1.0E+02-.7000))*PP((service_time/1.0E+01-.9000))
```

Figure 8.7. The piecewise approximated function for achieved parallelism. In the function, the PP represents the truncated basic spline function.

The pictures presented in Figure 8.6(a) and Figure 8.6(b) show approximations based on a few data points and therefore should be interpreted with care. In Figure 8.8, we show the utilization of the global controller again for different packet lengths and for various service times. In this figure, we see artifacts of the approximation for the utilization of the global controller: values are produced that are larger than 100% and smaller than 0%. Furthermore, the RSMs need not be as smooth as they appear to be in the figures. It is just as possible that the utilization is not even a function at all. To increase the accuracy of the value that is expected to be optimal, at least one new experiment should be performed with parameter values in the range closer to the selected values (e.g., the knee point).

8.6 Related Work

The generic DSE environment presented in this chapter gives us the opportunity to compare many different architecture instances automatically and systematically, which makes it possible for us to

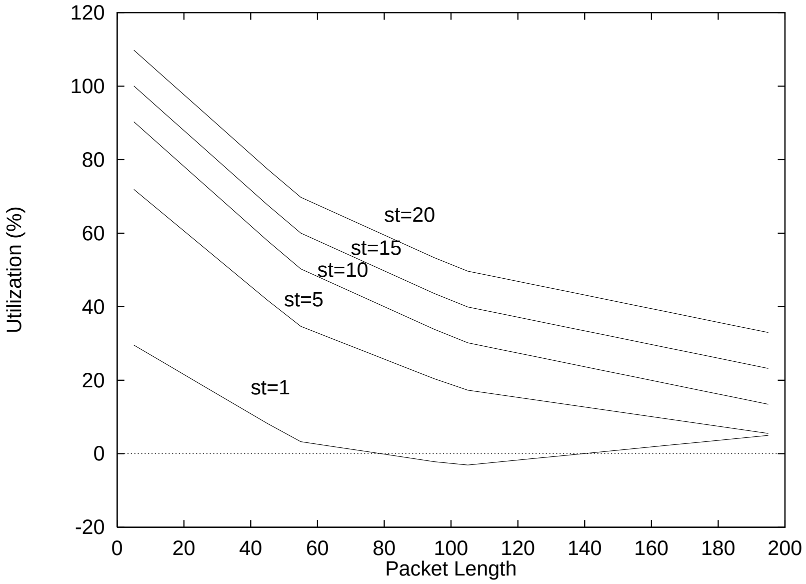


Figure 8.8. Utilization of the global controller as piecewise approximated function in Percentage versus Packet Length for various Service Times (st).

make good trade-offs to optimize a complete system. Currently, we optimize a system by investigating the obtained results visually. In the example previously presented, we only changed two parameters: packet length and service time. Normally, however, many (e.g., 5 to 10) parameters are changed, and finding some kind of optimum is then a very difficult multi-variable problem. Simple 3-D graphs as shown in Figure 8.6(a) and Figure 8.6(b) do not help us very much since they present a relationship under the assumption that other parameters are kept at a specific fixed value. In that manner, they do not reveal the relationship between all parameters at once.

Instead of optimizing using visual techniques, we can resort to standard analytical optimization techniques for multi-variables. Nevertheless, these techniques assume a smooth function, an assumption that is not applicable very often.

The presentation of multi-dimensional performance numbers is a problem in itself. This visualization problem is discussed in Pancake et al. [1995], as are interesting representation techniques. One of these techniques is a *Scatter-plot Matrix* [Reed et al., 1995], which contains an x-y scatter-plot for each parameter/performance metric pair. When the technique of *graphical brushing* is used within such a scatter-plot matrix, a cluster of points is brushed and all brushed points are automatically highlighted in all other scatter-plots. By modifying the brushing of clusters of points, one can observe interactively the effect that the clusters of points have on other performances.

Spence et al. [1995] have come up with many innovative ways of looking at optimization of systems and making effective trade-offs based on only visual representations: from designing robust electric circuits up to selecting houses from a database. Using special visualization techniques like the *Interactive Histogram* [Tweedie et al., 1994], the *Influence Explorer* [Tweedie et al., 1996] and the *Prosection Matrix* [Dawkes et al., 1995], they are able to present the multi-variable optimization problem in an intuitive way. These techniques reconstruct the RSM without assuming a function as model, as we did when using MARS. Furthermore, they span the design space by randomly picking points from the design space, instead of using statistical techniques as we have done. They assume that when they pick enough points (i.e., in the range of thousands of points), they obtain a good filling of the design space. This involves many iterations in the Y-chart, each of which takes minutes to hours to evaluate. The total evaluation time becomes exorbitantly large, making the present techniques unattractive.

Teich et al. [1997] use *Genetic Algorithms* to optimize architectures. This approach is more robust since it does not demand smooth functions, as analytical techniques do. Also, the view presented in Figure 8.2 makes the optimization of architectures directly applicable for genetic algorithms. The set of parameters matches directly with the notion of the "gene", the basic entity in genetic algorithms. However, genetic algorithms require many evaluations in order to reach an optimum. Since each iteration in the Y-chart takes minutes to hours, the evaluation time becomes prohibitively large, making genetic algorithms unattractive.

8.7 Conclusions

In this chapter we have showed how we perform design space exploration of stream-based dataflow architectures systematically and automatically, using a Y-chart environment and a generic DSE environment. We explained that design space exploration involves finding the inverse transformation from the performance back to the parameters. Because this inverse is in general difficult to determine, we construct response surface models to help us do it. We used a generic DSE environment to perform design space exploration.

To perform design space exploration, we had to carry out the following four steps: select parameter values in the design space efficiently, construct the Response Surface Models, manage the enormous amounts of data produced in running an experiment, and describe architecture descriptions in a parameterized way.

For the selection of parameter values in the design space, the generic DSE environment sets up an experiment as a fractional factorial experiment. It employs orthogonal arrays to span a design space using a minimal amount of trial runs; an important step since each execution of ORAS might take minutes to hours.

The generic DSE environment uses a multivariate regression modeling technique called Multivariate Adaptive Regression Splines (MARS), to construct the RSMs. The generic DSE environment uses MARS to approximate the relationship between parameter values and a specific performance metric by $\hat{f}$. This environment can visualize $\hat{f}$, representing the RSM.

So that it can manage the large amount of data produced in experiments, the generic DSE environment is built on top of the Nelsis design data management tool. One task of this tool is to manage the storage and retrieval of data; its main function, however, is to manage the relationship between data offered to and produced by tools.

We used the versatile scripting language Perl to describe architecture descriptions in a parameterized way. This very powerful scripting language provides high-level programming constructs like associative arrays, making the construction of different textual architecture descriptions very simple. Moreover, it acts like a UNIX shell script, which we use to activate various steps needed to evaluate an architecture instance for a set of applications using ORAS.

Using the generic DSE environment, we showed an example of design space exploration. For the Picture in Picture application, we set up an experiment in which we constructed 25 different architecture instances and measured the achieved parallelism in the architecture instance as well as the utilization of the global controller. We performed the exploration and showed the RSM $\hat{f}_p$ for achieved parallelism and the RSM $\hat{f}_u$ for the utilization of the global controller. The exploration showed that the amount of achieved parallelism in the architecture instance very much depends on the relationship between the service time of the global controller and the packet length. In $\hat{f}_p$, we identified a knee that represents a trade-off between the service time and packet length.

The RSMs presented are approximations. We showed the artifacts resulting from this approximation and indicated clearly that these Response Surface Models need to be interpreted with care.

As last, we discussed related work on design space exploration.

Bibliography

Peter Bingley and Wim van der Linden. Application of framework technology in system simulation environments. In *Proceedings of the Seminar 'Database Systems and Applications for the Nineties'*. Delft University of Technology, 1994.

K.A. Bush. Orthogonal arrays of index unity. *Annals of Mathematical Statistics*, 23:426 – 434, 1952.

Huw Dawkes, Lisa Tweedie, and Bob Spence. VICKI - the visualisation construction kit. In *Advanced Visual Interfaces*, Gubbio, Italy, 1995.

Jerome H. Friedman. Multivariate adaptive regression splines. *Annals of Statistics*, 19(1):1 – 141, 1991a.

Jerome H. Friedman. Multivariate adaptive regression splines (MARS modeling) software. Software can be obtained from http://stat.stanford.edu/software/software.html, 1991b. Version 3.5.

Robert H. Lochner and Joseph E. Matar. *Designing for Quality: An introduction to the best of Taguchi and western methods of statistical experimental design*. Chapman and Hall, 1990.

N. Logothetis and H.P. Wynn. *Quality through design; experimental design, off-line quality control and Taguchi's contributions*, volume 7 of *Oxford series on advanced manufacturing*. Clarendon, Oxford, 1989.

Art Owen. Orthogonal arrays for computer experiments, integration, and visualization. Technical report, Dept. of Statistics, Stanford University, 1991.

Art Owen. Orthogonal array design software. Software can be obtained from: http://stat.stanford.edu/software/software.html, 1994.

Cherri M. Pancake, Margaret L. Simmons, and Jerry C. Yan. Guest editors' introduction: Performance evaluation tools for parallel and distributed systems. *IEEE Computer*, 28(11):16 – 20, 1995. Theme Feature.

Daniel A. Reed, Keith A. Shields, Will H. Scullin, Luis F. Tavera, and Christopher L. Elford. Virtual reality and parallel systems performance analysis. *IEEE Computer*, 28(11):57 – 67, 1995.

R. Spence. MINNIE, a new direction in circuit design. *Electronics Weekly*, 1974. Issue 3.

R. Spence, L. Tweedie, H. Dawkes, and H Su. Visualisation for functional design. In *Proceedings Information Visualization'95*, pages 4 – 10, 1995.

J. Teich, T. Blickle, and L. Thiele. An evolutionary approach to system-level synthesis. In *Proceedings of the Fifth Int. Workshop on Hardware/Software Codesign*, pages 167 – 171, 1997.

K.O. ten Bosch, P. Bingley, and P. van der Wolf. Design flow management in the NELSIS CAD framework. In *Proc. 28th ACM/IEEE Design Automation Conference*, pages 711–716, San Francisco, 1991.

Lisa Tweedie, Robert Spence, Huw Dawkes, and Hua Su. Externalising abstract mathematical models. In *Proceedings of CHI'96*, Vancouver, Canada, 1996. ACM Press.

Lisa Tweedie, Robert Spence, David Williams, and Ravinder Bhoghal. The attribute explorer. In *Video Proceedings and Conference Companion, CHI'94*, pages 435 – 436, Boston, 1994. ACM Press.

Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 1992.

Chapter 9

Design Cases

Contents

THE Y-chart environment described in this thesis was used in two projects for the design of programmable architectures. In this chapter we describe these two projects and present results. One design case we consider is the *Prophid* video-processor architecture [Leijten et al., 1997; Leijten, 1998] for high-performance video applications. The other design case is the *Jacobium* processor architecture [Rijpkema et al., 1997] for array signal processing applications.

In Section 9.1, we illustrate the emphasis of each of the design projects: In the Prophid case it is on architecture modeling, whereas in the Jacobium case it is on application modeling. In Section 9.2, we explain the Prophid architecture and the benchmark application. We consider the Y-chart environment and the consequence it has for describing instances of the Prophid architecture and the benchmark. Following this, we present a design space exploration of the Prophid architecture executing the benchmark and we draw some conclusions. In Section 9.3, we look at the Jacobium processor architecture that executes Jacobi algorithms of which we give an example. We explain how we use the Y-chart environment and even how we extended it with additional tools. Using the SBF model, we describe Jacobi algorithms at different levels of granularity and show how this affects the development of the Jacobium architecture. Finally, we present conclusions.

9.1 Motivation

In this chapter, we discuss two different design cases of programmable architectures. Both architectures are similar to stream-based dataflow architectures, but the applications belong to different application domains. The purpose of this chapter is to show how the use of the Y-chart approach and Y-chart environment effect real design cases. It also serve to shows that the techniques developed in this thesis are indeed useful and capable of describing realistic design cases.

9.1.1 Application Characteristics

The Prophid and Jacobium architectures are similar to the stream-based dataflow architecture template presented in Chapter 2. For the Prophid architecture, this is not surprising since its architecture template was in principle taken as model for the development of the class of stream-based dataflow architectures. The Jacobium architecture template is a result of further development of previous architectures [van Dijk et al., 1993] and was inspired by developments in wireless communication architectures [Jain, 1997]. The designers of the Jacobium architecture knew about the Prophid architecture and were influenced by it.

However, the two architectures execute applications with different characteristics. This becomes clear when we look at the dependence graphs (DG) of two applications. The DG representation of video applications used in the Prophid case looks like a sequence of predominantly coarse-grained operators operating in a pipeline fashion. See, for example, the description of the Picture in Picture application in Figure 6.1. In contrast, array signal processing applications used in the Jacobium case must exploit correlation of data received from, for example, various antennas. Therefore, their DG representation consists of more fine-grained operators that are much more interrelated. See, for example, the dependence graph in Figure 9.10.

Application Descriptions

The use of the Y-chart environment in both cases requires that applications be available and described as networks of SBF objects. We experienced severe difficulties in finding realistic applications in both the Prophid and the Jacobium case. If applications were specified at all, the specification was not in the format of networks of SBF objects or in a format from which a networks of SBF objects can easily be derived. The specifications were given in the C-programming language or in Matlab, using global variables and data structures of the wrong type (i.e. matrices or lines) and in such a way that they hid any form of parallelism, due to the inherent sequential ordering of the imperative languages (see for more information Chapter 6, where we explain how we model applications).

The acquisition of applications was somewhat less troublesome in the Jacobium case. In that project, the Jacobium processor executes algorithms which belong to the class of Jacobi-type algorithms [Golub and Loan, 1989]. For this class, many algorithms have already been specified as nested-loop programs. Moreover, tools are available that can help to manually transform these nested-loop programs into the suitable format [Held, 1996], albeit with difficulty.

A consequence of not having a well-defined set of applications, especially in the Prophid case, is that design space exploration is performed using one application instead of a set of applications. Synthetic application descriptions have been considered, but the derivation of realistic synthetic benchmarks is a complex problem in its own right [Dick et al., 1998].

We do not intend to give a complete description of the design cases and their results. Instead, we want to show that techniques developed in this thesis are indeed useful and capable of modeling problems appearing in both cases.

In the Prophid case, we focus on whether we can describe an industrially relevant application as well as instances of the Prophid architecture and map the application onto these architecture instances. In the Jacobium case, we focus on the application development. We examine how we can describe Jacobi-type algorithms using the SBF model at different levels of granularity. To that end, we augment the Y-chart with additional tools, providing a trajectory allowing us to execute Matlab applications written as nested-loop programs on instances of the Jacobium architecture.

9.2 Case 1: The Prophid Architecture (Philips Research)

The objective of the Prophid case [Leijten et al., 1997; Leijten, 1998] at Philips Research is to make a weakly programmable video-processor for use in consumer video systems. It must be suitable for performing in real-time video applications like quality enhancement, frame resizing, and color adjustment.

We modelled an industrially relevant video application, referred to as the *benchmark*, using the application model approach developed in Chapter 6. We are interested in whether we can obtain performance numbers for instances of the Prophid architecture created using the architecture modeling approach developed in Chapter 5. In particular, we are interested in the trade-offs between communication strategy, packet length and buffer size.

We first explain the Prophid architecture and the benchmark. Then we look at the Y-chart environment used and at what consequence this has for describing the architecture instance and the benchmark. Following this, we look at a design space exploration of the Prophid architecture executing the benchmark. In this case we focus more on architecture modeling, which we represented in Figure 3.2 with the arrow with the lightbulb pointing to the box *architecture instance*.

9.2.1 Prophid Architecture

The architecture template of the Prophid video-processor architecture is shown in Figure 9.1. The Prophid architecture consists of a number of *co-processors* that operate concurrently on streams of data. The co-processors are divided further into *workspaces* where the actual signal processing takes place and they communicate streams with each other using input and output FIFO buffers and a switch matrix. The Prophid architecture employs a Time Division Multiplex (TDM) communication strategy, as explained in Chapter 2. The communication capacity of the switch matrix is thus divided into time slots and the global controller ensures that a stream coming from an output buffer is directed to the correct input buffer in the correct time slot. The global controller therefore connects directly to the switch matrix and, furthermore, it uses *terminals* which are simple 1:N and N:1 switches at the input buffer or output buffer side, respectively.

9.2.2 Prophid Benchmark

We want to execute the benchmark described by Witlox [1997], which is is an industrially relevant benchmark imposing a realistic workload on the Prophid architecture. The dependence graph of the benchmark is shown in Figure 9.2. The benchmark consists of three parts, indicated by PathI, PathII, and PathIII. In the benchmark, the four gray nodes (e.g., A, B, C and D) represent memory and the ten white nodes represent coarse-grained functions. In addition, two nodes at the head of the chain represent sources and one node at the tail represents a sink. The purpose of the benchmark is to combine two video signals into a single multi-window video signal. The two different input sources

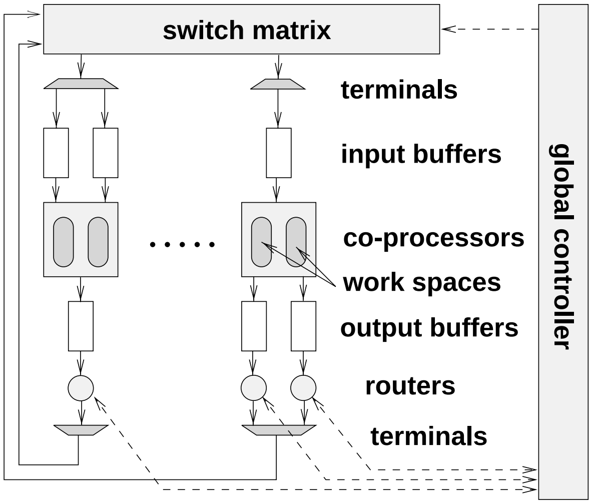


Figure 9.1. The architecture of the Prophid video-processor.

produce streams of samples that are processed by a number of coarse-grained functions such as frame resizing and quality enhancement, after which the two streams are combined into a single output stream. Other coarse-grained functions such as color adjustments and more quality enhancement are performed on the combined stream. Finally, the sink node consumes the stream.

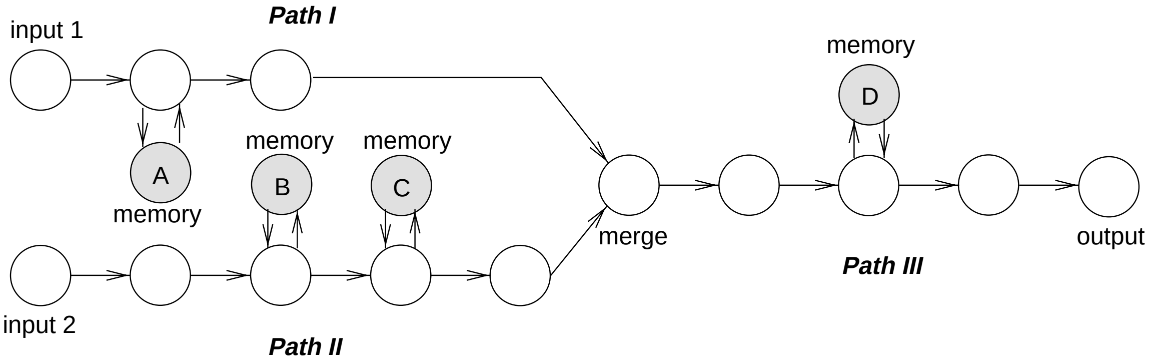


Figure 9.2. The Prophid benchmark.

9.2.3 The Use of the Y-chart in the Prophid Case

We used the Y-chart environment developed in Chapter 7 in this case and thus we need to describe the benchmark as a network of SBF objects using the modeling approach developed in Chapter 6. We also need to describe instances of the Prophid architecture using the modeling approach developed in Chapter 5. We discuss these two aspects in more detail.

Modeling the Prophid Benchmark as a Network of SBF Objects

The benchmark was not specified functionally correctly; only consumption/production patterns were specified for each node in Figure 9.2. The benchmark therefore describes a static application and it

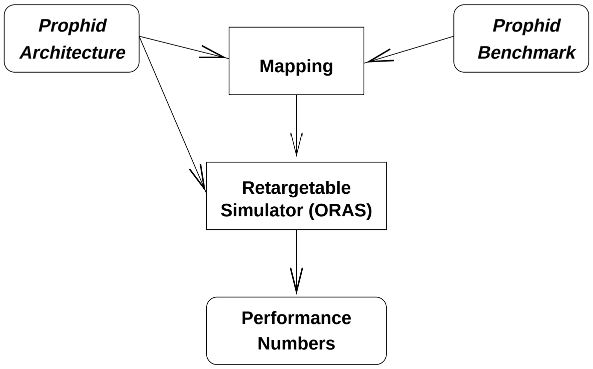


Figure 9.3. The Y-chart used in the Prophid case.

suffices to describe the nodes of the benchmark with SBF objects that use dummy functions. These *dummy functions* only consume and produce tokens using a particular pattern.

We constructed an SBF object for each node in the benchmark, including the memory nodes (we say more about modeling memory as SBF objects later). For these SBF objects, we specified the transition function and binding function (see Section 6.4) to capture the consumption/production pattern of each node in Figure 9.2.

Because the benchmark was not specified functionally correctly, it cannot describe a dynamic application. It can describe, for example, a frame resizing factor, which is a fixed number resulting in a fixed consumption/production pattern. We remark here that ORAS is, however, capable of executing fully functionally applications – thus including their dynamic behavior – in the context of an architecture instance. If applications become more dynamic – and they will – we will be able to describe these applications and simulate them correctly. The PiP case mentioned in Appendix B, for example, exhibits dynamic behavior (resizing of the image) and can be simulated by ORAS.

Modeling the Prophid Architecture

We had to describe the Prophid architecture in terms of the stream-based dataflow architecture template. This implies that we must be able to specify an instance of the Prophid architecture in terms of the architecture description language discussed in Section 5.5.

We described the Prophid co-processors as functional units and the workspaces as functional elements. Since the Prophid architecture uses a Time Division Multiplexed (TDM) communication structure, the functional units operate in a sample-switching mode (see Section 2.1). This means that the switching between workspaces on a co-processor takes place on a sample basis. To describe the TDM communication, we define the global controller as a TDM controller and the routers as TDM routers, which are both described in Appendix C. We do not have to describe the terminals shown in Figure 9.1 explicitly, because the global controller passes on an explicit reference to routers. Such a reference indicates to which input buffer a router has to write (see Section 5.6, where we explain the programming model of stream-based dataflow architectures).

The SBF objects of the benchmark map to functional elements, including SBF objects modeling memory. We assign a *latency* and an *initiation period* to each functional element to describe the timing characteristics of the functional elements (see Figure 7.8, where we showed how a functional element and it timing characteristics are specified in the architecture description language). The assigned values are initially obtained from designers when they are developing the co-processors.

9.2.4 Issues in Modeling the Prophid Architecture

We came to three modeling issues that we want to highlight in more detail. These modeling issues are typically encountered when modeling realistic applications and architecture instances. The three modeling issues are the modeling of memory in the Prophid architecture, the modeling of TV signals, and the modeling of the merging of two streams. These issues are discussed below.

Modeling Memory in the Prophid Architecture

In principle, modeling memory is a problem in a dataflow architecture because memory does not adhere to the data-driven execution scheme. A processing element activates solely based on the availability of data (see Section 2.1.3). Memory, however, can produce stored tokens even though no data is available to write into memory; therefore it does not operate in a data-driven fashion. Even so, we can still model memory as a functional element like any other functional element in the case of stream-based dataflow architectures.

We model memory as a functional element containing an asynchronous pipeline the size of the memory capacity. Unlike an ordinary asynchronous pipeline, we initialize the pipeline with data. Because a functional element has a read **process** and write **process** which are uncoupled when an asynchronous pipeline is used (see Section 5.4.10), the write **process** can send out tokens present in the pipeline before new data is read into the pipeline by the read **process**.

Notice that a functional element also contains an SBF object which can perform very complex reordering schemes in memory. The transition function of an SBF object is responsible for generating the appropriate read and write addresses for memory. An example of an SBF object that models memory was already given with the transpose functions in the Picture in Picture example described in Section 6.1. In that case, the transition function of the SBF object implements a particular sequence of read and write addresses that causes the transposition of an image.

Modeling a TV signal

A TV signal contains video pixels, but they have a particular shape in time, i.e. a *time shape* as shown in Figure 9.4. This figure shows a TV signal consisting of three components: a *line blanking*, a *field blanking*, and *video data*. The first two components are control signals included to give the electron beam inside a TV tube enough time to start writing new video pixels at the correct position on the screen. The line blanking delays new video pixels so the electron beam can start at the beginning of a new line. The field blanking delays new video data so the electron beam can start again at the top left position.

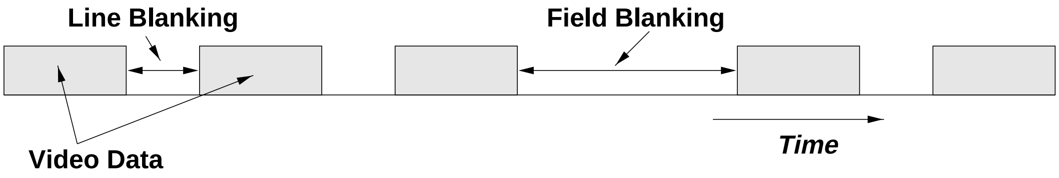


Figure 9.4. a TV signal consists of three components: a *line blanking*, a *field blanking*, and *video data* (the gray blocks).

The two most commonly used standards for TV signals are PAL and NTSC. As explained in Section 1.2, a TV field has a number of video lines whereby each line consists of video pixels. The number of lines and video pixels involved in the two standards are listed in Table 9.1. During a line blanking a functional element could process other streams instead of being idle while waiting for the end of a blanking. Even more time is available for processing during field blanking. This indicates an important opportunity for further improvement of the efficiency of a Prophid architecture. Hence, it is important that we can correctly model the time shapes given in Figure 9.4.

	Video Data		Line Blanking	Video Blanking
Standard	Video Pixels	Video Lines	(equivalent video pixels)	(equivalent video pixels)
PAL	832	281	193	31935
NTSC	832	241	193	27839

Table 9.1. Time Shape of TV Signals.

We model the time shape of TV signals using the controller of an SBF object. The SBF object obtained describes either node input1 or node input2 of the benchmark and maps to a functional element describing the source for pathI and pathII, respectively.

A source functional unit uses only output buffers, to which it writes. By using functions in SBF objects with no inputs or outputs, we modeled the line and field blanking periods accurately of the standard TV signals given in Table 9.1. Due to the fire-and-exit behavior of SBF objects (see Section 6.8), a transition always takes a number of cycles equal to the initiation period when instantiated on a functional element (see Section 7.5.1, where we explain how to augment SBF objects with time). Regardless of whether a function of an SBF object produces output at all, a transition always takes place; after all, a function with no inputs and outputs fires instantaneously. Therefore, a functional element is delayed a number of cycles equal to the initiation period each time a transition takes place. By calling a sequence of functions with no inputs or outputs as described by the controller (e.g., the binding and transition functions), we describe the time shape of TV signals cycle-accurately.

Modeling the Merging of Two Streams

The merging of the streams is not functionally specified for node merge. As a consequence, we can describe it only using a non-deterministic behavior. SBF objects, however, cannot describe non-deterministic behavior.

In the benchmark of Figure 9.2, the two streams of pathI and pathII come together in the merge to form a single new stream for pathIII. Since this merging is not functionally specified, the merging takes place in a non-deterministic way. In general, non-determinism is used to leave open the precise implementation of something like a merge. On the final Prophid architecture the implemented merge will definitely be a deterministic merge.

SBF objects cannot describe non-deterministic behavior, because of the blocking reads. The same property formulated differently: an SBF object describes a Kahn process and, as such, always describes a deterministic behavior. We remark that a CSP process would be able to describe a non-deterministic merge (see the discussion in related work on model of computations in Section 6.7).

Yet, to describe a non-deterministic merge using ORAS, we replaced the merge node in the description of the benchmark by three new nodes: two new sink nodes and one new source node. In that case, pathI and pathII end up in two new sinks. A new source produces a stream for pathIII. In this way, all three paths are uncoupled and describe a non-deterministic behavior.

Another approach to describing a non-deterministic merge is to relate the stream of pathI with the stream of pathIII. In this case, pathI describes the 'master' stream to which the stream for pathIII synchronizes. We then model the merge as a memory element. The stream in pathII ends up in a sink and is thereby uncoupled from pathI and pathIII.

9.2.5 Results

Given the fact that we realized a Y-chart environment for the Prophid architecture, we now use this Y-chart environment (see Figure 9.3) and the DSE presented in Chapter 8 to investigate which communication strategy works best: a time-division-multiplex (TDM) or a first-come-first-served (FCFS) communication strategy. For the TDM communication strategy, we also investigate the trade-off between the length of packets (i.e., the packet length) and the number of cycles making up the length of a time slot (i.e., the slot length).

For the Prophid architecture, we define a many-to-one mapping of the nodes of the benchmark to an architecture instance. In this many-to-one mapping, we define, for example, that two memories (the gray nodes A an B in the benchmark) map on one functional unit and the other two memories (the gray nodes C and D) map onto another functional unit. In the architecture instance, we fixed all options except for the *packet length*, *service time* of the controller, the *buffer size* of the buffers, and the communication strategy. If a TDM communication strategy is selected, we also define the parameter *slot length*. Table 9.2 shows the parameters with their ranges. Using these parameters we set up two experiments of 100 trial runs each. One experiment uses the TDM communication strategy and one experiment uses the FCFS communication strategy.

In each trial run, a scaled video sequence of 14,400 pixels is processed by the benchmark. The line and field blanking are scaled accordingly. We use the merge in which pathI and pathIII are related. Each experiment of 100 trial runs, took 1.2 hours to execute.

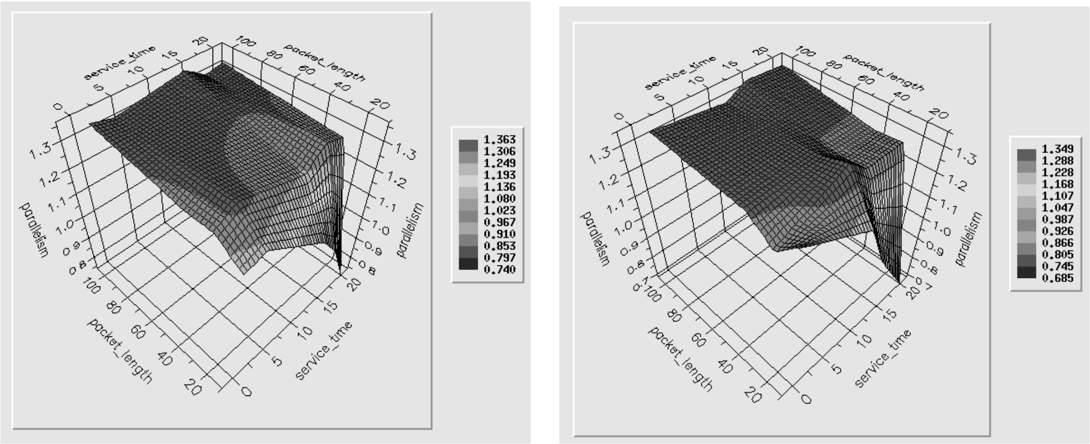
Parameter	Range
packet length	{ 10 . . . 100 }
slot length	{ 10 . . . 100 }
buffer size	{ 10 . . . 100 }
service time	{ 1 . . . 20 }
communication strategy	{ TDM, FCFS }

Table 9.2. Parameters used in the design space exploration of the Prophid architecture.

In Figure 9.5 we show the level of parallelism achieved when either a TDM (a) or a FCFS (b) communication strategy is used to execute the benchmark. We have chosen the slot length in Figure 9.5(a) to be 55 and the buffer size in cases (a) and (b) to equal 55. On the basis of the response surface models (RSMs), we concluded that both strategies realize the same level of parallelism. When the global controller has a service time shorter than 10 cycles per request and architecture instances use a packet length larger than 40 samples, the global controller is not a bottleneck and the highest level of parallelism is obtained.

Since we look at real-time applications, a more important performance metric is the achieved throughput for the inputs (i.e. input1 and input2) and the output (i.e. output) of the benchmark application of Figure 9.2. The RSMs that we present only relate to the experiment in which we used a TDM communication strategy. Furthermore, we look only at nodes output and input1 when we consider throughput.

In the benchmark, node input1 has a throughput constraint of 13.5 Msamples/sec and node output has a throughput constraint of 27 Msamples/sec. Thus, output produces a video stream at twice the



(a) TDM Communication Strategy

(b) FCFS Communication Strategy

Figure 9.5. The response surface model (RSM) for achieved parallelism with either a time-division-multiplex (TDM) communication strategy (a) or a first-come-first-served (FCFS) communication strategy (b).

speed at which a video stream enters input1. For the way we set up the benchmark this means that input1 should realize an initiation period of 12 cycles and that output should realize an initiation period of 6 cycles [1]. In Figure 9.6, we show the initiation period realized for input1 (a) and ouput (b). In these figures, the service time is equal to 4 cycles and the buffer size is equal to 55 samples. Note that the initiation period is not exactly 6 cycles, since we measure the average initiation period.

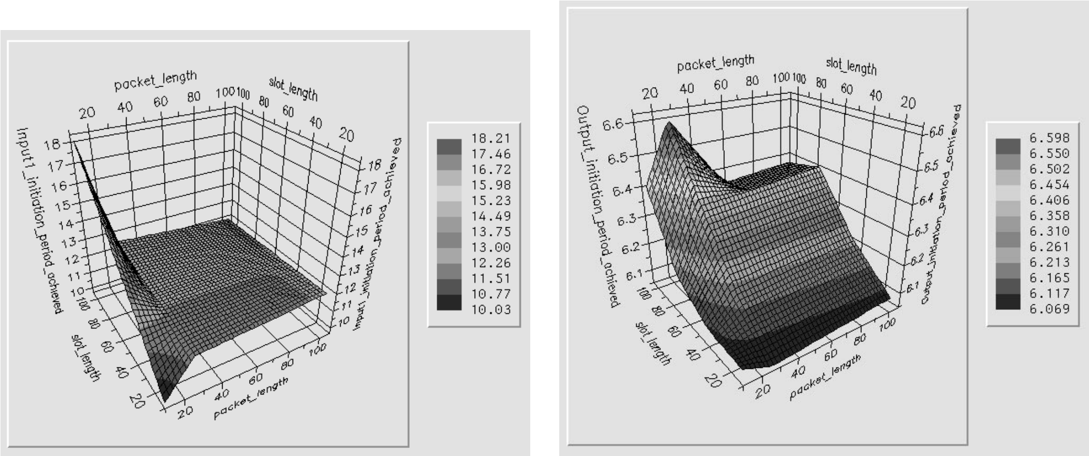
On the basis of the RSM in Figure 9.6(a), we conclude that if a packet length of 40 samples or more is selected, and the slot length has any value, input1 is able to satisfy an initiation period of 12 cycles. On the basis of the RSM in Figure 9.6(b), we conclude that if a slot length of 30 cycles or less is selected, and the packet length has any value, output is able to satisfy an initiation period of 6 cycles. By combining the results of both figures, we were able to conclude that a Prophid architecture satisfies the imposed real-time constraints of the benchmark when it employs a slot length of 30 cycles or less and a packet length of 40 cycles or more.

We want to realize an architecture instance of Prophid that uses the smallest possible buffers, since buffers take up silicon area. In Figure 9.7(a), we show the RSM expressing the initiation period realized by output for various buffer sizes and slot lengths. On the basis of this RSM, we concluded that buffers should have a capacity of at least 35 samples, irrespective of the slot length chosen.

Finally, we show the relationship between slot length and packet length and the achieved level of parallelism in Figure 9.7(b). Based on the RSM, we conclude that the achieved parallelism is completely dominated by the packet length when a slot length is selected shorter than 80 cycles. Therefore, to maximize the level of parallelism satisfying the real-time constraints, a large packet length should be selected.

To summarize: a Prophid architecture that executes the benchmark should use a large packet length, for example 100 samples per packet, a slot length shorter than 30 cycles, and should implement

[1] Recall that the initiation period is the reciprocal of throughput



(a) input1

(b) output

Figure 9.6. The initiation period realized for input1 (a) and output (b) of the benchmark as a function of the packet length and slot length.

buffers having a capacity of 35 samples. Note, however, that these numbers only apply when a global controller is used requiring 4 cycles per request, i.e., when the service time is 4 cycles.

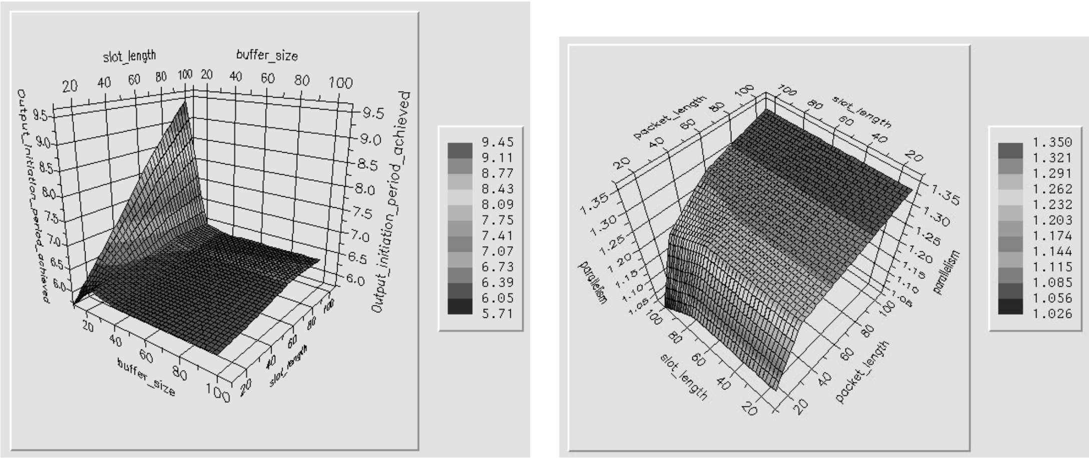
9.2.6 Conclusions

For the Prophid design, we showed that we can set up a Y-chart environment and we used this environment to perform a design space exploration. Although the results found in this exploration will not be used directly by Philips in further development of the Prophid architecture, they clearly show the potential of the Y-chart approach. We performed a design space exploration of the Prophid architecture and made various trade-offs explicit by evaluating 100 different architecture instances in 1.2 hours. Using these trade-offs, we showed that we could select parameter values leading to a feasible architecture instance.

In setting up the Y-chart environment, we showed that the SBF model is able to describe the benchmark. We also showed that we can specify instances of the Prophid architecture using the architecture description language discussed in Chapter 5. In describing an architecture instance, we encounter three difficult modeling issues: modeling memory, modeling the time shape of TV-signal, and modeling non-deterministic merging of streams. We showed that we could model all three issues.

9.3 Case 2: The Jacobium Processor (Delft University of Technology)

The objective of the Jacobium project [Rijpkema et al., 1997] at the Delft University of Technology is to make a weakly programmable and scalable processor, the *Jacobium Processor*, tailored to array signal processing applications in which so called *Jacobi-type algorithms* play a dominant role. The signal processing applications detect, recover or separate signals, or estimate system model parameters and are used in array processing [Justice et al., 1985] and time series analysis [Porat, 1993]. A char-



(a) Initiation Period

(b) Parallelism

Figure 9.7. (a) The RSM for initiation period of output as a function of buffer size and slot length. (b) the RSM for parallelism as a function of slot length versus packet length.

acteristic and common feature of these applications is that they perform matrix computations [Golub and Loan, 1989], which typically account for a heavy computational payload. Examples of such matrix computations include the *QR* and *SVD* matrix decompositions [Golub and Loan, 1989]. Both decompositions can be implemented by means of Jacobi-type algorithms. Jacobi-type algorithms use trigonometric computing techniques and have in common the use of plane rotations [2]. These plane rotations compute efficiently using a *Cordic* [Volder, 1959], which is the main – but not the only – computing element used in the Jacobium processor.

In the following, we first show the Jacobium processor architecture, followed by an example of a Jacobi algorithm in Matlab, namely the QR algorithm. Then we explain how we use the Y-chart environment and how we extended this Y-chart environment with additional tools allowing us to execute Jacobi algorithms described in Matlab on instances of the Jacobium architecture. We examine how we can describe Jacobi algorithms using the SBF model at different levels of granularity and how this effects the development of the Jacobium architecture. In this case we focus more on the application model, which we represented in Figure 3.2 as an arrow with the lightbulb pointing to the box *applications*.

9.3.1 The Jacobium Processor Architecture

The architecture of the Jacobium processor is shown in Figure 9.8. It consists of *Processing Elements* (PEs) that operate in parallel and that communicate streams of data to each other via a *communication network*, under the supervision of a *global controller*. Each processing element consists of a *router*, a *local controller*, a *compound node*, and a *local memory*, as can be seen in the inset in Figure 9.8. The Jacobium processor employs Cordics in the compound node as the basis computational elements to execute very efficiently the *vectorization* and *rotation* operations typically used in the Jacobi-type

[2] Plane rotations are also referred to as “Givens” rotations

algorithms. Besides a Cordic, the compound node may contain other hardware to compute particular functions, for example, multipliers and adders, as well as functions of large granularity. Compound nodes have a small instruction set that the local controller uses to put the compound node in a particular mode executing a particular function. The local memory is used to store tokens temporarily or to re-order tokens in streams. The local memory has a small instruction set [see, for example, Looye et al., 1998] that the local controller uses to obtain samples in a particular order from the local memory. These samples are consumed by the compound node, which produces new samples. The router routes these new samples through the architecture, under the supervision of the local controller.

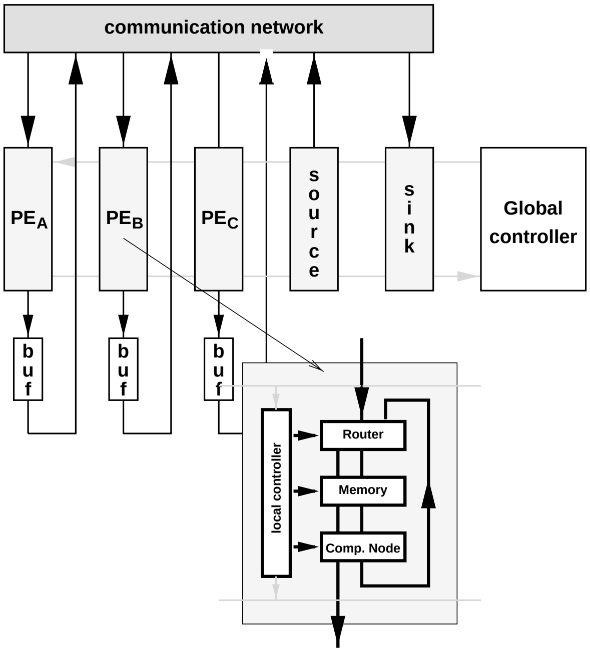


Figure 9.8. The architecture of the Jacobium processor.

Especially at the top level, the architecture template is very similar to the architecture template of stream-based dataflow architectures, yet the PEs differ from the PEs of the stream-based dataflow architecture. Nonetheless, we can model the operations of the PEs of the Jacobium architecture by means of the PEs of the stream-based dataflow architecture.

To realize this modeling, we rely on the special structure of an SBF object. Given a processing element, we can describe this processing element by means of a single SBF object. We model its local controller by the controller of an SBF object, its local memory by the state of an SBF object, and its compound node by the set of functions of an SBF object. This SBF object is instantiated onto a functional element. The router of the PE of the stream-based dataflow architecture containing the functional element models the router of the Jacobium PE. The output buffer of the functional unit in which the functional element resides models the output buffer of the PE. The modeling of a Jacobium PE by means of a stream-based dataflow PE is illustrated in Figure 9.9.

Because of this modeling, we can use SBF objects to describe processing elements, abstracting from the internals of the processing elements. This gives us the opportunity to reason on the design of Jacobium architectures at a higher level of abstraction, as we will show.

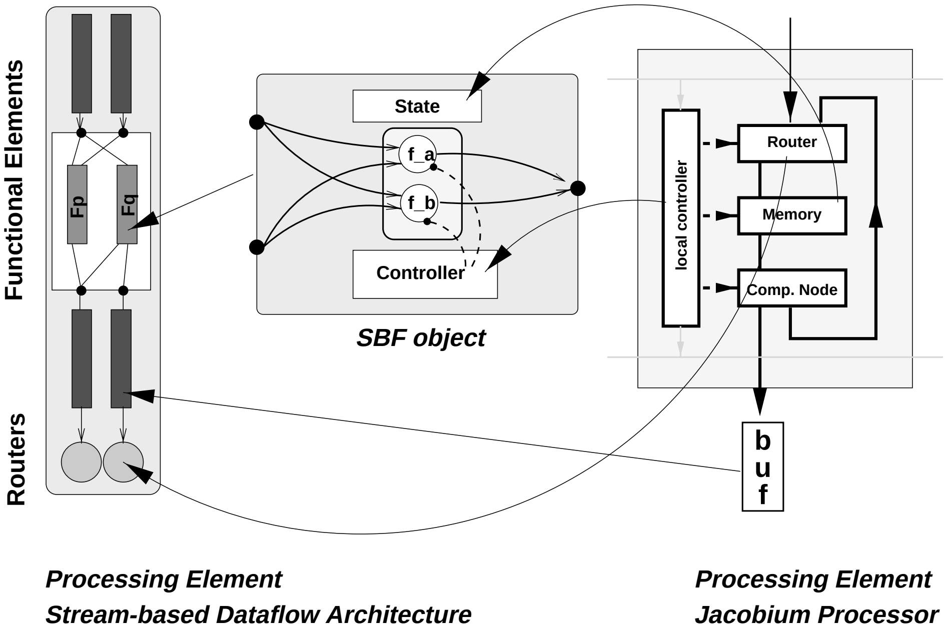


Figure 9.9. Modeling a processing element of the Jacobium architecture as a processing element of stream-based dataflow architectures.

9.3.2 Jacobium Applications

An example of a Jacobi-type algorithm is the *QR matrix decomposition* that can be found in adaptive beam-forming applications [Van Veen and Buckley, 1988]. Matlab code for a QR decomposition program is shown in Program 9.1. The program's loop bounds `K` and `N` are parameters. Two inner-loop iterators `j` and `i` form a triangular-shaped index space of size `N`. This triangular shaped space describes a single *QR update*. The outer-loop iterator `k` indicates which iteration of the QR update is currently taking place. Notice that the QR decomposition program in Program 9.1 uses matrices `R` and `X` and has a lexicographical sequential index ordering as dictated by the `for` loops. The structure of the QR algorithm is referred to as a *Nested Loop Program* [Held, 1996].

Program 9.1. QR ALGORITHM

```
for k = 1 : 1 : K,
  for j = 1 : 1 : N,
    [R(j,j),X(k,j),theta(j)] = Vectorize(R(j,j),X(k,j));
    for i = j+1 : 1 : N,
      [R(j,i),X(k,i)] = Rotate(R(j,i),X(k,i),theta(j));
    end
  end
end
```

The QR algorithm of Program 9.1 has a dependence graph representation as depicted in Figure 9.10. Instead of showing the complete dependence graph of the QR algorithm, we show only one k-plane, representing a single QR update. The complete DG would consist of `K` of these planes. Each and every k-plane depends on its predecessor. Each node in the DG represents a function from the QR algorithm: a gray node represents the function `Vectorize` and a white node represents the function `Rotate`. At the side of the triangle at the top of the figure, data (i.e., the X data) is arriving from external sources – say sensors of an `N`-antenna array (in this case `N`=6) – that propagates downwards through the plane. The values of 21 r variables (i.e., elements of the `R` matrix in Program 9.1) produced by the previous plane are updated using the functions `Rotate` and `Vectorize`. To rotate, each `Rotate` function needs to have an angle `theta` calculated by the function `Vectorize` on the diagonal of the triangle.

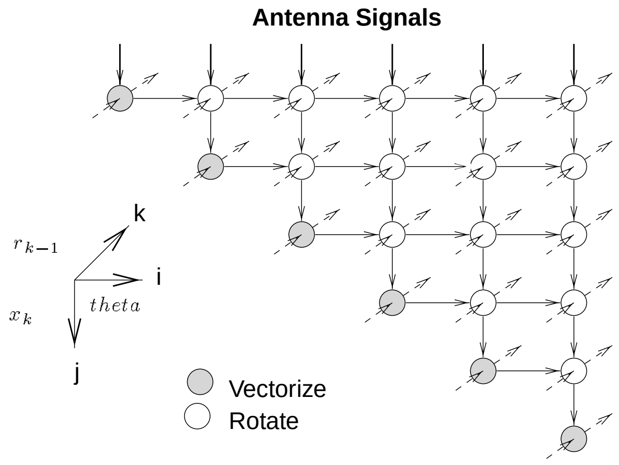


Figure 9.10. Dependence Graph Representation of one QR update k-plane of the QR-algorithm.

The dependence graph representation of the QR algorithm reveals features like regularity and locality. Unlike the description given in Program 9.1, it also reveals a high level of concurrency. These features – regularity, locality, and concurrency – are typical for all Jacobi-type algorithms.

9.3.3 The Use of the Y-chart Approach in the Jacobium Processor Case

We want to use the Y-chart approach to assist us in the design of the Jacobium architecture. For that purpose, we developed the environment shown in Figure 9.11. The core of the design environment consists of the Y-chart environment developed in Chapter 7. Thus, we again use ORAS as the retargetable simulator and we describe applications as a network of SBF objects, using the SBF model developed in Chapter 6. We augment the original Y-chart environment with additional elements to support the conversion from nested loop programs described in Matlab into networks of SBF objects. The added elements in the figure are:

- the Applications in Matlab box
- the Algorithmic Transformations box
- the Translation box
- the Validation boxes

Furthermore, we include the Library of SBF objects and the simulator SBFsim in the environment. Both elements are presented in Chapter 6, where we described the implementation of the SBF model.

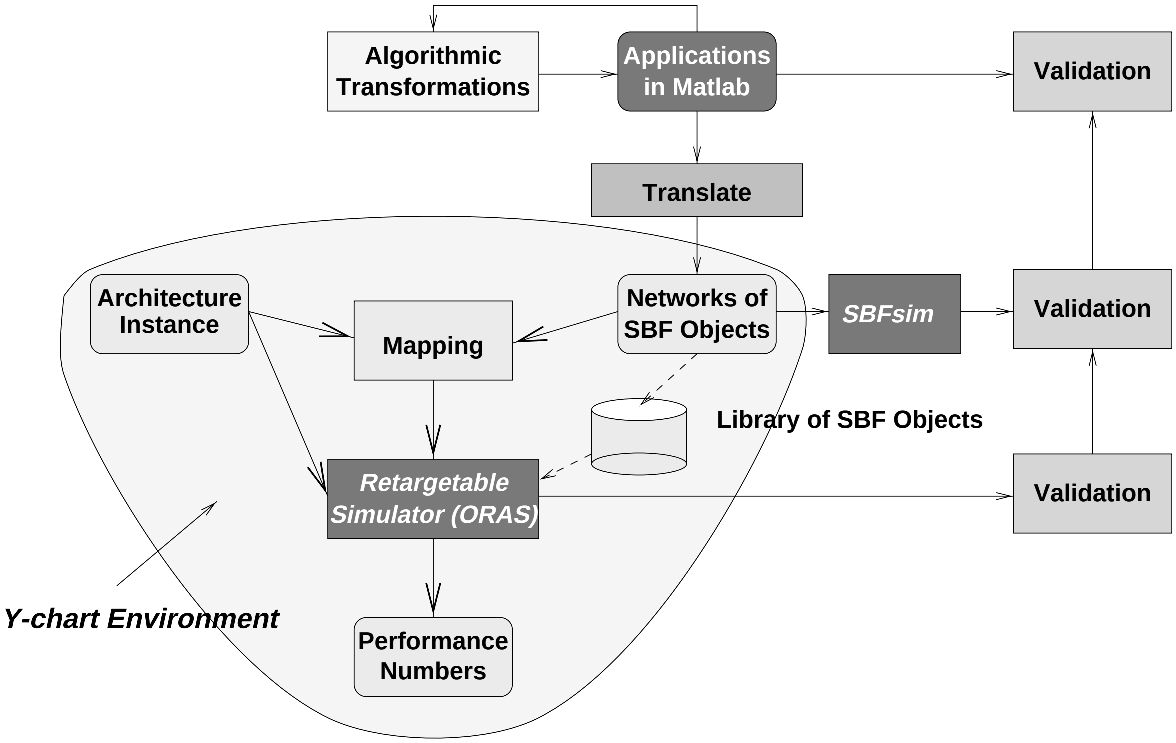


Figure 9.11. The extended Y-chart environment used in the Jacobium Processor case.

Applications in Matlab and Algorithmic Transformations

The environment is based on the assumption that we start from Jacobi applications written as nested loop programs in Matlab. These applications can be transformed to change characteristics of the original Jacobi application [see van Dijk et al., 1993].

Translation

Both the original and transformed Matlab specifications are, however, built on data types from the matrix algebra: that is, matrices and vectors. The translation of these types into streams of scalars is required. Moreover, the strict serial ordering of operations has to be removed to reveal the high degree of concurrency present in almost all Jacobi-type algorithms.

Thus, somehow, we need to translate the original imperative Matlab descriptions into a network of SBF objects. For that purpose, we rely on the tool *HiPars* [Held, 1996], developed originally as part of the HiFi project [Dewilde et al., 1989]. It converts static algorithms, described as parameterized nested loop programs, into a dependence graph representation. Part of such a DG description is already shown in Figure 9.10. In the resulting DG representation, the full concurrency present in the application is revealed and only scalar variables are communicated over the edges of the DG.

We can go from such a DG representation to a network of SBF objects. We will explain this translation in detail for the QR algorithm in Program 9.1. We can use predefined SBF objects from the library or we can create new SBF objects to represent the DG as a network of SBF objects. By storing these new objects in the library, we promote reuse of SBF objects.

Validation

The translation of a Matlab program describing a nested loop program to a network of SFB objects is not trivial and is not yet automated. For these reasons, we included a validation trajectory to check whether both the Matlab program and the SBF network have the same input/output behavior. Thus we verify the translation steps by providing the same input data to both the Matlab applications and the network of SBF objects and then comparing the output data.

We can map a network of SBF objects onto an architecture instance and use ORAS to obtain the performance numbers. Because ORAS also performs a full functional simulation, we can also validate the behavior at this level: ORAS should produce for the same input data the same output data as that produced by SBFsim or Matlab. This way we can check if the mapping was performed correctly.

9.3.4 Deriving a Network of SBF Objects from a Dependence Graph

We will show a very simple case in this section in which we express the QR algorithm of Program 9.1 as a collection of different SBF objects, each with (very) different characteristics. We will use the extended Y-chart environment to provide feedback on the different SBF objects realized.

Using the HiPars tool, we attain a DG representation of a nested-loop Matlab program. We ought to derive SBF objects from the DG representation that we can combine to model the original Matlab application as a network of SBF objects. This means, in the first place, that we have to *partition* the DG into smaller pieces, in which each piece makes up an SBF object with a particular set of functions. In the second place, for each identified SBF object we have to specify the controller functions, i.e., the transition function and the binding function.

Partitioning a Dependence Graph

Given a single k-plane of the QR algorithm as a DG representation, there are many ways we can construct a network of SBF objects executing the same QR application. Different partitions result in different grain sizes of the SBF objects, as was explained in Section 6.6. In Figure 9.12, we show three different partitions, each leading to different SBF objects with different grain sizes. The number of functions that an SBF object includes determines the grain size of that SBF object if we assume that the functions `Vectorize` and `Rotate` both require x RISC-like operations.

We now illustrate three possible partitions (a, b, and c) to describe the same k-plane of the QR algorithm. In specifying these partitions, we distinguish *regions* which describe a regular part of a DG containing the same kind of functions. Although two regions can describe the same function, they can differ as to where functions get their data from and where they write output data to. In that case we have *variants* of the same function (see Section 6.6).

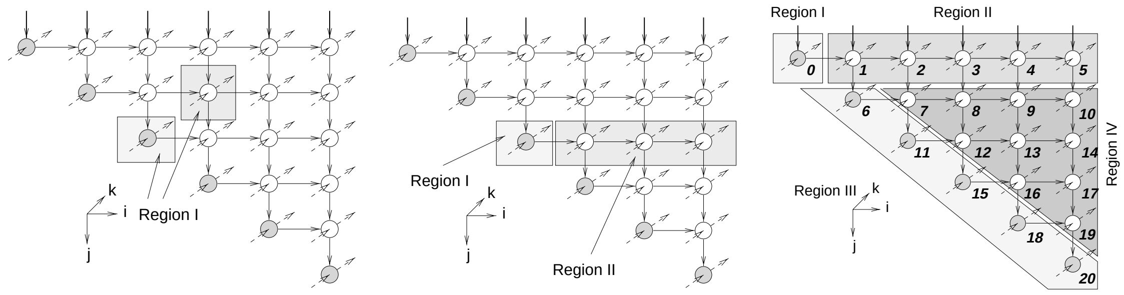
case (a): In this partitioning, we assume that each node of the DG defines an SBF object by itself. This results in two SBF objects: *SBFVectorize*, containing the function `Vectorize`, and *SBFRotate*, containing the function `Rotate`. Since both SBF objects contain only one function, the objects distinguish only one region. Both SBFVectorize and SBFRotate have a grain size of $1x$. To cover the DG we would need 6 instances of SBFVectorize and 15 instances of SBFRotate.

case (b): In this partitioning, we assume that each horizontal chain of nodes (j-line) in the DG defines an SBF object *SBFline*. SBFline contains two functions: `Rotate` and `Vectorize`. Each j-line consists of two regions: region I and region II. The function `Vectorize` is active in region I, and the function Rotate is active in region II. The variable $theta$ is kept internal to SBFline and hence it is stored in the state of SBFline. SBFline has a grain size of $6x$ because the longest line includes 6 functions. To cover the DG we would require six instances of SBFline.

case (c): In this partitioning, we assume that the complete triangular graph defines a single SBF object *SBFplane*. It contains the functions Rotate and Vectorize but each in two different variants, variant 1 and variant 2. SBFplane contains four different regions. In regions I, II, III, and IV the functions Vectorize_1, Rotate_1, Vectorize_2, and Rotate_2, respectively, are active. The function Rotate differs in Regions II and IV as to whether a function reads external data (e.g. from the sensors of an N-antenna array) or internal data. The same applies to the function Vectorize. In Region IV variables $theta$ and X are kept inside the SBF object. Variable $theta$ requires a single place to store its value, whereas variable X requires a FIFO structure of size 5. SBFplane has a grain size of $21x$ because it includes 21 functions. To cover the DG we would require one instance of SBFplane.

Although the different SBF objects have different granularities, the set of functions always consists of the same function Rotate and Vectorize. Because an SBF object describes a processing element, the compound node has to execute the set of functions defined by the SBF object. Since the three different SBF objects only use sets of functions consisting of Rotate and Vectorize, the use of a Cordic as the basic element of a compound node is a logical choice.

We want to remark that the SBF Network for the partitioning and assignment discussed in case (a) is intrinsically more parallel than the network discussed in case (c), as was explained in Section 6.6.



(a) SBFVectorize & SBFRotate

(b) SBFLine

(c) SBFplane

Figure 9.12. The dependence graph representation of the QR-algorithm partitioned in different ways, leading to SBF objects with different grain sizes.

Deriving the Controller Functions of an SBF object

Each SBF object covers a distinct part of a DG. For each part, we need to resolve an ordering of the function invocation that leads to the binding function μ and the transition function ω for the controller of each SBF object. We obtain the ordering by scheduling the nodes in the part of the DG covered by an SBF object. One – arbitrary – ordering is the ordering overlaid on the graph in Figure 9.12(c). Thus, each node has been assigned a number indicating its position in the execution order in which SBFplane will process nodes.

Given the ordering of Figure 9.12(c), we can derive the binding functions μ as

$$\mu(s) = \begin{cases} \text{Vectorize}_1, & \text{if } s = 0 \\ \text{Rotate}_1, & \text{if } s = 1, 2, 3, 4, 5 \\ \text{Vectorize}_2, & \text{if } s = 6, 11, 15, 18, 20 \\ \text{Rotate}_2, & \text{if } s = 7, 8, 9, 10, 12, 13, 14, 16, 17, 19 \end{cases} \tag{9.1}$$

and the transition function ω as

$$\omega(s) = s + 1 \pmod{20}. \tag{9.2}$$

The presented controller functions are defined for a k-plane DG having $N = 6$. We could also have defined the controller functions in a parameterized form, expressed in parameter N.

Given the three possible partitions, we need to decide which partition results in the best architecture given the set of Jacobi-type algorithms. The SBF objects shown in Figure 9.12(a) can easily be reused since the functions `Vectorize` and `Rotate` are used in most Jacobi-type algorithms. However, the price that is paid for this reusability is a large amount of communication of single scalar variables over the communication structure. The SBF objects presented in Figure 9.12(b) and Figure 9.12(c) keep some variables internal, reducing the amount of data that needs to be transported. This increases the complexity of the local controller and may cause an increase in the amount of local memory needed in a PE. As the SBF objects become more coarse-grained, tokens can very effectively be taken together into a packet, leading to a more efficient transportation of tokens. For example, the R values needed from a previous QR update are very effectively transported using packets. In addition, coarse-grained SBF objects can perform very complex operations, including the reordering of data streams [Garcea, 1998].

Pipelining

The SBF objects described in Figure 9.12 are executed on a compound node. A compound node typically uses a Cordic in the case of the Jacobium architecture. This Cordic is deeply pipelined [Hekstra and Deprettere, 1993] – 18 stages deep – to perform the trigonometric computations at a high throughput. Pipelining seriously affects the final performance of the Jacobium architecture, especially when long pipelines are used. The local data dependencies between nodes, as shown in the DGs of Jacobi-type algorithms, make it a very challenging task to obtain a high utilization of Jacobium architectures while maintaining a high throughput.

When we consider the Matlab description in Program 9.1, we notice that the functions are not pipelined at all. However, in the extended Y-chart environment they are, which makes it possible to obtain quick quantitative feedback on the utilization of an instance executed on a Jacobium architecture. By installing an SBF object (like the ones given in Figure 9.12) on a functional element (see Figure 7.8) and specifying a particular *initiation period* and *latency*, we can specify that the SBF object have a pipelined behavior. Furthermore, different processing elements in the Jacobium architecture can execute different k planes. As a consequence, pipelining takes place inside PEs as well as between PEs.

By measuring the performance of an architecture instance, we can investigate the effect of the levels of pipelining. When this is known, we can start to apply purposeful algorithmic transformations to alter Matlab programs, for example to change the shape of the DGs such that a higher utilization results [van Dijk and Deprettere, 1995].

9.3.5 Results

For the Jacobium case, we used the extended Y-chart environment given in Figures 9.11 to quantify the performance of the three partitions shown in Figure 9.12. We constructed three different architecture instances of the Jacobium architecture, each being able to execute the QR algorithm as given in Program 9.1 for 4 antenna signals. One architecture consists of 10 processing elements that execute the SBF objects of case (a), one architecture consists of 4 processing elements that execute the SBF objects of case (b), and one architecture consists of 1 processing element that executes the SBF object of case (c). We execute 1000 k-planes of the QR algorithm on each of these architectures and thus process 4000 antenna signals in total.

We determined the following performance metrics for the three architecture instances: the total execution time to process 4000 antenna signals, the level of parallelism achieved, the number of samples communicated in parallel over the communication structure, the average utilization of the processing elements, and the overall throughput realized by an architecture instance. In Table 9.3, we assume that the Cordics are not pipelined and that communicating a sample over the communication structure takes 1 cycle. In Table 9.4, we assume that Cordics are pipelined 18 stages deep. In Table 9.5, we further assume that the handling of packets by routers takes an additional 4 cycles per header and that the global controller has a *service time* of 1 cycle and a *capacity* of one request at a time (see Section 5.4.5, where we model the global controller).

Case	Total Execution Time (cycles)	Parallelism Oper/Cycle	Communication Samples/Cycle (max/avg)	Utilization PEs (%)	Throughput samples/cycle
A	2009	5.98	24/16	54	2.0
B	2509	3.31	9/6	84	1.60
C	44022	1.43	5/3	91	0.10

Table 9.3. Performance metrics for 3 Jacobium architectures with no pipelined Cordics.

Case	Total Execution Time (cycles)	Parallelism Oper/Cycle	Communication Samples/Cycle (max/avg)	Utilization PEs (%)	Throughput samples/cycle
A	19111	0.60	24/10	5.5	0.21
B	19057	0.46	9/6	11.7	0.20
C	44022	1.36	5/3	90.9	0.10

Table 9.4. Performance metrics for 3 Jacobium architectures with Cordics pipelined 18 stages deep.

Based on the numbers found, we can draw the following conclusions. Table 9.3, illustrates that the use of fine-grained processing elements, i.e., PEs on which we mapped the SBF objects *SBFVectorize* and *SBFRotate*, result in a large amount of global communication (16 samples/cycle) and a high level of parallelism (5.98 oper/cycle). In contrast, the use of coarse-grained processing elements, i.e., PEs on which we mapped SBF object *SBFplane* result in less global communication (3 samples/cycle) and a lower level of parallelism (1.43 oper/cycle).

We have included the pipelining of Cordics in Table 9.4. It shows that architectures (a) and (b) are seriously affected by the pipelining. The performance of architecture (a), for example, drops from

Case	Total Execution Time (cycles)	Parallelism Oper/Cycle	Communication Samples/Cycle (max/avg)	Utilization PEs (%)	Throughput samples/cycle
A	47326	0.25	17/11	2.4	0.08
B	45241	0.19	9/6	4.9	0.09
C	108944	0.55	5/2	36.7	0.04

Table 9.5. Performance metrics for 3 Jacobium architectures with Cordics pipelined 18 stages deep, with a global controller that has a service time of 1 cycle and a capacity of one request at a time, and with routers that require an additional 4 cycles per header to handle a packet.

5.98 to 0.60 operations per cycle. Architecture (c), on the other hand, is hardly affected by pipelining. Based on this table, we conclude that pipelining should not be neglected and that the extended Y-chart environment is useful and enables us to quickly investigate the effects of pipelining.

The results in Table 9.5 are based on the fact that communicating headers takes 4 cycles per header and that one router at a time can access the global controller. This affects the overall performance of all three architectures and the values for parallelism, utilization and throughput are about 60% lower than the values in Table 9.4. Although architecture (c) can use longer packets than architecture (a) or (b), its overall performance dropped 60%. Thus, packets longer that 21 samples are needed to reduce the influence of the headers. In absolute numbers, however, architectures (a) and (b) realize a higher throughput, but architecture (c) uses its resources most efficiently.

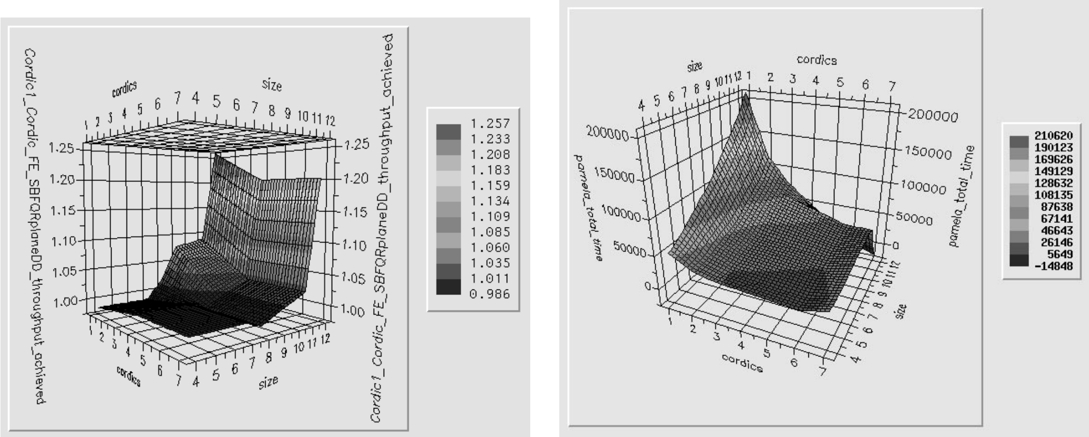
We also performed a design space exploration of the Jacobium processor in which we looked at architecture instances that use N coarse-grained processing elements to execute 1000 k-planes of the QR algorithm. We mapped the SBF object *SBFplane* that processes M antenna signals on each of these processing elements. We also changed the level of pipelining of the Cordics used in the processing elements and the size of the FIFO buffers. In the architecture instances, we assumed that it takes 1 cycle to communicate a sample over the communication structure and that a header requires an additional 4 cycles per packet. Table 9.6 shows the parameters of the exploration along with their ranges. We set up an experiment of 100 trial runs with these parameters. The experiment of 100 trial runs, took 50 minutes to execute. Notice that we did not include the packet length as parameter. In case of the Jacobi-algorithms, it depends on the number of antenna signals (M) used.

Parameter	Range
Bufffer Size	$\{ 4 \ldots 100 \}$
Cordics (N)	$\{ 1 \ldots 7 \}$
Problem Size (M)	$\{ 4 \ldots 12 \}$
Pipeline Depth	$\{ 1 \ldots 20 \}$

Table 9.6. Parameters used in the design space exploration of the Jacobium architecture.

Figures 9.13 shows the initiation period realized for a processing element (a) and the total execution time required by an architecture instance to process 1000 k planes (b) when a buffer size of 100 and a pipeline depth of 18 are selected. The initiation period reported is not equal to the throughput reported in Table 9.3 to Table 9.5; it gives the throughput that an individual processing element can realize. From figure (a), we concluded that processing elements operate with maximum throughput (e.g. 1 sample/cycle) on a stream of samples when a small problem size is used ($M \leq 7$). When a larger problem size is used, an architecture can maintain a throughput of 1 sample/cycle per processing element by using more processing elements. This works until a problem size larger than 9 is selected.

From figure (b), we concluded that when a small problem size is executed (e.g. $4 \leq M \leq 7$), adding extra processing elements is not efficient because the total execution time hardly decreases. When a large problem size is used (e.g. $M \geq 8$), additional processing elements are used more efficiently, especially when between 3 and 5 processing elements. Using more that 6 extra processing elements hardly improves the total execution time.



(a) Initiation Period of Processing Element

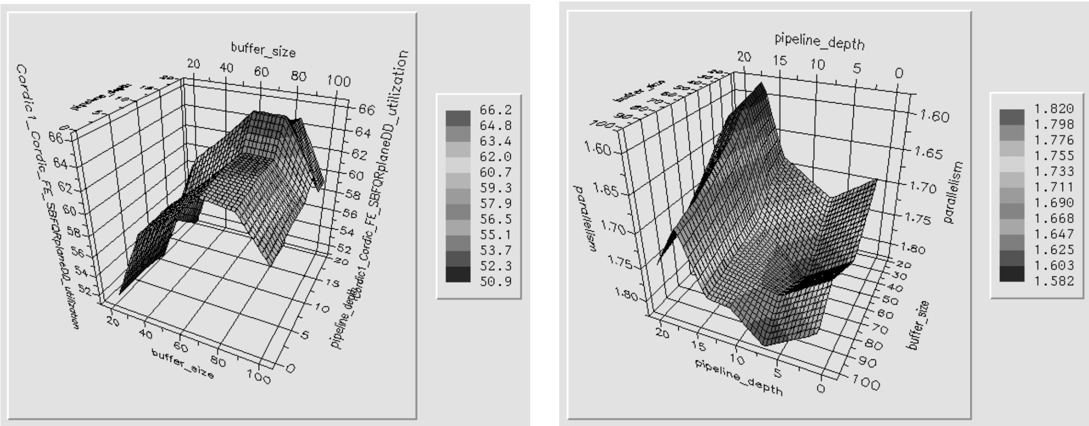
(b) Total Execution Time

Figure 9.13. The response surface model for the initiation period achieved for one processing element (a) or the response surface model for total execution time required to process 1000 k-planes (b) as a function of processing elements N and problem size M.

In Figure 9.14, we show the utilization of a processing element (a) and the achieved level of parallelism of architecture instances for different levels of pipelining and buffer sizes. In both figures, the problem size is equal to 12 and the architecture instances use 4 processing elements. From figure (a), we concluded that the processing elements are used most efficiently when a buffer size of approximately 70 samples is used, irrespective of the level of pipelining. From figure (b), we conclude that the level of pipelining influences very much the level of parallelism realized. For a problem size of 12 and 4 processing elements, the maximum level of parallelism is obtained when a pipeline depth of 5 stages is used. Furthermore, the deeper the pipeline, the lower the level of parallelism.

9.3.6 Conclusions

The Jacobium project is still running, but we can draw the following preliminary conclusions regarding it. We were able to use the Y-chart environment developed in this thesis to model the architecture of the Jacobium processor in terms of the architecture template of stream-based dataflow architectures. We extended this Y-chart environment, which allowed us to execute applications written in Matlab as a network of SBF objects on an architecture instance and determine how these networks



(a) Utilization

(b) Parallelism

Figure 9.14. The response surface model for utilization of a processing element (a) or the response surface model for realized level of parallelism (b) as a function of level of pipelining and buffer sizes.

performed on architecture instances. We used the extended Y-chart to quantify characteristics of three different partitions that executes one and the same QR algorithm, but with different granularity. We also quantified the effects of pipelining and global communication. For one partition, we performed a design space exploration and showed the we were able to derive relationships between architecture parameters and performance metrics.

Bibliography

P.M. Dewilde, A.A.J de Lange, A.J van der Hoeven, and E.F. Deprettere. HiFi: An object oriented system for the structural synthesis of signal processing algorithms and the VLSI compilation of signal-flow graphs. In *IFIP Int. Workshop on Applied Formal Methods for Correct VLSI Design*, 1989.

Robert P. Dick, David L. Rhodes, and Wayne Wolf. TGFF: Task graphs for free. In *Proceedings of 6th Int. Workshop on Hardware/Software Codesign*, Seattle, Washington, 1998.

Giuseppe Garcea. Derivation of dataflow networks for a specific domain of applications. Master's thesis, Delft University of Technology, Circuits & Systems Group, 1998.

G.H. Golub and C.F. Van Loan. *Matrix Computations, (2nd ed.)*. John Hopkins University Press, 1989.

Gerben J. Hekstra and Ed F. Deprettere. Floating point Cordic. In *Proceedings ARITH 11*, pages 130–137, Windsor, Ontario, 1993.

Peter Held. *Functional Design of Dataflow Networks*. PhD thesis, Delft University of Technology, 1996.

R. Jain. Personal Communication with Ed Deprettere, 1997.

J.H. Justice, N.L Owsley, J.L Jen, and A.C. Kak. *Array Signal Processing*. Prentice-Hall, 1985.

Jeroen Leijten. *Real-Time Constrained Reconfigurable Communication between Embedded Processors*. PhD thesis, Eindhoven University of Technology, 1998.

Jeroen A.J. Leijten, Jef L. van Meerbergen, Adwin H. Timmer, and Jochen A.G. Jess. Prophid: A heterogeneous multi-processor architecture for multimedia. In *Proceedings of ICCD'97*, 1997.

A. Looye, G. Hekstra, and E. Deprettere. Multiport memory and floating point cordic pipeline in Jacobium processing elements. In *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS'98)*, Boston, USA, 1998.

Boaz Porat. *Digital Processing of Random Signals*. Prentice-Hall, 1993.

Edwin Rijpkema, Gerben Hekstra, Ed Deprettere, and Ju Ma. A strategy for determining a Jacobi specific dataflow processor. In *Proceedings of 11th Int. Conference of Applications-specific Systems, Architectures and Processors (ASAP'97)*, pages 53 – 64, Zurich, Switzerland, 1997.

H.W. van Dijk and E.F. Deprettere. Transformational reasoning on time-adaptive Jacobi type algorithms. In M. Moonen and B. De Moor, editors, *SVD and Signal Processing, III*, pages 277–286. Kluwer Academic Publishers, Dordrecht, 1995.

Hylke van Dijk, Gerben Hekstra, and Ed Deprettere. On the systematic exploration of the space of Jacobi algorithms. Technical Report ET/NT/Fact-2, Dept. Electrical Engineering, Delft University of Technology, 1993.

B. Van Veen and K. Buckley. Beam forming: A versatile approach to spatial filtering. *IEEE ASSP Magazine*, 5(2):4–24, 1988.

J.E. Volder. The Cordic trigonometric computing technique. *IRE Trans. Electronic Computers*, EC-8 (3):330–340, 1959.

Bob Witlox. Performance analysis of dataflow architectures using petri nets. Master's thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1997.

Chapter 10

Summary & Conclusions

THERE is a trend toward increasingly programmable application-specific architectures. These architectures are becoming increasingly programmable to support multi-functional and multi-standard products. In the design of these architectures, it is no longer the performance of a single application that matters, but the performance of a set of applications. We have observed that general and structured approaches are lacking for the design of these increasingly programmable application-specific architectures.

We presented the stream-based dataflow architectures as an example of programmable application-specific architectures. We showed that this class of architectures fits into the category of dataflow architectures and that they represent interesting programmable architectures for high-performance, stream-based applications that are found in many multi-media applications.

In the design of stream-based dataflow architectures many architectural choices are involved, each leading to a particular architecture with a specific behavior and performance. For designers, it becomes increasingly complex to find feasible architectures, given the many choices involved and the required programmability.

We structured the many design choices by means of an architecture template which describes a class of architectures in a parameterized way and which has a well-defined design space. We considered the central problem in this thesis to be to provide a method to help designers find parameter values of the architecture template that result in a feasible design, given an architecture template, a set of applications, and design objectives like throughput and utilization as well as power and silicon area.

The methodology that we presented in this thesis is the Y-chart approach shown in Figure 10.1. It is a methodology in which designers make decisions and motivate particular design choices based on quantitative data. We showed that the Y-chart approach leads to a Y-chart environment which allows designers to quantify design choices for a particular architecture. By using such a Y-chart environment and changing design choices in a systematic way, designers explore the design space of a given architecture template for a set of applications. Furthermore, designers narrow down the design space of an architecture template in a stepwise fashion by using a stack of Y-chart environments, each at a different level of abstraction.

By exploring the design space at different abstraction levels, designers gain insight into the relationship between parameter values and the performance that is achieved. Based on this insight, designers make trade-offs between the many parameters leading to a feasible design. Therefore, the Y-chart approach results in better-engineered architectures in less time. These engineered architectures satisfy the imposed design objectives for the complete set of applications. Moreover, they use

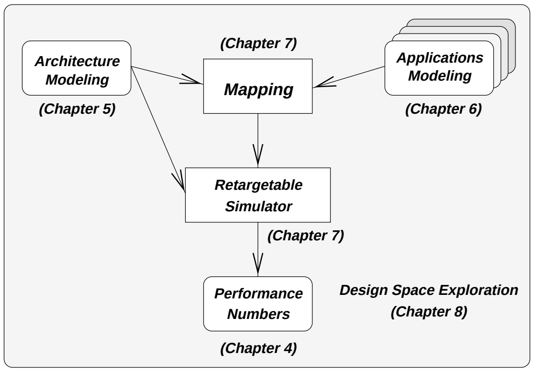


Figure 10.1. The Y-chart environment developed in this thesis

their resources more efficiently, resulting in a less expensive design in terms of the amount of silicon used.

In this thesis we realized a Y-chart environment for the class of stream-based dataflow architectures at a cycle-accurate level. We implemented the six components constituting a Y-chart environment. This included the use of a high-level performance analysis method, an architecture modeling approach, an application modeling approach, the construction of a retargetable simulator and a mapping approach, and the use of a design space exploration environment.

We used the Y-chart environment in two design cases and showed that designers can indeed compare alternatives in a quantitative way. The two design cases also showed that designers can indeed perform design space exploration, permitting them to consider trade-offs between parameter values and performance. Although the results found in the two cases will not directly affect the further development of the two designs, they clearly showed that we gained insight into the complex trade-off present. Similar results are very difficult to obtain using current design approaches. Finally, the two design cases showed that it was possible to analyze and change architectures at a high level of abstraction.

10.1 Discussion of the Y-chart Approach

Based on our use of the Y-chart environment in two design cases, we draw the following conclusions. First, we conclude that setting up a Y-chart requires significant effort. Secondly, we conclude that the Y-chart approach exhibits interesting general characteristics. Thirdly, we conclude that the Y-chart approach structures the design process of application-specific programmable architectures. Fourthly, we conclude that the Y-chart approach requires multi-disciplinary teams that challenge current design approaches.

10.1.1 Significant Effort

The Y-chart environments used in both the Jacobium and the Prophid cases are a direct result of a significant amount of time spent on modeling and developing of software. To realize the Y-chart environment, we developed the tools *SBFsim* and *ORAS*, which consist of a total of respectively 3,000 and 20,000 lines of C++, not taking into account the modeling of applications.

Although setting up a Y-chart environment requires significant effort, we were able to realize the complete Y-chart environment in slightly more than one and a half years because we (re)used existing tools. We used the PAMELA Run-time Library, which provided us with high-level primitives that made it possible for us to model stream-based dataflow architectures at a high level of abstraction. Another existing tool that we used was the generic DSE environment. This environment took care of setting up experiments efficiently, data management, and visualization of results, e.g., drawing the response surface models.

10.1.2 Characteristics of the Y-chart Approach

We conclude that the Y-chart exhibits general characteristics that are invariant with respect to the architecture or application researched. We summarize the characteristics as:

1. The Y-chart approach permits designers to quantify architectural choices, providing a sound basis on which design choices can be made.
2. The Y-chart approach permits designers to explore the design space of an architecture template not only for a single application, but for the whole set of applications.
3. The Y-chart approach allows designers to consider trade-offs and to balance between programmability and efficiency, finding an architecture instance that obeys set-wide design objectives.
4. The Y-chart approach takes into account the complete system – architecture, the set of applications, and the mapping of these applications – thus optimizing the complete system instead of only the architecture.
5. The Y-chart approach is invariant for a specific level of detail at which designers need to specify architecture instances when evaluating the architecture instance to render performance numbers.
6. The Y-chart approach requires that the architecture template be made explicit and thus that architecture templates can be reused in other design projects.

10.1.3 Structuring the Design Process

We conclude that the Y-chart approach structures the design process of programmable application-specific architectures. It stipulates the requirements that need to be satisfied in order to perform design space exploration of architectures.

In both the Jacobium and Prophid cases, we conclude that too much focus was placed on the architectural design. As a consequence, we experienced severe problems in obtaining specifications of applications in the appropriate format. The main reason these problems occurred was that developing applications in the correct format was not an integral part of the design case of either the Prophid or the Jacobium architectures.

In further development of programmable architectures, much more attention must be given to developing applications in a suitable format, for example, by making the release of applications in the appropriate format a clear deliverable within design processes. The development of libraries containing high-quality applications descriptions should be considered. This implies that applications should be documented, well-structured (for example, by using code reviews), and released and maintained over time by a person having the responsibility for particular applications.

A problem similar to the applications problem is mapping. Again, in both the Jacobium and the Prophid cases, mapping was not addressed properly since it was not an integral part of the design case.

The Y-chart approach, however, clearly states that designing programmable architectures requires that the architecture, the set of applications, and the mapping be addressed before any exploration takes place.

The Y-chart approach has already made an impact within Philips Research. There are design projects in which the Y-chart approach is used as the basis to structure the design process of new programmable architectures. In these design projects, the availability of benchmark applications and mapping were included from the beginning. In the near future, we expect that Y-chart environments will be realized for these design projects. For CPU design, and the Philips TriMedia processor in particular, the exploration has already been performed. For hardware/software codesign that includes CPUs and coprocessors, exploration will soon take place.

10.1.4 Multi-Disciplinary Teams

As shown in Figure 10.1, the Y-chart approach consists of six components, each of which represents a discipline on its own. The Y-chart approach brings these disciplines together from the very beginning of the design of a programmable architecture. Moreover, the Y-chart does not consider these disciplines in isolation, but clearly indicates how the various disciplines depend on each other and how they operate concurrently within the same design.

Setting up a Y-chart for a programmable architecture therefore requires that all six disciplines be mastered. This thesis shows that it is possible to integrate all these disciplines by constructing a Y-chart environment for stream-based dataflow architectures at the cycle-accurate level of abstraction. It also shows how one component representing a particular discipline influences the development of other components. For example, SBF Objects used in application modeling are designed in such a way that they can seamlessly integrate with functional elements to realize fully functional simulations. Therefore, we combined architecture modeling with application modeling.

Currently, the design of new general purpose processors require very large multi-disciplinary teams. It is not unusual in this area to have design teams of 100 to 300 people [Wilson, 1998] representing many different disciplines that interact with each other at different levels of abstraction at the same time or at least at overlapping times. Managing these interactions is becoming increasingly important and more difficult as designs become more complex [Hennessy and Heinrich, 1996]. In addition, less time is available for new designs because of time-to-market considerations. Similar observations will apply to the use of the Y-chart. Therefore, we have to establish clearly defined interfaces between the various components of a Y-chart environment as well as between components used in a stack of Y-chart environments.

Finally, we remark that, in our opinion, the way application-specific programmable architectures are designed should change radically. As pointed out above, the Y-chart approach combines different disciplines. Therefore, to get the full benefit of the Y-chart approach, such that we obtain better-engineered architectures, tightly integrated multi-disciplinary teams are required instead of the different teams of different disciplines as happens currently. An excellent example of how such tightly integrated multi-disciplinary teams should operate is described by Conklin [1996], who describes the design process of the DEC Alpha general purpose processor.

10.2 Further Research

We see many opportunities for further research, including:

More heterogeneous architectures We developed the Y-chart environment for stream-based dataflow architectures. An interesting line of research would be to investigate how techniques developed within this thesis can be used or be extended for more heterogeneous architectures that contain both dataflow and control concepts, possibly with different performance constraints (see, for example, Figure 1.3). Lieverse et al. [1998] have already shown that such extentions are possible. They used again the PAMELA run-time library for application modeling, but additionally, they used TSS (see Section 7.9) for architecture modeling.

Generalizing architecture modeling concepts We used the run-time library primitives and the building block approach to construct architecture instances that perform simulations quickly and efficiently. However, the run-time library works very well for stream-based dataflow architectures, but lacks the capability to describe *polling* efficiently (as indicated in Appendix C). Further research could determine how the run-time library could be extended such that it can describe polling, while remaining fast and efficient. Some suggestions for this research were presented in Section 7.9.

Presentation of multi-dimensional relationships The response surface models generated by the generic DSE environment presents 3-D graphs, while the response surface models actually represent higher-dimensional relationships. Further research could involve investigating how to present higher-dimensional relationships such that designers can more effectively make trade-offs. In Section 8.6, we already indicated that *scatter-plots* are an option.

Optimization Designers themselves inspect the Response Surface Models generated by the design space exploration. Further research might involve replacing visual inspection by integrated optimization tools within the generic DSE environment (i.e., replacing the “lightbulb” in Figure 8.1). These optimization tools would have to find feasible architectures using a minimum number of iterations, since each iteration in a Y-chart environment generally takes a considerable amount of time.

Use of the Y-chart approach in a real design by real designers The Y-chart environment used in the two cases proved the correctness of concept of the Y-chart approach. However, actual designers have not used it. Further research might include working with real designers using Y-chart environments and studying what kind of questions or difficulties arise at which levels.

Translation from Matlab to the SBF model In the Jacobium case, the translation from a Matlab application into a description in terms of the SBF model is not yet automated. Further research could be to automate the steps involved in the translation. This way a large pool of benchmark applications could easily be created, because many applications are already specified in Matlab. Rijpkema et al. [1998] have already identified the steps involved in automating this translation for piecewise regular algorithms. The partition of applications (see, for example, Figure 9.12) should also be automated by providing tools that create the correct SBF objects for a given partition.

10.3 Availability of Software

The source code of the software developed in this thesis, i.e., SBFsim and ORAS are freely available for further use. The software can be obtained by accessing address.

`http://cas.et.tudelft.nl/research/hse.html`.

Bibliography

Peter F. Conklin. Enrollment management: Managing the Alpha AXP program. *IEEE Software*, 13 (4):53 – 64, 1996. Reprint from Digital Technical Journal, 1992.

John Hennessy and Mark Heinrich. Hardware/software codesign of processors: Concepts and examples. In Giovanni De Micheli and Mariagiovanna Sami, editors, *Hardware/Software Codesign*, volume 310 of *Series E: Applied Sciences*, pages 29 – 44. NATO ASI Series, 1996.

Paul Lieverse, Pieter van der Wolf, Ed Deprettere, and Kees Vissers. Methodology for architecture exploration of heterogeneous systems. In *Proceedings of the ProRISC/IEEE Workshop on Circuits, Systems and Signal Processing*, 1998.

Edwin Rijpkema, Bart Kienhuis, and Ed Deprettere. From piecewise regular algorithms to process networks. In *Proceedings of the ProRISC/IEEE Workshop on Circuits, Systems and Signal Processing*, 1998.

Janet Wilson. Challenges and trends in processor design. *IEEE Computer*, 31(1):39 – 48, 1998.

Appendix A

Architecture Template in BNF

TO describe the Architecture Template of Stream-based Dataflow architectures, we use composition rules that state which Architectural Element Types are allowed to connect with each other and to which extend. To describe these composition rules we use the Backus-Naur Form (BNF) [Backus and Naur, 1959]. The complete Architecture Template is given below. See Section 5.5 for more information on describing an Architecture Template using composition rules in BNF.

```
configuration   :       ARCHITECTURE ID ’{’
                        controller
                        communication
                        peList
                        ’}’
                ;


controller      :       CONTROLLER ’{’ TYPE ’:’ FCFS ’(’ NUM ’,’ NUM ’)’ ’;’ ’}’
                |       CONTROLLER ’{’ TYPE ’:’ PHASE ’(’ NUM ’,’ NUM ’,’ ID ’)’ ’;’ ’}’
                |       CONTROLLER ’{’ TYPE ’:’ ROUNDROBIN ’(’ NUM ’,’ NUM ’)’ ’;’ ’}’
                ;


communication   :       COMMUNICATION ’{’ TYPE ’:’ SWITCH ’(’ NUM ’)’ ’;’ ’}’
                        ;

peList          :       aProcessingElement
                |       peList aProcessingElement
                ;

aProcessingElement:     PE ID ’(’ NUM ’,’ NUM ’)’
                        ’{’ body ’}’
                ;

body            :       inputBuffers
                        outputBuffers
                        router
                        functionalUnit
                ;

inputBuffers    :
                |       INPUTBUFFER ’{’ inputBody ’}’
                ;
```

```
inputBody       :       TYPE ':' BOUNDEDFIFO '(' NUM ')' ';'
                |       TYPE ':' UNBOUNDEDFIFO ';'
                ;

outputBuffers   :
                |       OUTPUTBUFFER '{' outputBody  '}'
                ;

outputBody      :       TYPE ':' BOUNDEDFIFO '(' NUM ')' ';'
                |       TYPE ':' UNBOUNDEDFIFO ';'
                ;

router          :
                |       ROUTER '{' TYPE ':' FCFS ';' '}'
                |       ROUTER '{' TYPE ':' PHASE ';' '}'
                |       ROUTER '{' TYPE ':' ROUNDROBIN ';' '}'
                ;

functionalUnit  :       fuItem feList '}'
                ;

fuItem          :       FU '{' TYPE ':' PACKET ';'
                |       FU '{' TYPE ':' SAMPLE ';'
                |       SOURCE '{' TYPE ':' BURST identifier ';'
                |       SOURCE '{' TYPE ':' STREAM identifier ';'
                |       SINK '{' TYPE ':' BURST identifier ';'
                |       SINK '{' TYPE ':' STREAM identifier ';'
                ;

feList          :       functionalElement
                |       feList functionalElement
                |       error
                ;

functionalElement:      FE ID '(' NUM ',' NUM ')' '[' NUM ']'
                        '{' feBody feType binding '}'
                |       error
                ;

feBody          :
                |       TYPE ':' SYNCHRONE ';'
                |       TYPE ':' ASYNCHRONE ';'
                |       TYPE ':' MEMORY identifier ';'
                |       error
                ;

feType          :       FUNCTION '{' TYPE ':' ID ';' '}'
                |       FUNCTION '{' TYPE ':' ID identifier ';' '}'
                ;

binding         :       BINDING
                ;

bindingRelations:        bindingInputs bindingOutputs
                ;

bindingInputs   :
```

```
                |       INPUT ’(’ inRelationList ’)’ ’;’
                ;

bindingOutputs  :
                |       OUTPUT ’(’ outRelationList ’)’ ’;’
                ;

inRelationList  :       inRelation
                |       inRelationList ’,’ inRelation
                ;

inRelation      :       NUM ARROW NUM
                ;

outRelationList :       outRelation
                |       outRelationList ’,’ outRelation
                ;

outRelation     :       NUM ARROW NUM
                ;

identifier      :
                |       ’(’ parameterList ’)’
                ;

parameterList   :       parameterItem
                |       parameterList ’,’ parameterItem
                ;

parameterItem   :       ID ’=’ NUM
                |       ID ’=’ ID
                ;
```

Appendix B

Picture in Picture Example

AN example of a Stream-based video application used in modern high-end TV-sets is shown in Figure B.1. It describes the *Picture in Picture* (PiP) algorithm which reduces a picture to half its size in both horizontal and vertical direction and places the reduced picture onto a full screen picture showing two images on a TV screen [Janssen et al., 1997]. The Picture in Picture application is discussed in Chapter 6.

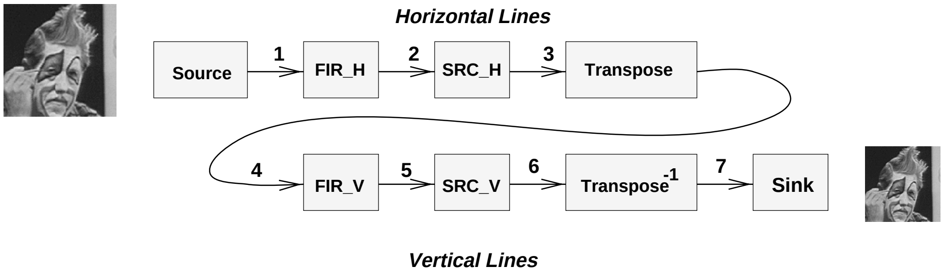


Figure B.1. The Picture in Picture Application

In the PiP example, a Source produces an infinite stream of video samples that are filtered by an N-taps Finite Impulse Response (FIR_H) Filter. Then the stream is passed through a Sample-rate Converter (SRC_H) that performs a down sampling of a factor two. Next, video images are transposed (Transpose): re-ordering samples in such a way that two consecutive samples belong to two different video lines. The stream then passes again through an N-taps FIR filter (FIR_V) and a SRC (SRC_V), this time to perform a vertical down sampling of a factor two. On the stream that results, the second transpose function ($\mathsf{Transpose}^{-1}$) performs a re-ordering such that consecutive samples now belonging to the same video line. Finally, the Sink consumes the samples.

B.1 One-to-one Architecture Instance

The architecture instance is given in this section. The Architecture Instance is able to execute the Picture in Picture (PiP) application shown in Figure B.1. We defined an architecture instance with eight Functional Units, as shown in Figure B.2. The eight Functional Units have a *one-to-one* mapping of a function from Figure 6.1 to its function repertoire such that function repertoire of each Functional Unit consists of one function.

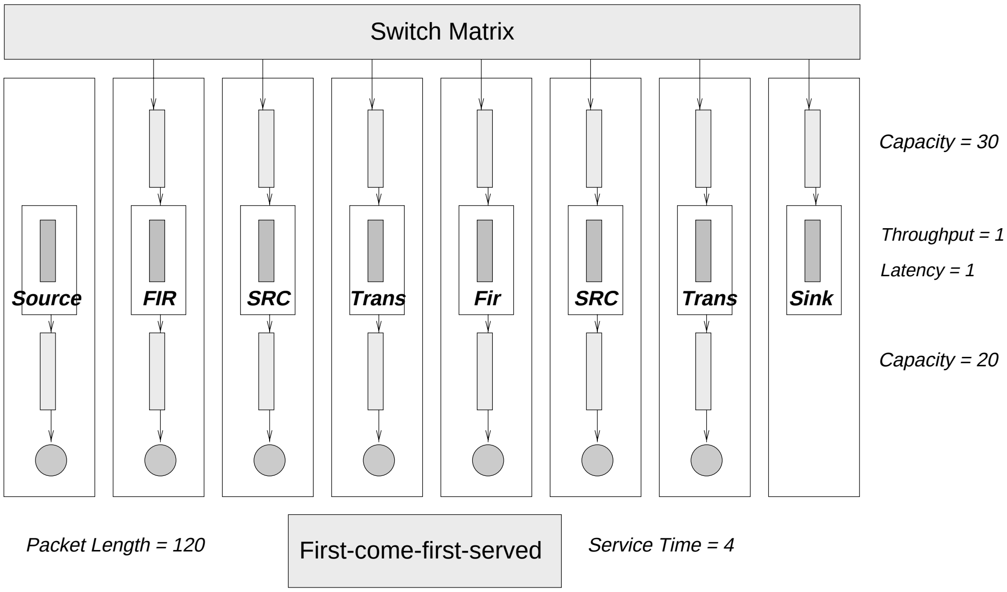


Figure B.2. The Architecture Instance for a One-to-one Mapping of the Picture in Picture application

B.1.1 Architecture Description

```
Architecture Dataflow {

   GlobalControl {
      PacketLength = 120 ;
   }

   Controller { Type: Fcfs( 1, 4 ); }
   Communication { Type: SwitchMatrix( 8 ); }

   ProcessingElement Source(0,1) {
      OutputBuffer { Type: BoundedFifo( 30 ); }
      Router { Type: Fcfs; }
      SourceUnit {
         Type: Burst(packets=760,base=1);
         FunctionalElement Source(0,1)[1] {
            Function { Type: PGMsource(initiation_period=2,latency=2); }
            Binding {
               Output ( 0->0 );
            }
         }
      }
   }
   ProcessingElement FIR_H(1,1) {
      InputBuffer { Type: BoundedFifo( 30 ); }
      OutputBuffer { Type: BoundedFifo( 20 ); }
      Router { Type: Fcfs; }
      FunctionalUnit {
         Type: Packet;
         FunctionalElement FIR_H(1,1)[1] {
            Type: Synchrone;
            Function { Type: FirFilter(initiation_period=1,latency=18); }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
            }
         }
      }
   }
   ProcessingElement SRC_H(1,1) {
      InputBuffer { Type: BoundedFifo( 30 ); }
      OutputBuffer { Type: BoundedFifo( 20 ); }
      Router { Type: Fcfs; }
      FunctionalUnit {
         Type: Packet;
         FunctionalElement SRC_H(1,1)[1] {
            Type: Synchrone;
            Function { Type: DownSample(initiation_period=1,latency=1,factor=2); }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
            }
         }
     }
   }
   ProcessingElement TRANSPOSE(1,1) {
      InputBuffer { Type: BoundedFifo( 30 ); }
      OutputBuffer { Type: BoundedFifo( 20 ); }
      Router { Type: Fcfs; }
      FunctionalUnit {
         Type: Packet;
         FunctionalElement TRANSPOSE(1,1)[1] {
            Type: Asynchrone;
            Function { Type: Transpose(initiation_period=1,latency=1,line=60); }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
            }
        }
      }
   }
   ProcessingElement FIR_V(1,1) {
      InputBuffer { Type: BoundedFifo( 30 ); }
      OutputBuffer { Type: BoundedFifo( 20 ); }
      Router { Type: Fcfs; }
      FunctionalUnit {
         Type: Packet;
         FunctionalElement FIR_V(1,1)[1] {
            Type: Synchrone;
            Function { Type: FirFilter(initiation_period=1,latency=18); }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
            }
         }
     }
   }
   ProcessingElement SRC_V(1,1) {
      InputBuffer { Type: BoundedFifo( 30 ); }
      OutputBuffer { Type: BoundedFifo( 20 ); }
      Router { Type: Fcfs; }
      FunctionalUnit {
         Type: Packet;
         FunctionalElement SRC_V(1,1)[1] {
            Type: Synchrone;
            Function { Type: DownSample(initiation_period=1,latency=1,factor=2); }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
            }
         }
      }
   }
   ProcessingElement TRANSPOSE_INV(1,1) {
      InputBuffer { Type: BoundedFifo( 30 ); }
      OutputBuffer { Type: BoundedFifo( 20 ); }
      Router { Type: Fcfs; }
      FunctionalUnit {
         Type: Packet;
         FunctionalElement TRANSPOSE_INV(1,1)[1] {
            Type: Asynchrone;
            Function { Type: TransposeInv(initiation_period=1,latency=1,step=5,line=60); }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
            }
         }
      }
   }
   ProcessingElement Sink(1,0) {
      InputBuffer { Type: BoundedFifo( 30 ); }
      SinkUnit {
         Type: Burst(packets=120);
         FunctionalElement Sink(1,0)[1] {
            Function { Type: PGMsink(initiation_period=2,latency=2,height=60,width=60); }
            Binding {
               Input ( 0->0 );
            }
         }
     }
   }
}
```

B.1.2 Mapping

```
}
Mapping test {
   1: 2,  0, FIR_H_inBuffer_0;
   2: 3,  0, SRC_H_inBuffer_0;
   3: 4,  0, TRANSPOSE_inBuffer_0;
   4: 5,  0, FIR_V_inBuffer_0;
   5: 6,  0, SRC_V_inBuffer_0;
   6: 7,  0, TRANSPOSE_inBuffer_0;
   7: -1, 0, Sink_inBuffer_0;       /* Dummy Buffer */
}
```

B.2 Many-to-one Architecture Instance

The Stream-based Dataflow architecture also permits the sharing of Functional Units between different streams: *many-to-one* mappings. More than one application function can map to a Functional Unit. In Figure B.3, we show an architecture instance that also can execute the Picture in Picture application of Figure B.1 but uses sharing of Functional Units. In this architecture instance, each Functional Unit has a function repertoire of two Functional Elements each executing identical functions (e.g., Sample Rate Conversion (SRC_H) and (SRC_V)).

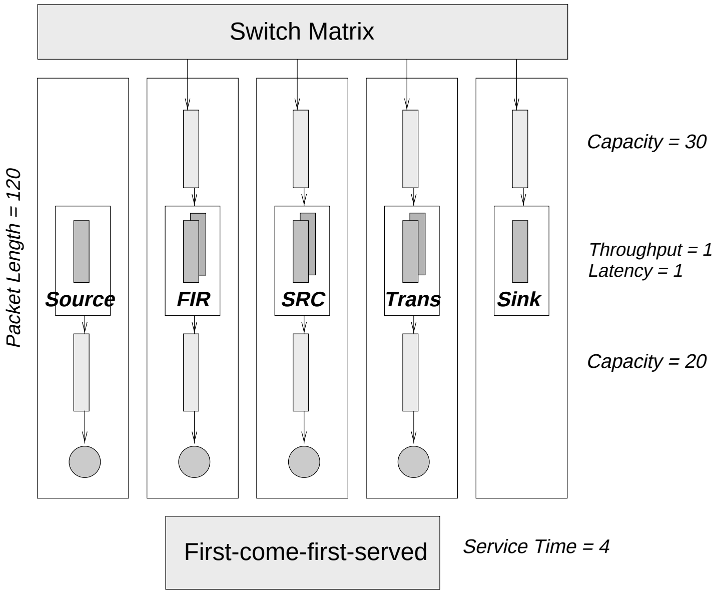


Figure B.3. The Architecture Instance for a Many-to-one Mapping of the Picture in Picture application

B.2.1 Architecture Description

```
Architecture Dataflow {

   GlobalControl {
      PacketLength = 120 ;
   }

   Controller { Type: Fcfs( 1, 4 ); }
   Communication { Type: SwitchMatrix( 8 ); }

   ProcessingElement Source(0,1) {
      OutputBuffer { Type: BoundedFifo( 30 ); }
      Router { Type: Fcfs; }
      SourceUnit {
         Type: Burst(packets=760,base=1);
         FunctionalElement Source(0,1)[1] {
            Function { Type: PGMsource(initiation_period=2,latency=2); }
            Binding {
               Output ( 0->0 );
            }
         }
      }
   }
   ProcessingElement FIR(1,1) {
      InputBuffer { Type: BoundedFifo( 30 ); }
      OutputBuffer { Type: BoundedFifo( 20 ); }
      Router { Type: Fcfs; }
      FunctionalUnit {
         Type: Packet;
         FunctionalElement FIR_H(1,1)[1] {
            Type: Synchrone;
            Function { Type: FirFilter(initiation_period=1,latency=18); }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
            }
         }
         FunctionalElement FIR_V(1,1)[1] {
            Type: Synchrone;
            Function { Type: FirFilter(initiation_period=1,latency=18); }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
            }
         }
      }
   }
   ProcessingElement SRC(1,1) {
      InputBuffer { Type: BoundedFifo( 30 ); }
      OutputBuffer { Type: BoundedFifo( 20 ); }
      Router { Type: Fcfs; }
      FunctionalUnit {
         Type: Packet;
         FunctionalElement SRC_H(1,1)[1] {
            Type: Synchrone;
            Function { Type: DownSample(initiation_period=1,latency=1,factor=2); }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
            }
         }
         FunctionalElement SRC_V(1,1)[1] {
            Type: Synchrone;
            Function { Type: DownSample(initiation_period=1,latency=1,factor=2); }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
            }
         }
     }
   }
   ProcessingElement TRANSPOSE(1,1) {
      InputBuffer { Type: BoundedFifo( 30 ); }
      OutputBuffer { Type: BoundedFifo( 20 ); }
      Router { Type: Fcfs; }
      FunctionalUnit {
         Type: Packet;
         FunctionalElement TRANSPOSE(1,1)[1] {
            Type: Asynchrone;
            Function { Type: Transpose(initiation_period=1,latency=1,line=60); }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
            }
        }
         FunctionalElement TRANSPOSE_INV(1,1)[1] {
            Type: Asynchrone;
            Function { Type: Transpose(initiation_period=1,latency=1,step=5,line=60); }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
            }
         }
      }
   }
   ProcessingElement Sink(1,0) {
      InputBuffer { Type: BoundedFifo( 30 ); }
      SinkUnit {
         Type: Burst(packets=120);
         FunctionalElement Sink(1,0)[1] {
            Function { Type: PGMsink(initiation_period=2,latency=2,height=60,width=60); }
            Binding {
               Input ( 0->0 );
            }
         }
     }
   }
}
```

B.2.2 Mapping

```
}
Mapping test {
   1:  2, 0, FIR_inBuffer_0;
   2:  3, 0, SRC_inBuffer_0;
   3:  4, 0, TRANSPOSE_inBuffer_0;
   4:  5, 1, FIR_inBuffer_0;
   5:  6, 1, SRC_inBuffer_0;
   6:  7, 1, TRANSPOSE_inBuffer_0;
   7: -1, 0, Sink_inBuffer_0;       /* Dummy Buffer */
}
```

Bibliography

Johan G.W.M. Janssen, Jeroen H. Stessen, and Peter H.N. de With. An advanced sampling rate conversion algorithm for video and graphics signals. In *IEE Sixth International Conference on Image Processing and its Applications*, Dublin, 1997.

Appendix C

Limitations of the RTL

THE RTL describes asynchronous communication between processes very efficiently. It is, however, less efficient when describing synchronous communication and control behavior like *polling*, which potentially limits the usability of the RTL. The reason why the RTL cannot describe polling efficiently is that it lacks the notion of the *state of the system*.

C.1 State of the System

When we use the RTL, we cannot determine the state of the system at a particular time instance. In Figure 4.7, the process shown is running at time instance t=5 and it observes the system while there are still processes in the RPQ that need to execute at the same time. One of these processes could change the value of a variable at t=5. Hence, when the running process observes a variable in another process without making use of **pam_P** and **pam_V** (i.e, when it does not explicitly synchronize with that process), it is not able to determine whether the observed variable still holds the old value or whether it has already acquired a new value.

C.1.1 Polling

When the notion of the state of the system is lacking, it is difficult to describe *polling*. In polling, a process checks a condition and performs a particular action based on this condition. When polling is used, a process controls the progress of another process without synchronization. Therefore, one process cannot make another process block.

The need for polling arises when we want to describe, for example, the time division multiplexed policy of the communication structure (see Section 2.1.3 where we explain the behavior of a TDM communication protocol). The communication structure divides the bandwidth of each channel into N time slots of x cycles. All routers are pre-assigned a time slot on a channel that they can use to send a packet or part of a packet to input buffers of functional units. The global controller is responsible for routers being enabled when their time slot is available.

To describe this enabling process, we model the global controller as an active element instead of a passive element as is done in Program 5.3. The global controller could use **pam_V** to activate the appropriate routers for a time slot, but when it wants to reclaim control because the time slot has finished, it cannot simply use **pam_P**. A router that is blocked, for example, because it wanted to read a token from an empty output buffer, also causes the global controller to block and the global controller cannot reclaim control of all routers at a predetermined time instance. Therefore, the global controller resorts to polling to control the routers.

C.1.2 Priority Scheduling

To circumvent the limitation of not having a state of the system, the RTL process scheduler should schedule processes based on *priority*. It should takes processes with the same time stamp from the RPQ starting at the highest priority down to the lowest priority. Using priority, we can establish a dependency relation between processes within the same time instance, without using semaphores. Using priorities, we can instruct the process scheduler that it should execute process B before process A, by giving process B a higher priority than process A.

The current implementation of the RTL does not provide priority-based scheduling. If we need some kind of priority scheduling, we can make use of the fact the RTL represents time as a real number instead of an integer. By running some processes only on integer time instances, i.e., $t = \{1, 2, 3, \ldots\}$ while other processes run at time instances $t = \{1.5, 2.5, 3.5, \ldots\}$, a simple 2-level priority scheduling mechanism is realized within the RTL. Although it is not a very elegant method, it does provide a level of priority scheduling. This priority scheduling mechanism is used in ORAS to describe the behavior of the TDM global controller and router.

C.2 Implementing Polling in the RTL

By adding three new PAMELA-statements to the RTL, we have made it possible to describe polling in the RTL by using semaphores. In Program C.1, we show the three statements **pam_set**, **pam_reset** and **pam_test**. The first two statements set or reset a *binary semaphore*, giving the semaphore respectively the value *true* or *false*. The last statement, **pam_test**, performs a blocking test on the condition: if the condition is false the process blocks, otherwise the process proceeds.

Program C.1. EXTENSIONS TO THE RTL TO SUPPORT POLLING

```
void pam_test(pam_sema* aSem )
{
    pam_P ( aSem );
    pam_V ( aSem );

    /* Sync the Timing, to realize 2-level priority scheduling mechanism  */
    if ( pam_root->time != ((int) pam_root->time) ) { pam_delay (0.5);}
}
void pam_set(pam_sema* aSem)
{
    if ( pam_T (aSem) != 1 )
       pam_V ( aSem );
}
void pam_reset(pam_sema* aSem)
{
    if ( pam_T (aSem) == 1 ) {
       pam_P ( aSem );
    }
}
```

The **pam_test** statement first executes a **pam_P** on semaphore aSem, which is followed immediately by executing a **pam_V** on that same semaphore. Thus if the condition is set, the process can proceed without altering the condition. If the condition is not set, the process blocks on the **pam_P** statement. We will explain the function of the conditional time delay in statement **pam_test** when we explain the behavior of the TDM global controller in Section C.3.

The only way to change the status of a condition is by using the **pam_set** and **pam_reset** statements. These statements set and reset the condition while avoiding a process block. For that purpose, we conditionally set or reset the binary semaphore involving the **pam_T** statement to test the status of the semaphore (whether or not the semaphore is blocking).

C.3 Modeling the TDM global controller

We used the three new statements to describe the TDM behavior of the global controller. Besides a routing program, the TDM global controller also contains a *time slot assignment* table as given in Table C.1. This table shows in which time slot (i.e. time slot 1, 2 or 3) which router out of the six is allowed to send data over the communication structure. Since only two routers are active within a time slot, a communication structure with two channels would suffice.

Time Slot	Router0	Router1	Router2	Router3	Router4	Router5
1	off	on	on	off	off	off
2	off	off	off	on	on	off
3	on	off	off	off	off	on

Table C.1. Example of a time slot assignment with 3 time slots to control 6 routers

We model the TDM global controller as an active element (as described in Program C.2) while it still uses the passive method described in Program 5.3. In the **process** description of the TDM controller, we use the array `tdmTable` containing the binary semaphores that the TDM controller needs to activate in a particular time slot as given in Table C.1. These semaphores connect to the routers given in Table C.1.

In Program C.2, the **process** of the global controller activates for each time slot all the routers that need to be activated in a time slot by deactivating the routers from the previous time slot. Then the **process** gives the routers `slotLength` cycles of time to transport data over the communication structure.

Program C.2. The TDM Global Controller as Process

```
process  TDM_Global_Controller {

    // Activate the correct Routers
    foreach binary_semaphore ∈ tdmTable[timeSlot] {
        pam_set( binary_semaphore );
    }

    // Deactivate the Routers of the previous Time Slot
    timeSlot = (++timeSlot)%numberOfTimeSlots;
    foreach binary_semaphore ∈ tdmTable[timeSlot] {
        pam_reset( binary_semaphore );
    }

    // Each time slot takes slotLength cycles
    pam_delay( slotLength );
}
```

To cause the **process** to always execute after all other processes have finished at a particular time instance, we use the 2-level priority scheduling mechanism previously explained in Section C.1.2. The **process** only runs at time instances *t+0.5* whereas all other processes execute on *t*. Hence, when the global controller **process** executes, it observes the correct state of the system. Because the TDM controller sets and resets the semaphore on time steps that are non-integer values, these values propagate to the process running on the integer time steps. To prevent this from happening, we synchronize processes that use the **pam_test** statement in such a way that they run again on integer time steps. To do this, we extended the **pam_test** statement with a special conditional statement as given in Program C.1. If the time of a process is not an integer, the process is delayed 0.5 cycles. Notice however, that such tests seriously affect the execution speed in a negative way.

C.3.1 TDM Controlled Routers

A TDM controlled router can send data over the communication structure only when it is activated by the TDM controller. This requires a small adjustment with respect to the First-Come-First-Served router program given in Program C.3. The program for the TDM router given in Program 5.5 does not show the complete code, but only the part where data is written into the input buffer `bufferOut` of a functional unit. In the new program description, each `write` statement is preceded by a **pam_test** statement, which tests the condition of the semaphore set by the TDM controller. If the condition is set, the router **process** falls through the check without changing the condition. If the condition is not set, the router **process** blocks on the check until the TDM controller sets the condition.

Program C.3. PART OF THE PROCESS OF A TDM ROUTER

```
// Send out the New Header
pam_test ( semaphore );
bufferOut.write( header);
pam_delay ( 4 );
// Determine how many Samples are in a Packet
numberOfSamples = header.getlength();
// Read the rest of the packet!
for numberOfSamples do {
    aSample = bufferIn.read();
    pam_delay ( 1 );
    pam_test ( semaphore );
    bufferOut.write( aSample);
}
```

C.4 How VHDL differs from the RTL

The hardware description language VHDL [1993] is often used to describe hardware. It is particularly suited to describing low-level control statements, which makes it a popular language. VHDL supports control behavior like polling. We therefore investigate in what way the VHDL simulator engine differs from the RTL simulation engine.

A VHDL simulator implements a simulation cycle which consists of two phases [Pick, 1995]: an *action phase* and a *reaction phase*. In the action phase, the scheduler evaluates processes, thus creating new events. These new events cause the VHDL scheduler to evaluate processes that react on these events in the reaction phase. Before the reaction phase starts, all variables in the processes have adopted new values reflecting the new state of the system. Hence, VHDL describes polling very effectively.

The VHDL simulation engine uses three different queues: a *time-ordered queue*, a *signal-event-monitoring queue*, and a *process-pending queue*. The scheduler, which is part of the VHDL simulation engine, starts by evaluating processes with the same time-stamp in the time-ordered queue. Signals change during the execution of processes, causing events on the event-monitoring queue. The scheduler continues to evaluate processes until all processes with the same time-stamp have evaluated. At that time, all variables in the system have adopted new values reflecting the new state of the system. Next the VHDL scheduler wakes up processes that should react on events stored in the signal-event-monitoring queue. The awoken processes are stored in the process-pending queue. Next, the scheduler starts the reaction phase by executing all processes in the process-pending queue. These processes observe a system in which all variables reflect the new state of the system. Processes that execute in the second phase can cause new events to occur. Therefore the scheduler has to iterate a number of times, which is referred to as the VHDL δ-mechanism [Pick, 1995]. After no more events take place and all processes have executed, the reaction phase has finished and a new simulation cycle starts.

A VHDL simulation engine must perform a large amount of bookkeeping and sorting to implement the two-phase simulation cycle. This provides a large amount of flexibility, but at the expense of a considerable amount of time being spent on these activities. The RTL scheduler, on the other hand, executes a process as soon as an event (i.e. a **pam_P**, **pam_V**, or **pam_delay**) takes place, dramatically reducing the amount of bookkeeping and sorting needed. As a consequence, although it is difficult to describe polling, the RTL has an excellent simulation speed compared to the speed of VHDL simulations.

Bibliography

Joseph Pick. *VHDL, Techniques, Experiments, and Caveats*. Series on Computer Engineering. McGraw-Hill, 1995. pages 11 – 44.

VHDL. *IEEE Standard VHDL Language Reference Manual*. IEEE Computer Service, 445 Hoes Lane, P.O. Box 1331, Piscataway, New Jersey, 08855-1331, 1993. IEEE Std 1076-1993.

Index

Acknowledgments

This Ph.D. thesis is the result of research conducted mainly at Philips Research Laboratories in Eindhoven. As a Ph.D. student from the group Circuits & Systems at Delft University of Technology, I considered it a privilege to work in this inspiring research institute. I was exposed to both industrially relevant problems and to an excellent academic environment.

For this opportunity I am indebted to many people. I want to thank my promotor Patrick Dewilde for giving me the opportunity to pursue a Ph.D. within the group Circuits & Systems, an excellent group of which I am a proud member. I also want to thank Eric van Utteren en Cees Niessen, who, as group leaders of the group VLSI Design Automation and Testing, gave me the opportunity to work in their group. Finally, the partial support of Philips Research and the Ministry of Economic affairs, the Netherlands, is hereby greatly acknowledged.

I was able to work within a team that consisted of people of which I have learned at lot: Ed Deprettere, Kees Vissers and Pieter van der Wolf. I want to thank Ed Deprettere for his support as copromotor and his relentless enthusiasm to learn and understand new things; Kees Vissers for his open mind, his insights, and his standards in research; and Pieter van der Wolf for structuring my work and keeping me with both feet on the ground. I enjoyed working with the three of them and want to thank them for being excellent mentors.

During my promotion, I had the opportunity to interact with many people that I want to acknowledge. There were from many groups, both from Delft and Philips Research. First of all I want to thank all persons of the group Circuits & Systems with whom I have worked and in particular I want to thank Paul Lieverse and Daniel van Loenen from Delft, and Giuseppe Garcea from the University of Florence, Italy, who all have done a great job during their Masters assignment. I want to thank Neil Smith from UC Berkeley for his extensions to SBFsim and I want to thank Edwin Rijpkema for the discussions we had on the Jacobium Project. I also want to thank Arjan van Gemund from the CARDIT group for his help on the use of PAMELA and the PAMELA Run-time Library. From Philips Research, I want to thank my colleagues and in particular Gerben Essink, Erwin de Kock, and Wim Smits. I also want to

thank the people involved in the Prophid project: Jeroen Leijten, Adwin Timmer, and Jef van Meerbergen. Peter Bingely and Wim van der Linden helped me with the generic DSE environment, and I want to thank them for their help. I also want to thank Andre van der Avoird for the many discussions we had on our work and about working at Philips Research. I also want to thank Laurel Beecher for editing my thesis.

I want to acknowledge the fellow Ph.D. students at Philips with whom I have shared room 4.085: Hans van Gageldonk, Paul Lieverse, Bob Witlox, Robert Arendsen, and Ramon Clout.

Finally, I want to thank my family and my friends for supporting me in my Ph.D. and for continuously showing interest in my work. Special thanks are for my wife-to-be Karien, who supported me all those years, in particular, by reading work I had written and by showing understanding when again some deadline had to be met.

Bart Kienhuis,
26 November 1998,
Amsterdam.

Samenvatting

Er is momenteel een trend waarneembaar dat applicatie specifieke architecturen steeds meer programmeerbaar worden, om multifunctionele en multistandaard produkten te kunnen ondersteunen. Het kenmerk van deze nieuwe architecturen is dat ze een set van applicaties kunnen ondersteunen in plaats van één enkele applicatie. We hebben echter ondervonden dat goede, algemeen toepasbare ontwerp methoden niet voorhanden zijn voor dit soort architecturen.

Als voorbeeld van z'n programmeerbare architectuur, laten wij in dit proefschrift de *stroom gebaseerde dataflow architectuur* zien. Deze architectuur past in de categorie van dataflow architecturen en is zeer geschikt voor stroom gebaseerde, hoge prestatie applicaties die gevonden worden in bijvoorbeeld real-time multimedia toepassingen.

Bij het ontwerp van deze stroom gebaseerde dataflow architecturen moeten erg veel keuzen gemaakt worden, waarbij elke keuze leidt tot een andere architectuur, met een eigen specifiek gedrag en prestatie. Nu wordt het voor ontwerpers steeds moeilijker om die keuzen te maken zodat de steeds complexer wordende architectuur nog voldoet aan alle ontwerp eisen, waaronder programeerbaarheid.

Wij hebben de vele ontwerp keuzen gestructureerd met behulp van een *architectuur template*. Deze beschrijft de totale klasse van stroom gebaseerde architecturen met behulp van parameters en heeft dus een duidelijk begrensde ontwerp ruimte. De probleem stelling van dit proefschrift is om een methode te ontwikkelen die ontwerpers helpt bij het vinden van de parameter waarden van een architectuur template, zodanig dat een haalbare architectuur gevonden wordt in de ontwerp ruimte. Een haalbare architectuur voldoet aan de opgelegde ontwerp eisen waaronder: doorzet en utilisatie, maar ook gebruikt vermogen en silicium oppervlakte.

In dit proefschrift presenteren wij als methodiek de *Y-chart aanpak*, waarmee ontwerpers keuzen kunnen kwantificeren. De Y-chart aanpak genereert een objectieve basis voor het motiveren en maken van keuzen. De Y-chart aanpak leidt tot een *Y-chart omgeving* en wij presenteren in dit proefschrift z'n Y-chart omgeving voor de klasse van stroom gebaseerde dataflow architecturen. Een Y-chart omgeving bestaat uit 6 componenten en elke component wordt besproken. Zo bespreken we een hoog niveau methode waarmee prestatie analyse gedaan kan worden, een architectuur modelerings aanpak, een applicatie modelerings aanpak, een aanpak voor het construeren van een retargetable simulator, en een aanpak om applicaties af te beelden op een specifiek instantie van de architectuur template. Tenslotte bespreken we ook hoe we de ontwerp ruimte van architecturen op een systematische manier kunnen exploreren.

We hebben de ontwikkelde Y-chart omgeving gebruikt in twee ontwerp cases. Daaruit blijkt dat de Y-chart omgeving inderdaad ontwerp keuzen kwantificeert. Bovendien laten de twee ontwerp cases zien dan we de ontwerp ruimte van architecturen snel kunnen exploreren, op een hoog niveau van abstractie. De getallen die gepresenteerd worden, zullen niet direct tot de verdere ontwikkeling van de twee architecturen leiden, maar geven wel duidelijk aan waartoe de methode in staat is. Het geeft ontwerpers inzicht in de vele complexe afwegingen tussen parameter waarden en prestatie getallen die aanwezig zijn in dit soort programmeerbare architecturen. Soortgelijke resultaten zijn zeer moeilijk te verkrijgen met de hedendaagse ontwerp methodes; en als ze te verkrijgen zijn heeft dat veel meer ontwerp- en simulatie tijd gekost.

Curriculum Vitae

Bart Kienhuis was born on August 9, 1968 in Vleuten, The Netherlands. After finishing his HAVO degree at the “St Gregorius” Community School in 1986, he received his VWO degree from Community School “De Klop” in 1987. In that same year, he started his study Electrical Engineering at Delft University of Technology. As a student from the Circuits & Systems group, he did his M.Sc thesis on the *HiPars* compiler that converts nested loop programs into parallel programs. After he graduated in 1994, he started to work towards his Ph.D., again as a student of the Circuits & Systems group and spent almost all of his research time at Philips Research Laboratories in Eindhoven, working on a joint project.

From February 1st, 1999, Bart has accepted a Post-Doc position at UC Berkeley within the group of Professor Edward A. Lee.

DIMES
Delft Institute of Microelectronics
and Submicron Technology