Execution Platform Modeling for System-Level Architecture Performance Analysis

Vladimir Dobrosav Živković

Execution Platform Modeling for System-Level Architecture Performance Analysis

PROEFSCHRIFT

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus prof.mr. P.F. van der Heijden,
volgens besluit van het College voor Promoties te verdedigen op
dinsdag 23 September 2008
klokke 15.00 uur

door

Vladimir Dobrosav Živković
geboren te Aleksinac, Servië
in 1970

The work in this thesis was carried out between 2000 and 2004 in the Archer project supported by Philips Semiconductors (now NXP Semiconductors).

Execution Platform Modeling for System-Level Architecture Performance Analysis
Vladimir Dobrosav Živković. -
Thesis Universiteit Leiden. - With index, ref. - With summary in Dutch
ISBN/EAN: 978-90-9023450-2

Text editing services: Neville Young, Technical Writer, Pretoria, South Africa

Printing services: DPP-Utrecht, Utrecht, The Netherlands

To my wife Vesna, and my son Petar
To my Mum and Dad, Dragica and Dobrosav
To my Mother in Law and Father in Law, Marija and Slobodan

Contents

Acknowledgments

During the course of the research presented in this thesis, at Leiden University (2000 - 2004), I was supported through the `Archer` (at the beginning also known as `Spade II`) research grant from Philips Semiconductors (now NXP).

Many thanks to the next people for the interesting scientific and technical discussions we had: Pieter van der Wolf, Erwin de Kock, Ondrej Popp, Wido Kruijtzer, Ad Peeters, Gerben Essink, Denis Alders, Andrei Radulescu, Kasia Nowak and Paul Stravers from Philips Research; Paul Lieverse from TU-Delft; Andy Pimentel from University of Amsterdam; Peter Knijnenburg, Luuk Groenewegen, Herbert Bos, and Bart Kienhuis from LIACS.

I would like to thank the following fellow Ph.D. students from LIACS at Leiden University whom I have shared room 122 with: Alexandru Turjan, Todor Stefanov, Claudiu Zissulescu, Laurentiu Nicolae, and Dmitry Cheresiz. I always will remember the very interesting discussions we had about our research work and survival in the Netherlands. I am particularly grateful to Alexandru Turjan, for his unselfish acts of help at the beginning of our stay in the Netherlands.

I am also greatly indebted to many teachers, colleagues and friends from my homeland: Mile Stojčev and Goran Lj. Djordjević from the Faculty of Electronic Engineering (FEE) at the University of Niš, Serbia; Bane Vasić from University of Arizona; Jelena Vučković from Stanford University; Dejan Milenović from ABB Process Industries Products, Switzerland; Daniela Milović, Aleksandar Prvulović, Zoran Marković, Dejan Dimitrijević, and Miloš Kostić.

As a foreigner in the Netherlands, I have been and still I am dependent on a practical help, co-operation, and information sharing with other expats. The following people proved to be very supportive and I show my gratitude to all of them: Bojan Leković, Slobodan Mijalković, Zoran Stojanović, Miodrag Djurica, Marko Cvetković, Milan Petković, Dejan Stojanović, Stanislav Jovanović, Božidar Stanković, Vladimir Erić, Dejan Ognjanović, Marko Smiljanić, Marija de Roo Janković, Aleksandar Berić, as well as Janet Boldger, Erik Reid, Koos Ellis, Roelof van Wyk, Ettore Benedetti, Werner Strydom, and David Borland.

Last, but not least, I thank my family and my close relatives for supporting me during the course of my Ph.D: my parents, Dobrosav Lazar Živković and Dragica Čedomir Živković, for giving me life in the first place, for educating me with all aspects, from the ethical heritage, over general culture, through literature, to sciences, for unconditional support and encouragement to pursue of my own life path, even when my interests and intentions went boundaries of conventional and expected; my sister Danijela Tasić, for believing in me; my cousin Zoran Stevanović, for supporting my family and myself when it was necessary; my parents in law, Slobodan Burazor and Marija Burazor for supporting me and believing in me.

Two special persons I want to express my greatest gratitude to, my wife Vesna for her support and patience, and my son Petar:

Petar, my son, many times this work appeared to be an obstacle our joined fun, playing, and learning. Thank you for all attention, understanding, and respect of my work and deeds. I must admit there have been moments when I wanted to give up of everything, but you, my son, have been and remained my main motive to continue. Never give up Petar, always finish your part of the job, do not allow unfinished businesses to hunt you later on...

Thank you all.

Vladimir Dobrosav Živković
Leiden, September 23, 2008

Chapter 1

Introduction

Get your facts first, then you can distort them as you please. [1]

1.1 Summary

The uninterrupted increase of the capacity of silicon has resulted in radical changes in the level of applications which an embedded system has to support as well as in the system's level of complexity. The consumer-electronics industry now comprises many electronic gadgets consisting of a single chip with various functions embedded: a radio transceiver, a network interface, multimedia functions, security functions. In addition, the chip contains the ”glue” needed to hold it together along with a design which allows the hardware and software to be reconfigured for future applications. In conclusion, today's embedded systems are integrated on a single chip instead of their previous implementation as a standard microprocessor-based board.

As a result, such Systems-on-Chip (SoC) are heterogeneous. That is, they are embedding different types of processing units (programmable, reconfigurable, dedicated), and different types of communication networks (buses, cross-bar switches, shared and dedicated networks). Not only has the once inflexible hardwired system become ”soft”, but the solid border between software and hardware is rapidly fading away [1]. This ”softening” creates problems for SoC-based systems because they are becoming extremely complex. To illustrate the SoC design problems, we quote a few statements by Chris Rowen (Tensilica) [2]: *(i) Design complexity vs. designer productivity: A well-recognized SoC design-gap, which lies between the growth in the chip complexity (58% for 5 year period) and productivity growth (21% for 5 year period) in logic design tools, widens every year. (ii) Application complexity: Standard*

[1] A quote of Samuel Langhorne Clemens, better known by the pen name Mark Twain (1835 A.D.-1910 A.D.), American humorist, satirist, writer, and lecturer. By many American experts regarded as ”the father of American literature”.

communication protocols are rapidly increasing in complexity. (iii) Hardware and software validation: All embedded systems now contain significant amounts of software. (iv) Design-bugs: SoC design-bugs can literary kill a company. This was confirmed to an extent at the Transaction-Level Modeling panel (TLM) [3], where designers said that more effort at the system-level - to cut ”time-to-market” - is urgently needed. We agree and claim that a sound mapping exploration strategy can give some reassurance in this developing situation.

To master the complexity of the exploration of various mapping alternatives, it is essential that higher levels of abstraction are included in the design hierarchy and that all relevant applications/architectures are effectively and efficiently captured in the models that are used at these levels. The models must be generic enough (at least for the application domain considered) to encompass the various features that go with different mappings. Moreover, although at high levels of abstraction, application and architecture models are coarse grained and parametrized, it is imperative that architecture components should be Intellectual Properties (IPs) wherever possible. This may imply that input and output data types in application model tasks and architecture model processing units are quite different. Mapping should still be straightforward in such cases. To the best of our knowledge, no mapping approaches offer such facilities.

This chapter focuses on explaining research-background and related work needed to understand our application, architecture, and mapping modeling approach as detailed in this thesis. We focus on the parallelism and heterogeneity of architecture, the abstraction level needed to efficiently explore such architectures, and the existing system-level methods and approaches.

1.2 Embedded Systems: Definitions, Design, and Exploration

Embedded systems have become a highly significant part of everyone’s daily lives. They are literally everywhere: from the various electronic gadgets such as personal-digital assistants, mobile-phones, MP3-players, i-Pod-devices, identification and banking smart-cards, through television & entertainment sets, gaming devices, to various measurement and acquisition instruments - all different in scale and size. Embedded systems extend to telecommunication, military and space-exploration equipment. Hence, it is difficult to arrive at a single coherent definition of embedded systems. Obviously, there appears to be no size, area, cost or similar restriction when speaking about embedded systems. Yet, one special characteristic distinguishes them from other types of systems: they are all tightly connected to their environment.

Embedded Systems. *Embedded systems are digital computer-based systems that embed their functionality into environments they operate, and due to their tight-relation to these environments, they differentiate from any other category of digital computer-based systems.*$\diamond$

The operation of an embedded system can be easily described by the following sequence: (1) acquire the inputs from the environment using sensors, keyboards, on-off triggers, analog-to-digital converters or any other input converters; (2) process these inputs using the embedded functionality to produce the corresponding results; and (3) convey those results to the environments using primitive light-and-sound outputs, actuators such as relays or robotic arms, networks, digital-to-analog conversion, audio output, video output or any other output device or format. Although the sequence is a simple one, meeting the functional requirements needed for any of today's embedded systems is not simple at all.

Embedded systems are reactive, often real-time systems. According to [4], a real-time system must satisfy explicit (bounded) response-time constraints or it risks erroneous behavior, including failure. Embedded systems also must meet required performance constraints or they risk failure of the embedding system or loss of Quality-of-Service (QoS); An MP3 player will not produce the required audio-quality if performance requirements are not met; a Set-Top-Box (STB) will not be able to descramble scrambled digital-video data; a data stream will not be acquired properly by a Digital Acquisition System; a robot hand will not react on time; parameters indicating the failure of some other digital hardware will not be processed in time. Hence, we can say that embedded systems are a special sub-group of real-time systems.

High performance requirement is particularly challenged by: (1) the high volume of data going into and out of an embedded system, (2) varying data rates of inputs, and finally, (3) the power hungry behaviors (algorithms) built into the system. These aspects may be considered to be an ill-affordable system cost which may conflict with performance requirements.

However, performance requirements are not the only concern. For mobile devices, size, weight, and power consumption are equally important. Additionally, integrity and privacy may play such an important role that security constraints may become dominant [5]. Finally, for devices whose configuration (structure and topology) may change when activated the reconfigurability is the most important [6].

All these factors make the understanding of embedded systems patently difficult. To help both designers and scientists in their understanding, analysis, exploration and design of newer and better embedded systems, a specialization of embedded systems towards specific applicability domains is made.

Domain Specific Embedded Systems. *If a group of embedded systems shares a certain commonality, such as e.g. application domain, and due to this commonality they can interchange or re-use parts of their implementations among themselves, these systems are called domain specific embedded systems.*$\diamond$

Designing domain specific embedded systems makes life easier; there is no real need to be concerned about text processors and Graphical User Interfaces (GUI) in an STB, but the task of decoding the MPEG-2(4) stream must be performed perfectly. Conversely, some word-processor applications and GUIs are expected on hand-held Personal Assistant (PA) gadgets, but there is no need for extreme decoding and stream processing features. In this way, reducing (removing) unnecessary embedded tasks makes it possible to save both silicon real estate and limited resources. However, even though domain specific applications do not require General Purpose Platforms (GPP) (such as are used in high-level CPUs), today's domain specific applications are still hungry for performance (resources) and this implies that modern domain specific embedded systems need multiple processing resources.

Multiprocessor Embedded Systems. *If an embedded system comprises multiple processing components which operate in parallel, then the embedded system is called a multiprocessor embedded system. Moreover, the components may be different types in which case the system* *is called heterogeneous.*◇

1.2.1 Embedded Systems Design

Embedded systems design has become far more complex than in the early days when they were simple micro-controller-on-PCB[2] designs. The *ad-hoc* design approach that was common then is no longer possible. As quoted in Section 1.1: ”SoC design-bugs can literary kill a company.” Modern embedded system design requires thorough simulation verification of an SoC before it is delivered to production-lines because the Non-Recurring Engineering costs (NRE) are too high. Moreover, non-functional behaviors such as power dissipation, Quality-of-Service (QoS), integrity and Real-Time (RT) constraints are now of primary importance. Therefore, the major goal of embedded system design is to cope with both functional and non-functional aspects. In addition to that, design and implementation costs must not grow with system complexity. This in itself demands a foresighted design paradigm which avoids prototyping by relying on abstract model-based and exploration-based designs.

1.2.2 Design Space Exploration

Given (user) requirements and constraints, there are - in principle - many systems that can implement these demands. All these systems constitute points in a ’performance-cost’ design space. Design Space Exploration (DSE) is a method aiming at identifying those points that are optimal in some way or another.

Approaching this search for optimal points by considering each and every point in the space is not feasible. Instead, one has to find a strategy that guides the search in the path from requirements and constraints to optimal implementation candidates by pruning the design space while proceeding. This approach, which is a real paradigm shift, was introduced in [7] and called the *Abstraction Pyramid* view. This view is reproduced here in Figure 1.1 for convenience.

The base of the pyramid represents the complete design space for the application domain. This space is, at least in principle, reachable from user requirements and constraints that are at the top of the pyramid. Specification, exploration and design then proceeds at discrete levels of abstraction as represented by the parallel cuts. At each level, level-specific models are used to explore the system instances (also referred to as *platform* instances) with levels of confidence that are within pre-defined bounds. Selected instances narrow down the reachable design space, as illustrated by the inner pyramids in Figure 1.1. Transition from one level of abstraction to the next one down implies a number of refinements of both the parameters and the accuracy measures. The cost of model construction and evaluation is higher at the more detailed levels of abstraction whilst the opportunities to explore alternatives are significantly greater at the higher levels of abstraction. Exploration and design at higher levels of abstraction is called system-level [3] exploration and design. At the system-level, parametrization and concurrency are typically coarse-grained, and performance/cost measures are coarse metrics

[2]Printed Circuit Board.

[3]System-level is $\langle application, architecture, mapping\rangle$.

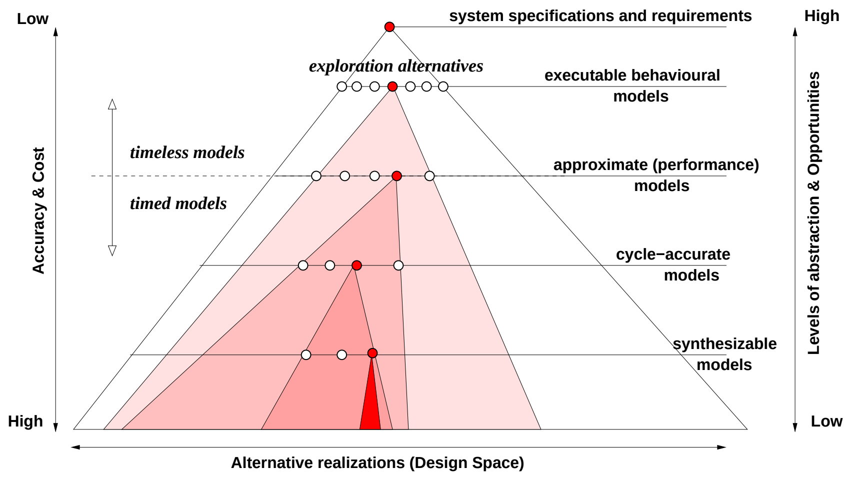

Low
system specifications and requirements
High
exploration alternatives
executable behavioural models
timeless models
approximate (performance) models
timed models
Accuracy & Cost
cycle−accurate models
Levels of abstraction & Opportunities
synthesizable models
High
Low
Alternative realizations (Design Space)

Figure 1.1: The Abstraction Pyramid.

as well. For example, a processor unit is characterized by a latency and a throughput value, parallelism is at the level of tasks, and performance and cost are measured in terms of, say, throughput and the number of processing units.

We now describe briefly the levels of abstraction in Figure 1.1.

Top level - Level of specifications and requirements

This level of abstraction is essentially an expert level or so called *back of the envelope specification* (user requirements and constraints). The system is seen as specified by the user without any technology or implementation hints. In software engineering it is also known as *level zero* (L0) requirements.

Level of behavioral models

This is an entry point to a design process. The models at this level are *executable*. The system being modeled is still decoupled from time and resource-constraints, so that the numbers obtained from the executions are rather 'qualitative' (amount of messages communicated and amount of abstract operations executed) than 'quantitative' (system performance). Nevertheless, the behavior can be expressed in some high level parallel language. At this level, performance is purely functional.

Level of approximate performance models

The level of ”approximate-accuracy” [4] provides more opportunities to the designer to explore alternative solutions, anticipating the transformation from executable behavioral (*untimed* [8]) models and cycle-accurate models. In [10] this level of abstraction is introduced as an ultimate way to avoid the so-called “guru approach,” where the embedded system designer jumps from the conceptual or behavioral model straight to the cycle-accurate model. In contrast, an incremental narrowing of the design-space reduces the risk of landing on non-optimal points.

In this thesis we claim that before going down to lower levels of abstraction, the designer should perform a thorough exploration at the level of approximate performance abstraction. This exploration prunes the design space in such a way that the designer will then have only to focus on a significantly reduced set of design possibilities when moving down to the next level of abstraction.

Level of cycle-accurate models

The level of cycle-accurate models is also known as a bus-cycle accurate level. At this level, communication between system components and computations within components are evaluated on a scale of Register Transfer Bus Cycles [5].

While this approach level of exploration provides a great deal of confidence, the processing power that is required to run exploration simulations in the case of complex and demanding applications is overwhelming [11]. Therefore, we argue in this thesis that the designer should use models at this level only after he has significantly pruned the design space at the upper abstraction levels.

Level of synthesizable models

This level of abstraction is the ”ultimate” implementation specification level. Almost all consumer-electronics products today are designed taking only cycle-accurate and synthesizable levels into account. These are the levels where a traditional designer feels comfortable and becomes sufficiently confident with the obtained performance numbers. Raising the levels of abstraction leads to new challenges in dealing with the conversion of specifications and exploration on higher levels of abstraction to specifications at synthesizable level.

Now that we have introduced the Abstraction Pyramid paradigm, it remains to decide whether (and on what levels) we should rely on *analytical* or *simulations exploration methods*.

[4] It is sometimes called *time-approximate* level [8] or even *performance model* level [9].
[5] In Computer Architecture this is known as RTL.

1.2.3 Analytical Exploration Methods

As indicated earlier, modern embedded systems are increasingly complex. Aspects related to resource sharing, communication buffering and timing constraints are fairly complicated to deal with when it comes to modeling and to evaluating them.

One can deal with these aspects by using analytical modeling and quantification methods. These are based on Network Calculus Theory [12]. In this approach, data is modeled in terms of data characteristics; resources are modeled as black-boxes that transform data to data and transform available capacity to remaining capacity. The analysis then solves a set of equations that confirm or deny the attainment of the pre-defined objectives.

A quite different usage of analytical exploration is illustrated in the `Design Trotter` framework [13]. There, a designer can establish ’metrics’ to guide the embedded design and synthesis tools towards an efficient application architecture matching. The metrics are computed through data and control dependency analysis on: local-and-global data transfers, on data-processing, and on control operations at all abstraction levels. The application specification [6] is parsed into a Hierarchical Control Data Flow Graph (HCDFG), which consists of the lower-level Control Data Flow Graphs (CDFG), which again consists of so-called elementary nodes (the aforementioned representations are equivalent to CDFGs defined in Chapter 2). Once the HCDFG hierarchy is created *average parallelism metrics*, *memory orientation metrics* and *control orientation metrics* are calculated in a bottom-up manner, from the lowest level of hierarchy to the highest level of hierarchy. The results form the application characterization, and hence they help to direct the SoC design for this application. This approach is known as Multi-Granularity Metrics.

Analytical methods are very efficient when the component black-box relations between input quantities and output quantities (service costs, availability, etc.) can be expressed in terms of relatively simple, say linear, equations. Because of these assumptions, analytical methods are only feasible at high levels of abstractions where the objective is to ’estimate’ performance and cost before a more detailed exploration of the estimation-based pruned design space can be addressed.

1.2.4 Simulation-based Exploration Methods

Analytical methods have their limitations. In particular, when going down the abstraction levels, analytical methods may have to rely on simulation to get more detailed information about the component’s input-output capacity (see [12]). Thus, analytical methods are not feasible at all levels of abstraction. Sooner or later, simulation is mandatory. Of course, simulation at the lower levels of abstraction is costly. Therefore, simulation can be conceived at the approximate-performance level.

In this thesis we focus on a simulation-based exploration method which is compliant with the so-called *Y-chart* approach to a system exploration [14]. In the Y-chart approach, a system (model) comprises an application (model), an architecture (model) and mapping transformations which associate the application (model) and the architecture (model) together. See

[6]This is usually a C-code functional specification.

Figure 1.2. The application (model) is purely transformable, i.e., it only expresses functional behavior. The architecture (model) is purely reactive i.e., it only expresses non-functional behavior which includes latency and throughput, resource availability, power consumption, etc. The Y-chart, then, takes the parameters from the two models and the transformation set to conduct a quantitative performance/cost analysis. The numbers that are returned by the analysis may be used to tune application and architecture models and to make mapping transformations.

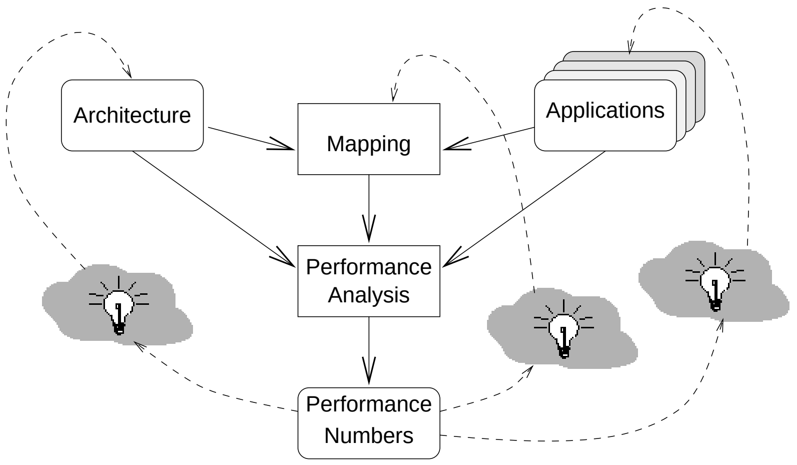


Figure 1.2: The Y-chart approach (Kienhuis): a *design space exploration* process.

This approach permits multiple applications to be mapped onto a candidate architecture as well as to map an application onto a variety of architectures. In a framework in which the Y-chart approach is implemented, the top three boxes in Figure 1.2 appear as applications layer, mapping layer and architecture layer, respectively. The mapping layer translates *representations* of components in the application model to *representations* of components in the architecture model. For example, a mapping transformation may convert communication semantics in the application model to communication semantics in the architecture model.

1.3 System Modeling

The Y-chart model [7] is applicable at each and every level of abstraction [15]. It was originally introduced by Gajski [15] as a generalization for design-for-synthesis. See Figure 1.3.

In Gajski's approach, 'system' is defined using various abstraction levels, where each level contains objects common for that abstraction level and where higher level objects are hierarchically composed out of lower level ones. At each abstraction level the design can be described in the form of either a behavioral or structural model and both models are defined by the number of details at that abstraction level. In the Y-chart model, design is the process

[7]It is worth noting that we distinguish between *Y-chart model* and *Y-chart approach*.

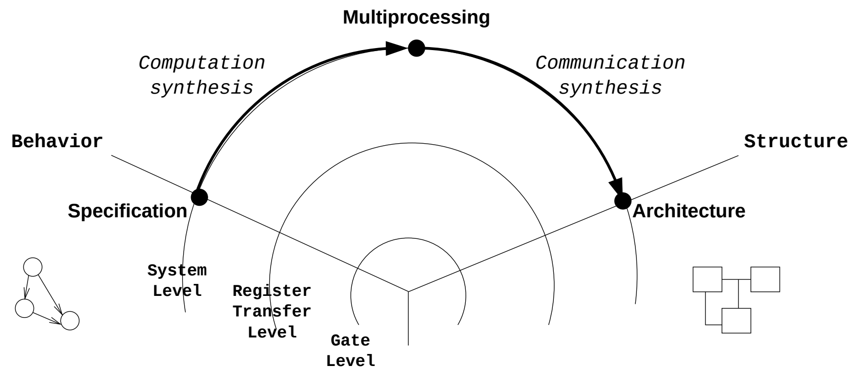


Figure 1.3: The Y-chart (Gajski): a generalization of a *synthesis* process.

of moving from a behavioral model to a structural model under a set of constraints and where structural objects are each designed at the next lower level. This is why this approach is sometimes called synthesis Y-chart.

The application and architecture models are independently chosen, yet they should match in the sense that applications should be specified in parallel language when the architectures are parallel architectures. In any case, both the application and the architecture can be conveniently modeled in terms of so-called *Models of Computation*.

1.3.1 Models of Computation

According to the National Institute of Standards and Technology (NIST):

”Models of Computation (MoC) are formal, abstract definitions of a computer. Using a model one can more easily analyse the intrinsic execution time or memory space of an algorithm while ignoring many implementation issues. There are many models of computation which differ in computing power (that is, some models can perform computations which are not possible in other models) and differ in the cost of various operations.”

From the above we derive our own definition for Models of Computation.

Models of Computation. *Models of Computation give a formal semantics concerning the way computations communicate between or follow each other. They allow for reasoning - to answer ’what-if’ questions. They may also be used for abstract specifications of computations.* [16]◇

Models of Computation that are relevant for our needs are listed below.

Finite-State Machines

Finite-State Machines (FSM) are graphs, the nodes of which represent states and may perform computations on input events, and the arcs of which represent transitions between states. The number of states and possible state-transitions is *finite*. Finite-State Machines may become intractable when the number of states grows large.

Parallel Models of Computation

Parallel models of computation are graphs of nodes that perform computation and arcs that exchange data between the nodes. Computation nodes are either (mathematical) functions or sequential processes. The various models differ in the way nodes communicate data among each other.

Process Network Models (PN)

A PN is a network of processes that mutually exchanges data using some sort of synchronization. An example of a fairly general PN is the Communicating Sequential Processes MoC (CSP) [17] which uses the 'rendezvous' or synchronous message passing synchronization method. The CSP model is non-deterministic, and is usually event-driven. Since today's heterogeneous embedded systems are not purely data-driven but also control-driven, the MoC's such as the CSP model are important as well.

An example of deterministic PN is the Kahn Process Networks (KPN) MoC [18] in which the processes operate autonomously and concurrently and communicate through unidirectional Point-to-Point (PtP) channels that buffer data in unbounded First-In-First-Out (FIFO) queues. Processes synchronize by means of blocking reads, i.e., a process read blocks when attempting to read from an empty channel. Each process can compute data in its own local memory, allowing the overlapping of process executions - this is usually described as globally-asynchronous, locally-synchronous.

Many MoCs have been proposed in the literature that are special cases of the KPN model. They can be classified in two groups: (1) Data-Flow Process Networks [19], and (2) Data-Flow Graphs (DFG) [20]. In a DFG, the processes are actually (mathematical) functions, called actors, that have well-defined firing rules which dictate token consumption and production conditions. The most well known DFG is the Synchronous Data-Flow (SDF) [21], in which every actor consumes a fixed number of tokens from its input channels and produces a fixed number of tokens for its output channels. A global schedule and FIFO channel sizes can be decided at compile-time - 'bounded buffer-size execution' [22]. More expressive DFGs have been proposed, namely Boolean Data-Flow (BDF) [23], Integer Data-Flow (IDF) [24], and Data-Flow combined STAte machine controlled Reconfiguration (DF*) [25]. With these FGs, there is a trade-off between expressiveness and compile-time analysis opportunities. In Data-Flow Process Networks, the processes are characterized by a repetitive invocation of actor functions. KPNs and their special cases are *data-driven*, and are typically *data-streaming* oriented.

Concurrent FSM Models

Opposed to the streaming data-driven applications are the *control-driven* applications. The control-driven applications can be modeled by the FSM model, yet this model may become intractable, unless a concurrent FSM model is introduced. Concurrent FSMs communicate by sending data availability signals. Examples are: (1) State-charts and ROOM-charts, originating from the real-time software design, and (2) Co-Designed FSMs, originating from the digital signal processing design.

State-charts [26] is a broad extension of conventional formalism of FSM. State-charts are relevant for large and complex discrete event systems, such as multi-computer real-time systems, communication protocols, and digital-control units - all of them commonly known as *reactive systems*. In state-charts states and transitions are described in a modular fashion, allowing for: clustering (generating super-states), orthogonality (i.e., concurrency) and refinement (i.e. ’zoom’ capabilities). Due to these features, state-charts allow for both *top-down* and *bottom-up* design approaches. The communication in state-charts is based on *broadcast* communication mechanism. That is, one state generates an event and all other states sense it, acting in response if specified. This is unlike the MoC CSP, where an explicit rendezvous channel has to be established, with a single sender and a single receiver. Therefore, state-charts are more efficient for describing ’interrupt-driven’ behavior than any other parallel MoC. Finally, state-charts can be easily extended or integrated with the other representations. For instance, incorporating Temporal Logic (TL) [27] into state-charts allows for verification.

ROOM-charts [28] are an integral part of the wider methodology used for modeling of real-time systems, called Real-time Object Oriented Modeling (ROOM). ROOM-charts are inspired by the state-charts formalism. Yet, ROOM-charts contain more formalisms to describe real-time constraints of a system than state-charts. Additionally, ROOM-chart models use the so-called “principle of separating internal control from function”, and due to this, ROOM-charts are very convenient for modeling today’s heterogeneous embedded systems as well. Finally, the ROOM-charts model is aimed at Object-Oriented Language code-synthesis (e.g., C++ code). Therefore, the parts and features of the ROOM-charts are strongly typed, and the ultimate goal is either an executable model of the system (Simulation-based Exploration) or the final real-time software image (the final product).

Co-Designed Finite State Machine (CDFSM) representation is introduced to embedded system designers by the `Polis` method [29]. A CDFSM is a specialized FSM that incorporates the unbounded delay assumption: for a classic FSM only the idle phase can have any duration between zero and infinity. The other phases all have a duration zero. An FSM also instantaneously reacts on input events. In CDFSM, the transition phase can have any duration between one time unit and infinity - all other phases can have any duration between zero and infinity. A CDFSM also takes a non-zero unbounded time to perform its tasks. The CDFSM MoC is also described as globally-asynchronous, locally-synchronous. The system is modeled as a network of interacting CDFSMs communicating through events: 1) receiving an event is analogous to blocking, 2) sending an event is analogous to not blocking, and 3) the events are broadcast to all connected CDFSMs.

1.3.2 System-Level Modeling Terminology

In Section 1.3.1 we have introduced models of computation that are appropriate for abstract modeling of system behavior. In this subsection we present in more detail the terminology and concepts of system-level modeling. Recall that a system (model) is conceived as consisting of an application (model), an architecture (model), and a set of (mapping) transformations that associate the two models together. The mapping transformations convert application representations to architecture representations. The application and architecture representations can be conveniently modeled using Transaction-Level Models (TLMs).

Transactions and Transaction-Level Models. *A transaction refers to a data or event exchange between two architecture components. As a result, models of architecture components which are involved in transactions are said to be modeled as Transaction-Level Models. Communication among components is modeled by channels and its details are separated from the details of computation and the cost of various operations.* [3]⋄

In TLM, application representation primitives are converted to architecture level primitives. For example, a process in an application model may be represented as a sequence or `read`, `execute`, and `write` abstract instructions. A processor in the architecture model onto which that process is mapped may be represented as a sequence of `check-data`, `load-data`, `signal-room`, `execute f0`, `execute f1`, `execute fn`, `check`-room, `store-data`, `signal-data` (see [30]). Different architecture models may interpret `read`, `execute` and `write` application primitives in different ways (see Chapter 3). The TLM is aware of these alternative architecture primitives and takes care of the appropriate conversions.

The TLM components significantly reduce the amount of detail in an architecture model. For example, in Chapter 3 we rely strongly on the TLM concept, and as a result of using TLM in our model, when two (or more) components need to communicate they communicate nothing else but events. A real data item is neither processed nor communicated in our architecture model, and hence, no additional simulation-time costs by processing or communicating data are introduced. The architecture model processes only newly generated delay and synchronization events. A delay event appears when an architecture transaction delay expires.

To explain transaction delays, we first have to introduce a new concept - a concept of Platforms.

Platforms. *A platform is a parametrized architecture that is a composition of library components. The library provides component types and rules to interconnect components. It also provides software to manage the composition of components.*⋄

Obviously, introducing platforms not only provides a level of abstraction where we can easily make comparisons to other platforms, but it also allows us to distinguish between environmental characteristics such as technology, flexibility, and tooling. A familiarity with the similarities and differences between platforms helps us to make time-to-market predictions and explore the accuracy of the predictions.

Platforms can be hardware platforms and software platforms.

Hardware Platform. *A hardware platform consists of a set of computation units and a communication, synchronization, and storage infrastructure. Roughly speaking, a hardware plat-*

form is a parametrized hardware architecture in which the parameters are typically number and type of units, communication and synchronization primitives and protocols, and storage methods.◇

Software Platform. *A software platform consists of a set of computation services, such as: inter-process communication, memory management, process scheduling, file system and input/output services. A software platform provides applications with unified and hardware independent interfaces, maintains a system state coherency, and supervises the execution of applications. A software platform may be: (1) an operating system, (2) a virtual machine, or (3) a micro kernel. In all three cases a software platform is a multiprogramming paradigm for an embedded multiprocessor system.*◇

Roughly speaking, a software platform is an abstraction of the underlying hardware platform for the cases when the hardware platform is programmable or reconfigurable in time [8].

In this thesis, we are interested in the mapping of stream-based applications [9] onto multi-processor architectures. Applications are modeled as Kahn Process Networks (or specialized versions) and architectures are modeled as parametrized architecture templates. The software platform: (1) provides soft real-time services, (2) supports the chosen programming model, (3) copes with the schedules that maximize the overall value/performance and (4) supports system-calls that can cope with the high-bandwidth requirements of stream-based applications [31]. A particular model of such an operating system is presented in Chapter 3.

1.4 Problem Statement

Now that we have introduced the concepts of system-level, transaction-level, and platform-based modeling, it remains to clarify why we rely on these concepts and how we do so.

Why

Next generation (embedded) systems on a chip will be multi-processor systems. These are systems that comprise of a set of heterogeneous processing units that operate concurrently and communicate over some communication, synchronization and storage infrastructure. These systems are too complex to be specified by an expert designer and designed by state-of-the-art design methodologies. This approach is so error-prone that the non-recurrent costs (prototyping, debugging, re-design) would block any form of return on investment (see Section 1.2.1). To overcome this problem we abstract the applications and the architectures. In addition, we also abstract the way that the applications associate with the architecture - that is, we abstract their association. As a consequence, the exploration of the design space is at abstract levels too. In the Y-chart model this is called system-level and in Figure 1.3 it is indicated by the bold solid arrow-headed lines.

[8]As opposed to 'reconfigurability in time', a hardware platform can be reconfigurable in space - as FPGA devices are. From the viewpoint of this thesis, reconfigurability in space is modeled purely as a feature of hardware-platforms.

[9]Stream-based applications are sometimes also called *continuous media applications*.

How

Due to us having to deal with abstract, parametrized system models, we choose to distinguish between issues as proposed in the Y-chart approach - issues which are further refined in the computation models where a distinction is made between computation and communication as well. In the scope of this thesis the architecture models are considered to be at the transaction level (see above). At this level we can abstract the internals of the computation units and focus on transactions among units (the communication, synchronization, and storage infrastructure). Similarly, we have to provide a model of the applications so that we can specify them at the level of abstraction where we will be dealing with them. We emphasise that the application is irrespective of any specific hardware architecture, though the application and hardware architecture models must match in the sense that they can be easily related. However, because the application model should be irrespective of any specific hardware architecture, the matching between application model and hardware architecture model will never be perfect. As a consequence, the relating of application models and hardware architecture models requires transformations which take application model representation primitives to architecture model representation primitives. These transformations constitute what we call the mapping process.

1.4.1 Objectives and Research Topics

The main objective of this thesis is to develop models and methods that lead to fast and accurate, abstract, design-space exploration multiprocessor systems-on-chip which are used in high-throughput, streaming applications. Central to these models is a Y-chart, with three clearly separated entities: Architecture, Applications and Mapping (see Figure 1.2). Relating these three means determining their models and representations, as well as the required transformations to overcome differences between the primitives of the entities. Hence, we identify Models, Representations and Transformations as the main research subjects for this thesis. That is:

- Models - Applications and architectures are modeled independently. However, they should be compatible in the sense that applications are modeled in a parallel language when architectures are parallel architectures. The question is: *What are these models?*
- Representations - Applications and architectures are associated with each other. This requires that application and architecture components are represented in such a way that the application model can drive the architecture model. The question here is: *What are these representations?*
- Transformations - Because application models and architecture models do not necessarily match, transformations should be provided to translate application representations to architecture representations. The question here is: *What are these transformations?*

Given Application/Architecture models and Mapping representations and transformations, a Performance/Cost Analysis Method must be provided such that a subsequent design-space exploration can be built on it in a fast and accurate way. Thus, we end thus subsection with the final question: *What is that Method?*

1.5 Solution Approach

The approach to the solution in this thesis is depicted in Figure 1.4. We explain it in this section.

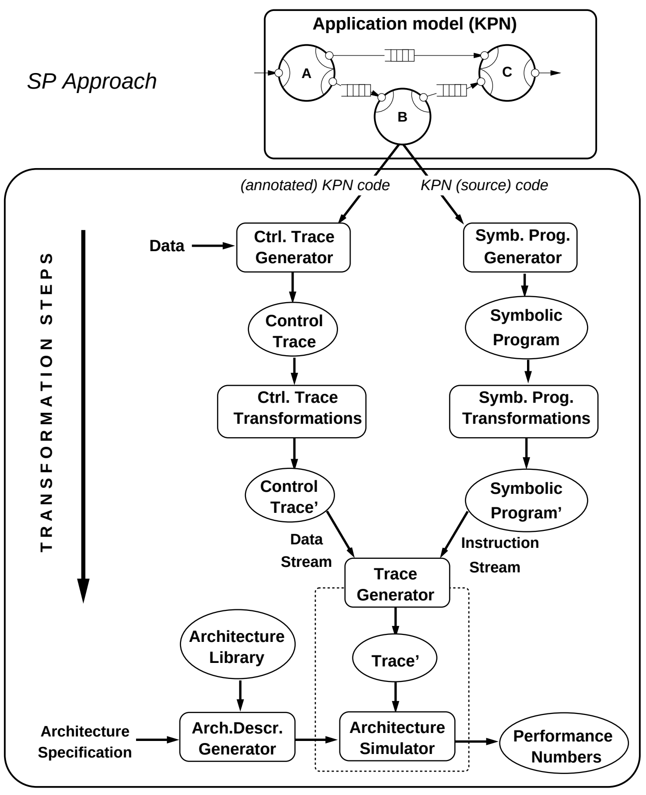


Figure 1.4: The Symbolic Program approach (SP approach) [11, 32]. This approach allows designers (1) to perform design-steps as in the case of detailed design (indicated as Transformation Steps), (2) to run fast simulations of architectures being explored, and (3) to reuse the same application and architecture representations irrespective of the supplied data input (one of the ideas of the Y-chart approach [14].

Because we target streaming application systems, we believe that the KPN MoC is an appealing model for specifying the functional behavior of the system. We call this the application model, which is purely transformative. The architecture part of the system is modeled as an admissible composition of components taken from a library of components. These components only model the 'cost' of the application's workload in terms of resources, transaction delays, throughput, service availability, etc. We associate application and architecture models together by letting the application components generate Symbolic Programs as well as Control Traces that provide information regarding the outcome of data-set dependent conditions for a given input stream. The idea of recovering or preserving the control flow and data-dependencies from the original application representation by means of Symbolic Programs (SP) has been introduced in [32].

Our architecture model components are executable and interpret the combined symbolic programs and control traces in terms of non-functional behavior. However, because the architecture model does not necessarily match the application model, symbolic programs and control traces may have to be transformed to yield information which the architecture components are able to interpret. They combine information from transformed symbolic programs and information from transformed control traces to data-specific Symbolic Introduction traces, and interpret the incoming instructions in terms of performance and cost of services (modules) that are internal to the components. The ideas of modeling and exploring architectures by interpreting symbolic program representations has been introduced in [11, 32].

To conclude, our approach to the solution is directed at Models, Representations and Transformations. We do not discuss Performance Analysis in this thesis.

1.6 Related work

Several design-space exploration methods at abstract levels have been proposed in the literature. The approaches mentioned below are closely related to the approach presented in this thesis.

1.6.1 Spade

The `Spade` methodology [7], [33] is a System-level Performance Analysis and Design-space Exploration methodology. The `Spade` methodology follows the Y-chart approach introduced in Figure 1.2. The `Spade` design flow is illustrated in Figure 1.5. In this flow, we recognize the application modeling, architecture modeling, mapping and performance analysis. We now briefly comment on the various parts in Figure 1.5.

`Spade` uses KPN MoC [18] to model the functional behavior of an application. The application model represents the workload that is imposed on an architecture. The workload consists of two parts: communication workload (`read` and `write`) and computation workload (`execute`). The architecture model in `Spade` is component based. It qualifies aspects of non-functional behavior, such as delays and throughput.

`Spade` supports an explicit mapping step, where application processes and channels are mapped on architecture components. For the purpose of performance/cost analysis `Spade` performs a co-simulation of application and architecture. In `Spade` methodology this is called Trace-Driven Execution (TDE). The application model generates traces of Symbolic Instructions (SI). These traces are, hence, representations of the processes in the application

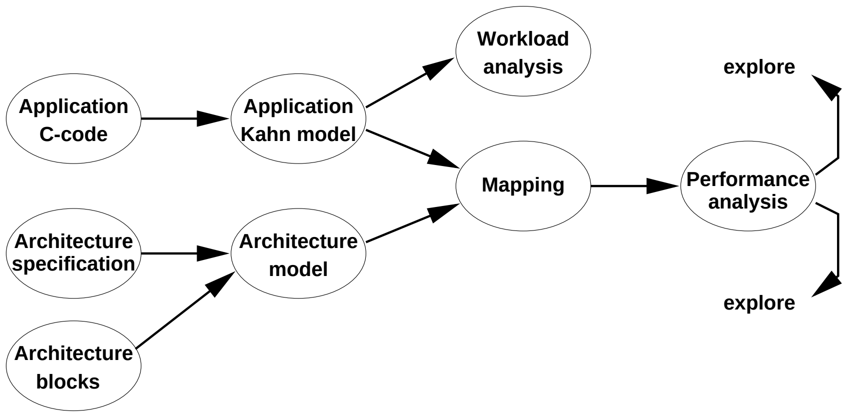


Figure 1.5: The SPADE design flow

model. The application SI traces are translated to architecture SI traces by an explicit TDE simulation-time transformation engine. Then, the architecture SI traces are interpreted by the architecture, which returns performance numbers.

`Spade` models have two major disadvantages: (1) SI traces do not preserve dependencies between instructions (loss of information), and (2) the architecture model is too close to the application model (loss of generality).

1.6.2 Sesame

`Sesame` [34] is a successor of `Spade`. Like `Spade`, `Sesame` models the applications as KPNs and represents a KPN process as a trace of abstract instructions. In contrast to `Spade`, `Sesame` uses an even-driven simulator [35], which is much faster than a bus-cycle accurate simulator. Moreover, an architecture in `Sesame` is defined by the `Pearl` modeling language and that makes the architecture modeling more flexible than a library-based approach (such as `Spade`).

In order to partially recover data-dependencies from the `Spade`-like SI traces, `Sesame` relies on the Integer Data Flow graph (IDF) representation [36]. `Sesame` replaced the TDE type of mapping with the so-called ”virtual processor representation”, which is the IDF implementation of ideas in [30]. Hence, the task of a virtual processor is to refine `read`, `execute`, and `write` SI traces into a partially ordered trace of `check-data`, `load-data`, `signal-room`, `execute f0`, `execute f1`, `execute fn`, `check`-room, `store-data`, `signal-data` instructions.

`Sesame` uses evolutionary algorithms to find Pareto optimal architectures [37]. In this way `Sesame` provides a method to steer DSE towards a simulated solution.

Our approach differs from the `Spade` and `Sesame` approaches in that we represent the application process and the architecture processor as symbolic programs rather than as symbolic instruction traces. Symbolic programs can fully separate data-dependent and data-independent information while symbolic instruction traces cannot. See Figure 1.4. In the SP-approach, data-dependent information (e.g., variation of data input contents and of data format) is isolated in a control trace, while data-independent information (such as the application process structure in terms of control-and-data dependencies) is isolated in a symbolic program. On the contrary, a symbolic instruction trace combines data-dependent and data-independent information. Hence, a variation of input data implies various TDEs (in the `Spade` case) or various IDFs (in the `Sesame` case) for a single application process. To avoid that, symbolic-instruction based methodologies imply more severe restrictions than modeling architectures and application-on-architecture mappings. On the contrary, the SP approach can cope with any architecture, as long as it is a composition library component. If there is no applicable composition in the SP component library, then additional SP components can be added [10]. Therefore, `Spade` and `Sesame` components are dealing with the effects of particular behavior (*traces of application execution*) instead of with the source of behavior (*pure application representation*). Due to this, the `Sesame` cannot cope with general mapping (see Sections 3.6.2 and 4.6).

1.6.3 MTG-DF*

MTG-DF* is a modeling methodology which combines the Multi-Thread Graph (MTG) approach [38] with the Data-Flow combine STAte machine controlled Reconfiguration (DF*) model [25].

The DF* model is an extension of SDF [21] so that: (1) each process has multiple states which are executed in a fixed sequential order, (2) each state has its own producer/consumer conditions and implementation, (3) transitions and producer/consumer communication appear only when state-conditions are satisfied, and (4) the last state is either followed by the first state (cyclo-static execution order) or some other state. In principle, DF* states can execute in parallel. We see the value of the DF* model more at the Intra-task level than at the Task-level (See Chapter 2). We acknowledge that the DF* model had some influence when creating our symbolic programs.

The MTG representation models embedded software as a graph of multiple threads of execution. Therefore, the MTG representation is a parallel application representation too. However, unlike in the cases where the representations are originating from the Kahn model and where inter-process communication is based on unbounded FIFO channels, the inter-process communication in MTG is split between synchronization via semaphores and data communication via shared memory. These are explicitly visible and their non-deterministic nature is fully exposed. This is also why the MTG representation is less abstract than our symbolic program representation (see in Section 2.4), or `Spade`-trace representation [7]. Due to the level of details the MTG representations are regarded as so-called *gray-box* representations, where *black-box* representations stay for fully abstracted representations of application sources and where *white-box* representations stay for to-the-last detail synthesizable representations of application sources.

[10]Resolving the missing behaviors by enlarging the library contents is a common practise for `Sesame`, too. The only difference is that newly added `Sesame` components still miss the proper separation of representation of data-dependent vs. data-independent information, while SP-components do not miss this.

The MTG-DF* approach is synthesis-driven and hence, it is too detailed for simulation-based DSE of MPSoCs. Additionally, the main goal of this approach is software which immediately excludes the DSE of the all-in-hardware architectures. Therefore, we refer to the MTG-DF* from the point of view of the *graph* representations it uses rather than anything else [11].

1.6.4 Ptolemy

The *Ptolemy* framework provides methods and tools for the modeling, simulation, and design of complex computational systems [39]. It has been developed by the University of California at Berkeley. It focuses on heterogeneous system design using MoCs for modeling both hardware and software. Important features are the ability to construct a single system model using multiple MoCs which are interoperable, and, the introduction of disciplined interactions between components, where each of them is governed by a MoC. The interoperability between different MoCs is based on *domain polymorphism*, which means that components can interact with other components within a wide variety of domains (MoCs). Also, the Ptolemy methodology does not have the objective to describe existing interactions, but rather imposes structure on interactions that are being designed. Components do not need to have rigid interfaces, but they are designed to interact in a possible number of ways. Particularly, instead of verifying that a particular protocol in a single port-to-port interaction can not deadlock, Ptolemy tends to focus on whether an assemblage of components can deadlock. Designers are supposed to think about an overall pattern of interactions, and to trade off expressiveness for uniformity.

The Ptolemy work and the work presented in this thesis connect at the part of modeling heterogeneous systems: (1) Both promote interoperability of MoCs - the complex architecture behaviours are modeled using different models of computation which interact over rigid interfaces, and (2) both are kind-of Component-based Design (CbD) approaches - the particular system instances are built as assembles of smaller components, each of which contributes in the particular aspect of the system architecture.

1.6.5 Some Additional DSE Methods

The work of this thesis dates to the period between the years 2000 and 2004, and, hence, it is clear that more development has happened between that time and Today. We feel a responsibility to mention the new activities in the field of DSE and Modeling for DSE-purposes. The DSE methods we mention in this section are based on the DSE methods overview paper of Matthias Gries [40]. This overview paper recognizes two kind of methods in a way how they relate to the Y-chart in Figure 1.2:

1. Methods that deal with *the evaluation of a single design*, represented by the performance analysis step in the chart. These methods range from purely analytical methods to cycle-accurate and RTL simulations. To shorten the DSE runs and to be able to focus

[11]There is a similarity in the way how the MTG-DF* representation is used to describe an embedded software application and the way how SP-architecture modules are described. However, the purpose/aim and the origin of these approaches are different.

on the resource utilization, these methods sometimes assume correct-by-construction synthesis steps prior to simulations. Examples of such methods and/or frameworks are the earlier mentioned `Spade` and `Ptolemy`, but also `MESH`, `StepNP` and `SEAS` which also use the abstract architecture models. The former uses HLLs to describe architecture model components, while the later two use ISSs and HDLs (respectively) for the same purpose.

Some of the analytical approaches are also falling into this category, e.g. the approach [41] where computation and communication system events (symbolic instructions) are first augmented and then simulated, or the approach [42] which uses the four event stream models (periodic, jitter, burst, and sporadic) to model internal component scheduling and then through their transformations create the formal analysis of the global system-scheduling and buffer memory of the heterogeneous system being modeled.

2. Methods for *the coverage of the design space* by (more or less) systematically modifying the mapping and the analysis to the mapping and architecture in the chart. These methods only slightly alter an application representation, just enough to adopt or refine it in order to match the facilities of the architecture representations. These alterations are usually required to establish a feasible mapping. On the other hand, while DSE run only the workload (so-called input data sets) changes while the application functionalities do not change. Examples of such methods and/or frameworks are the earlier `MTG-DF*` and `Sesame` which search for Pareto-optimality [37], but also some `MILAN` which has different tools for DSE pruning (see later in this section).

The paper [40] deals with in-depth of all commonly-known (modern and/or legacy) DSE methods. We, however, focus here on system-level simulations and abstract performance models only. Hence, we will mention only small subset of the methods and frameworks available. In addition, we decided to focus on the approaches that somehow (either via MoC and modeling choices or via simulation-techniques) link to the work we present in this thesis.

StepNP

StepNP stands for SysTem-level Exploration Platform for Network Processing. StepNP has been developed by STMicroelectronics in collaboration with a couple of universities. It targets a system-level exploration of streaming applications, multiprocessor network-processing architectures, and SoC tools [43]. It provides well-defined interfaces between multi-processor architecture components in terms of interconnects (functional channels, NoCs), processors (simple RISC), memories and coprocessors. It also has a custom Operating System (OS) that provides support for concurrency and multi-threading, so the existing Instruction Set Simulators (ISS) can be integrated via additional wrappers. The targeted applications should be described using the MIT Click modeling paradigm [44], originally intended for building flexible and configurable routers. Thus, the application is assembled from packet processing elements where each individual element implements simple router functions like packet classification, queuing, scheduling, and interfacing. Complete application representations are then built by connecting elements into a graph which models packet. StepNP uses synthesizable `SystemC` models to provide path to the hardware [40].

SEAS

SEAS stands for a System for Early Analysis of SoCs. This is an IBM framework that allows for the composition of high-abstraction level virtual components with the aim to estimate the performance, area, and power dissipation of the resulting SoC [45].

Early analysis begins with a designer specifying a system-level description of the SoC design. The designer needs first to identify the SoC components. The components, including buses and power-management circuitry, are selected from a given technology library of cores. The core library components include the necessary performance, power, and physical information. As a result of this step, each virtual component has a real-design linked to it. The second step for the designer is to describe interconnecting of the components. Describing the connections between the chip interface and the constituent cores is a complex task that requires extensive knowledge of a bus architecture and each core in order to understand how each pin should be connected. Support is usually provided by automating this process as much as possible. Finally, the designer needs to describe clock domains. This is especially required since clocks have to be considered during floor-planning and power dissipation modeling. The evaluation in terms of speed and power is done by means of `SystemC` simulation.

MILAN

MILAN is a model-based, extensible simulation framework that combines tools for DSE pruning with the simulators at different levels of abstractions (Matlab, `SystemC`, C, SimpleScalar Assembly, etc.). It provides a unified environment capable of modeling a large class of embedded systems and applications, seamlessly integrating different widely-used simulators into a single framework [46]. The simulators include various Trace-Driven as well as Task-Level performance evaluation tools, but also the third party engines. The application are specified at the high-abstraction level by means of hierarchical data flow graphs, where each graph node is represented according to the level and specification-language of the simulation target. The two very important additional MILAN's features are: (1) its capability to enable rapid evaluation of different performance metrics such as power, latency, and throughput, and (2) its support for rapid evaluation of a large design space.

MESH

MESH is Carnegie Mellow University a performance DSE framework [47]. In MESH, hardware building blocks, software, and schedulers/protocols are seen as three abstraction levels that are modeled by software threads on the simulation host. Hardware modeling threads are running periodically why software and scheduler modeling threads are running sporadically. The software threads are delivering load (time budgets) to the hardware threads; these budgets are estimated through the application specification profiling.

The abstract performance architecture modeling is at the concept level very similar with the Models of Architecture (MoA) we present in this thesis (see Chapter 3).

SPW

SPW is the Signal Processing Worksystem created by CoWare Inc. It allows hierarchical compositions of components written with respect to either Synchronous Data-Flow (SDF) or Dynamic Data-Flow (DDF) MoC formalisms. Components are available either as parts of design-libraries or explicitly coded FSM rules using HLL (`Matlab`, `C++`, `SystemC`) or HDL (`VHDL`, `Verilog`). The DSE flow is based on iterative refinement of modeling and simulation at different abstraction levels.

The approach we present here also employs FSM based architecture models (see Chapter 3), as well as the exploration flow through modeling and simulations iterative refinement (see Chapter 4).

Chapter 2

Symbolic Programs

Today's scientists have substituted mathematics for experiments, and they wander off through equation after equation, and eventually build a structure which has no relation to reality. [1]

2.1 Summary

The aim of this chapter is to provide representations of component behaviors in a KPN application model that are suitable to drive an architecture model in a Y-chart design-space exploration approach. KPN process behaviors can be represented in various ways. These ways are shown in Figure 2.1.

One way is to generate a *trace* of the Application Programming Interface (API)process, in which *Reading*, *Executing* and *Writing* actions appear as `read`, `execute`, and `write` Symbolic Instructions (SI). This is the way `Spade` and `Sesame` represent process behaviors. The representation model is rather poor, because it cannot cope with data-dependent expressiveness of the process behavior. A representation that overcomes this problem is the Control Data Flow Graph representation model (CDFG). However, a CDFG representation contains all low-level details of an application process and, hence, it is not the best choice for abstract design-space exploration.

The main differences between these representations are in terms of *completeness* and *abstraction*. The SI representation is easy to use because it abstracts application details quite well, and focuses designer's attention on the actual application computation and communication sequences. This is convenient for Design Space Exploration (DSE - see Chapter 1, Section 1.2.2) of complex application-onto-architectures mappings. However, the SI repre-

[1]These words were taken from "Form of Modern Mechanics and Inventions," printed in July, 1934 A.D. The words belong to Nikola Tesla - Никола Тесла - (1856 A.D.-1943 A.D.), Serbian scientist, inventor and engineer. In honor to him in 1960 A.D. the tesla (symbol T) was accepted as the SI derived unit of magnetic flux density (or magnetic induction) and defines the intensity (density) of a magnetic field.

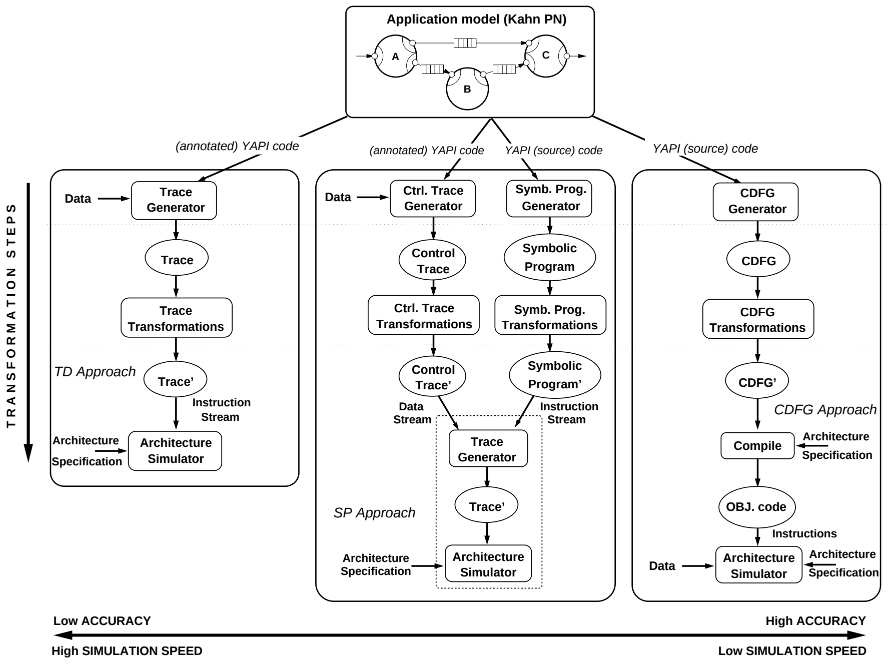


Figure 2.1: The Symbolic Program approach vs. SIT & CDFG approaches.

sentation also leads to inaccuracies in performance estimations due to a lack of completeness - the SI representation is static and reflects the sequential execution of a process in its totally ordered Symbolic Instruction trace.

CDFG representations, on the other hand, lead to complete and synthesizable representations. This is desirable from an accuracy point of view. However, deriving and maintaining CDFG representations is time consuming and too detailed for efficient and fast abstract-level DSE [2]. Data-Flow Graph (DFG) representations are less detailed and partially ordered. Due to these qualities they are used in the Sesame framework as intermediate representations. But - these representations are inadequate because they do not, conveniently at least, model conditional constructs [48].

In this chapter, we introduce a novel representation of process behaviors, which we call Symbolic Programs. A *symbolic program* is an abstract non-executable CDFG, plus a *control trace* which memorizes the outcomes of data-dependent constructs. We claim that Symbolic Program representations are both abstract and complete, and therefore, preferable to DFG, CDFG, and SI representations. See Table 2.1.

[2]The whole point of a DSE method is to steer designer's decisions by depicting the pros and cons of particular choices. As such, the method must not be too time consuming nor extremely complex - otherwise it stops to be a DSE method and becomes a design method.

$\frac{Abstraction\rightarrow}{Completeness\downarrow}$	*NO*	*YES*
YES	CDFG code	Symbolic Program
NO	DFG code	Symbolic Instruction Trace

Table 2.1: Process representations compared in terms of completeness and abstraction. The *completeness* means that a representation contains both control and data-flow elements, while *abstraction* means that element details are hidden-or-removed.

2.2 Introduction

Because we advocate the Y-chart approach, we have to find a way to associate an application model with an architecture model. Both models consist essentially of computation, communication and storage components. In the application model, these are processes with FIFO buffered channels between the processes. In the architecture model, they are computation units, communication units, synchronization units and storage units (See Chapter 3). These two models are linked to each other by means of a functional behavior representation of the application model components which drive the architecture model components. This is done in such a way that the architecture model components can interpret the application's functional behavior representation as far as non-functional aspects such as delays, throughput, service cost and availability are concerned. In this chapter we propose to represent the functional behavior of application components as Symbolic Programs (SP). These are abstract non-executable CDFGs that can be converted to Symbolic Instruction Traces (SIT), by merging the SP representation with a Control Trace that memorizes the outcomes of data-dependent constructs.

The remainder of this chapter is organized as follows. Firstly, in Section 2.3, we introduce entities used by symbolic programs in a top-down manner, starting with the highest level possible - i.e. the application network - and ending with the lowest level possible - the smallest details used by symbolic programs. Then, in Section 2.4 we explain the details of symbolic programs, their syntax and semantics. Finally, in Section 2.5, we provide briefly those symbolic program transformations used prior to and during the mapping process (See Chapter 4).

2.3 Definitions and Terminology

This section provides a glossary of terms found in this thesis. To begin, we interchangeably use terms application model components and processes.

Process. *A process is a computational component of a concurrent (parallel) application model, particularly a KPN model, that performs a certain behavior, expressed via an ordered graph of repetitive Read, Execute and Write statements. Its behavior has a representation that can be interpreted, possibly after transformations, by architecture model components. A process has its own exclusive memory space which is not visible to other processes belonging to the same application representation* $\diamond$

The applications in this thesis are modeled by means of application representations.

Application Representation. *An application representation is an abstract view of the behavior of application model components and their communication interfaces.*◇

When two or more processes need to communicate, they use inter process communication.

Inter Process Communication. *Inter process communication (IPC) is a well-defined protocol that processes have to adhere to when they exchange data. The protocol is wrapped into globally accessible interfaces, which must guarantee: (1) exclusivity of the internal process memories, and (2) atomicity of the accesses to the global IPC information* ◇

The process representations and the IPC representations are important because they drive the mapping process and the exploration process. Thus, the better we represent them the better (easier, more accurate, more beneficial) the mapping and exploration will be. Particularly, the application KPN consists of application processes and application FIFO channels. We opt for usage of control and data intermediate representations to represent processes and specially tailored `read` and `write` symbolic instructions to represent the IPC interfaces.

Intermediate Representation. *An intermediate representation is a program for an abstract machine which has three important properties: (1) it is easy to produce, (2) it is easy to translate into the target-machine program and (3) it supports its purpose (e.g. transformation).* [49] ◇

Intermediate representations can appear in a variety of forms. Most of them use basic-blocks and control-points to capture information about the application behavior given by the application model.

Basic Block. *A basic block is a sequence of consecutive statements in which a flow of controls (dependencies) enters at the beginning and leaves at the end without halt or possible branching, except at the end.* ◇

This is illustrated in Figure 2.2.

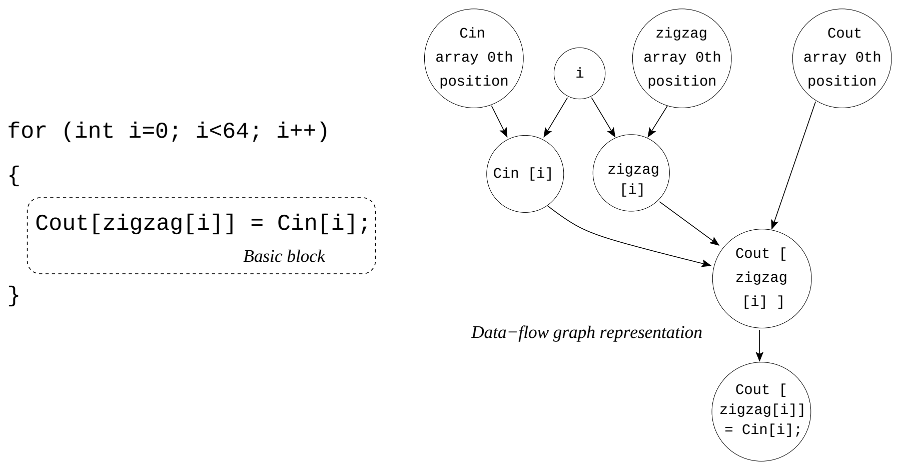
The left part of Figure 2.2 shows a part of the Zig-Zag routine [3] in the JPEG still image codec standard [50]. Figure 2.2 illustrates a basic block as we see it. The DFG of this basic block is shown on the right in Figure 2.2. It is worth noting that it expresses a partial order of the operations within the basic block.

The DFG in Figure 2.2 is acyclic due to its basic block features: no control flow nodes are allowed. Symbolic programs do rely on control point annotation - the control points form an obligatory part of this application representation as we will see in Section 2.4.

Control Points - Unconditional & Conditional. *A node in an intermediate representation of an application program is said to be a control point if its purpose is to direct the control flow in the program. When directing is done independently of any input data, the control point is said to be unconditional. Otherwise, when some input data evaluation is required, the control point is conditional* ◇

The code-sample in Figure 2.2 contains control points, for example, the `for` statement on the left hand-side. Note that conditional control points are always dependent on some data

[3] The Zig-Zag routine is used to facilitate entropy coding in the JPEG standard.



```
for (int i=0; i<64; i++)
{
  Cout[zigzag[i]] = Cin[i];
}
```

Figure 2.2: Illustration of the basic-block definition in the JPEG standard Zig-Zag routine, [50].

(e.g., the `for` statement depends on an index `i`). However, there is a difference in the way data-dependency is established. In the case of the earlier `for` statement, the values of the `i` index are known *a priori* to the program execution - the loop boundaries are static. In some other cases that may not be so - the `if` conditional control is data-dependent on the run-time data input (the `symbol` value).

Data-dependent Control - Static & Dynamic. *A conditional control point is a static control point if the data value is known before execution time. Otherwise, it is a dynamic control point* ◇

This is illustrated in Figure 2.3.

```
1 for (l=1; l<64; l++)
2 {
3    symbol = get_symbol(HUFF_ID(AC_CLASS, AC_HT));
4    if (symbol == HUFF_EOB)
5    {
6        break;
7    }
8    ...
```

Figure 2.3: A code sample that illustrates conditional control and unconditional control

Figure 2.3 illustrates both static and dynamic data-dependent controls. The code shown there is part of the variable-length decoding of the AC coefficients in the JPEG standard, [50]. Line 1 is a for-loop head where the number of iterations is controlled. Thus, Line 1 illustrates a conditional control point. Line 4 is an *if* selection statement which guards the program flow between Lines 5 and 7. Since the retrieved symbol is evaluated (compared with a constant), this is also a conditional control. Moreover, since the symbol value is not known *a priori*, this is a dynamic control. Finally, Line 6 illustrates the unconditional control point (a jump), which directs the program flow towards the exit point, explicitly known in advance.

2.4 Symbolic Programs

Symbolic Programs. *A symbolic program (SP) is a bipartite representation of a process consisting of (1) a structural element - an abstract CDFG and (2) a behavioral element - which is a control trace that represents the outcomes of conditional statements for a particular data set* $\diamond$

Symbolic programs are abstractions of process behaviors which were earlier given either as low-level Control Data Flow Graphs or as source-code descriptions. The SP representation is inspired by the idea of *abstract execution* [51]: (1) the application model (KPN) is executed to produce traces of control data and (2) the control data traces are used later, when simulating the execution of an SP mapping onto an abstract architecture model. By using the SP representation, we want to capture the behavior and the structure of the application process. We explain the way a symbolic program represents an application process using as reference the code-example in Figure 2.8 and the corresponding symbolic program in Figure 2.12.

2.4.1 SP Structure

The C/C++ code in Figure 2.6 provides an executable specification of the 'Vectorize' process in the adaptive QR matrix decomposition algorithm [52]. This code was automatically generated by `Compaan` tool. The execution of this code generates the communication (`read`, `write`) and computation (`execute`) loads. Since the code is static (e.g., the sequences of `read`, `execute`, and `write` SIs are not dependent on the input data values) it is possible to rewrite it into a little bit different code - Figure 2.8, and still produce the same sequences by execution of the rewritten code. It is worth noting here that rewriting code from Figure 2.6 to Figure 2.8 corresponds to 'detection of variants' source code transformation explained later in Section 2.5.1. However, to keep things simpler we use Figure 2.8 to explain the SP structure and how it is derived.

This code implements a Kahn process: `read` function calls [4] and `write` function calls [5] obey the Kahn semantics [6]. The internal computation is wrapped in the `Vec` function call. The structure of the symbolic program text [7] is depicted in Figure 2.12 and is given in accordance with the SP syntax (see Section 2.4.4).

The source code is arranged so as to expose the binary decision tree of the application process. This helps when abstracting source code details. For example, the code section between lines 6 and 10 in Figure 2.8 is collapsed to only two `read` IPC nodes. Similarly, the code section between lines 16 and 18 is a single computation node - a `Vec` function call. Consequently, the source code in Figure 2.8 and the abstract representation in Figure 2.12 have the same structure.

[4]Lines: 8,9,14, and 15.

[5]Lines: 22,23,28, and 36.

[6]i.e.a processes is blocked when trying to consume data from empty channels and the inter-process communication is done through FIFOs only

[7]This is a textual or code view of the SP. It is also possible to view SPs as their corresponding abstract CDFGs by drawing directed arcs each following a series-parallel relation in the SP text. Please, be aware that these CDFGs are **not executable**.

However, no functional information is available in the abstract SP text in Figure 2.12. That is, the C/C++ code maintains loop index values, as indicated in lines 2 and 3, and depending on these loop index values the code performs certain actions as indicated in lines: 6,11,18,20,25,31,34 and 38. On the contrary, the SP text does not process or compute any index values; the `condition` text only marks the place of the corresponding control points [8] in the C source code, but neither arithmetic nor logical computation takes place. Nevertheless, the structure is preserved, and so are the dependencies. Notice that a designer can recover the edges of the corresponding abstract CDFG by following series-parallel markings [9], IPC access point identifiers [10] and variable names [11]. This is much more than an SIT representation[12] can offer.

2.4.2 SP Behavior

Mapping of the application model on an architecture model associates components of the two models together: the application model produces SP representations (e.g., Figure 2.12) and the architecture model has to interpret them. However, an SP is, by definition, independent of any particular data-set that drives the application, whilst the architecture component has to interpret the SP for a specific data-set. It, therefore, also needs information regarding the outcomes of data-dependent constructs. The outcomes are stored in a control trace, which - when combined with the SP text - can provide an SI trace that is valid for a particular data-set and can be interpreted by the architecture components. The control trace appears as annotation in the binary decision tree structure of the C/C++ code. In Figures 2.4, 2.6 and 2.8 the annotations are given by C comments. It is worth noting that the annotations of C/C++ code in Figures 2.6 and 2.8 correspond uniquely to the conditions of SP texts in Figures 2.7 and 2.12, respectively. During the process execution in an unconstrained domain [13], the inserted annotation code produces a sequence of tokens - the process *control trace*. This control trace (CT) is data-dependent, meaning that a different control trace may (and usually does) appear when process input-points are fed with a different data-set. This is illustrated in Table 2.2 . The table shows that for the JPEG decoding KPN network (for details see Chapter 4, Section 4.5.3) although the same SPs are used for the decoding, the CTs are different (observing both their sizes and their contents) since different images are used.

Table 2.2 illustrates that CTs separate data-dependent process information from the process control structure. The CT examples shown so far assume that the only information available CTs is whether some selection/repetition condition was evaluated as `'true'` or `'false'` for some input data set - see Figure 2.6. However, there are dynamic algorithms that contain more complicated data dependencies. The example in Figure 2.4 is a C/C++ excerpt from the $raster$ process of the JPEG decoding process network in Figure 4.10. The corresponding peace of the SP text is shown in Figure 2.5. The sample CTs for the $raster$ process for the images from Table 2.2 is given as follows:

[8] `for`, `if`, or `else`.
[9] In an SP text, ';' marks **sequential** relations between SIs, and '||' marks **parallel** relations among SIs.
[10] The port arguments are indicated by the decimal numbers immediately after `read` and `write` calls.
[11] The arguments of `read`, `write`, `execute` nodes, as well as the arguments of `condition` nodes.
[12] See `Spade` in Section 1.6.1.
[13] Unlimited resources.

$\frac{Images\rightarrow}{SPs\downarrow}$	*philips* (resolution: 50×67)	*shuttle* (resolution: 669×1004)
frontend	$\frac{44142By}{b7cd3a3f51717e1d232648f5ace71fcf}$	$\frac{5231953By}{48550a1b99ff6f754ff60377d328a715}$
vld	$\frac{447656By}{c45b364f780b04ab101940eb57f467d9}$	$\frac{51572812By}{096320bd9335766a222a944401561657}$
iq	$\frac{42074By}{3f2889cb78ebabb798fc5b2a383db55d}$	$\frac{5451634By}{67838c7ea0a73e1c59be3fc59327b242}$
izz	$\frac{39605By}{ca801b7f6261c4b67e727627d87abd9c}$	$\frac{5239085By}{1a2984cfc8e74b599459c10293ee7f2f}$
idct2d	$\frac{124085By}{efebf544c999aa1baf6fbe1235497cd8}$	$\frac{16368521By}{bfcf058cc2633e78feb702a1906f2213}$
raster	$\frac{32618By}{89ff1f47f19c261cadc1c7cca6d208fb}$	$\frac{4132831By}{2242765377c02221c65b49915203a49e}$
yvs	$\frac{1891By}{fb150456dd49132eba84357c1a386733}$	$\frac{30136By}{0280577950309f0623e2e876da0a3afb}$
cbvs	$\frac{1397By}{23aed27ef2ef826fda2161615cf667e8}$	$\frac{22104By}{38496317e3be52f37e4f9bb2c91c32d9}$
crvs	$\frac{1397By}{19ca11e047693f1dac3e82546ba9f098}$	$\frac{22103By}{67935bec3f5ffa9fd4d99d62e652ce2c}$
yhs	$\frac{50925By}{6d88fa4935e8396af1883deaf5255791}$	$\frac{10085185By}{6cb466555b629a28ebf7fc025c72cc48}$
cbhs	$\frac{34175By}{1320c8a02d914d983b98aacce89eb75e}$	$\frac{6731825By}{5e123a5d02c81ae3e91ba1231734a58d}$
crhs	$\frac{1397By}{1320c8a02d914d983b98aacce89eb75e}$	$\frac{6731825By}{38496317e3be52f37e4f9bb2c91c32d9}$
matrix	$\frac{16755By}{d092b01f2d52e66319b42a126b41755e}$	$\frac{3358385By}{d8c6d1caaa66bf93f2f162be7f138382}$
sink	$\frac{17425By}{0dad373c9e573d88e964798661ee2ae2}$	$\frac{3368425By}{06851224e67855ccb1bad9b634762f9d}$

Table 2.2: The illustration of data-dependent Control-Trace feature using the ratio of values $\frac{filessizes}{contentssignature}$ for the fixed JPEG decoding network of SPs given in Figure 4.10. The CT files sizes are taken from the file system by `'ls -al'`, while the file content signature is obtained by running the '`md5sum`' digest program on the contents of CT files. The Message Digest 5 (MD5) algorithm implements a cryptographic hash-function which ensures that when operating on an arbitrary-length data returns a distinctive 128-bit hash value [5]. **Note:** One can observe that for the same SP text (e.g., vld), the JPEG decoding network running on the two different JPEG images produces CTs of different sizes (`ls`) and data content values (`md5sum`).

```
0:   for (int l=0; l<8; l++) {
1:   {  /* branch C12(l) taken */
2:      if ((x<X[ci]) && (X[ci]-x >= 8))
3:      {  /* branch C13(x X) taken */
4:          read (IP7, &stripe[ci][(8*v+l)*X[ci]+xi+8*h], 8);
5:          /* branch C14(x X) NOT taken */
6:          /* branch C15(x X) NOT taken */ }
7:      else if ((x<X[ci]) && (X[ci]-x < 8))
8:      {  /* branch C13(x X) NOT taken */
9:          /* branch C14(x X) taken */
10:         read (IP7, &stripe[ci][(8*v+l)*X[ci]+xi+8*h], X[ci]-x);
11:         read (IP7, dummy, 8-(X[ci]-x));
12:         /* branch C15(x X) NOT taken */ }
13:     else
14:     {  /* branch C13(x X) NOT taken */
15:          /* branch C14(x X) NOT taken */
16:          /* branch C15(x X) taken */
17:          read(IP7, dummy, 8); } }
18:   /* branch C12(l) NOT taken */
```

Figure 2.4: A code sample of the JPEG $raster$ process. This code sample illustrated data-dependent `read` operations at Lines 10 and 11 - the amount of data read is decided at the run time based on $X[ci] - x$ and $8 - (X[ci] - x)$, respectively.

- $philips.jpg$ ...$(c12 \equiv true) \prec (c13 \equiv false) \prec (c14 \equiv true) \bigwedge (R7(6) \prec R7(2)) \prec (c15 \equiv false)$..., and
- $shuttle.jpg$ ...$(c12 \equiv true) \prec (c13 \equiv false) \prec (c14 \equiv true) \bigwedge (R7(1) \prec R7(7)) \prec (c15 \equiv false)$...,

where '$\prec$' indicates that the group (a data event enclosed with '(' and ')') on the left of the $\prec$ operator precedes the group on the right of the $\prec$ operator. It is worth noting that the `read` SIs at Lines 4 and 5 in Figure 2.5 are dependent on the `condition 14` and the run-time evaluations of the expressions $X[ci] - x$ and $8 - (X[ci] - x)$ in Figure 2.4. These evaluations depend, again, on the array X and the scalar-variable x, which values change depending on the data-input. Since the control is already linked to data input (e.g., the resolution of $philips.jpg$ is 50×67, and the resolution of $shuttle.jpg$ is 669×1004 - the file sizes are already shown in Table 2.2 , the $raster$ CTs for each of these two images are used to convey the results of the former expression evaluations. As a consequence, the $philips.jpg$ CT sequence says to a Program Unit (see Chapter 3, Section 3.4.2) and its Depth First Traversal engine (see Appendix A) that 6 tokens are to be read from the port 7 and put to the $stripe$ memory, followed by 2 tokens to be read from the port 7 and put to the $dummy$ memory - look above for $(R7(6) \prec R7(2))$. For the $philips.jpg$ CT sequence says to a Program Unit that 1 token is to be read from the port 7 and put to the $stripe$ memory, followed by 7 tokens to be read from the port 7 and put to the $dummy$ memory - look above for $(R7(1) \prec R7(7))$. The JPEG decoding network in Figure 4.10 contains a few such processes, so their corresponding SP texts and CTs make use of this behavior.

```
0:  loop condition 12 (l) {
1:     condition 13 (x X ci) {
2:        read 7 (stripe, 8); }
3:     condition 14 (x X ci) {
4:        read 7 (stripe,0);
5:        read 7 (dummy, 0); }
6:     condition 15 (x X ci) {
7:        read 7 (dummy, 8); } }
```

Figure 2.5: A SP text sample corresponding to the C/C++ sample of the JPEG $raster$ process in Figure 2.4. **Note:** Due to the fact that data-dependent `read` operations at Lines 10 and 11 decide on amount of data to be read at the run time, the corresponding SP text in here at Lines 4 and 5 (respectively), have both value 0 for the amount of data to be read from port 7 - so called 'budget'. The 0 value says to the corresponding Program Unit and its Depth First Traversal Engine to acquire the 'budget' value from the corresponding CT.

The SPs are not a so-called 'silver bullet' to resolve all the application representation modeling issues. As indicated in [9], Sequence-Parallel representations cannot cover for all partially-ordered cases that one application specification may have. As we indicate later on in Section 2.5, not all information is kept in application SPs, and, therefore, only very few classes of SPs can be transformed successfully without knowing the original C/C++ code. For example, it would be beneficial for SP-transformations if the SP `condition` statements could capture the exact expressions that are evaluated in the original C/C++ code. However, due to the possible complexities of C/C++ specifications that requirement cannot always be met. The currently supported *behavior-capturing capabilities* of SPs are elaborated in more detail in Sections 2.4.3 and 2.4.4.

Regardless of the limited behavior-capturing capabilities, SPs posses one special feature that distinguishes them from the other application representations; SPs use 'separation of concerns' to separate input data set dependent and independent behaviors. This feature has been shown earlier in Figure 2.1 in a more abstract manner. Disassociation of process structure and control helps in creating more flexible architecture models (see Chapter 3).

```
/* The Compaan-like C code */
void ND_5 :: main() {
   for ( int k = 1 ; k <= -2*k+T+1 ; k += 1 )
   {  /* branch C1(k,T) taken */
      for ( int j = 1 ; j <= N ; j += 1 )
      {  /* branch C2(j,N) taken */
         read( RP_11, a );
         if ( j-2 >= 0)
         {  /* branch C3(j) taken */
            read( RP_12, b ); }
         else
         {  /* branch C3(j) NOT taken */ }
         if ( j-1 == 0)
         {  /* branch C4(j) taken */
            read( RP_13, b ); }
         else
         {  /* branch C4(j) NOT taken */ }
         Vec( a, b, c, d );
         if ( -3*k+T >= 0)
         {  /* branch C5(k,T) taken */
            write( WP_19, c ); }
         else
         {  /* branch C5(k,t) NOT taken */ }
         if ( -j+N-1 >= 0)
         {  /* branch C6(j,N) taken */
            write( WP_18, d ); }
         else
         {  /* branch C6(j,N) NOT taken */ }
      /* branch C2(j,N) NOT taken */ }
   /* branch C1(k,T) NOT taken */ }
}
```

Figure 2.6: The `Compaan`-generated C/C++ source code of the 'Vectorize' process in the adaptive QR matrix decomposition algorithm [53]. **Note:** All conditional statements (selections and repetitions) are annotated by hand with "branch taken" or "branch NOT taken". The *else* branches that do not exist in `Compaan` are added for easier annotation.

2.4.3 Semantics

Looking at Figure 2.1, SP representations have a front-end role for the mapping process: An application KPN model behavior is converted to a set of SPs and a set of related Control Traces, and then these SPs and Control Traces are successively transformed towards targeted mapping. This is indicated as *transformation steps* in the same figure. In addition to this, Figure 2.1 brings an extra message: In contrast to SI representations, SP representations including their Control Traces are complete representations in which instructions (conditional as well as non-conditional) are partially ordered. Moreover, SPs can very well be hierarchically defined [14].

```
/* Symbolic Program text */
main {
   loop condition 1 (k T) {
      loop condition 2 (j N) {
         read 11 (a, 1);
         condition 3 (j) {
            read 12 (b, 1); }
         condition 4 (j) {
            read 13 (b, 1 ); }
         execution 0 (in a in b out c out d, 1);
         condtion 5 (k T) {
            write 19 (c, 1); }
         condition 6 (j N) {
            write 18 (d, 1); } } } }
```

Figure 2.7: The SP text of the `Compaan`-generated C/C++ source code of the 'Vectorize' process from Figure 2.6. **Note:** All conditional statements (selections and repetitions) have the identical annotations as they have in the C/C++ source code. From the structure of the SP text and its correlation with the original `Compaan` C/C++ code, it is implicit that the SP-text and the C/C++ code share the same abstract CDFG. It is also implicit that the nodes of the SP text are totally ordered and that the interpretation of the compound (composite) nodes - the nodes indicated between '{' and '}' - happens in the Depth First Traversal manner.

Hierarchy. The hierarchical features introduce structure in the SP, which helps to cope with SP representation complexities. The SP hierarchy replicates the *separation of concerns* concept of classical structural (procedural) Software Engineering. Each C procedure or even each basic-block could be a sub-SP text [15]. For example, the `execute` statement in Figure 2.12 - line 10 - may itself result in an SP through a `program_call`.

Partial order. The instructions in an SP are partially ordered. However, the instructions being abstract, their underlying detailed behavior is still assumed to be sequentially specified, i.e., as given in the source code. For example, the `execute` statement in Figure 2.12 - line 10 - has a sequential specification of 'Vectorize' in the source code given in Figure 2.8. Architecture components, on the other hand, may be capable of executing an instruction in parallel, or may execute the instruction in a sequential order that is different from the given one. This requires conversion of *strictly ordered* instructions to partially ordered instructions (and possibly back to strictly ordered instructions) as proposed in [30] and [36]. Figure 2.13 shows the parallel counterpart of the sequential SP in Figure 2.12

Information preserving. Looking back at Table 2.1, both SIT and SP representations are abstract representations, which implies that some information is necessarily lost. Thus, though the SP representation is complete (see the same table), translations, transformations and abstractions used when an SP is derived will result in some information loss - the coarser that abstractions are, the greater the information loss will be. The claim of this chapter is that the amount of lost information in an SP representation is much smaller once compared to a SIT representation. In other words, the information preservation of an SP representation is better than the information preservation of a SIT representation. We illustrate the above statement

[14]We re-invent the idea of so-called *structured procedural design* in domain of Software Engineering.

[15]The `function` text description serves this purpose.

```
/* Variant-friendly C code */
void ND_5 :: main() {
   for ( int k = 1 ; k <= -2*k+T+1 ; k += 1 )
   {  /* branch C1(k,T) taken */
      for ( int j = 1 ; j <= N ; j += 1 )
      {  /* branch C2(j,N) taken */
         if ( j-2 >= 0)
         {  /* branch C3(j) taken */
            read( RP_11, a );
            read( RP_12, b );
            /* branch C4(j) NOT taken */ }
         else
         {  /* branch C3(j) NOT taken */
            /* branch C4(j) taken */
            read( RP_11, a );
            read( RP_13, b );
         }
         Vec( a, b, c, d );
         if ( -3*k+T >= 0)
         {  /* branch C5(k,T) taken */
            if ( -j+N-1 >= 0)
            {  /* branch C7(j,N) taken */
               write( WP_19, c );
               write( WP_18, d );
               /* branch C8(j,N) NOT taken */ }
            else
            {  /* branch C7(j,N) NOT taken */
               /* branch C8(j,N) taken */
               write( WP_19, c );
            }
            /* branch C6(k,T) NOT taken */ }
         else
         {  /* branch C5(k,T) NOT taken */
            /* branch C6(k,T) taken */
            if ( -j+N-1 >= 0)
            {  /* branch C9(j,N) taken */
               write( WP_18, d );
               /* branch C10(k,T) NOT taken */ }
            else
            {  /* branch C9(k,T) NOT taken */
               /* branch C10(k,T) taken */ }
          }
      /* branch C2(j,N) NOT taken */ }
   /* branch C1(k,T) NOT taken */ }
 }
```

Figure 2.8: The C/C++ sequential source code of the ’Vectorize’ process. **Note:** This code has the identical behavior with the code in Figure 2.6, but it is ’variant friendly’. Such code is used for so-called ’variant detection transformation’ modeling and for generating the CTs for the case study.

by the following example.

We made an analytical comparison to show how large a simulation error one can get by using only SITs, given that the final system can exploit the partial ordering not addressed by SITs. The SP in Figure 2.13 has been transformed twice at the source-code level. The first transformation results in some SP instructions being partially ordered [16]. The second transformation [17] further enhances the parallelism within the SP. These two cases are compared against the strictly ordered SP, which corresponds to the classical SIT [7]. The comparison is done by means of analysis only in the following way:

1. The following assumptions were made:
   - All `read` and `write` SIs are considered to take the same amount of time - $(\forall i : i = 1 : R_i, W_i \equiv C)$, where R_i and W_i are the individual SIs and C is the amount.
   - All `execute` SIs are considered to take the same amount of time - $(\forall i : i = 1 : E_i \equiv E)$, where E_i are the individual SIs and E is the amount.
   - No blocking appears.
   - All sequential SIs are contributing their amounts to the total delay amount by means of a simple addition $+$.
   - All parallel SIs are contributing their amounts to the total delay amount by means of a $\max$ function.
2. The three SPs were observed: the sequential SP as in Figure 2.12, the partially ordered SP as in Figure 2.13, and the parallel scheduled SP as in Figure 2.14.
3. The total delay for SP in Figure 2.12 can be approximated as: $D_{SP1}(x,y) = y \cdot (4 \cdot x + 1)$, where $x = \frac{C}{E}$ and y is the number of the loop iterations.
4. The total delay for SP in Figure 2.13 can be approximated as: $D_{SP2}(x,y) = 2 \cdot x + y \cdot \max(x,1)$.
5. The total delay for SP in Figure 2.14 can be approximated as: $D_{SP3}(x,y) = y \cdot (2 \cot x + 1)$.
6. The relative improvement $i_j(x,y)$ is defined as $\frac{|\Delta D_j(x,y)|}{D_j(x,y)}$ for each of the cases, as follows:
   - The relative improvement of SP2 vs. SP1: $i_1(x,y) = \frac{|y \cdot (2 \cdot x+1) - y \cdot (4 \cdot x+1)|}{y \cdot (4 \cdot x+1)}$
   - The relative improvement of SP3 vs. SP1: $i_2(x,y) = \frac{|2 \cdot x + y \cdot \max(x,1) - y \cdot (4 \cdot x+1)|}{y \cdot (4 \cdot x+1)}$
7. The `gnuplot` engine was used to create the graphical representations of the relative improvements. See Figures 2.9 and 2.10.

[16]See *detection of variants* in Section 2.5.
[17]See *loop scheduling* in Section 2.5.

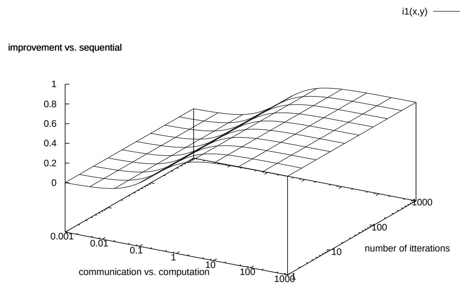


Figure 2.9: SP2 vs. SP1 - $i_1(x, y)$.

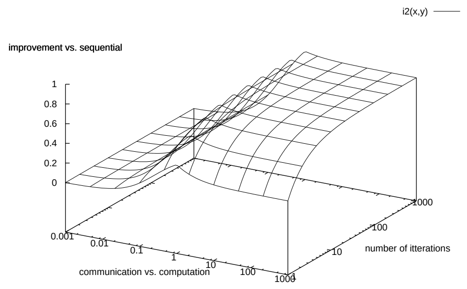


Figure 2.10: SP3 vs. SP1 - $i_2(x, y)$.

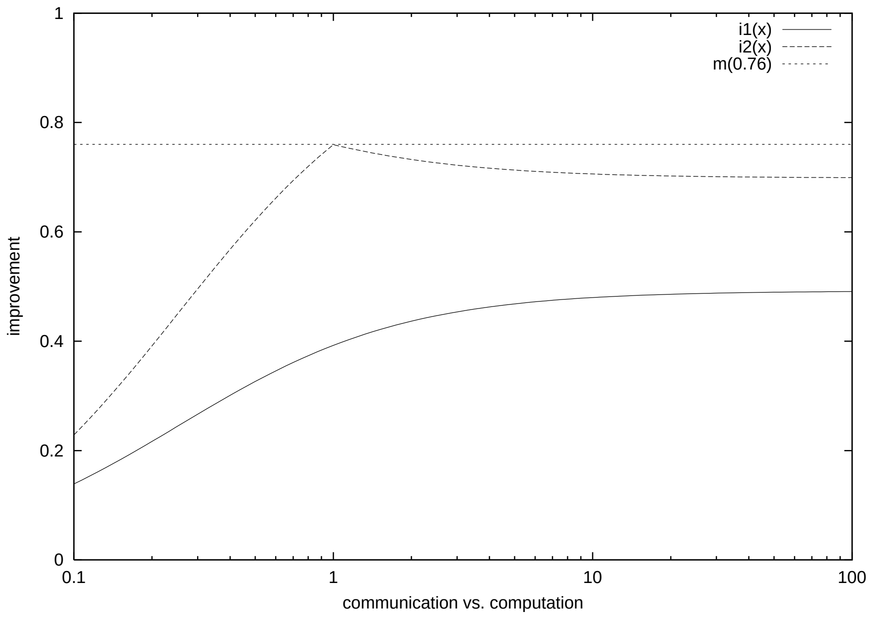


Figure 2.11: Relative improvements (for 10 iterations only) in the worst case execution time, measured with the range of values $(0, 1]$ - 1 indicating the perfect match - when using SP partial-order annotations against the pure sequential SIT. The SIT-like SP is shown in Figure 2.6. The first improvement $i1(x)$ is achieved by detecting the variants of partially ordered `read`, `execute`, and `write` SIs - see Figure 2.13. The second improvement $i2(x)$ requires a task-level examination because it represents a rescheduled SP. The maximum achieved improvement for the SP code in Figure 2.14 is indicated by the constant $m(0.76)$.

Figures 2.9 and 2.10 show that modeling of the partial order for system-level performance exploration is important. To help better understanding of these two figures we have created a joined plot for $y = 10$ (the number of the iterations). The results are shown in Figure 2.11. As

```
main {                                               //     e.g.,:
   loop condition 1 (k T) {                          //     c1 T
      loop condition 2 (j N) {                       //     c2 T
         condition 3 (j) {                           //     c3 T
            read 11 (a, 1);                          //
            read 12 (b, 1); }                        //
         condition 4 (j) {                           //     c4 F
            read 11 (a, 1);                          //
            read 13 (b, 1); }                        //
         execute 0(in a in b out c out d, 1);        //
         condition 5 (k T) {                         //     c5 T
            condition 7 (j N) {                      //     c7 T
               write 19 (c, 1);                      //
               write 18 (d, 1); }                    //
            condition 8 (j N) {                      //     c8 F
               write 19 (d, 1); } }                  //
         condition 6 (k T) {                         //     c6 F
            condition 9 (j N) {                      //
               write 18 (c, 1); }                    //
            condition 10(j N) { skip; } } } } }      // ... the next value for ’c1’
```

Figure 2.12: **Left:** SP text of the ’Vectorize’ process, in the adaptive QR matrix decomposition algorithm. **Right:** A peace of CT associated with the SP text on the left and generated by the ’Vectorize’ process in Figure 2.8 for some fictive data set-up. **Note:** `condition` N on the left corresponds to `c`N on the right; T stands for ”true” or ”branch taken”, while F stands for ”false” or ”branch NOT taken”. The inner conditions that are eliminated by the ”false” value of the outer conditions (e.g., `condition` 9 eliminated due to `condition` 6 evaluated as ”false”) are not in the presented CT piece.

```
main {
   loop condition 1 (k T) {                          //     c1 T
      loop condition 2 (j N) {                       //     c2 T
         condition 3 (j) {                           //     c3 T
            read 11 (a, 1) ||                        //
            read 12 (b, 1); }                        //
         condition 4 (j) {                           //     c4 F
            read 11 (a, 1) ||                        //
            read 13 (b, 1); }                        //
         execute 0(in a in b out c out d, 1);        //
         condition 5 (k T) {                         //     c5 T
            condition 7 (j N) {                      //     c7 T
               write 19 (c, 1) ||                    //
               write 18 (d, 1); }                    //
            condition 8 (j N) {                      //     c8 F
               write 19 (d, 1); } }                  //
         condition 6 (k T) {                         //     c6 F
            condition 9 (j N) {                      //
               write 18 (c, 1); }                    //
            condition 10(j N) { skip; } } } } }      //
```

Figure 2.13: Parallelized SP code of the Vectorize process presented earlier in Figure 2.12. **Note:** The CT on the right is the identical one to the CT in Figure 2.12. This is due to the fact that the original code is static with regards to loop-indexes and conditions dependent on loop-indexes. Consequently, for such cases no CT transformations are required.

one can see, the improvement [18] varies between 0 and 0.76 with respect to the type of process-

[18]This should be understood as a *potential concurrency* so that there exists a possibility to execute some process

ing element being used in the system-level simulation. This means that absence of modeling features in SITs can seriously jeopardize simulation accuracy. The SP, on the contrary, preserves application process information so that more options are available and accuracy can be improved.

2.4.4 Syntax

Figure 2.12 is an SP text representation of the C/C++ code of an application process. Its SP syntax is the subject of this subsection.

The symbolic program syntax must be expressive enough to specify abstract CDFGs, leaving enough room for further manipulation. This syntax should not support fine-grain operations which are customary in CDFG representations. Thus, it is necessary to find a minimal set of expressions that are sufficient to support both requirements: expressiveness and granularity. Of course, symbolic programs are abstract, and therefore, only `read`, `execute`, and `write` are inherently partially ordered.

To summarize, SPs are composed of SIs representing already mentioned communication events (`read` and `write`) and computation events (`execute`), as well as outcomes of control events (Control Trace). SIs are partially ordered, whilst Control Trace information is totally ordered.

The symbolic program syntax can conveniently adhere to the syntax of the standard parser-generator tool called Yacc [54]. The implementation details of symbolic programs expressed in the Lex-Yacc format are given in Appendix A

2.5 SP Transformations

In the preceding sections we have introduced the symbolic program representation. We claim that symbolic programs are well suited to associating application model components and architecture model components together. Application model SP representations can be easily transformed to SI representations by combining them with the corresponding CTs, and these SIs can be further interpreted by architecture model components that do not necessarily match the application model components.

SP transformations are source-to-source transformations. The input is an SP and the output is a transformed SP. The transformation mimics high-level design decisions on the source code from which the SP was derived. The SP transformations may be: (1) Intra-task (i.e. intra-process) transformations, or (2) Task-level (i.e. process-level) transformations. Transformations illustrated in this section are **only applicable to a certain set of SPs** - namely, to the set which SPs do not have dynamic data-dependencies within conditions of selection and repetition statements.

code concurrently - the choice of whether this is going to happen is still dependent on the mapping model.

2.5.1 Intra-task Transformations

Intra-task transformations represent high-level abstractions of some well known ILP [19] techniques. For example, if the processor model onto which an SP is mapped can process more than one SI in parallel, then the employed SP transformations may expose this characteristic explicitly in the SP. Such transformations are for example: *Detection of variants* [20], and *loop scheduling*. Notice that transforming an SP usually implies *control-trace transformation* as well.

The ILP techniques which abstract the manifestations which we try to model are based on a dependency analysis of both the arguments of SIs and the arguments of conditionals in a particular SP. We explain these transformations using the examples in Figures 2.6, 2.13, and 2.14. The SP in Figure 2.6 may be improved in two ways: Firstly, by applying the detection of variants if they are not already detected, some SIs become partially ordered as shown in Figure 2.13; secondly, by scheduling the loop iterations of the SP shown in Figure 2.13, some additional SIs become partially ordered and the number of control-points decreases. The resulting SP is depicted in Figure 2.14. It is worth noting again that the SP we used for illustration does not contain data-dependent conditions, as well as, that it has been already structured in a form of the binary decision tree.

```
main {
   loop condition 1 (k T) {
      read 11 (a, 1) ||
      read 13 (b, 1);
      read 11 (a, 1) ||
      read 12 (b, 1) ||
      execute 0(in a in b out c out d, 1);
      loop condition 2 (j N) {
         read 11 (a, 1) ||
         read 12 (b, 1) ||
         execute 0(in a in b out c out d, 1) ||
         write 19 (c, 1) ||
         write 18 (d, 1); }
      execute 0(in a in b out c out d, 1) ||
      write 19 (c, 1) ||
      write 18 (d, 1);
      write 18 (c, 1); }
```

Figure 2.14: The pipe-lined SP code of the 'Vectorize' process earlier given in Figure 2.12.

Detection of Variants. The transformation which performs the detection of variants is formally explained in [55]. Here, we present the modeling effect of this type of transformation and we do it by way of example only. We assume that 'variants' can be detected either manually or the original C/C++ code is given in the form of a binary decision tree (as it is the case in Figure 2.12). Since SPs keep the dependency information by their definition, sometimes such tree can be developed manually. When the manual detection of variants is employed, after examining the dependencies, the selection statements of an SP are merged in such a way that the direct product of this merging represents a decision tree whose leaves are symbolic `read-execute-write` instructions. When these instructions are guarded by the same conditional, and when they do not depend on each other, we say that they represent a *variant*.

[19] Instruction-Level Parallelism

[20] 'Variants' are basic blocks associated with leaf conditions in a leaf in a binary decision tree.

As a direct advantage of this transformation, the SIs within a single variant can be processed in parallel (see Figure 2.13). Note that each SI contains dependency information, i.e., where the SI arguments come from and where they go to. In Figure 2.12, the `execute` takes the `b` argument either from FIFO channel 12 or from FIFO channel 13. Obviously, within a single loop iteration there is a total order between the `execute` and the `read` SIs. However, `read` SIs from FIFO channels 11 and 12, or 11 and 13, can overlap. That is why it was possible to replace the sequential symbol ";" at lines 5, 8, and 13 in Figure 2.12 with the partial order symbol "$\|$" at the same lines in Figure 2.13. However, as a disadvantage, the size of the symbolic program grows because each variant is now a `trace statement` which contains at least a single bundle of SIs[21].

Loop Scheduling. SP loop scheduling is possible only in a limited amount of cases (in the scope of this thesis we are actually interested only in the case illustrated in Chapter 4, Section 4.5.2). Loop scheduling techniques modify the order of occurrence of certain SIs in order to allow better utilization of processor component resources. As a side effect, they may reduce the number of conditionals in the loop body (see Figure 2.14). Note that different loop iterations of the SPs in Figures 2.12 and 2.13 may overlap, so the application of *software pipelining* [56] on the SP with parallel variants (Figure 2.13) produces an SP with pipelined instructions (Figure 2.14). Note, however, that loop scheduling techniques must not be applied when there is a dependency cycle [22] at the task level. Figure 2.15 illustrates the case where a cycle between processes $SP3$ and SPX via FIFO channels 12 and 18 exists. Here, there is a risk that a deadlock may occur because the loop-body of the SP in Figure 2.14 is skewed in such a degree that multiple (in this case two) `read 12` instructions appear before a single `write 18` takes place. Consequently, process SPX cannot perform any `read 18`, and hence, it may block without producing enough tokens in FIFO 12. Thus, the software pipe-lining technique applied to process loops must not be considered as an intra-task transformation only, because changes in communication behaviors affect system properties.

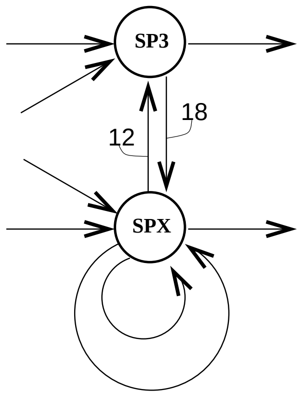


Figure 2.15: A deadlock caused by the re-scheduling transformation: $SP2 \mapsto SP3$

[21]The partially ordered SIs - see the syntax.

[22]This is the case when the architecture component relies solely onto compile-time reordering and it does not support run-time reordering.

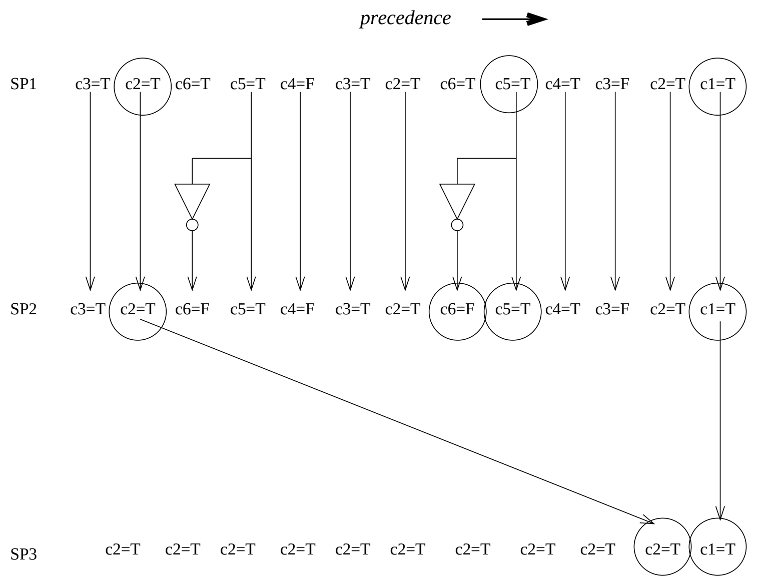

precedence
SP1
c3=T c2=T c6=T c5=T c4=F c3=T c2=T c6=T c5=T c4=T c3=F c2=T c1=T
SP2
c3=T c2=T c6=F c5=T c4=F c3=T c2=T c6=F c5=T c4=T c3=F c2=T c1=T
SP3
c2=T c2=T c2=T c2=T c2=T c2=T c2=T c2=T c2=T c2=T c1=T

Figure 2.16: Transformations of control traces with regards to transformations of $SP1$

Control-trace transformation. We have mentioned in Section 2.4.2 that SPs come with control information in the form of control traces, for each application process (or a task). Thus, when intra-task transformations, such as those in Figures 2.6, 2.13, and 2.14, are applied to an SP, the corresponding control traces have to be transformed as well since the intra-task transformations affect the SP control structure (e.g. its decision tree). Moreover, if such SP control structure changes can be expressed formally (e.g., in terms of logical expressions over original control structure in the starting/not-transformed SP), the same formalisms can be used to automate the transformation of the control trace - an example is given in Figure 2.16. The consecutive transformations of the SP in Figure 2.6 to the SP in Figure 2.13 ($SP1 \mapsto SP2$), and the SP in Figure 2.13 to the SP in Figure 2.14 ($SP2 \mapsto SP3$), include the corresponding transformations of control traces as shown in Figure 2.16. Note that each conditional control point of the SP in Figure 2.6 ($SP1$), is enumerated as a c_i, where i is unique in the scope of $SP1$. In other words, c_1 stands for the outer loop condition $k < T$, c_2 stands for the inner loop condition $n < N$, and so on. A similar enumeration is applied for the SP in Figure 2.13 ($SP2$) and the SP in Figure 2.14 ($SP3$).

When the transformation $SP1 \mapsto SP2$ is applied, the control trace of $SP1$ needs to be transformed as well. This new trace is labeled as $SP2$ in Figure 2.16. Since the transformation did not change the SP structure from the point of view of loops, loop-related trace events are unchanged. However, because two selection statements at the end of the inner loop are merged, the corresponding trace events, namely c_5 and c_6 in the control traces $SP1$ and $SP2$ have a different meaning and, therefore, a transformation is required. This is illustrated as a logical negation in Figure 2.16. Furthermore, when transformation $SP2 \mapsto SP3$ is applied, the selection statements are lost, the trace events that correspond to these selections, namely

c_3, c_4, c_5, and c_6, are simply ignored. Moreover, because software pipe-lining is applied (prolog and epilog code sections are generated at the expense of the number of inner loop iterations), the first two c_2 events in $SP2$ are also ignored. The resulting $SP3$ control trace is shown in Figure 2.16.

2.5.2 Task-level Transformations

Task-level transformations are transformations that affect the application model structure [23]. That is, the topology of the application network is changed, resulting in processes (in other words their SPs) and channels being removed or created. In the former case the transformation may be seen as *process merging*, while in the latter case as *process splitting*. In both cases, conditions have been given under which these transformations can be automated (See [57] and [58]).

In general, process merging implies that a valid schedule among merged processes has to be found. Such valid schedules cannot in general be obtained from the partially ordered SPs that represent the processes to be merged. A property of KPNs is that they are compositional, which also implies that a KPN is equivalent to a single process. Deriving a PN from a single process is much easier than deriving a single process that is equivalent to a PN.

We use [59] and [60] as the base for explaining process splitting in this thesis. In [59] split processes appear as a result of a transformation of the sequential program from which a KPN originates. In [60] process splitting is performed directly on a PN. However, both methods are not generally applicable.

Finally, note that each of the aforementioned task-level transformations also imply the transformation of associated control traces. For example, in the case of the merging of processes, a new control trace is formed by merging corresponding control traces. In the case of splitting a process, the control trace is also split to provide control traces for the split-processes. In Chapter 4, Section 4.5.2, we illustrate a process-splitting transformation applied to a 2-Dimensional Inverse Discrete Cosine Transform (2D-IDCT), resulting in a network of two 1-Dimensional IDCTs (1D-IDCT).

[23]This is also known as a *repartitioning* of the application code.

Chapter 3

Architecture Modeling

Who stands on a hill, even a small one, sees more than he who stands below the hill. [1]

3.1 Summary

The aim of this chapter is to provide an architecture modeling paradigm for embedded systems. Because we are dealing with levels of abstraction above RTL, we need models that adequately represent the underlying architectures. Abstract models are adequate when they can predict behaviors and costs in terms of relevant metrics. Without an adequate model of architecture, we cannot reason about our decision-making and its consequences in terms of performance and cost in abstract design space exploration. Our modeling is aiming at subsequent performance analysis and design space exploration in a computational way, i.e., using simulation and not in an analytical way. Because of that, some ingredients in the modeling paradigm will often be referred to as *representations* rather than *models*.

Thus, we represent or model an architecture in terms of that part of a system not containing its functional behavior. Roughly stated, an architecture consists of a number of computational units and a communication, synchronization and storage infrastructure. It also includes software that is necessary for operating the architecture.

Our architecture model describes architectures that are specific for an application domain that is called *streaming applications*. Recall that the two major characteristics of embedded systems are that they are application domain specific information processing systems and that they depend on their environment (input data characteristics). Streaming applications have functionalities that can be modeled conveniently in terms of the Kahn Process Network model

[1] In original: ”Ко на брдо ак’ и мало стоји, више види н’о онај под брдом.” These were the words of the historical character Bishop Danilo in the epic ”The Mountain Wreath”, written by Petar II Petrović Njegoš - Петар II Петровић Његош - (1813 A.D.-1851 A.D.) Serbian Orthodox Prince-Bishop of Montenegro and among the greatest poets of the Serbian language.

of computation (MoC). The streaming architectures on which they are mapped could also be specified in terms of the commonly addressed set of MoCs, yet they are more heterogeneous than the application models are or have to be. This chapter focuses on the heterogeneity of architecture models.

3.2 Introduction

Architecture modeling is crucial for the design space exploration method we envision in this thesis. Architecture exploration is the third step in a three step procedure: modeling, analyzing and finally exploring. Our approach is based on simulation and one objective is to provide methods that can achieve fast analysis and exploration, as well as high accuracy given the abstraction level that we are considering and the approach we are advocating. We rely on two key concepts: the first one is separation of concerns [61], meaning that we try to keep application modeling, architecture modeling and mapping issues as orthogonal as possible; the second concept is that of component based design [61]. Both concepts lead to re-use of components, in the sense that modifications and refinements do not have consequences in all parts of the system models, i.e., can be kept local and do not ask for very large component libraries. Our modifications are in terms of architecture composition modifications and these in turn are in terms of selection of components from a library of components and admissible interconnections of selected components.

3.2.1 Architecture definition

The classic definition of an architecture that has been promoted by the Instruction Set Architecture community is in terms of functionality - it can be built from instructions. The consequence of this is that there is no relation between the timing behavior of the implementation and its realization. The realization is changed frequently as a consequence of technology advances, while instructions are not changing (significantly) over time. Thus, the relation between functionality and timing has become completely decoupled over time so that it is almost impossible to analyze timing behavior. In embedded systems design, timing is a crucial issue, especially when there are real-time constraints imposed by the environment and when one has to predict timing behavior using abstract models that cannot be well specified if the relation between functional behavior and timing behavior is obscure.

We, therefore, define the architecture in a different and indeed, almost opposite way. We do not include functional behavior in the definition of an architecture. Functional behavior is specified and modeled in isolation. The architecture, then, is defined as the way implementation resources are organized, how they co-operate and how they are scheduled (time-behavior). This definition applies to all levels of abstraction, though the precise meaning of organization, co-operation and time-behavior does depend on the level of abstraction we want to deal with. The higher the level of abstraction, the more we have to rely on models that are abstract themselves and can not, therefore, contain details that are only known at the lower levels of abstraction. Thus, when components are abstractly modeled, organization and co-ordination can only be expressed at the same level of abstraction and modeling and timing behavior can only be as accurate as the models can predict.

Clearly, abstraction reduces complexity and cost of modeling and simulating. On the other hand, it leads to the problem of coping with accuracy and level of confidence. It is, therefore, extremely important to have means to predict accuracy in such a way that it can be analyzed in terms of bounds. This in turn certainly depends on the models that are used. Hence, the ultimate challenge is to find an architecture model that is expressive and flexible but still allows for reasoning about predictability and accuracy, given a number of well chosen metrics. The following two chapters offer our answer to this challenge by way of an insight into how our architecture model is assembled, how it can be used and what the expected level of predictability is(see Sections 4.5.1 and 4.5.3).

3.2.2 Targeted Architectures

Multiprocessor architectures in embedded systems are usually not general purpose architectures but are tailored to match - to a certain extent - a particular application domain. Because the application domain we are dealing with in this dissertation is the domain of multimedia, the multiprocessor architectures that fit this domain are architectures that can sustain heavy computations as well as high data throughputs without incurring obstruction from control events that may interfere with the main data stream processing. Such architectures are sometimes called *streaming* multiprocessor architectures. It is important to notice that we are not considering application-specific architectures but architectures that fit an application domain. The implication of this condition is that it should be possible to target a single, possibly configurable architecture when transforming several applications from the application domain to parallel implementations. Thus we consider architectures that are built on components that support computationally intensive tasks and which can exchange large amounts of data among themselves. The communication of data between producing and consuming components occurs - at least in principle - over channels that support FIFO-like buffering with blocking write and blocking read synchronization protocol. A representative example of the architectures we are addressing here consists of a set of computational units and a communication, synchronization and storage infrastructure. Notice that we will not be dealing with design issues. Considering architecture alternatives is confined to selecting components from a component repository and connecting them in different yet admissible and dependable ways. This also includes alternatives in terms of the system's software protocols.

3.2.3 Model Structure

The architecture model structure is a composition of component models, some of which are for computation, some for communication and synchronization and some for storage. Remember that we want to model embedded multiprocessor (streaming) architectures as defined in Section 3.2.1. A component model is defined as a set of constituent modules (services), an input stream representation (tokens and their types) and an operational representation (symbolic programs). The composition of component models is based on well-defined admissible composition rules and interfaces. This is the structural *what* issue. The answer to the question of *why* we structure the model of the architecture this way is that we want to separate concerns as much as possible in such a way that performance analysis and design space exploration is not only feasible, but also effective and efficient in terms of modeling effort, simulation effort and accuracy. The representation of the basic ingredients is necessary and sufficient to predict the architecture's timing behavior as confidently as a simulation based prediction method can do. Later in the chapter, we shall elaborate on *how* we materialize the representations to achieve that goal.

3.2.4 Model Behavior

An architecture model does have a behavior that results from the dynamic behavior of the component's constituent modules and the interaction of the components through their interfaces. From our definition of what an architecture model is, it should be clear that it's *behavior* is to be interpreted in terms of behavior in *time* rather than *functionality*. Functional behavior is represented by symbolic instructions or symbolic programs extracted from the functional model and which must be interpreted by the architecture model. The functional model exposes its own concurrency, event ordering and synchronization primitives and protocols. The architecture model exposes its parallelism, resource constraints and operational delays. It interprets the former into the latter, possibly after transformations of the functional representations. This is what we can say here about the *what* of the behavior. Again, *why* we deal with behavior in the way we do is mainly because we want to separate concerns: the architecture's behavior model predicts whether an application can execute timeously, given current workloads and workloads added by the envisioned application; the application model predicts functional behavior and exposes this behavior to the architecture in terms of representations. The question of *how* this is accomplished is addressed in the remainder of this chapter.

3.2.5 Contribution

Recall that our streaming architectures are comprised of computational units, a communication, synchronization and storage infrastructure, and system software. The computational and infrastructure components are selected from a repository of re-usable components. Although the repository may contain a vast amount of components, we only rely on a relatively small set of modules and admissible interconnection rules (interfaces) to compose the structure of a so-called *Transaction Level Model* [2] of an architecture and its components. Of course, components have behavior and so has the architecture. We separate functional behavior from time behavior, the latter being dealt with in the architecture model. Functional behavior is represented in terms of symbolic programs that may have to be transformed before they can be interpreted, in terms of time and execution delays, by the component modules in the architecture model which is aware of aspects such as resource sharing, contention and control.

A large portion of our modeling paradigm is compatible with the Model of Computations paradigm [3] and is compliant with the Transaction Level Modeling semantics of *SystemC* [62].

[2] See Chapter 1, Section 1.3.2.
[3] See Chapter 1, Section 1.3.1.

We claim that our modeling paradigm can accurately predict the timing behavior of many conceivable streaming architectures (see Sections 4.5.1 and 4.5.3).

3.2.6 Chapter Organization

This chapter is organized as follows. First, in Section 3.3 we deal with *the how* question related to the structure of our architecture model: how is it materialized? Then, in Section 3.4 we turn to the how question related to the behavior of our architecture model: how is it modeled? Next, in Section 3.5 we demonstrate how we model some representative architecture types. Finally, in Section 3.6 we list briefly, by means of examples, some related architecture modeling approaches.

3.3 Architecture Model Structure

In Section 3.2.3 we dealt with *the what* and *the why* questions related to the structure of our architecture model: what is the problem and why is it so? Here we deal with the approach to this problem, so we introduce the model itself in these terms.

The structure of our architecture model consists of components and relations among these components. We define components based on their main purpose: *processors* - used to perform computations, *router interfaces* - used to establish the communication routes, *arbiters* - used to synchronize shared interconnections and *storage* - used to function as as a global shared memory. The relations comply with the *Transaction Level Modeling* [62] way of interconnecting components i.e., components interact through well defined interfaces, where each component has an appropriate number of interface ports. The interconnection of components is based on well-defined admissible composition rules and interfaces. This is illustrated in Figure 3.1. Processor components, storage components and arbiters can only be connected to router interface components, as indicated with the encircled numbers 1, 2 and 3 respectively. Note that the router interface components are essentially dealing with the *communication workload* of the mapped application model and that, thus, they have two modeling roles: *data transfer* to and from processor components to other processing components or storage components and *synchronization* of these transfers. This, however, represents a model behavior and it will be covered in Section 3.4 with the rest of the behaviors.

We introduce the different components in the sections that follow.

3.3.1 Processor Components

Processor components in Figure 3.1 are the architecture components onto which application processes are mapped. Actually, our methodology supports mapping of symbolic program representations of application processes onto processor components rather than mapping application processes themselves. Therefore, in the modeling, analysis and exploration phases, processor components interpret symbolic programs in terms of execution times. As representations of application processes are rich in behavioral information (see Chapter 2), the

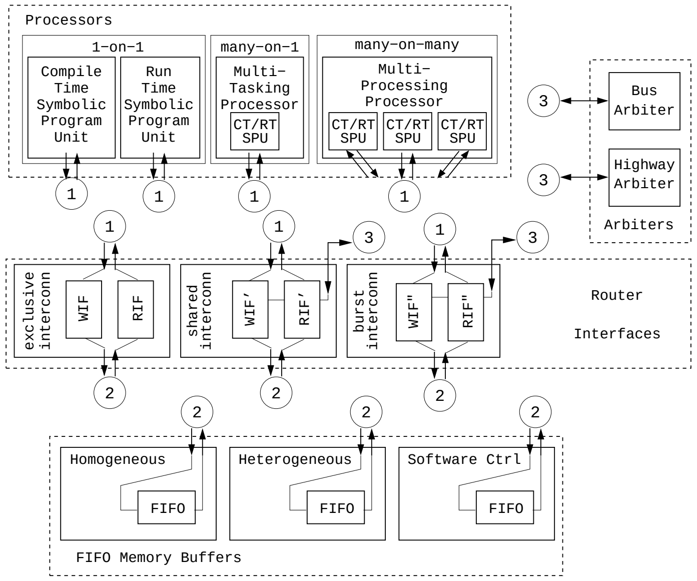


Figure 3.1: The architecture model structure - components and relations. Encircled numbers indicate which components should be connected to each other.

processor components are themselves composite modules, rather than being characterized by operation counts and cumulative delays. Thus the processor components model timing behavior of distinct entities such as symbolic instruction fetching, dispatching, execution and resource sharing.

Figure 3.1 reveals that we distinguish between three types of processor components. These three types are established depending on (1) the maximum number of mapped application processes which a processor component can handle at once and (2) how many parallel execution flows a processor component can handle. The types in Figure 3.1 are enumerated in terms of the types of mapping relation of $|processes|$-on-$|processors|$ such as: the *one-on-one*, the *many-on-one* and the *many-on-many*. Please note that ”many-on-many” does not include ”one-on-one” as is the case with the conventional ’many-to-many’ and ’one-to-one’ relations from Entity-Relation (ER) model [63]. In our architecture model ”many-on-many” and ”one-on-one” represent disjoint (non-overlapping) sets of processor models.

The one-on-one processor type can only model the execution of a single process mapped onto it. The many-on-one processor type can model the interleaved execution of a number of processes that are mapped onto it. Finally, the many-on-many processor type can model the overlapped execution of a number of processes that have been mapped onto it. Many-on-one corresponds to *multitasking*, while many-to-many corresponds to *multiprocessing*. In both multitasking and multiprocessing cases, the light-weight *operating systems* are modeled [4] as well and integrated in the processing units. Please note that the *one-on-many* mapping relation is not explicitly shown here. However, it is worth noting that one-on-many is available

[4]See Section 3.5.3.

as a sub-type of the processor type with the one-on-one relation (see VLIW later on).

We further distinguish between two one-to-one processor subtypes: the *Compile Time Symbolic Program Unit* and the *Run Time Symbolic Program Unit* (see Figure 3.1). For the compile time symbolic program unit, concurrency must be exposed in symbolic program representations just before their mapping onto the processor (e.g. Very Long Instruction Word or VLIW - see [64]). For the run time symbolic program unit, concurrency is extracted by the processor itself (e.g. super-scalar - see [65]).

The four processor types are built from lower-level modules which are specialized Finite State Machines that model particular behaviors. Because not all of the behaviors are present in every one of the considered processor types, it is a specific partition, combination and relation of these behavioral modules that defines the model of a particular processor type. There is a *common core* in all these processor types around which the different flavors are built. We call this core the *Symbolic Program Unit* (SPU). Amongst the many-on-one, many-on-many, and run-time processing units, the SPU core [5] is the simplest, and thus serves as the basis from which all the other processing units are derived.

3.3.2 Communication Router Interfaces

Communication interfaces create a context [6] in which the Inter Process Communication (IPC) is refined and then executed. The IPC refinement [7] refers to the conversion of the IPC of an application representation to the IPC of an architecture representation. Processes in the application model communicate by means of `read` and `write` primitives in which data transfer and synchronization are intertwined. Components in the architecture model, on the other hand, have separated data transfer and synchronization protocols. When communicating data from the memory space of one process to the memory space of another process, an IPC connection must be established first. The part of the processor component that executes an IPC `read` symbolic instruction connects to the part of the Read Interface (RIF) component that controls the data retrieval from the storage component addressed by that symbolic instruction. A few different RIF components are shown Figure 3.1. The RIF differences are explained later in Section 3.3.5. The same holds true for the part of the processor component that executes an IPC `write` symbolic instruction: it connects to the part of the corresponding Write Interface (WIF) component that controls data deployment to the storage component addressed by the write symbolic instruction. A few different RIF components are shown in Figure 3.1. The WIF differences are explained later in Section 3.3.5. The controlling of the RIF and WIF components effectively separates *data transfer* from *synchronization*.

Thus two operations can overlap whenever the connection is established: transferring data to and from the processor to the associated interface and transferring data from the WIF to the corresponding RIF through the storage component. The synchronization that goes with data transfer is dependent on the synchronization protocol(s) that the architecture supports. On an abstract level, the synchronization of the transfer of data to and from a processor

[5] See Section 3.4.2.

[6] A *context* is a special environment necessary to properly translate-&-interpret the Inter Process Communication of a mapped symbolic program.

[7] This is a run-time refinement - it is performed by the model while it runs.

component to its interface component is modeled in terms of a master-slave rendezvous protocol, where the processor is the master and the interface is the slave. The synchronization between interface components and a storage component is modeled in terms of condition-synchronization protocols: *check-room/signal-data* for the WIF and *check-data/signal-room* for the RIF. In the event that data transfer to/from a storage component goes over a shared bus, a shared bus synchronization protocol, bus-claim/bus-release, is injected between conditional synchronization pairs. We discuss the synchronization and data transfer protocols further in Section 3.3.5 where we deal with component-level interfacing.

As can be seen from Figure 3.1 we distinguish between three types of processor interconnects: *Exclusive*,*Shared* and *Burst*. The first is *cross-bar* like, the second one similar to being *bus-based* and the third one is likened to a *burst-bus*.

It is worth noting that, as is the case with processor component types, the various communication interface types are built on modules, some of which may or may not play a role depending on the type considered. We will come to the timing and dynamic behavior of the constituent modules in Section 3.4.

3.3.3 Arbiters

As already mentioned in the previous subsection, there exists a situation where a shared bus has to be modeled. In this case, the architecture component library provides the arbiter component. The arbiter component models a shared bus synchronization protocol - bus-claim and bus-release. The model uses the integer-valued P-V semaphore [66]: 1) bus-claim wraps the P call, 2) bus-release wraps the V call.

In the case of a single bus (or *Bus Arbiter* in Figure 3.1), when the arbiter component grants the bus to an interface component (the bus owner), every other interface component that claims the bus afterwards will be blocked until the bus owner releases the bus. When the interface component is bus owner it can transfer data through it.

In the case of multiple bus-lines (or *Highway Arbiter* in Figure 3.1), when the arbiter component grants the bus to an interface component, every other interface component that claims the bus thereafter will be granted access too, until the specified number of simultaneous data-transfers is reached. When the number of simultaneous data-transfers becomes equal to the number of bus owners, new bus claims are blocked and wait for release of the bus by current bus owners.

Multiple bus-arbiters can co-exist safely in the same architecture model instance [8]. That is, an interface component can use services from different bus arbiters. However, a single bus arbiter must be used for data-transfers to a particular storage component. It is not permitted to transfer data to/from the particular storage component via different buses. A FIFO buffer is connected to same bus on both of its ends.

[8]In terms of Object-Oriented Design, our architecture model should be understood as a class-template and a particular architecture description based on our model should be understood as an object-instance.

3.3.4 Storages

The storage components are organized as FIFO memory buffers. Each buffer has a writer port and a reader port. The interface components (WIF and RIF in Figure 3.1) communicate with each other using the protocol associated with these ports. This protocol is described in the next section along with the other protocols.

We differentiate FIFO storages in two ways: 1) based on the size of tokens that they store in transit and 2) based on the synchronization interface. The size of tokens reaching and leaving the buffer may be either homogeneous or heterogeneous. Homogeneous FIFOs consume equally large tokens on both (read and write) ends, while heterogeneous ones do not. With respect to the synchronization interface, an embedded processing core may or may not be required. That is, the conditional synchronization protocol (wait-for-data/room, signal-room/data) is executed either on a programmable core or in a specialized hardware.

In our architecture model, all FIFOs communicate using the same protocol, but they differ primarily in the way data packets are treated. When data is written to a FIFO, it may or may not have to be rearranged to fit a predefined token size. Similarly, when data is read from a FIFO, it may require a similar treatment. The ability to store, load and synchronize exchange of tokens of variable size requires an extra modeling effort for the implementation of the *check-room/signal-data* and *check-data/signal-room* protocol actions. Alternatively, the architecture model may communicate tokens that have the same size as the tokens communicated in the application model. That would not require any overhead in the FIFO components, but may lead to unrealistic memory requirements (FIFO size) and performance numbers (excessive FIFO blocking delays because of lack of room or of data). Finally, one could build an application model which takes into account all architecture FIFO communication artefacts. This is, indeed, also possible, but it is totally inadequate for the architecture modeling and exploration purposes. Remember that our decision to follow the Y-chart approach separates application and architecture models. This means that the application model is independent of the architecture model and vice versa.

The above discussion raises several questions, one of which is whether the *check-room/signal-data* and *check-data/signal-room* protocol actions are implemented in hardware or in software. This is especially interesting in the case of *heterogeneous* FIFOs, where the performance of these actions may significantly impact the performance of the rest of the system. Designs where the FIFO conditional synchronization protocol is executed on that programmable core (e.g., the inter-processor communication in the `spaceCake` architecture [67] architecture are known in practice. Therefore, with respect to the materialization of the FIFO conditional synchronization, the FIFO may require the software core or the specialized hardware for the conditional synchronization protocol.

More details are provided in the following section.

3.3.5 Interfacing Architecture Components

As already mentioned, the model components described so far communicate through well defined component interfaces [9]. There are three interface types, as depicted in Figure 3.1. Each of them goes with a particular protocol:

1. The *push-pull* protocols - the handshake protocols between processing units [10] as masters and interface units [11] as slaves.
2. The *room-data* protocol - the condition synchronization protocols between interface units when communicating data through FIFOs [12].
3. The *claim-release* protocol - the resource-sharing protocol between interface units when transferring data over the shared-resource [13].

One possible sequence involving all these protocols is illustrated in Figure 3.2. interface points among different architecture components are indicated by circled numbers; they correspond to the enumerations in Figure 3.1. To explain the protocol sequence illustrated in Figure 3.2 we use Dewey decimal numbers [68].

On the left hand side of the figure is illustrated a pull master-slave protocol. The pull protocol *pulls* data into a processor component from the corresponding read-interface component. On the right hand side of the figure, a push protocol illustrated. The push protocol *pushes* data from a processor component to the corresponding write-interface component. Both of the protocols consist of three parts. Firstly, the connections have to be opened: Dewey sequences $1.0 \mapsto 2.0 \mapsto 2.1 \mapsto 1.1$ for the pull-side and $6.0 \mapsto 5.0 \mapsto 5.1 \mapsto 6.1$ for the push-side. Secondly, the real push or pull handshake protocols: sequences $1.2 \mapsto 2.2 \mapsto 2.8 \mapsto 1.3$ and $6.3 \mapsto 5.3 \mapsto 5.9 \mapsto 6.4$, respectively. Thirdly, the connections have to be closed explicitly: sequences $1.5 \mapsto 2.10$ for the pull-side and $6.5 \mapsto 5.10$ for the push-side. Note that when looking into the write connection, data is moved from the processor component to a local memory of the write interface component prior to any handshake activity: the sequence $6.2 \mapsto 5.2$. On the contrary, when moving data from the read interface component to the processor component, the data is moved from a local memory of the read interface component to the processor component after all handshake activities are completed: The sequence $2.9 \mapsto 1.4$.

In the center of Figure 3.2, the condition synchronization protocols are illustrated. These are protocols that are convenient for the modeling of synchronization. The reader side (visible through the previously established pull connection) checks on the availability of data in a global memory buffer [14]. If the data is not in the memory buffer the reader side blocks until

[9]The component interfaces are the interfaces from the component-based design point of view, unlike the router interface components we introduced in Section 3.3.2. Consequently, whenever we speak in terms of inter-component interfacing we refer to *component interfaces*, whereas when we speak in terms of the model description and assemblies we refer to *interface components* or *routers*.

[10]See Section 3.3.1.

[11]See Section 3.3.2.

[12]See Section 3.3.4.

[13]See Section 3.3.3.

[14]See Section 3.3.4.

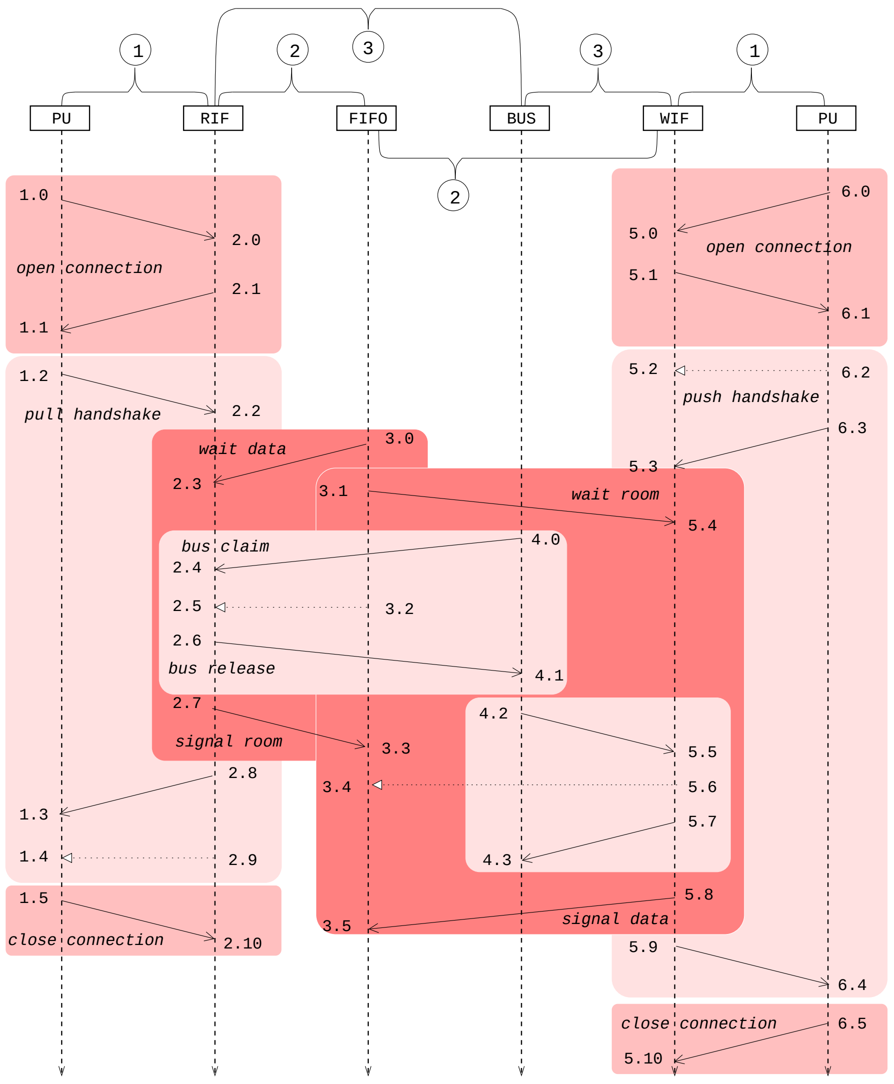


Figure 3.2: The architecture model structure - sequence diagram describing the component interfaces.

it is notified that the data has arrived: Dewey sequence $3.0 \mapsto 2.3$. Similarly, the writer side (push connection) checks on the availability of room in the global memory buffer. If there is no available space in the memory buffer the writer side blocks until it is notified that sufficient space is available: The sequence $3.1 \mapsto 5.4$. In the particular case illustrated in Figure 3.2, neither of the two checks (data-to-read or room-to-write) give rise to blocking, which implies that the memory buffer (FIFO) has both enough data and room. When data is read from the memory buffer or when data is written to it, the signaling part of the synchronization protocol takes place: Dewey sequences $2.2 \mapsto 3.3$ and $5.8 \mapsto 3.5$ respectively. The former serves to update (atomically) the number of available data tokens. The latter serves to update (also atomically) the available space in the buffer. Note that in the case of Figure 3.2 data is transferred through the bus. Therefore, the load and the store activities (data transfers) are synchronized by the conditional synchronization protocol but they are only possible when the bus protocol is fulfilled.

Finally, there is a small part in Figure 3.2 where the resource sharing protocol is illustrated. This is a single bus protocol that allows only a single bus owner. For example, after the read interface has discovered that data tokens are available in the memory buffer, it claims the bus. When it has been given control over the bus, it loads the data from the global memory into the local memory of the read interface. After that, the read interface releases the bus, allowing a next bus request to take over the bus. The respective activities are depicted by the following Dewey sequence $4.0 \mapsto 2.4; 3.2 \mapsto 2.5; 2.6 \mapsto 4.1$ (the semi-colon ';' sign forces the load data transfer between the bus claim and the bus release). On the contrary, the write interface side is blocked on the bus request - therefore it must first wait for the read interface to finish its data transfers. After that, it gains control, stores new data into the global memory buffer and releases the bus afterwards. This is illustrated by the following Dewey sequence: $4.2 \mapsto 5.5; 5.6 \mapsto 3.4; 5.7 \mapsto 4.3$.

3.4 Architecture Model Behavior

The behavior of our architecture model is with respect to time aspects only. The performance analysis is then based on that behavior.

Time in our model is configurable via a set of parameters. These parameters determine the way conditions necessary for some actions in the model are addressed. This orders these actions in the model and affects the possible states and state transitions in the model. To deal with the conditions, ordering and state in our architecture model, we define the following elements: Threads of Execution, Asynchronous Inter-thread Communication Channels, Synchronous Inter-thread Communication Channels and Concurrent Finite State Machines. Lastly, we describe the time-performance measurement model.

3.4.1 Architecture Model Element Behaviors

Finite State Machine. *A Finite State Machine (FSM) is a sequential MoC defined by an ordered relation among the actions described by the 5-tuple* $\langle S, T, X, Y, A\rangle$ *where:* S *is a* *set of FSM states, T is a set of FSM transition condition predicates, X is a set of state entry actions, Y is a set of state exit actions and A is a set of transition actions* [28]. ⋄

Thread of Execution. *A Thread of Execution is an ordered sequence of actions which refer to a single program and data memory context. Each thread defines its own ordered sequence which can be executed concurrently with sequences of other threads*. [69] ⋄

Synchronous Inter-thread Communication Channel. *A Synchronous Inter-thread Communication Channel is the information exchange where one execution thread sends a certain message to other blocked threads of execution and, at the same time, unblocks them. In computer engineering this is known as a conditional synchronization*. [70] ⋄

Asynchronous Inter-thread Communication Channel. *An Asynchronous Inter-thread Communication Channel is the information exchange where one execution thread, the caller, activates a sequence of actions in the context of the other thread of execution, the callee, and where - whilst this sequence is not explicitly part of the callee thread native sequence - it can still be used to: (1) copy-back to caller all the needed insight information of the callee and (2) alter the execution flow of the callee thread*. [70] ⋄

Concurrent Finite State Machine. *Concurrent Finite State Machines (CFSM [15]) are threads of execution represented with FSM behavior, one FSM per thread and where interactions are based on the Inter-thread Communication paradigm, both Synchronous and Asynchronous.* ⋄

3.4.2 Processor Modeling

Generally speaking, an application process is a High-Level Language (HLL) specification of one or more threads compiled and linked for a particular processor and a particular platform. However, the application process in a binary form would not serve the architecture exploration purpose because it is specific to a single processor-platform pair. Consequently, changes in the processor or platform specifications, or in the application source imply new tools (compilers) or at least recompilation. Sometimes this is acceptable because the compilation process may be fast and automated [16], but sometimes this is not the case [17]. Thus, processor components in our architecture model do not execute the application processes that are eventually mapped onto them. Instead, our processor components interpret process representations in terms of timing behavior. The representations we use are Symbolic Programs [18] (SP) and transformations that are applied to them are part of the application-to-architecture mapping process. See Chapter 4.

A symbolic program captures the details the functional behavior of an application process without implying resource-limitations. The processor model needs to *interpret* the symbolic program(s), to *extract* the partial order within the stream(s) of symbolic instructions, to *schedule* the symbolic instructions according to the resource availability and to *delay* according to

[15]Note that Polis Co-design Finite State Machines [29] and our Concurrent Finite State Machines are identical.

[16]A single `make` command that can rebuild the whole image.

[17]For example, consider the case of a heterogeneous architecture with both dedicated, semi-programmable and fully-programmable processors. The changes in the dedicated or customized components would impact the recompilation duration.

[18]See Chapter 2.

the annotations in these instructions. Therefore, the main features we model within the processor model are: symbolic instruction extraction, fetching, dispatching (distribution) and execution and resource sharing, all these in terms of time dynamic behavior. Based on the above features, we determine the low-end threads of execution and the interfaces of such threads in the processor model. In the next subsection we describe the timing behavior and the dynamic behavior of the processor model in terms of threads and channel-events (see Section 3.4.1).

Symbolic Instruction Extraction & Fetching

When a symbolic program (SP) is assigned to a processor model instance, a part of the processor model extracts symbolic instructions from the SP. We call it the program unit (PU). This is a thread whose behavior is described by the SP syntax (see Section 2.4.4) rather than by a CFSM rule. The PU thread has two parts: the parser part and the traversal part.

The parser part creates a *parse tree* out of the symbolic program. The parse tree models a conventional program memory. The creation of the parse tree corresponds to the process loading. The other part of the program unit traverses the parse tree based on the SP control trace (see Chapter 2). This corresponds to the symbolic instruction extraction. Remember, a symbolic program is a general representation, valid for many data-sets. It is the control trace that is bound to a particular data-set. This will help to produce a particular sequence of symbolic instructions. When the tree traversal hits a symbolic instruction, the PU thread writes its content into the FIFO channel to which the PU is connected. This corresponds to the symbolic instruction fetching. The implementation details of the PU thread are given in Appendix A, Section A.3.

Symbolic Instruction Scheduling

The fetched symbolic instructions arrive queued according to the order expressed in an SP. We call a front-end controller (FECTRL) the part of the processor model that reacts to such an input stream of symbolic instructions (SI). The controller delivers SIs with a certain degree of concurrency. That degree should match the degree of concurrency available in the processor model instance. For example, the processor instance can or cannot perform two differently designated `read` symbolic instructions in parallel. Similar criteria hold for the other types of SIs. This corresponds to the symbolic instruction scheduling. The scheduled symbolic instructions are written to the FIFO channel that conveys them to a following controller in the processor model instance (Symbolic Instruction Dispatching).

The re-ordering of input instructions into output instructions can be modeled by means of a user-configurable delay parameter [19]. The implementation details of the FECTRL thread are given in Appendix A, Figure A.6.

[19]The value of this parameter can also be zero. If the parameter value is zero, then re-ordering is considered to be instantaneous.

Symbolic Instruction Dispatching

Once fetched and scheduled, symbolic instructions have to be dispatched according to their type and sub-type [20]. The part of the processor model that performs this job is the Back-End Controller (BECTRL) thread. The BECTRL thread reads the scheduled symbolic instructions and dispatches them to their ultimate destination threads (Read Units, Execute Units, or Write Units). It applies a block to the output until the execution of the dispatched symbolic instructions is completed. For example, if three different symbolic instructions have been submitted to the execution simultaneously, the BECTRL thread dispatches them and applies a block until all three are completed.

Dispatching the symbolic instructions to the destination threads can be modeled by means of a user-configurable delay parameter. The implementation details of the BECTRL thread are given in Appendix A, Figure A.7.

However, it is possible to parallelize symbolic instruction execution even further. For example, it may be possible to commit the next instruction to a Read Unit without having to wait for the completion of an Execute Unit. For this purpose, we make a few modifications in the BECTRL thread. The idea is to queue the results of the executions in a synchronous data-flow manner, such that these results can be used later, when needed. If the results are not available, then the execution of an instruction which needs them stalls until the results become available. The implementation details of the modified BECTRL thread are given in Appendix A, Figure A.7.

Symbolic Instruction Execution

As a dispatched symbolic instruction may be either a `read`, a `write`, or an `execute`, the thread that executes it may be similarly either a Read Unit, a Write Unit or an Execute Unit.

The Read Unit thread (RU) reacts on a read symbolic instruction and processes it through several stages: (i) the RU makes a request to communicate with the FIFO global memory component [21] which is logically identified in the instruction; (ii) it performs the pull protocol [22] to read data, (iii) it creates releasing events on both the dispatcher and the RIF component ports. The RU thread is suspended in three cases: 1) on the input, if there is no read instruction; 2) on the output, when opening the connection to the FIFO component and 3) again on the output, when it performs the pull data protocol with the RIF component. The data retrieval from the RIF into the processor can be modeled by a user-configurable delay parameter. The implementation details of the RU thread are given in Appendix A, Figure A.9.

The Write Unit thread (WU) reacts on a write symbolic instruction and processes it through several stages: (i) the WU makes a request to communicate with the FIFO component which

[20] `read 12 (...)` means that data from the logical FIFO 12 are to be read - `read` is a type, `12` is a subtype. Similarly, `execute 12 (...)` means that function 12 is to be executed. However, rather than being function arguments the sub-types refer to distinguishable processing resources. That is, `read` and `write` sub-types bind application Kahn buffers to architecture FIFO components and `execute` sub-types correlate to different functionality with the different computation elements within a processor components.

[21] See Section 3.3.5

[22] See Section 3.3.5

is logically identified in the instruction; (ii) it performs the push protocol to write data; (iii) it creates releasing events on both input and output channels. The WU thread is suspended in three cases: 1) on the input, if there is no write symbolic instruction; 2) on the output, when opening the connection to the FIFO component and 3) again on the output, when it performs the pull data protocol with the WIF component. The data submission from the processor to the WIF can be modeled by a user-configurable delay parameter. The implementation details of the WU thread are given in Appendix A, Figure A.10.

The Execute Unit thread (EU) reacts on an execute symbolic instruction and processes it through several stages: (i) it waits for a certain amount of time [23]; (ii) it creates releasing events on the input channel. The EU thread is suspended on the input if there is no execute instruction. The implementation details of the EU thread are given in Appendix A, Figure A.11.

Interrupt Modeling

Although the KPN application model does not know about interrupts, the processor model does know. This is required for the cases of many-on-one and many-on-many mapping relations (see Section 3.3.1). The processor interrupt logic is modeled using the Programmable Interrupt Controllers threads (PIC). There are two kinds of them available: the Read Interrupt Controller thread (RIC) and the Write Interrupt Controller thread (WIC). Both are associated with the same CFSM, hence we refer to both of them as PICs. The modules implement the 'master' part of the read/write interrupt methods[24]. These methods are employed to submit the network data-status or the network room-status to the Distributed Operating System (DOS) channel [25]. Specifically, the PIC applies a block until a change is detected in the data-status or in the room-status of the global FIFO component. The events are generated directly by the router interface components (WIF and RIF). The interrupt generation can be modeled by a user-configurable delay parameter. The implementation details of the PIC threads are given in Appendix A, Figure A.12. The PIC thread usage is exemplified in Section 3.5.3, Figure 3.7, as WIC and RIC.

Processor Synchronous-Event Channels

The processor synchronous event channels model interconnection data paths between processor modules which model SI fetching, scheduling, dispatching, execution and interrupts according to definitions for synchronous channels established in Section 3.4.1. There are four internal synchronous processor channels: (i) A Read-Write Blocking FIFO channel (RWB FIFO), (ii) a Peek Read Write Execute channel (peek-RWE), (iii) a Blocking Dispatcher channel (BD) and (iv) a Symbolic Instruction Operands Crossbar channel (SIOC).

The Read-Write Blocking FIFO channel provides two interface methods: (1) a method to

[23]Equal to the product of the configured amount of time and the amount indicated in the instruction - so-called *budget*.

[24]Remote procedure call *raise* which activates the asynchronous event handling in the DOS channel - see Appendix A

[25]See Section 3.4.2.

read data from the channel and (2) another method to write data to the channel. This channel is used to transfer symbolic instructions between the PU thread and the FECTRL thread, as well as between the FECTRL thread and the BECTRL thread. See Section 3.5.1.

The Peek Read Write Execute channel provides four interface methods: (1) a method to poll for available symbolic instructions, (2) a method to peek the type of a symbolic instruction from the channel, plus (3&4) the aforementioned RWB FIFO-channel read/write methods. This channel is used between the FECTRL thread and the modified BECTRL thread. See Section 3.5.2.

The Blocking Dispatcher channel provides four interface methods, two for the dispatcher (BECTRL) side and two for the execution unit (RU, WU, EU) side. The dispatcher uses one method to deliver the symbolic instructions and the other to wait for the finalization report. The execution unit uses one method to accept a new symbolic instruction and the other to report on finalization of that instruction. The BD channel is used between the BECTRL thread on the one side and the configured number of the RU, WU and EU threads on the other side.

The Symbolic Instruction Operands Crossbar channel provides two interface methods: (1) a method to get the oldest value of a symbolic instruction operand from a head of the operand FIFO and (2) another method to put the latest value of a symbolic instruction operand to the back of the operand FIFO. A thread that requests the operand value is a 'requester' (e.g., WU or EU). A thread that provides the operand value is a 'provider' (e.g., RU or EU). The synchronization in the SIOC channel is reminiscent of Integer Data Flow (IDF) MoC [24]. A requester will remove as many operand tokens from the operand FIFO(s) as expected in the write or execute symbolic instruction. A provider will provide as many operand tokens to the operand FIFO(s) as expected in the read or execute symbolic instruction.

Processor Asynchronous-Event Channels

The processor synchronous event channels model interconnection data paths between processor modules which model SI scheduling, dispatching, execution, and interrupts according to definitions for asynchronous channels established in Section 3.4.1. Essentially, the asynchronous channels are needed to model a processor with an operating system on-top. Our operating system model targets *homogeneous* multiprocessors, e.g., the cases of many-on-one and many-on-many mapping relations (see Section 3.3.1).

When modeling an operating system, one has to be aware of three different operating system types that may appear on a homogeneous multiprocessor [71]: (1) separate-supervisors, (2) master-slave and (3) symmetric. Separate-supervisor and master-slave are not truly parallel software platforms - the former has serious load-balancing issues, the latter does not scale well and has Amdahl's Law implications. Unlike these two, a symmetric software platform scales nicely and allows for modular (micro-kernel) implementations. Moreover, the symmetric software platform complies with the main requirements needed to support stream-oriented applications [31]. This is why our operating system model, Distributed Operating System (DOS) is implemented as a symmetric software platform.

The DOS channel virtualizes the RU, WU and EU threads for each symbolic program, isolating a symbolic program from all other symbolic programs with which it may share processor resources. The interconnection and the global memory are abstracted partly by the DOS channel, and partly by the channel contained in the router interface (WIF and RIF) components [26].

To achieve this, the DOS channel has to exclude explicit blocking conditions because they may produce biased symbolic program schedules [27]. Excluding explicit synchronization implies that the asynchronous event synchronization protocol has to be used in the underlying channel implementation [28]. Therefore, the DOS channel consists of a single thread, which performs the polling and a set of communicating signal handlers, which handle predefined asynchronous events. The asynchronous events contain feedback information about: (a) idle symbolic processing unit cores [29], (b) globally available room and data [30] and (c) non-empty symbolic instruction streams [31]. The signal handlers prevent artificial deadlocks by conveying this feedback to the scheduler.

The multiprogramming scheme creates additional overhead for the processor model. The symbolic programs are scheduled according to some *scheduling policy*. The switching among the symbolic program instruction streams creates a model of a *context switching* delay. Furthermore, if the processor is a homogeneous multiprocessor, then the scheduling may imply a *migration* delay, because a symbolic program may continue its execution on a different Read-Write-Execute thread than before. See Appendix A, Figure A.13 for the implementation details.

Processor Composite Modules

A composite module is a hierarchical module that conveniently groups the low level threads into processor model parts. This makes the processor model itself easier to understand. We define two composite modules: the Symbolic Program Stream module and the Symbolic Program Unit Core module.

The Symbolic Program Stream composite module (SPS) provides a hierarchical model of the context of a single process, on top of the processor model which supports multiprogramming. This module connects to the DOS channel explained earlier [32]. Since the DOS channel is based on asynchronous events, minor modifications are required in the threads which the SPS module uses. Namely, the FECTRL thread uses an asynchronous interface (I/F) at the output [33]. For the implementation details, see Appendix A, Figure A.14. The structure of the SPS module is depicted in Figure 3.3.

[26]See Section 3.4.3.

[27]E.g., in `Spade` the available 'native' SI trace schedules guarantee artificial deadlock-free executions, but are only a subset of valid schedules modeled. This implies that all other valid schedules are left unmodeled. The explanation and exemplifications of this issue are available in [72].

[28]See Section 3.4.1 for asynchronous events

[29]See Section 3.4.2, Figure 3.4 and Section 3.5.3, Figure 3.7.

[30]See Section 3.4.2 and Section 3.5.3, Figure 3.7.

[31]See Section 3.4.2, Figure 3.3 and Section 3.5.3, Figure 3.7.

[32]See Section 3.5.3, Figure 3.7.

[33]The asynchronous I/F at the output serves as the master routine for the DOS channel, or in other words, it keeps the DOS channel up-to-date with the available symbolic instructions in the symbolic program.

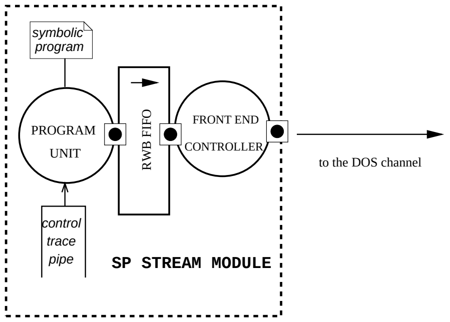


Figure 3.3: The composite SPS processor model module (Program Unit refers to the PU thread, Front-end Controller refers to the FECTRL thread, and the RWB FIFO refers to Read-Write Blocking FIFO).

The Symbolic Program Unit Core (SPU core) composite module represents a single processing core. This is quite useful when modeling homogeneous multiprocessors - performed by simply replicating this module. It connects to the DOS channel at the bottom end [34]. Since the DOS channel is based on asynchronous events, minor modifications are required in the threads which the SPU-core module uses. Namely, the BECTRL thread uses two asynchronous I/F's at the input [35]. For the implementation details, see Appendix A, Figure A.15. The structure of the SPU core module is depicted in Figure 3.4.

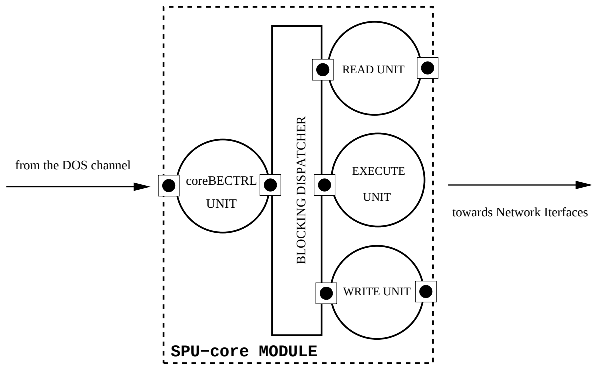


Figure 3.4: The composite SPU-core model module (*coreBECTRL* refers to a modified version of the BECTRL thread, necessary to match the DOS asynchronous channel).

[34]See Section 3.5.3, Figure 3.7.

[35]One asynchronous I/F at the input serves as the master routine for the DOS channel, or in other words, it keeps the DOS channel up-to-date with the status of the SPU core (busy/idle). The other asynchronous I/F at the input serves as the slave routine for the DOS channel, which reads the next available symbolic instructions for the SPU core.

3.4.3 Router Interface Modeling

A router interface model defines a platform-dependent context for the Inter-Process Communication (IPC). The IPC context we consider is a virtualization of bus and global memory components for one or more processes mapped onto a single processor.

Recall that the application model limits inter-process communication to FIFO-based IPC only. The `read` and `write` symbolic instructions directly abstract the FIFO read/write IPC. However, such symbolic `read`/`write` instructions are executed inside the processor model. The question is: Why do we need to model yet another application-process related context (router interfaces) when we already have the processors for that purpose? The answer is rather intuitive: the refinement of the IPC symbolic instructions also depends on other non-processor components - mainly those in the interconnecting network. When data needs to be moved between the network and a processor (or multiple processors), processors are only partially involved. Data needs to be loaded and stored into the global FIFO memory. Also, on the way to/from the global memory, arbitration for data-transfer lines takes place. These issues were described when interface protocols were discussed (See Section 3.3.5). Hence, the status of these network components and their dynamics act in a way which is complementary to the processor components in the case of IPC. Network dynamics and global memory capacity influence the schedules of the SP executions: `read` SI's stall processor components until sufficient data is available in the global memory and `write` SI's will stall processor components until sufficient space is available in the global memory. These stallings are the result of the finite network throughput and the finite global memory capacity. Thus, these constraints affect the way processor components push and pull data via the network and memory. By considering both processor and network components, the IPC modeling picture is complete.

In this subsection we provide an insight into the behavioral elements implementing the above IPC refinements in router components. We mention them following the order in which the IPC refinement is carried out: connection, synchronization and transfer. In addition, we show how network-generated interrupts are modeled.

Opening and Closing FIFO Connections

The opening and closing of read and write IPC connections is implemented inside the FIFO-Input Controller (FICTRL) and the FIFO-Output Controller (FOCTRL) threads. The two are essentially the same: the only differences consist of (i) the unit to which they are connected (the RU or the WU processor thread) and (ii) their purposes (to establish either a read or a write connection). Thus, we exploit their similarities to provide a general explanation of the dynamic features of these threads.

Initially, a thread waits for a request to come from the processor unit. In the case of the FICTRL thread, the request comes from the RU processor thread; in the case of the FOCTRL thread, the request comes from the WU processor thread (see Section 3.4.2). As soon as the request arrives, the thread configures the appropriate connection based on the information available within the interface component context [36]. The connection configuration

[36]Interface components are also known as *routers*: WIF and RIF. See the explanation of the connection opening

is performed according to the received ‘logical’ (application) FIFO identification number. Eventually, the requester is given the acknowledgement that the connection is ready-to-use.

The opening of the FIFO connection can be modeled by means of a user-configurable delay parameter. A FIFO connection is considered to close instantaneously. The closing is done after read/write IPC have been completed (see Figure 3.2). The implementation details of the FICTRL and FOCTRL threads are given in Appendix A, Figures A.16 and A.17.

Synchronizing FIFO Connections

The synchronization over FIFO connections is implemented inside the FIFO-Input Unit (FIU) and the FIFO-Output Unit (FOU) threads. The two are essentially the same: the only differences are (i) the unit they are connected to (either to the reader or to the writer side of the FIFO component) and (ii) their purpose (to load the data from or to store the data in the FIFO component).

The FIU thread is connected to (1) the read FIFO component side and to (2) the pull event-synchronization channel. The FOU thread is connected to (1) to the write FIFO component side and to (2) the push event-synchronization channel. In addition to these, the threads can be connected to the bus arbiter component. That is, each time the FIFO is accessed by either of these threads, bus arbitration takes place.

The FIFO connection synchronization can be modeled by means of a user-configurable delay parameter. The implementation details of the FIU and FOU threads are given in Appendix A, Figures A.18, A.19 and A.20.

Network Interrupt Modeling

We considered interrupt modeling in Section 3.4.2. The Read and Write interrupts are generated by some bookkeeping activities in the router components: the data and room status information are acquired using the FIU and FOU threads. These status acquisitions (1) must be done in a non-blocking way (i.e., status check must not interfere with the main IPC tasks of the router component threads) and (2) should appear only when the relevant room and data changes in the FIFO appear (i.e., ideally they should be performed only once, when the status has already meaningfully changed - sufficient room or data spotted.).

These two ”rules” appear to be imprecise because we may not know explicitly when to generate an interrupt. Furthermore, the processor DOS channel is fundamentally dependent on the presence of the status information (see Section 3.4.2). Thorough examination of the interaction between processor components and the router components provides the following guidance. Both writing-to and reading-from the FIFOs are operations generated at the processor side; recall that, ‘push’ and ‘pull’ channels are blocked until processor WU and RU threads generate requests. It is worth noting that when storing data to a FIFO component, the processor component first writes to the router component and then it may possibly apply a block. Conversely, when loading data from a FIFO component, the processor component

protocol shown in Figure 3.2.

may first apply a block if there is no data and then continue with the loading sequence. This is visualized in Figure 3.2, in the push and pull protocol sequence parts. Consequently, (1) room status can be obtained from the FOU thread of the write router component by acquiring room status of a FIFO at the beginning of the push sequence and (2) data status requires an independent polling mechanism which can be activated whenever there is a change in the FIFO status from ‘empty’ to ‘data-has-just-arrived’.

Router-Interface Synchronous-Event Channels

The synchronous-event channels of the router-interface component model interconnection data paths between internal router modules. This models the opening/closing/synchronising connections and network-interrupts according to definitions for synchronous channels established in Section 3.4.1 .They are thus named *synchronous-event channels*. There are two internal synchronous event channels available: (1) A Pull Channel (PULL) and (2) a Push Channel (PUSH). They model the pull and push protocols, respectively. The PUSH and PULL channels are important parts of the router-interface components.

The channels have the same structures but complementary roles. The PULL channel provides the means for the processor RU threads to pull data from the FIFO’s. The FIU thread connects to the other side of the PULL channel and provides the interface to the designated FIFO component. Similarly, the PUSH channel provides the means for the processor WU threads to push data to the FIFO’s. The FOU thread connects to the other side of the PUSH channel and provides the interface to the designated FIFO component.

3.4.4 Global FIFO Memory Modeling

To model storage in our architecture model we use the Global FIFO Memory component. The global FIFO memory consists of a set of FIFO memory components. A FIFO memory component is a buffer with two explicit access points: One for the writer and one for the reader. Data being written are always appended (unless there is no room). Data being read are always removed from the front (unless there is no data). Lack of room or data in the buffer sets the synchronization primitives `check-room` and `check-data` to a blocking state. Since they are part of the FOU and the FIU threads inside of router-interface components, any further queuing and removing activity for that buffer is blocked too. As soon as such condition disappears, the corresponding threads in the peer interface components [37] unblock any pending data operation for that buffer and trigger `signal-data` or `signal-room` primitives. As a result, FIFO components convey data according to a conditional-synchronization protocol (*check-room/signal-data* and *check-data/signal-room* - see Section 3.3.5).

A FIFO component is always connected to two configured router-interface components such that only the particular FIU thread inside of read router-interface can load data from the FIFO memory and that only the particular FOU thread inside of write router-interface can store data into the FIFO memory - the FIFO connection cannot migrate over different router-interfaces

[37] A peer of a FOU thread in a WIF interface component connected to the writer side of a FIFO is a unique FIU thread in a RIF interface component connected to the reader side of the same FIFO and the other way around.

nor over different threads within a single router-interface. The execution of the conditional-synchronization protocol primitives can be modeled by means of a user-configurable delay parameter. If these primitives require such user-configuration, it is implied that they are implemented in software (e.g., a dedicated processor, not visible to a user, executes these primitives). Otherwise, the primitives are implemented in hardware, using particular logic blocks.

3.4.5 Bus Arbitration Modeling

The Arbiter component models the bus contention. It consists of: 1) a semaphore [66] with its initial value set to a number of bus-lines and 2) a priority queue [73]. When a bus-line is requested and the semaphore value is not zero, the semaphore value is decremented. In this case, a bus-requester does not block and can proceed with a transfer over the bus-lines. Otherwise, if the current bus-semaphore value is zero, then the bus-requester applies a block until the bus-semaphore is incremented, the latter being an indication that bus-lines have been released. To make the bus schedule realistic, each bus contender queues its request based on the time-stamp[38] value. A requester releases the bus when it completes its transaction. This event increments the bus-semaphore value and wakes up all waiting (blocked) requesters. However, only the contender with the oldest request will get the next bus access. It is worth noting that the criteria by which the requests in the queue are managed, correspond to the bus schedule. Depending on the implementation of this feature, different bus schedules can be derived.

3.4.6 Measuring Performance

Our architecture model is abstract and non-functional. The reason is simple - during the exploration process we are interested in performance numbers, not in produced data. Therefore, the application functionality is modeled by a set of abstract architecture instructions, each of which corresponds to a particular computation delay. Typically, application processes exchange tokens (data types) which must be translated into architecture data types, e.g., bits, bytes, words, double words, etc. As a consequence, a single application SI may have to be translated in a sequence of architecture SIs. Hence, the first step is to define an abstract instruction set.

Abstract Instruction Set. *An abstract instruction set (AIS) is a pair $\langle F, S \rangle$ used to specify symbolic instructions, where F is a set of functions, the execution of which the architecture model needs to simulate and S is a set with specifications of processor and network word sizes.* $\diamond$

Apart from architecture SIs, the architecture model instance executes various communication related delays. They can either be fixed assigned delays or implicitly generated delays. The former are given as architecture structure component parameters. For example, an execute SI must be specified in terms of a fixed delay. The latter results from the model instance execution (simulation) and various time behaviors of the components. For example, a delay

[38]A time-stamp marks the 'time' when the bus-request has arrived at the arbiter component.

that appears as a consequence of blocking on either memory or data availability falls into this category.

We use fixed delays for scheduling, process migration, context switching, interrupt, communication connection switching, a single bus word transfer, a communication setup and both data/room checking and signaling. Some of these may be ignored or are not applicable for some platform instances (e.g., hardware accelerators and co-processors do not need scheduling, context switching or other multi-programming functionalities).

CFSM Performance Measurement

The measuring mechanism of performance numbers is closely related to a dynamic TLM model of the component CFSMs and the inter-thread communication channels. We collect and accumulate both the explicit (fixed) and the implicit delays by summing and subtracting the end-to-end time differences for each architecture functionality deemed to be a delay. This produces a 'running' time of an individual CFSM in a certain state. The CFSM running time is a sum of running times in all its states.

For example, looking at Figure A.9 from Appendix A, the execution of a single `read` symbolic instruction by an RU thread inside of a processor component results in the unrolled sequence of FSM states: $IDLE \mapsto SETUP \mapsto STALL \mapsto RUN \mapsto IDLE$, where "$\mapsto$" defines the total order between the states from left to right. The delay of the $SETUP$ state can be assigned by means of user-configuration, and the delay of the $STALL$ state is implicit due to the fact that it depends on the conditional synchronization with a separate component (a read router-interface component connected to the processor component). Finally, depending on the user configuration, the RUN state can contribute to both the explicit delay (so called "budget" of a `read` SI) and the implicit delay (storing data coming from the outside in the specific internal `read` operand FIFO). Each time a `read` SI arrives, all states are affected according to (1) the assigned delay parameters and (2) implicitly generated delays due to conditional synchronization. As a result of these delays the RU CFSM running time is altered, the other CFSMs interfacing the RU module are also altered and finally, the total system simulated time is altered.

Equations 3.1 and 3.2 more formally express the measures; T_S stands for the running time of state *S*, $delay_i$ expresses a fixed-parameter delay (*i* indexes through all delays of state *S*), *update* accounts for a collection of implicit delays caused by condition-synchronization (*j* indexes through all updates of state *S*) and T_M stands for the running time of the module *M* with states S_k (*k* indexes through all states of the module *M*).

$$T_S = \sum_i delay_i + \sum_j update(j) \tag{3.1}$$

$$T_M = \sum_k T_{S_k} \tag{3.2}$$

Component Performance Measurement

However, running time must be calculated differently for a component than for a module since modules may run concurrently. The running time of the component cannot be derived by a simple sum of all running times. Rather, we look for an end-to-end delay because it gives us the running time of the component. The start time is found as a minimum of start-up time-stamps of all modules within the component. Each module [39] acquires this start-up time at the start of the execution. The end time is found as a maximum of stop time-stamps of all modules within the component. Each module acquires this stop time when it is blocked and there are no inputs available [40].

For example, looking into the processor with compile-time pipelining of symbolic instructions (see Section 3.5.1), each of its CFSM modules, PU, FECTRL, BECTRL and the sets of RUs, WUs and EUs, have their specific start-up time-stamps and stop time-stamps. The processor component running time is determined by a difference between the highest stop and the lowest start-up time-stamp values. If an RU module has a lowest start-up time-stamp and the PU module has the highest stop time-stamp, then these two time-stamps determine the processor component running time.

Equation 3.3 expresses more formally this end-to-end measure; T_B stands for the running time of the component *B* (which may be a processor or a interface), $max(\bigcup_i T_{E_i})$ represents the *end_time* of the component *B* (*i* indexes through all modules of *B* component, T_{E_i} refers to the stop time stamp of the module *i* and *max* extracts the maximal value) and $min(\bigcup_i T_{O_i})$ represents the *start_time* of the component *B* (*i* indexes through all modules of *B* component, T_{O_i} refers to the start time stamp of the module *i* and *min* extracts the minimal value).

$$T_B = max(\bigcup_i T_{E_i}) - min(\bigcup_i T_{O_i}) \tag{3.3}$$

The running time of the whole architecture is calculated in the following way: we look for the maximum end time of all components [41] and the minimum start time of all components [42] and we define the difference between this maximum and minimum as the running time of the architecture *W* (i.e., T_W in Equation 3.4).

$$T_W = max(\bigcup_i T_{B^E{}_i}) - min(\bigcup_i T_{B^O{}_i}) \tag{3.4}$$

[39] A module is described by a single CFSM.

[40] This phenomena we named the artificial deadlock.

[41] $T_{B^E{}_i}$ in Equation 3.4, where *i* indexes through all components in the architecture and *E* stands for an end time stamp.

[42] $T_{B^O{}_i}$ in Equation 3.4, where *i* indexes through all components in the architecture and *O* stands for a start time stamp.

Architecture Timing Model

Based on the CFSM timing model and the component timing model, we have the following definition of the timing model.

Timing Model. *The timing model of the architecture model instance is defined as the 3-tuple* $\langle A, D, T \rangle$ *where,* A *is the abstract instruction set for that instance,* D *is a set of assigned delays for each state of a CFSM, for each CFSM in a component, for each component in the architecture and* T *is a set of calculated performance measurements (*T_S*,* T_M*,* T_B*,* T_W*) obtained during the system simulation.* $\diamond$

It is worth noting that A and D (abstract instruction set and assigned delays) are established through a mapping prepossessing phase called *calibration*. This is not a trivial task at all, and due to that, it is probably impossible to automate. The calibration has a major impact on accuracy of an architecture model instance, since it is driving the configuration of that instance. There is some research work done in the area of calibration of DSE [74], but none of it's results can be acquired ”as is” in our architecture model. An exemplification of calibration issues is given in the next chapter, Section 4.5.3.

3.5 Examples

In this section we present examples of different flavors of components (processor-types, router-types, bus-types) which can be instantiated based on the structural and behavioral architecture elements described earlier.

3.5.1 Model of the Processor Compile-Time Pipelining

The processor with compile-time pipelining of symbolic instructions is an architecture component which: (1) represents a model of the single processor core of an embedded multiprocessor, (2) can execute only a single SP, (3) can execute many symbolic program instructions in parallel but only when they are specified in the SP as bundles [43] of mutually independent SIs - *compile time* and (4) can reuse internal resources to read (write) different FIFOs - application (logical) FIFO identifiers are decoupled from the identification of architecture FIFO components [32].

The processor with compile-time pipelining of symbolic instructions consists of the following threads: the extraction and fetching thread (PU in Section 3.4.2), the symbolic instruction scheduling thread (FECTRL in Section 3.4.2), the symbolic instruction dispatching thread (BECTRL in Section 3.4.2) and a specified number of execution threads of read, execute and write type (RU, EU and WU in Section 3.4.2, respectively).

The topology description of this processor type component (i.e., which threads are interconnected by which channels) is as follows (see Figure 3.5):

[43]This term is commonly used by the compiler community when they refer to scheduling a set of partially assembly instructions on top of a VLIW processor [64].

- the PU thread is connected to the FECTRL thread through a read-write blocking channel (RWB),
- the FECTRL thread is connected to the BECTRL thread through a read-write blocking channel (RWB),
- the BECTRL thread is connected to a configured number of RU, EU, WU threads by the blocking dispatcher channel (BD).

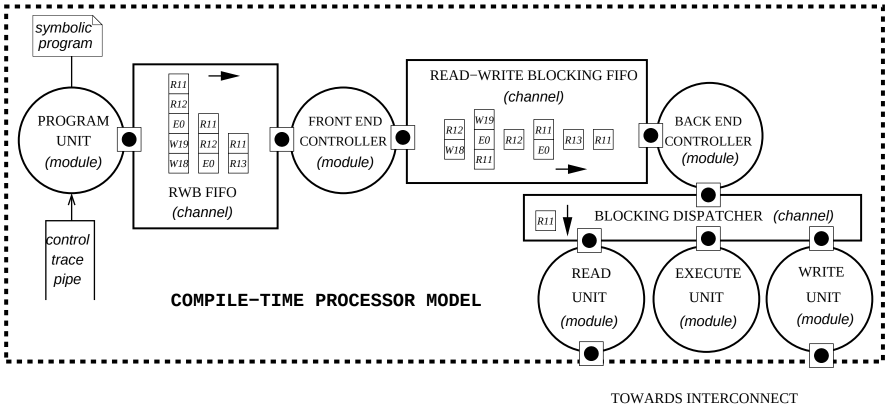

symbolic program
PROGRAM UNIT (module)
control trace pipe
RWB FIFO (channel)
R11 R12 E0 W19 W18 R11 R12 E0 R11 R13
FRONT END CONTROLLER (module)
READ−WRITE BLOCKING FIFO (channel)
R12 W18 W19 E0 R11 R12 R11 E0 R13 R11
BACK END CONTROLLER (module)
BLOCKING DISPATCHER (channel)
R11
READ UNIT (module)
EXECUTE UNIT (module)
WRITE UNIT (module)
COMPILE−TIME PROCESSOR MODEL
TOWARDS INTERCONNECT

Figure 3.5: Internal structure of the processor with compile-time instruction scheduling.

The configuration parameters of this processor component are given in the mapping chapter, Chapter 4.

3.5.2 Model of the Processor Run-Time Pipelining

The processor with run-time pipelining of symbolic instructions is an architecture component which: (1) represents a model of the single processor core of an embedded multiprocessor, (2) can execute only a single SP, (3) can execute many symbolic program instructions in parallel irrespective of whether or not they are specified in the SP as bundles of mutually independent SIs - if they are not, then the parallelism is established at *run-time* and (4) can reuse internal resources to read (write) different FIFOs [75].

The processor with run-time pipelining of symbolic instructions consists of the following threads: the extraction and fetching thread (PU), the symbolic instruction scheduling thread (FECTRL), the symbolic instruction dispatching thread (a modified version of BECTRL, Section 3.4.2), the dispatching threads for each type of symbolic instruction (R-BECTRL, E-BECTRL, W-BECTRL) and a specified number of execution threads of read, execute and write type (RU, EU and WU, respectively). In addition, the processor with run-time pipelining of symbolic instructions makes use of the following channels: the read-write channel with non-blocking methods to examine the possibility of partial retrieval of an SI-bundle [44] and a

[44]This is a Peek-RWE channel, Section 3.4.3. Note that this channel transports partially ordered instructions - SI-bundle.

data-operand-flow channel which keeps the data [45].

The topology description of this processor model (i.e., which modules are interconnected by which channels) is as follows (see Figure 3.6.):

- the PU thread is connected to the FECTRL thread through a read-write blocking channel (RWB),
- the FECTRL thread is connected to the BECTRL threads through a peek read-write-execute channel (Peek-RWE),
- the BECTRL thread is connected to the R-BECTRL thread through a read-write blocking channel (RWB),
- the BECTRL thread is connected to the E-BECTRL thread through a read-write blocking channel (RWB),
- the BECTRL thread is connected to the W-BECTRL thread through a read-write blocking channel (RWB),
- the R-BECTRL thread is connected to a configured number of RU threads through a particular instance of the blocking dispatcher channel (BD),
- the E-BECTRL thread is connected to a configured number of EU threads through a particular instance of the blocking dispatcher channel (BD),
- the W-BECTRL thread is connected to a configured number of WU threads through a particular instance of the blocking dispatcher channel (BD),
- the RU threads are connected to the EU and WU threads and the EU threads are connected to the WU threads through a symbolic instruction operand crossbar (SIOC).

The configuration parameters of this processor component are given in the mapping chapter, Chapter 4.

3.5.3 Model of the Programmable Multi-Processor

The programmable multi-processor is an architecture component which: (1) represents a composition of multiple identical (homogeneous) single processor cores in an embedded multiprocessor, (2) can execute multiple SPs, (3) can migrate the SP executions over different processor cores at run-time, (4) can execute many symbolic program instructions in parallel only if they are specified in the SP as ordered SIs and (5) can reuse internal resources to read (write) different FIFOs. Later in this thesis we use the programmable multi-processor component when modeling many-on-many mappings [46].

The programmable processor component is shown in Figure 3.7. It consists of the following composite modules, threads and channels: a set of composite modules which model the

[45]This is an SIOC channel, Section 3.4.3.
[46]See Chapter 4, Section 4.5.3.

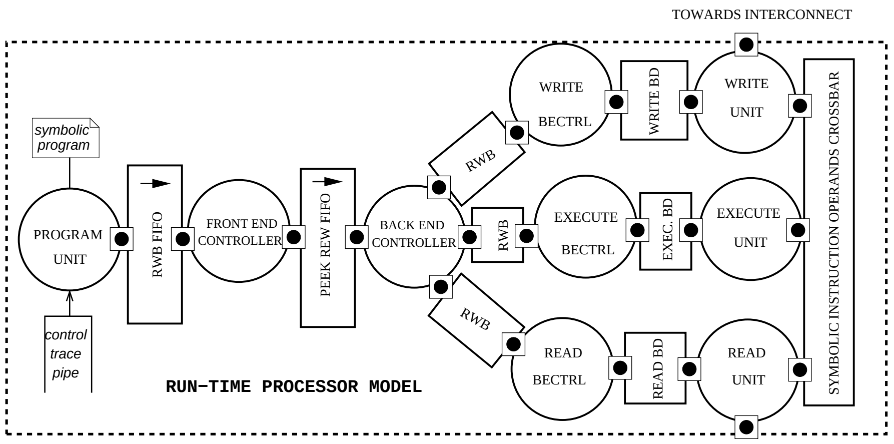


Figure 3.6: The processor with run-time instruction scheduling.

multiple program memory spaces (see SPS in Section 3.4.2), the operating-system channel (see DOS in Section 3.4.2), the modules to report on the network status (see RIC and WIC in Section 3.4.2) and a set (a configured number) of single processor cores (see SPU-core in Section 3.4.2).

The topology description of the processor model (i.e., which units are interconnected by which channels) is as follows (see Figure 3.7):

- the number of SPS modules (each SPS module corresponds to a single application process) are connected to the number of SPU-cores (each SPU-core corresponds to a single processor core in a multiprocessor) through the distributed operating-system channel (DOS),
- the RIC thread is connected to the SPU-cores through the DOS channel,
- the WIC thread is connected to the SPU-cores through the DOS channel.

The configuration parameters of this processor component are given in the mapping chapter, see Chapter 4.

3.5.4 Model of the Routing Interfaces for a Point-to-Point Network

The point-to-point routing interfaces are architecture components which: (1) model the routing over an interconnect of exclusive data-transfer lines, (2) move data from the processor IPC context to the network IPC context and vice-versa, (3) synchronize data accesses to the global FIFO memory buffers and (4) move the data to and from the global FIFO memory. There is no resource contention in this case - the performance of the network architecture, considered in isolation of the application model (representation), is fully dependent on the global FIFO memory capacity.

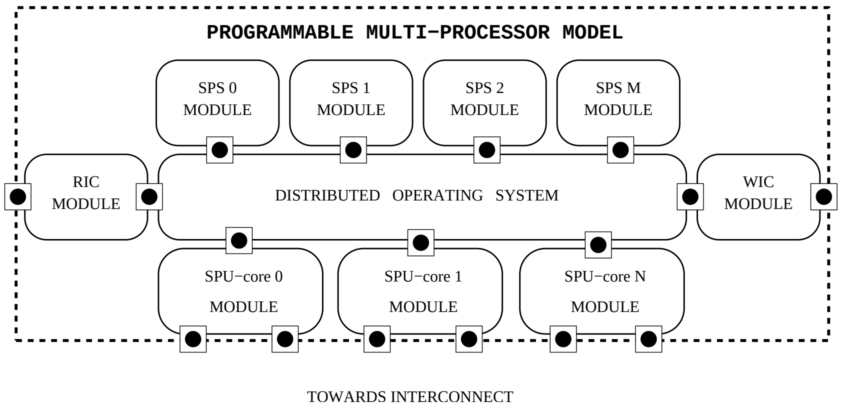


Figure 3.7: The programmable (multi)processor.

There are two complementary types of point-to-point routing interface: 1) The read network interface (RIF), through which all refined `read` symbolic instructions are routed and 2) write network interface (WIF), through which all refined `write` symbolic instructions are routed.

The RIF router (Figure 3.8) consists of: one pull channel (PULL in Section 3.4.3), a configured number of FIFO input-controller modules (FICTRL in Section 3.4.3) and a configured number of FIFO input-modules (FIU in Section 3.4.3). The number of FICTRL modules in the RIF component is equal to the number of RU modules in the processor component connected to that RIF component. The number of FIU modules in the RIF component is equal to the number of FIFO components connected to that RIF component [47].

The configuration parameters of this processor component are given in the mapping chapter, Chapter 4.

The WIF router (Figure 3.9) consists of: one push channel (PUSH in Section 3.4.3), a configured number of FIFO output-controller modules (FOCTRL in Section 3.4.3) and a configured number of FIFO output-modules (FOU in Section 3.4.3). The number of FOCTRL modules in the WIF component is equal to the number of WU modules in the processor component connected to that WIF component. The number of FOU modules in the WIF component is equal to the number of FIFO components connected to that WIF component [48].

The configuration parameters of this processor component are given in the mapping chapter, Chapter 4.

[47]If the number of FICTRL modules is r and the number of FIU modules is f, then the rule which applies is: $r \leq f$.

[48]If the number of FOCTRL modules is w and the number of FOU modules is f, then the rule which applies is: $w \leq f$.

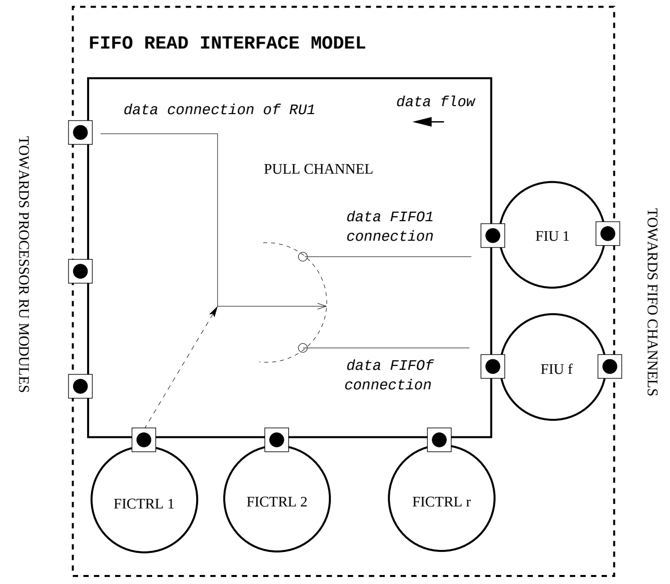


Figure 3.8: Point-to-point read routing interface.

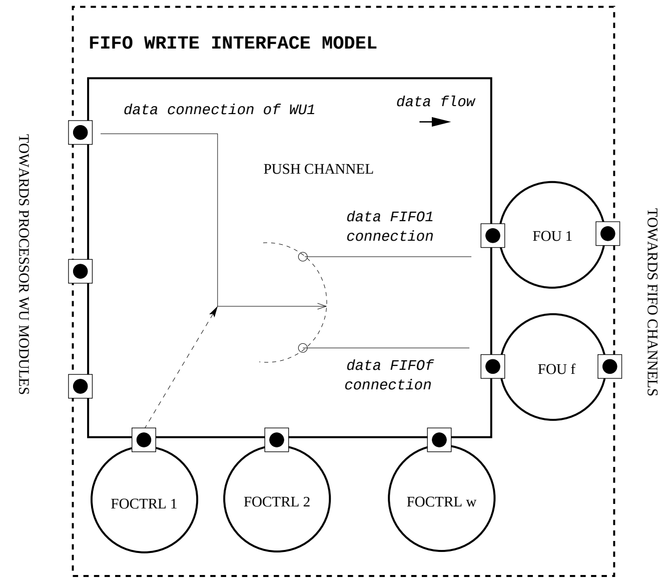


Figure 3.9: Point-to-point write routing interface.

3.5.5 Model of the Routing Interfaces for a Shared Bus Network

The shared bus routing interfaces are architecture components which: (1) model the routing over shared data-transfer lines, (2) move data both ways between the processor IPC context and the network IPC context, (3) synchronize data access to the global FIFO memory buffers and (4) move data to and from the global FIFO memory. There is possibly a resource contention when using shared bus and that affects performance of the network architecture.

The model of shared bus router component is based on the point-to-point component model introduced in Section 3.5.4 and conveniently modified to match the shared bus communication case. The changes affect the FIU and FOU modules inside the RIF and WIF routing components respectively: before data transfer can take place, the shared data transfer lines must first be claimed and, after data transfer has finished, the shared lines must be released (see the bus-protocol example in Section 3.3.5, illustrated in Figure 3.2).

3.5.6 Model of the Routing Interfaces for a Burst Bus Network

The burst bus routing interfaces are architecture components which: (1) model the routing over the shared data-transfer lines, (2) move both ways between the processor IPC context and the network IPC context, (3) translate data-tokens to fit the packet size of the burst bus, (4) cache data in the routing interfaces, (5) synchronize data access to the global FIFO memory buffers and (6) move data to and from the global FIFO memory in predefined chunks of data. In this case, there is not only a resource contention but there are also data delays caused by buffering of data which affects performance of the network architecture.

The model of the burst-bus router component is based on the shared-bus component model introduced in Section 3.5.5 and conveniently modified to match the burst bus communication case. The changes affect the FIU and FOU modules inside the RIF and WIF routing components respectively. However, unlike the modifications in Section 3.5.5, which were symmetrical (the same change applies for both the RIF component and the WIF component), the burst bus changes are asymmetrical. The change at the WIF component side accounts for the fact that the WIF component must provide more concurrency and smarter buffering internally than is the case for the WIF component in the point-to-point connection (Section 3.5.4) and the WIF component in the shared-bus connection (Section 3.5.5). The change at the RIF component side is rudimentary; in the burst bus case the FIU module of an RIF component may have already retrieved more data than it really needs so that when the next symbolic `read` instruction arrives the RIF component will immediately deliver the already retrieved data to the processor.

There are pros and cons when transferring data using burst-packets. The pros are: 1) Data is retrieved by a burst-RIF component earlier than what is needed by the processor RU modules - this implies that data is delivered faster than with non-burst buses; 2) The number of bus-transactions is less than the number of transactions in the case of non-burst buses. This is because data is already fetched (or cached within an RIF component) so less bus-transfers are needed. There are two cons. Firstly, since the data is transferred in bursts, WIF components must buffer a few consecutive transfers to the same FIFO buffer. Buffering is dictated by the burst-word size which may not ideally match the size of the data indicated in `write` symbolic instructions. In order not to cause a hangup in the WIF component due to lack of data in the burst packet, the burst-translation (buffering) is guarded by a buffering interval. So, if for a long time there are no more `write` symbolic instructions targeting the particular FIFO, the burst data-transfer will take place with the amount of data being buffered. This means that infrequent and small `write` symbolic instructions will de-grade the network performances. Secondly, if the amount of buffered data is not equal to the burst-packet size (the case of insufficient data in WIF), the remaining space in the packet, when the burst occurs, will be filled with useless data. Therefore, the efficiency and the throughput may be lower than expected.

3.5.7 A Heterogeneous System

A heterogeneous system consist of a heterogeneous multiprocessor and a heterogeneous communication network. It can be modeled using all the previously described components. Here, we give a brief description (example) of such a system.

Let us assume that the aim is to model a coprocessor-based embedded system, consisting of a programmable processor (see Section 3.5.3), global memory accessible via burst bus (see Section 3.5.6), coprocessors (see Sections 3.5.1 and 3.5.2) and a dedicated memory used for communication between processor and coprocessors (see Section 3.5.4). The structural model of this system [49] is shown in Figure 3.10. The communication network is surrounded by a dashed line. The top-part of the network in Figure 3.10 is the network part with dedicated channels. The bottom part of the network in Figure 3.10 is the bus-based network part.

The ability to model heterogeneous embedded system architectures is crucial because - as we will see in Chapter 4 - heterogeneous embedded multiprocessors are the ultimate architectures for mapping streaming applications. In addition, designers may acquire very important information about the ‘cost vs. performance’ ratio using the heterogeneous embedded system modeling scheme. That is, they are able to estimate e.g., whether mapping some part of the application onto a pure hardware processors and dedicated channels is justified from the performance improvement point of view or not, as well as whether it is acceptable from the cost impact point of view or not.

3.6 Related Work

Closely related to the architecture modeling described in this thesis are the other symbolic instruction driven architecture models: `Spade` and `Sesame`. The difference comes from the representation used to capture these symbolic instructions: instead of symbolic programs these two models use linearly (i.e. totally) ordered traces. The symbolic instruction traces drive the (non-functional) architecture model which interprets the transformed symbolic instruction in terms of performance and cost values.

[49]The programmable processor is marked as `MT-proc.`, coprocessors are marked as `compile-T`, the global memory is light-shaded and the dedicated memory is dark-shaded.

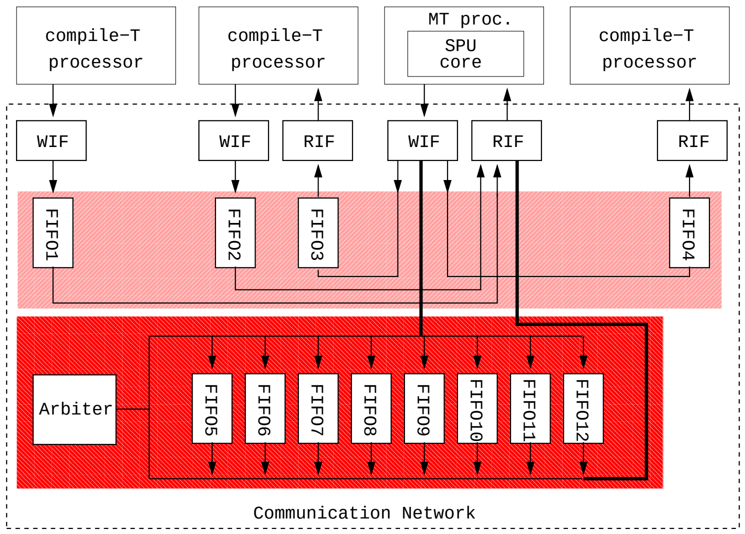


Figure 3.10: Model of a heterogeneous system.

3.6.1 Spade Architecture Modeling

In this section we present a brief description of the architecture modeling in the Spade methodology. The methodology was mentioned briefly in Chapter 1, Section 1.6.1. For the methodology, we refer to [7] and [33].

The Spade architecture modeling resembles a *pioneering* approach. It was the first to make use of the separation of concerns by using application and architecture symbolic instructions. Therefore, the architecture model focussed fully on timing rather than on functional performance.

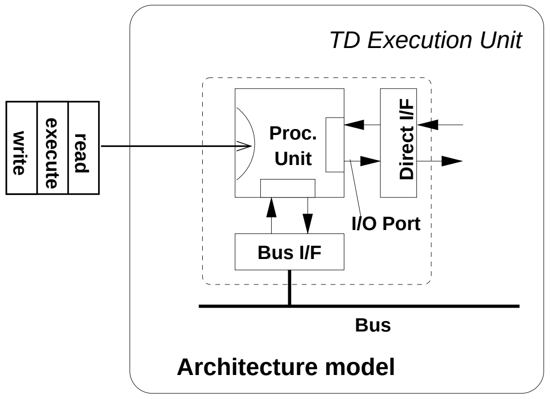


Figure 3.11: The Spade TDU related execution unit

The `Spade` architecture model is based on the *trace-driven* (TD) approach [76]. The architecture components are generic building blocks without any functional behavior but with a parametrized timing (delay) behavior. The processors are modeled using the `Spade` TD execution unit (TDU). The network interfaces, interconnections and memory are basically *Point-to-Point* (PtP) FIFO connections, which may be temporally scheduled by a semaphore (a bus model).

A TDU accepts symbolic instructions from the architecture trace of symbolic instructions (see Figure 3.11). The traces of symbolic instructions are mentioned in Chapter 2, Section 2.1. The symbolic instructions may either be the non-refined symbolic instructions `read`, `write` and `execute` from the application trace, or may be their refined versions `check-data`, `load`, `signal-room`, `checkroom`, `store`, `signal-data` and `execute` obtained by applying trace transformations in the mapping layer [30]. In both cases, there is a strict order among the symbolic instructions within a trace: either symbolic instructions are strictly ordered, or at least, there is a strict order between the `load-execute-store` sequences. Hence, rather than re-arranging (rescheduling) the strictly ordered trace, a TDU executes the symbolic instructions in the given order. This implies that potential concurrency hidden in the application process cannot be exploited by TDUs: neither do traces keep such information, nor do TDUs make use of such information. Indeed, in [30] a set of possible trace transformations were presented as a possible way out of the above restriction. However, recall that trace symbolic instructions do not capture process control constructs. Possible trace transformations are, therefore, limited in number. Consequently, the number of realistic architectures that can be modeled using `Spade` architecture modeling is rather limited.

3.6.2 Sesame

In this section we present a brief description of the architecture modeling in the Sesame methodology. For the methodology, we refer to [36].

The main differences between `Spade` and `Sesame` architecture are: (1) the `Sesame` architecture models are implemented rather using the *discrete event simulation language* [35] than the *cycle-driven simulation language* [7] and (2) the `Sesame` architecture models are based on refined application symbolic instructions [30]. This allows for significant modeling-space and simulation-time improvements over `Spade`. Nevertheless, the architecture components still follow the basic `Spade` idea: they are essentially non-functional timing (delay) blocks.

Comparing to other the models described in this thesis, we can conclude that the concept of `Sesame`'s virtual processors [36] is strongly influenced by the symbolic program paradigm [11, 32]. However, unlike our model, which refines application symbolic instructions into architecture symbolic instructions in the architecture components, the `Sesame` model expects this to be done in the mapping model, before reaching the architecture component. Moreover, each symbolic trace is bound to a dedicated architecture component and this makes context-switching and context-migration unrealistic or even impossible [50]. Finally, the virtual processors are essentially bound by the particular Synchronous Data Flow (SDF) graph [77]: the input trace-sequence is essentially a repeatable pattern of `read`, `execute` and `write`

[50] In other words, homogeneous multiprocessing cases or even multitasking cases are not realistically modeled.

symbolic instructions. If the trace-sequences are random or are not periodic, the SDF graph becomes increasingly complex. At this point, `Sesame` tries to solve this problem by following the main ideas of symbolic programs and control points - it re-uses the CDFG-like *selection* and *iteration* constructs, but it names them differently: `CASE-BEGIN/END` and `REPEAT-BEGIN/END` [36]. However, the SDF graph may be sensitive to the data-set because the separation of concerns has not been applied on the traces, so that both the data-set dependent and data-set independent elements are still in a single trace of symbolic instructions. Thus, the same application, which is highly data-dependent, may produce a different `read`-`execute`-`write` sequence when processing a different data input (e.g. a different JPEG image or a different MPEG stream), which also implies that the architecture instance below the virtual processor may vary (may differ) from one data set to the another data set. In such a case, the number of the SDF graphs needed is non-determinate, which also implies that the model is not reusable because it is not generic. On the contrary, in our model this issue cannot appear because the architecture model components at the simulation time decide what is the `read`-`execute`-`write` sequence to be executed and, thus, they are insensitive to any relative pattern within this sequence.

Chapter 4

Mapping Modeling

E Pluribus Unum. [1]

4.1 Summary

This chapter deals with methods to relate application models to architecture models. These methods transform the representation of the application model to the representation of the architecture model. Together they constitute what we call the mapping of the application on the architecture. The application representations are mainly process representations in the form of symbolic programs. The architecture representations are architecture specific symbolic programs that differ from the application specific symbolic programs. The mapping transformations aim to bridge the mismatch between the two symbolic programs. The chapter presents the various mapping steps and mapping techniques and illustrates the process by means of case-studies.

4.2 Introduction

Because we separate the application model from the architecture model, there is in general no match between these models, except for the fact that the application is specified in a parallel language [78] or model of computation [18] and the architecture is specified in terms of interconnected components [32]. These components in the architecture behave in general in a different way to the process counterparts in the application. The two behaviors have thus to

[1] *From many, (comes) One*. In original: *color est e pluribus unus*. The phrase originally came from ”Moretum”, a poem attributed to Publius Vergilius Maro - known in English as Vergil - (70 B.C.-19 B.C.) a classical Roman poet who completed Aeneid. Much later, the motto was selected by the first Great Seal committee in 1776, at the beginning of the American Revolution.

be related in one way or another. In other words, a transformation of the application behavior into the architecture behavior needs to be performed in order to deal with unavoidable mismatch between the two models. This transformation we call *the mapping of the application on the architecture*. In our case, we want to apply transformations on the symbolic program representations of application processes and their architecture component counterparts. The mapping process starts out with the assignment of application process and communication channels to architecture processor components and storage components, respectively. Once an application process is assigned to an architecture processor component, the symbolic program that represents the application process has to be transformed to the corresponding architecture processor component symbolic program that is composed of processor component native instructions which are - in general - different from the instructions in the application symbolic program. Of course, the 'native instructions' of the processor component are purely symbolic - that is, they do not encompass any functional behavior, only latency and throughput annotations. To illustrate a gap between application and architecture symbolic instructions we provide the following example: An application instruction `read` may translate to an architecture instruction tuple ⟨`check-data`, `load-data`, `signal-room`⟩ as given in [30]. The mapping of an application to an architecture is considered to be the third independent part in a system model. Thus, in Design Space Exploration and Performance Analysis of a System, one can, and in general will, deal with alternatives in all three parts: application model, architecture model and mapping transformations, all at the appropriate level of abstraction and in an *exploration-driven* embedded system design process. The latter is different from a synthesis-driven embedded system design process although the exploration-driven approach should not be such that a designer cannot rely on it to perform synthesis.

As a matter of fact, the user of a DSE methodology [2] should be able to recognize DSE mapping based on his understanding of what the mapping stands for in a traditional synthesis driven design process. That is, there should be a strong resemblance between: (1) The mapping actions in traditional embedded system design processes, and (2) DSE mapping steps. In a traditional synthesis-driven embedded system design process, applications are specified using a high level language [3] (HLL) and architectures (SoC) are chosen based on their availability, stated performance, reconfigurability, and support for a software development kit (SDK). In the most general sense, SDK includes tools such as compilers, linkers and OS images. The traditional mapping (for embedded system synthesis), then follows as an iterative process consisting of:

1. Binding - the assignment of the application processes to architecture processing units and the application IPC to architecture communication resources,
2. Porting - rewriting the HLL specification in order to meet the hardware description language (HDL) or SDK requirements and the architecture features,
3. Translation - manual and automatic (tool) actions which will produce architecture specific object (instruction) code for each bound and ported application entity,
4. Building - the creation of the *system image* which can be loaded and executed on that particular architecture,

[2]The builder of an embedded system.

[3]C, C++, Java, Matlab, ...

5. Integration - making of loaders and start-up scripts and embedding them together with the system image into the SoC, and finally,

6. Performance Test & Verification - running the integrated SoC system with whatever set of data in whatever environment in order to verify the behavior and to test the performances.

We argue that the DSE mapping described in this thesis resembles most of the traditional mapping for synthesis steps, and thus, it fits the user's needs better than some other DSE methods [34]. The user does not need to fundamentally change his understanding of 'mapping', since the traditional design-for-synthesis steps are preserved. The only requirement for the user is to keep in mind that a *system* **is** a simulation program and **not** a synthesized system image, as well as that the modeled architecture is characterized in terms of latencies and throughput and not in terms of functionalites. In any case, the triple $\langle\, application\ model$, $architecture\ model, mapping\ transformations\,\rangle$ is again a model, the system-model.

4.2.1 Chapter Organization

This chapter is organized as follows. Firstly, we roughly introduce the mapping specification in Section 4.3. Secondly, in Section 4.4 we define and illustrate the steps needed to create the mapping specification. Thirdly, in Section 4.5 we show some exploration cases and their results. Finally, in Section 4.6 we list and explain the main contributions of our approach versus some related mapping modeling approaches.

4.3 Mapping Specification

The mapping of an application model instance onto an architecture model instance is completed when a designer creates [4] a so-called *mapping specification*. Roughly speaking, the mapping specification establishes unambiguous *workload-versus-resource* relations, namely: (1) a relation between the computational workload of an application representation and the computation resources of an architecture model, and (2) a relation between the communication workload of an application representation and the communication resources of an architecture model.

Once the mapping specification is ready, it is parsed and a `SystemC` file is generated. By compiling this file with the library of architecture models (see Chapter 3) the simulation program is created. By running this simulation program on a particular data-set (see Chapter 2) we get simulation numbers for that data-set.

In the following section we identify the steps needed to establish a mapping specification, and consequently, to create a simulated system program.

[4] In order to create a mapping specification a designer may use a tool-set, follow a mapping methodology, or rely on his *know-how* i.e. mapping-experiences.

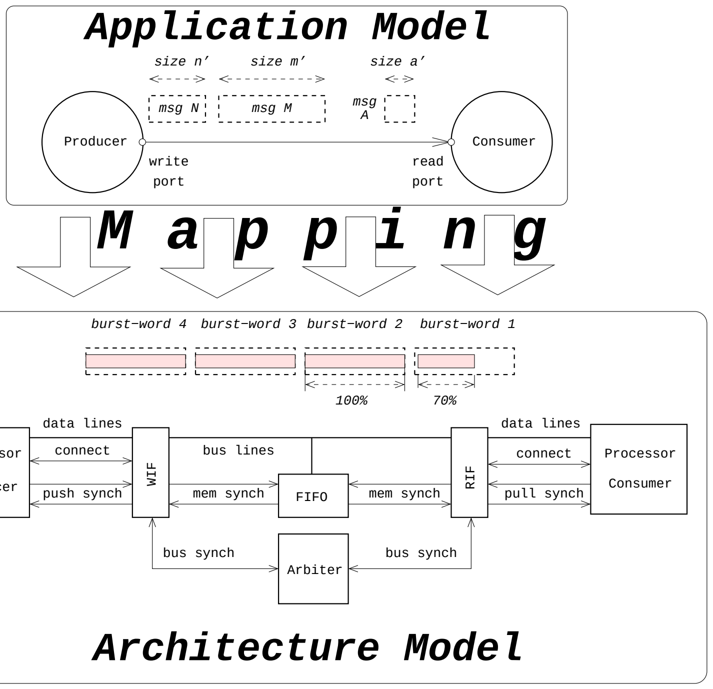


Figure 4.1: Mapping of the application model onto the architecture model

4.4 Mapping Steps

Figure 4.1 illustrates that a mapping is needed to associate application models with architecture models.

The application model is the Kahn Process Network (KPN). Therefore, the application behavior is preserved, but no resource constraints have been decided yet (See Chapter 2). Having such application model helps to create multiple data-sets cheaply; the output of a KPN for a given input data is invariant to the KPN schedule. Hence, the application simulation data can be generated independently of mappings. If the application KPN does not change, the symbolic programs do not change either, so the application simulation data is limited to data-dependent control traces.

The architecture model is based on a library of components (See Chapter 3). The architecture components exchange data based on architecture SIs. However, these SIs are different to the SIs coming from the application representation. Architecture components are unaware of functional behavior, they can only produce latency (delays) and throughput (synchronization). Moreover, the data used by the architecture components has precise granularity, while the data in the application has varying length, it is abstract, and it is generally too complex for immediate use in the architecture. Therefore, it is necessary to translate annotations coming from symbolic programs into architecture SIs.

In practice, the mapping is an iterative process consisting of a set of subsequent steps (see Section 4.2) rather than a high-level, coarse-grained, monolithic job. Similarly to the practice, a DSE mapping shown in Figure 4.1 cannot be a monolithic job since the gap between application model and architecture model is too big. Therefore, we have to identify steps for our mapping model similar to the existing steps in this so called 'traditional mapping for synthesis'. Identifying the mapping modeling steps is dependent on the following two aspects: (1) what is the modeling goal (accuracy vs. abstraction) and (2) what are the flow types (platform vs. budgeting). We have indicated in Chapter 1 that our modeling method aims at both abstraction and accuracy. Regarding the second aspect, our method flow is rather platform-based since the mapping input is fixed to a large extent and the main architecture functions are described in terms of architecture primitives (architecture model is a repository of parametrized library components). As a matter of fact, this is the one of the reasons our mapping model resembles realistic mapping implementation steps.

We recognize the following four steps in our mapping modeling:

- Binding - the assignment of the components of application representation to the components of architecture representation,
- Matching - translating abstract data types and high-level application behaviors into architecture model data and instruction types,
- Refinement - translating application symbolic programs into architecture symbolic programs of suitable granularity and quality, and
- Transforming - pre-processing architecture symbolic programs and architecture control traces in order to expose more parallelism among symbolic instructions of symbolic programs, to create smaller symbolic programs, to reduce the number of loops in symbolic programs, to reduce number of conditional constructs, etc.

The order in which these steps are applied may vary for different mappings. Some steps may be performed more than once or they may not be performed at all - depending on a mapping case. For example, matching, transformation and refinement steps may be repeated many times in order to create an appropriate mapping.

4.4.1 Binding Step

The application KPN model executes the application on a single data set, resulting in the pairs $\langle symbolic\ program, control\ trace \rangle$, where only $control\ trace$ depends on the data set (see Chapter 2). The architecture model is a network of interconnected architecture components (see Chapter 3). Binding is the assignment of: (1) symbolic programs and control traces to processor components, and (2) KPN channels to architecture communication components (routers, FIFOs and arbiters). For example, in the example shown in Figure 4.1 the binding is of type 1-on-1, where `Producer` process is bound to `Producer Processor` and `Consumer` process is bound to `Consumer Processor`, and the channel between `Producer` and `Consumer` is bound to a `FIFO` component, between `Producer Processor` and `Consumer Processor`.

4.4.2 Matching Step

Having symbolic programs bound to the architecture components leaves many issues open such as: (1) the relationship between the width of the processor component words and the annotated size in the symbolic instructions of the symbolic program(s) bound to that processor, (2) the types of the processor component primitives versus the number of different `execute` symbolic instructions of the symbolic programs(s) bound to that processor, and (3) the relationship between the width of the communication interface words and the annotated message size in `read` and `write` symbolic instructions of the symbolic program(s). Therefore, the usual step that follows the binding step is a translation of the data types and primitives (symbolic instructions) from the bound symbolic programs to the data types and primitives (symbolic instructions) of the CFSMs of the architecture components (see about *abstract instruction set* and *time measurement* in Chapter 3, Section 3.4.6). We call this step Matching.

Firstly, we need to examine an SI-subtype for each symbolic instructions-type. In the case of an `execute` SI the SI-subtype differentiates among different computation functions, and in the cases of `read` and `write` SIs the SI-subtype differentiates among different target FIFOs. When all possible SI-subtypes belonging to SIs within an SP bound to a particular processor component are detected, we need to associate the input and the output arguments with these SI-subtypes. In the case of `read` and `write` SIs these are single arguments since only a single data item can be read from or written to a FIFO. In the case of an `execute` SI these will in general be a list since an `execute` can take many inputs and produce many outputs. However, there are two exceptions where an `execute` contains either one-or-more outputs or one-or-more inputs: A source process (or processes) of the application KPN, and a sink process (or processes) of the application KPN. Finally we need to examine budgets in these SIs [5]. Budgeting of an SI determines the volume of communication or computation which the SI will model. In the cases of `read` and `write` SIs the budget number represents a token size to be read from or written to the target FIFO. In the case of `execute` SI, the budget number represents the worst case execution time for the annotated computation function. Therefore, the budget says either how many times an EU CFSM in the processor component is going to repeat a computation delay, or how many times an RU CFSM or a WU CFSM in the processor component is going to repeat a communication delay. It is worth noting that the computation delay may refer to a single instruction, a basic block, or an atomic routine (depending on the choice) whilst the communication delay refers to the transfer of a single communication word to-or-from a processor. For instance, in the example shown in Figure 4.1, the matching is done between the sizes of the application messages (*msg A*, *msg M*, *msg N* and their sizes *size a'*, *size m'*, *size n'*, respectively) and the burst-words in the architecture (*burst-word 1*, *burst-word 2*, *burst-word 3*, and *burst-word 4*). Note that some burst-words are 100% filled with valid data, while some others are not - nevertheless, all of them cause the same delay.

Obviously, the most critical task in this step is to establish the budget such that the matching between an SI and an architecture primitive is as good as possible at this level of abstraction [6]. This is also known as a *calibration*. The calibration is not directly part of DSE, but it is rather a pre-processing step for DSE. It is a matter of relating parameters at two levels of

[5] See Figure 4.1, different `size`s model different communication budgets in the application model part.

[6] Remember, in Chapter 3, we defined our models as *Transaction-Level Models*.

abstraction: (1) The abstraction level of the application model, and (2) the abstraction level of the architecture model. In this sense, we could say that the ’calibration’ is a form of matching and that it could be done for all components in the architecture model library, once and forever. However, actual architectures cannot always be specified in terms of the available library components, and the calibration step is really difficult in these cases. Therefore, the matching step for calibration purposes must be repeated as many times as needed (e.g., until simulation results of the calibration case reach saturation, or until simulation results of the calibration case reach some reasonable or sought accuracy.).

4.4.3 Refining Step

This step always exists due to the fact that there is a mismatch between the application model representation and the architecture model representation. For example, the symbolic programs may base computation and communication on tokens that are atomic for the application model whereas they are composite tokens in the architecture model. Furthermore, the components communicate using primitives other than `read` and `write`. In Chapter 2 we have already defined the horizontal refinement as the creation of more instructions of the same type that act on the tokens of granularity which are smaller than the original instruction. Similarly, we have defined the vertical refinement as the creation of more instructions of different types that act on the tokens of the same granularity as the original instruction. For example, in the example shown in Figure 4.1 we show the vertical refinement between: 1) `read` and `write` symbolic instructions visible as read/write ports at the application side, and 2) complex partially ordered sequences of `push-pull`, `signal-wait data-room`, `connection`, `bus claim/grant` synchronization protocols and `data` and `bus transfers`.

Obviously, the refining step is closely linked with the matching step. For example, by means of the horizontal refinement we change the size-annotation in the SIs, while we determine the appropriate size-annotation by matching. In this chapter we exemplify these refinement strategies.

4.4.4 Transforming Step

In a sense, binding, matching, and refining are also transformations. Nevertheless, the Transforming Step refers to the transformations which affect the order among SIs in a symbolic program. In general we consider two sorts of transformations: (1) platform-independent, and (2) platform-dependent. The former do not require any prior knowledge of architecture (buffer sizes, timing-info, or similar) while the latter do. Platform-independent transformations are in a way part of the application modeling. The SP example shown in Chapter 2, Figure 2.13, is a platform-independent detection of variants of partially ordered SIs within a symbolic program. However, in order to derive some more sophisticated SI scheduling, we need architecture timing and architecture constraints as well. Therefore, the platform-dependent transformations can take place only after binding, matching, and refining steps.

4.5 Mapping Cases

The target architectures in the case-studies have been modeled using the components introduced in Chapter 3. To reach the appropriate mappings, however, we had to iteratively conduct different experiments and several interviews with SoC designers. The reason lies in the "specifications" which SoC designers had produced: they were not generic and not abstract enough for any TLM architecture modeling paradigm. That is, the values of parameters in our model are difficult to derive from the detailed SoC specifications and descriptions which are derived by the designer without any reference to the model.

This is related to 'calibration'. The calibration is both a *necessity* and a *problem*: (1) It is necessary because abstract models are parametrized and some of those need to be given values which must be obtained from the actual component, and (2) it is a problem because it is often difficult to extract parameter values from (low-level) component specifications.

Each of the following three cases is conducted to show or verify certain aspects of our architecture and mapping modeling paradigm. For the simple case described in Section 4.5.1, the main goal was to show efficiency and accuracy of our mapping methodology given that: (1) the platform is implemented in a so-called Field Programmable Gate Array (FPGA), and (2) the application model is represented by means of Symbolic Programs. The case described in Section 4.5.2, illustrates mapping-refinements of the original application-specification without rewriting the original code. Finally, the case described in Section 4.5.3 illustrates how such high-level architecture exploration methods can successfully be used to model a heterogeneous multiprocessor on chip (MPSoC).

4.5.1 Case-study: Adaptive QR Matrix Decomposition

The objective of this case is to model embedded system mappings which can be explained as:

"Create the accurate one-on-one mappings of Kahn PNs onto multi-processor with compile-time pipelining of symbolic instructions [7], where the application process networks are pre-created and cannot be changed."

The restriction on changing application process networks implies that the mapping transformations can happen only at the level of symbolic programs (i.e., we must apply the Transforming Step). Thus, in this sub-section we give (1) a description of the mapping case we conducted and (2) experimental results to support our claims about accuracy, efficiency and the exploration power of the mapping approach presented in this thesis.

The case is based on an algorithm commonly used to solve an over-specified set of linear equations in a least squares sense. This algorithm is known as adaptive QR matrix decomposition [79]. In signal processing practice, this algorithm is used for calculating weights in an adaptive beam-forming system [80]. We performed system-level exploration of different mappings of the QR algorithm onto an FPGA platform as described in [52]. For an understanding of this case it is necessary to give a specification e.g., in the form of a sequential algorithm in `Matlab`. See Figure 4.2. The $r(m,n)$ are entries of an upper triangular matrix

[7]See Chapter 3, Sections 3.3.1 and 3.5.1.

R of size $N \times N$ that is updated at each k-step, the $x(k,p)$ are entries of a vector of size N that are taken from a source consisting of N sensing devices called antenna data in the remainder of this section, and $\theta(p)$ is a vector of size N that represents the orthogonal matrix Q of size $N \times N$ in the decomposition $X = QR$, where X is the stack of all vectors of size N collecting the $x(k,p)$ entries. For the case of simplicity we have assumed that X, Q, and R are real-valued.

```
for k=1:1:K,
   for j=1:1:N,
      [r(j,j),x(k,j),θ(j)]=Vectorize(r(j,j),x(k,j));
      for i=j+1:1:N,
         [r(j,i),x(k,i),θ(j)]=Rotate(r(j,i),x(k,i),θ(j));
      end
   end
end
```

Figure 4.2: A QR matrix decomposition `Matlab` code sample.

Description of The Case

We modeled three different mappings of the adaptive QR algorithm onto an FPGA platform. For the first mapping, the algorithm is modeled as a process network of four communicating processes. The network is shown in Figure 4.3, part 1. For the second mapping, the algorithm is modeled as a process network of eight communicating processes. The network is shown in Figure 4.3, part 2. Finally, for the third mapping, the algorithm is modeled as a process network of twelve communicating processes. The network is shown in Figure 4.3, part 3. All the networks were derived automatically from the sequential algorithm in Figure 4.2, using the `COMPAAN` tool-set [53].

We represented the networks using symbolic programs and control traces. We modeled the FPGA platform using components from the repository of the architecture model components depicted in Figure 3.1. In the experiments, we use the following architecture plus mapping specifications (for each mapping there is one architecture plus mapping specification):

1. **Binding:** The number of processor components in the architecture is equal to the number of processes in the QR process network. In other words, each application process is mapped onto a single processing unit in a 1-on-1 fashion (see *one-to-one* in Chapter 3, Section 3.4.2).

2. **Binding:** The number of FIFO components in the architecture is equal to the number of channels in the QR process network. Each application FIFO channel is mapped onto a single FIFO component in a 1-to-1 fashion.

3. **Binding:** There is no resource sharing, neither for computation (an operating system is not needed since there are no different threads on any processing unit) nor for communication (a bus is not needed since all buffers are dedicated).

4. **Transforming:** The number of simultaneous `read` and `write` operations in the architecture is explicitly shown in the symbolic programs (see the example in Chapter 2,

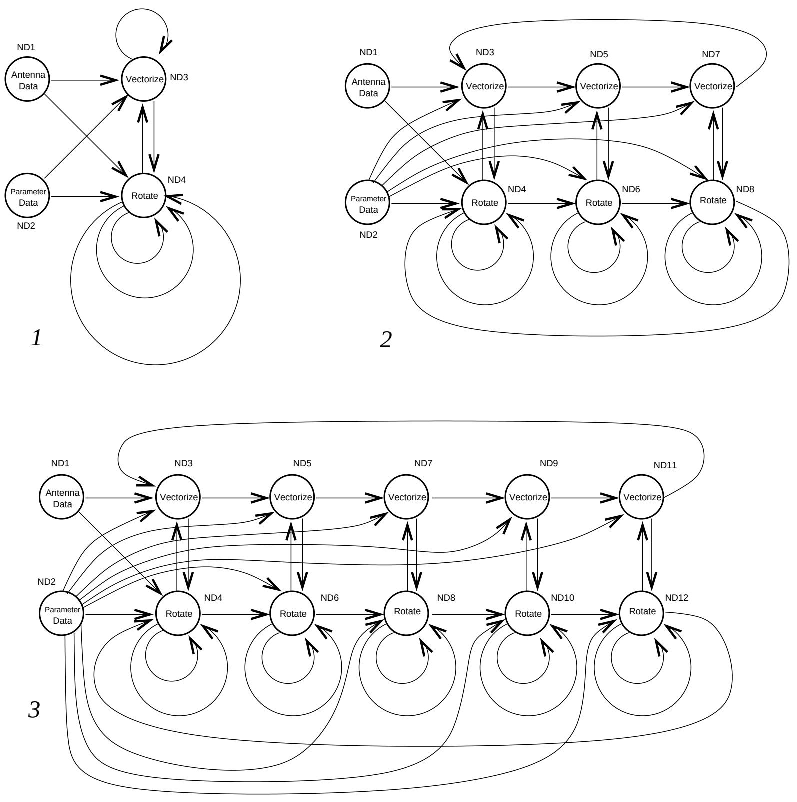


Figure 4.3: The three application QR process networks (Derivation of these process networks is not subject of this thesis, for more information about that please refer to [59].)

Figure 2.13).

5. **Matching:** Each operation (`read`, `write`, `execute`) takes a single processing unit cycle when executed in the architecture.

6. **Matching:** From the architecture network point of view, `read` and `write` operations cause additional delays: a cycle for switching and a cycle for a FIFO buffer access. The FIFO buffer access cycle appears only when blocking on the FIFO takes place.

7. **Matching:** FIFO buffers in the architecture are sized so as to provide enough space (in this case study, for the three mappings the FIFO buffer sizes are always 256 tokens).

Based on the above mapping specification, the nine simulation programs for the nine QR-on-FPGA mappings have been synthesized: (1) There are three application process networks as shown in Figure 4.3; (2) There are two different SP representations for each application network - the first one contains totally ordered symbolic instructions, the second contains partially ordered symbolic instructions; (3) There are three different architecture and mapping specifications for each application-architecture mapping candidate. The first contains specifications for the multiprocessor which cannot execute simultaneously multiple symbolic instructions of the same type (`read`, `execute`, `write`). The second contains specifications for the multiprocessor which can execute simultaneously multiple symbolic instructions of the same type. The third contains specifications for the multiprocessor which can both execute simultaneously multiple symbolic instructions of the same type and pipeline communications to FIFO components. These different mapping specifications are the result of the Calibration - we needed three iterations to determine the correct matching & transforming parameters before the synthesized simulation programs provided us with a relative error of about +1.5%. We describe these results in the next section.

Results

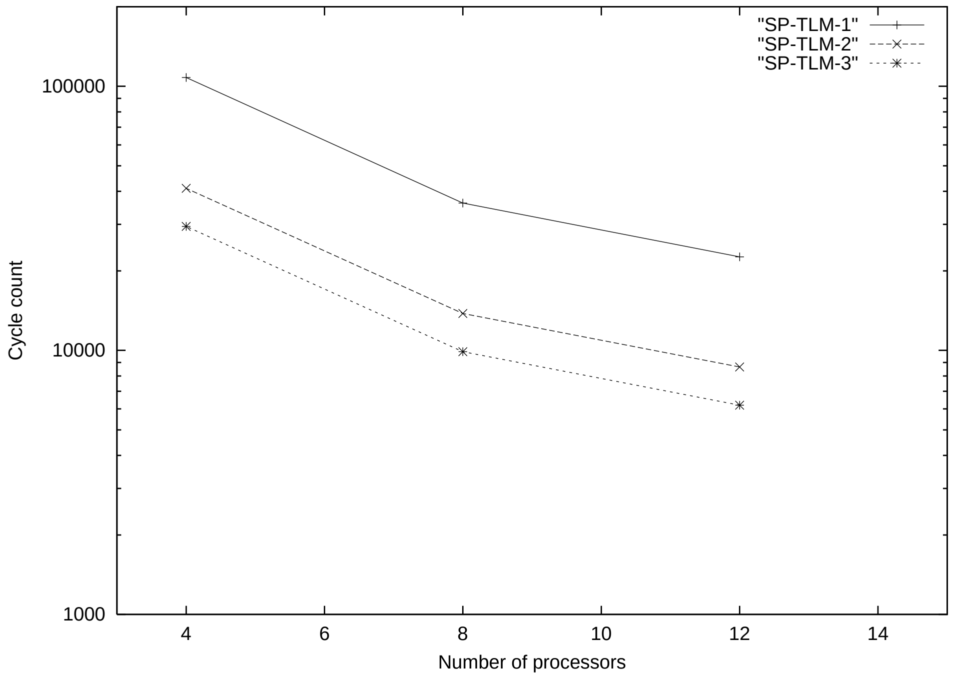


Figure 4.4: The simulation results of the adaptive QR matrix decomposition case-study.

Number of processors	FPGA cycle count	TLM cycle count	relative error
4	29281	29458	0.6%
8	9771	9884	1.2%
12	6111	6202	1.5%

Table 4.1: Cycle-count: the FPGA mapping vs. the SP-TLM-3 mapping

QR onto the FPGA	QR with the TLM model
10 hours	10 seconds

Table 4.2: Required mapping & simulation times: QR on FPGA vs. QR on TLM. We consider *mapping* equal to *compilation*, and we consider *simulation* equal to *execution*. The necessary preparations and adaptations of the application and architecture models have not been taken into account.

We run the executables of our nine mappings, and the results are shown in Figure 4.4. There are three SP-TLM labels [8] which refer to three gradually differing mapping cases: "SP-TLM-1" refers to mappings of totally ordered SPs onto multiprocessors which cannot execute simultaneously multiple symbolic instructions of the same type; "SP-TLM-2" refers to mappings of partially ordered SPs onto multiprocessors which can execute simultaneously multiple symbolic instructions of the same type; and, "SP-TLM-3" refers to mappings of partially ordered SPs onto multiprocessors which can both execute simultaneously multiple symbolic instructions of the same type and pipeline communications to FIFO components.

To quantify the results we show the comparison of the mapping case "SP-TLM-3" versus actual FPGA mappings of the adaptive QR matrix decomposition in Tables 4.5.1 and 4.5.1.

Table 4.5.1 shows the number of cycles needed to complete executions of the different QR networks on the FPGA platform [52] vs. the number of cycles needed to complete the executions of different QR networks on the TLM based model of this platform. As can be seen, the TLM architecture model is able to predict the performance of the real FPGA platform executing the adaptive QR algorithm with a relative error of about +1.5%. For the case in hand, a larger error would have revealed a major flaw in the method. Table 4.5.1 shows that simulation speed is excellent.

[8]"SP-TLM" stands for *Symbolic Program* onto *Transaction Level Architecture Model*.

4.5.2 Case-study: Mapping 2D-IDCT Specification to IP-primitives

The objective of this case is to model embedded system mappings which can be explained as:

”Create the flexible one-on-one mappings of a particular application Kahn PN onto: (1) multi-processor with compile-time pipelining, and (2) multi-processor with run-time pipelining of symbolic instructions [9], where: (1) the application process network is pre-created and cannot be changed, and (2) the granularity of computations and communications between the application model and the chosen architecture models differ significantly.”

The restriction on changing application process networks implies that the mapping transformations can happen only at the level of symbolic programs. The restriction on granularity enforces the Refining Step (see Section 4.4.3). Thus, in this sub-section we give (1) a description of the mapping case we conducted and (2) a refinement of the 2D-IDCT mapping model without modifying the high-level specification. It is worth noting that with this case we model ’fictive’ architectures, and due to this, we cannot reason about the accuracy of the case results as we did with the case in Section 4.5.1. Here, we can reason only about performance impacts of refinement and transformation choices on the simulated system in isolation, i.e, in the scope of simulation models, and which are already available in [75].

The Two Dimensional Inverse Discrete Cosine Transform (2D-IDCT) is part of image compression methods, one of which is a standard described in [81]. 2D-IDCT appears in many Multimedia applications and is a critical path function [50].

At some level of abstraction, the 2D-IDCT application is specified as a 3-process PN [18], as shown in Figure 4.5 (processes `Source` and `Sink` do not play any role here - they are illustrated for the sake of delivering input data and collecting output data).

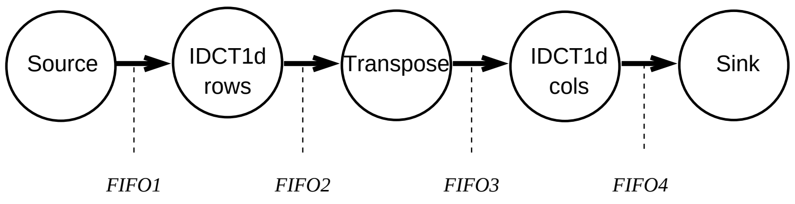


Figure 4.5: The 2D-IDCT Kahn Process Network

In this graph, `1D-IDCT` is the One Dimensional Inverse Discrete Cosine Transform, which transforms a time-domain block of 8×8 image pixels to a frequency-domain block of 8×8 values in a *row-by-row* fashion. `Transpose` performs the transpose of the output blocks of the first `1D-IDCT` and then delivers the transposed blocks to the second `1D-IDCT`. The second `1D-IDCT` then also applies row-by-row transformations, which, due to the transposition, corresponds to a *column-by-column* transformation on the output of the first `1D-IDCT`. The (unbounded) channels between the two producer-consumer pairs exchange these blocks.

[9]See Chapter 3, Sections 3.3.1, 3.5.1, and 3.5.2.

Symbolic Program Representation of the 2D-IDCT Khan PN

The listing in Figure 4.6 below, is a Symbol Program that represent all three processes in Figure 4.5.

```
main {
   loop condition 0 ($i_N$)
   {
      read  m ($i_0$, 64);
      execute  f ($i_0$ $o_0$, b);
      write  n ($o_0$, 64);
   }
}
```

Figure 4.6: Symbolic program template for both the IDCT1D and the Transpose tasks.

The numbers “64” and “b” indicate the token size in number of pixels read from (written to) the input (output) channel, and the execution budget, relative to the `read` and `write` execution budgets, of the function being executed by the process, respectively. The 1D-IDCT and the Transpose functions have different budgets[10]. As indicated earlier, each SP is associated with an accompanying control trace. From the structure of the SP in Figure 4.6, it can be seen that the corresponding control trace is trivial in this case because there is only one control point.

Architecture Specifications & Mapping Descriptions

We conducted two different experiments: (1) mapping of the 2D-IDCT specification onto a multiprocessor with *compile-time pipelining* processors, and (2) mapping of the 2D-IDCT specification onto a multiprocessor with *run-time pipelining* processors. In these experiments, we use the following architecture plus mapping specifications (for each mapping there is one architecture plus mapping specification):

1. **Binding:** We assumed a 1-on-1 mapping of SPs onto processors and application channels onto FIFO components.
2. **Binding:** There is no resource sharing, neither for computation (an operating system is not needed) nor for communication (a bus is not needed since all buffers are dedicated).
3. **Refining:** 1D-IDCT processors operate on rows (8-pixel data-token) rather than on blocks ($8 \times 8 = 64$ data-token). That is, the architecture `check-data` and `check-room` FIFO synchronization primitives operate on rows, and the `signal-room`, and `signal-data` FIFO synchronization primitives operate on blocks. Conversely, in the other tasks (Source, Transpose, Sink) the `check-data` and `check-room` synchronization primitives operate on blocks, and the `signal-room`, and `signal-data` synchronization primitives operate on rows.

[10]The execution budget parameter of the 1D-IDCT tasks is 8, and the execution budget parameter of the Transpose task is 1.

4. **Refining:** 1D-IDCT implementations in the processors are as in [82], and can be represented by a sequence of four different `execute` symbolic instructions.

5. **Matching:** FIFO buffers in the architecture are sized such that they provide enough space (in this case study, for the three mappings the FIFO buffer sizes are always 256 tokens).

Compile-time Transformation of Symbolic Programs

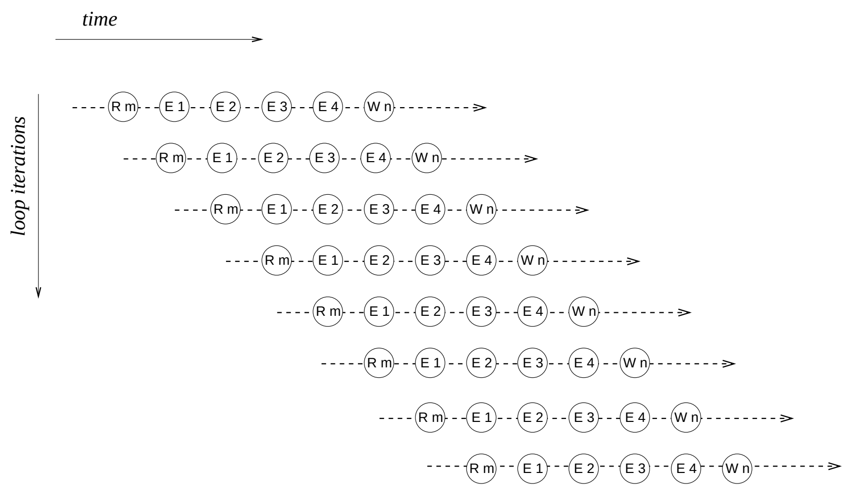


Figure 4.7: The transformed loop body of the IDCT1D SP

The idea of this transformation is to schedule the execution of the SP shown in Figure 4.6 as shown in Figure 4.7. After applying this transformation the SP template has changed and the resulting SP template is shown in Figure 4.8.

The transformation illustrated in Figures 4.7 and 4.8 is so-called "software pipelining", allowing overlapping of symbolic instructions at run-time [56]. Each symbolic instruction, delimited by the ";" terminal, may express parallelism (mutual independence) among symbolic operations delimited by the "||" terminal. This implies that no dependency checks (argument checking) in the architecture model are performed at run-time. Notice that mutually independent symbolic operations in an explicit parallel symbolic instruction need not have equal evaluation times. The next symbolic instruction is only scheduled when the slowest symbolic operation in the current symbolic instruction terminates.

Run-time Transformation of Symbolic Programs

Another way of modeling the behavior shown in Figure 4.7 is to detect at run-time a possible *overlapping* of `read`, `execute`, and `write` symbolic instructions. The appropriate processor model is provided in Chapter 3 Section 3.5.2. The transformation applied here is a simple refinement: an expansion of the loop-body in the SP template shown in Figure 4.6. The processor model, on the other hand, is now more complex because it has to produce at run-time the pipelined execution order (compared to the compile-time processor where the compiler is more involved). The result of the refinement of the SP is shown in Figure 4.9. It is worth noting that the execution flow of this SP is the same as in Figure 4.7.

```
main {
    loop condition 0 $i_N$
    {
        read  m ($i_0$, 8);

        read  m ($i_0$, 8) || execute $f_1$ ($i_0$ $I_1$, 2);

        read  m ($i_0$, 8) || execute $f_1$ ($i_0$ $I_1$, 2) || execute $f_2$ ($i_1$ $I_2$, 2);

        read  m ($i_0$, 8) || execute $f_1$ ($i_0$ $I_1$, 2) || execute $f_2$ ($i_1$ $I_2$, 2) ||
        execute $f_3$ ($i_2$ $I_3$, 2);

        read  m ($i_0$, 8) || execute $f_1$ ($i_0$ $I_1$, 2) || execute $f_2$ ($i_1$ $I_2$, 2) ||
        execute $f_3$ ($i_2$ $I_3$, 2) || execute $f_4$ ($i_3$ $o_0$, 2);

        read  m ($i_0$, 8) || execute $f_1$ ($i_0$ $I_1$, 2) || execute $f_2$ ($i_1$ $I_2$, 2) ||
        execute $f_3$ ($i_2$ $I_3$, 2) || execute $f_4$ ($i_3$ $o_0$, 2) || write $f_n$ ($o_0$, 8);

        read  m ($i_0$, 8) || execute $f_1$ ($i_0$ $I_1$, 2) || execute $f_2$ ($i_1$ $I_2$, 2) ||
        execute $f_3$ ($i_2$ $I_3$, 2) || execute $f_4$ ($i_3$ $o_0$, 2) || write $f_n$ ($o_0$, 8);

        read  m ($i_0$, 8) || execute $f_1$ ($i_0$ $I_1$, 2) || execute $f_2$ ($i_1$ $I_2$, 2) ||
        execute $f_3$ ($i_2$ $I_3$, 2) || execute $f_4$ ($i_3$ $o_0$, 2) || write $f_n$ ($o_0$, 8);

        execute $f_1$ ($i_0$ $I_1$, 2) || execute $f_2$ ($i_1$ $I_2$, 2) ||  execute $f_3$ ($i_2$ $I_3$, 2) ||
        execute $f_4$ ($i_3$ $o_0$, 2) || write $f_n$ ($o_0$, 8);

        execute $f_2$ ($i_1$ $I_2$, 2) || execute $f_3$ ($i_2$ $I_3$, 2) || execute $f_4$ ($i_3$ $o_0$, 2) ||
        write $f_n$ ($o_0$, 8);

        execute $f_3$ ($i_2$ $I_3$, 2) || execute $f_4$ ($i_3$ $o_0$, 2) || write $f_n$ ($o_0$, 8);

        execute $f_4$ ($i_3$ $o_0$, 2) || write $f_n$ ($o_0$, 8);

        write $f_n$ ($o_0$, 8);
  }
}
```

Figure 4.8: Unrolled and pipelined symbolic program template.

```
main {
   loop condition 0 ($i_N$)      // the original loop
   {
      loop condition 1 ($i_M$)   // the newly introduced loop
      {
         read  m ($i_0$, 8);      // the refined read (lines)

         execute $f_1$ ($i_0$ $I_1$, 2); // the first pipeline stage
         execute $f_2$ ($i_1$ $I_2$, 2); // the second pipeline stage
         execute $f_3$ ($i_2$ $I_3$, 2); // the third pipeline stage
         execute $f_4$ ($i_3$ $o_0$, 2); // the fourth pipeline stage

         write  n ($o_0$, 8);     // the refined write (lines)
       }
   }
}
```

Figure 4.9: Unrolled symbolic program template.

4.5.3 Case-study: JPEG Decoding Network on MPSoC

The objective of this case is to model embedded system mappings which can be explained as:

”Create the acceptable mapping models of (1) a realistic application onto (2) a complex and challenging distributed shared memory architecture with and without operating system included [11], where the mapping excluding operating system is done in one-to-one fashion similarly as in the earlier cases, and the mapping including operating system is done in many-on-many fashion, and where the communication mechanism in the modeled architecture can modify the mechanisms available in the architecture component library.”

The *acceptable* accuracy means that the maximal difference between our simulation numbers and the numbers given to us as the real architecture performance, has to be within $\pm$20%. The idea is to have a realistic application, which is dynamic and rich with dependencies (see Chapter 2, Section 2.3) mapped onto a model of a realistic high-end MPSoC architecture [12]. Our choice of the realistic is a JPEG decoder. JPEG is an acronym for Joint Picture Experts Group [50]. The architecture modeled here is called Wasabi architecture [83].

For our JPEG-on-`Wasabi` study, we identify two sub-cases: in the first case, we exclusively use hardware accelerator resources (all-in-hardware case); for the other, we rely only on software core resources (all-in-software case). The rationale behind this choice is the following:

- We need to accurately model the `Wasabi` communication network at system-level and to do that we create a one-on-one all-in-hardware mapping case because it will help us to translate the low-level `Wasabi` specifics into the high-level manifestations which we can model with our architecture components.
- We need to accurately model the `Wasabi` processors (software cores and hardware accelerators) at a high level of abstraction and to do that we create a **calibration** sub-case - Producer-Consumer, which will expose the irregular behaviors of the `Wasabi`

[11] See Chapter 3, Sections 3.5.3.

[12] By *realistic architecture* we mean the ILP-level simulated high-end multiprocessor model.

processors. Later, using this sub-case we will extrapolate additional high-level manifestations from these low-level irregularities.

- We need to model a symmetrical shared memory multiprocessor and to do so we will re-use previously estimated parameters of the `Wasabi` processor and communication network and we will model the operating system based solution, a so-called many-on-many all-in-software mapping case.

As a first step, we gather a proper `Wasabi` parameter which is set to be suitable for our architecture models. Moreover, we investigate some details of the actual architecture behavior that may be overlooked in our models.

As mentioned earlier, in order to capture the system-level parameters and the behavior of the communication network in `Wasabi`, we run the all-in-hardware case. This relies on the fact that hardware accelerators behave according to user-annotated delays. Such delay-annotations relate the computation load of the JPEG application and can be roughly estimated from the C/C++ code. In this way, both `Wasabi` and our models experience the same computation load. From the `Wasabi` related documentation and `space-Cake`[13] Instruction Set Simulation (ISS) we can estimate the communication network parameters. This can be regarded as the Matching Step for the communication network.

Only at this point do we proceed with the all-in-software case. By means of previously estimated network parameters, we calibrate the computation load. In this way, we identify the model parameters and the behaviors related to MIPS-es and the Operating System.

We expected that the default communication network behavior captured by our model would not be accurate enough to predict the `Wasabi` communication network behavior. For this reason we also have conducted a simple Producer-Consumer experiment to expose and refine the actual `Wasabi` communication network behavior.

The remainder of the section is organized as follows: firstly we present the JPEG decoder model; secondly, we set out the `Wasabi` block diagram. Then we study the Producer-Consumer sub-case. Finally, we summarize results and indicate the pros and cons of the model which is highlighted in this case.

The JPEG Application Specification

We reuse the JPEG decoder specification already introduced in [58]. The application is modeled using Kahn Process Networks. A graphical representation of this process network is given in Figure 4.10.

Big bobbles with text labels represent processes (e.g., DMX stands for de-multiplexer, VLD stands for variable length decoder, etc.). Little white and black bobbles with numbers represent ports (e.g., o1 stands for output port 1, while i5 stands for input port 5, etc.); rectangles with 3 edge boundaries represent unbounded FIFO channels; text labels specify types of tokens being transferred through these FIFOs, and arrows indicate data-flow direction.

[13] `space-Cake` is going to be introduced later in this section.

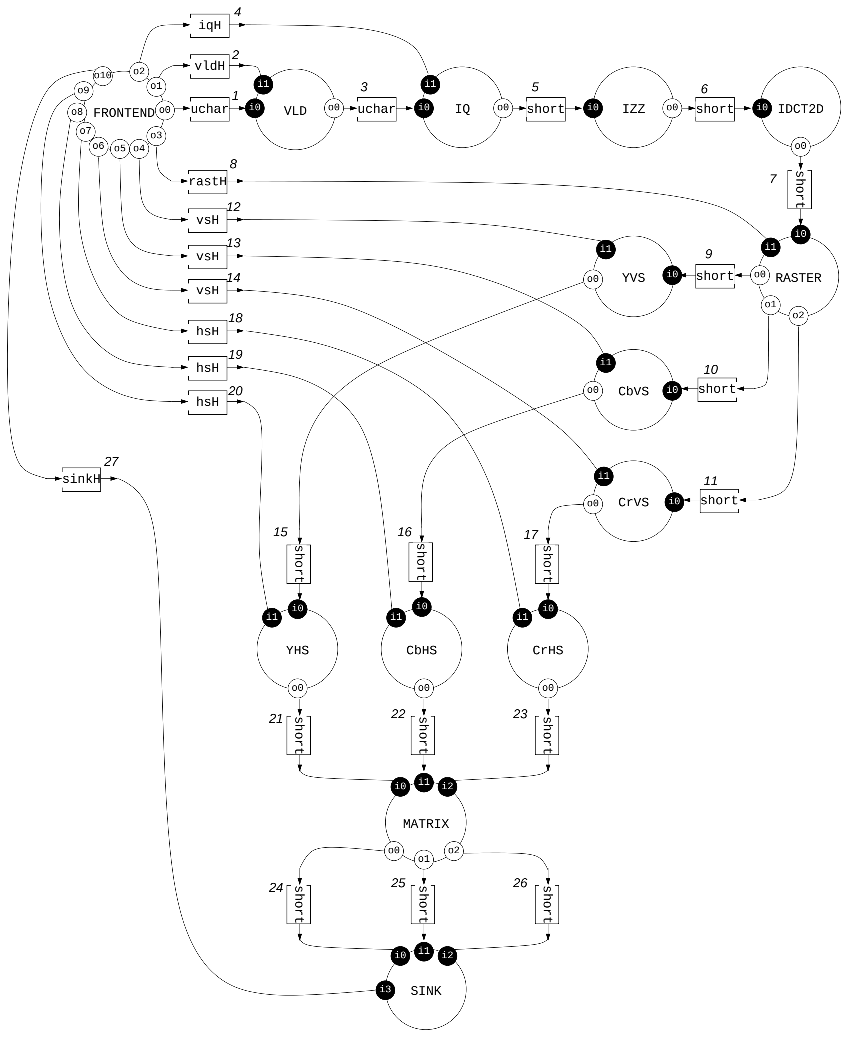


Figure 4.10: The JPEG decoder process network

Note that each FIFO is enumerated with an integer number. Consequently, data streams through FIFOs 1, 3, 5, 6, 7 (in sequence), then 9, 10, 11 (in parallel), then 15, 16, 17, (in parallel), then 21, 22, 23, (in parallel), and finally, 24, 25, 26 (in parallel). The remaining FIFOs serve to provide once-per-picture parameter initializations through so-called ”headers”. It is worth noting that these headers are not fixed, but rather are derived by the Frontend process. That is, the network is not tuned for processing a particular JPEG image format. In the signal processing community this is also known as ”parametrized data-flow modeling” [84].

The `spaceCake` - `Wasabi` Block Diagram

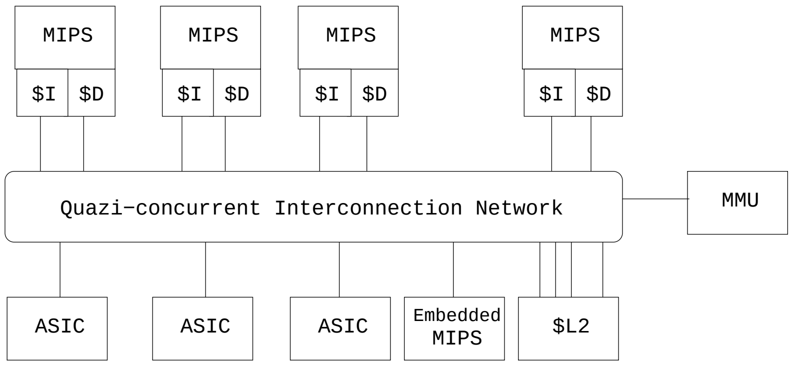


Figure 4.11: Block diagram of the Wasabi MPSoC architecture

Our primary objective is to model the `Wasabi` architecture as a target architecture. The `Wasabi` multiprocessor system on a chip (MPSoC) is designed in the Philips Research Laboratories (for more data refer to [83]). This MPSoC represents a single tile of a more complex system, called `space-Cake`, see [67]. This is the reason both names appear when we discuss the modeled architecture.

The `Wasabi` tile is a MPSoC consisting of: a number of programmable MIPS processor cores with an integrated L1 Harvard cache (i.e., separated instruction and data caches), a snooping bus-based interconnection network (ICN) which connects the cores and the L2 cache, and a memory-management unit (MMU) interface to the off-chip DDR [14] memory. Apart of MIPS-es, dedicated processor cores (or accelerators) may also be used in Wasabi. Thus, this architecture is heterogeneous, and suitable for multimedia and video processing applications such as JPEG compression.

The block based architecture view is given in Figure 4.11. Note that the ICN in Figure 4.11 provides a certain level of concurrency, which is very interesting for our modeling environment. Also, note the Embedded MIPS block in the same figure. It serves as a software synchronization core in situations where hardware accelerators have been employed. This block should not be overlooked, since it is a source of unpredictability, given the amount of interaction it has with the caches.

Producer-Consumer Calibration Sub-case

Figure 4.12 illustrates the case when our default communication network behavior [15] is used to model the `spaceCake` `Wasabi` communication network. The parameters are set according to the detailed `Wasabi` description [83]. The *x*-axis represents the total amount of tokens

[14]*DDR* - Double Data Rate - is a synchronous dynamic random access memory technology used for high speed storage of the working data of a computer or other digital electronic device.

[15]The *default* behavior is described in Section 4.5.1.

being exchanged via the FIFO channel. The *y*-axis reports processor component cycles. Such data have been measured by executing the Producer-Consumer network [16].

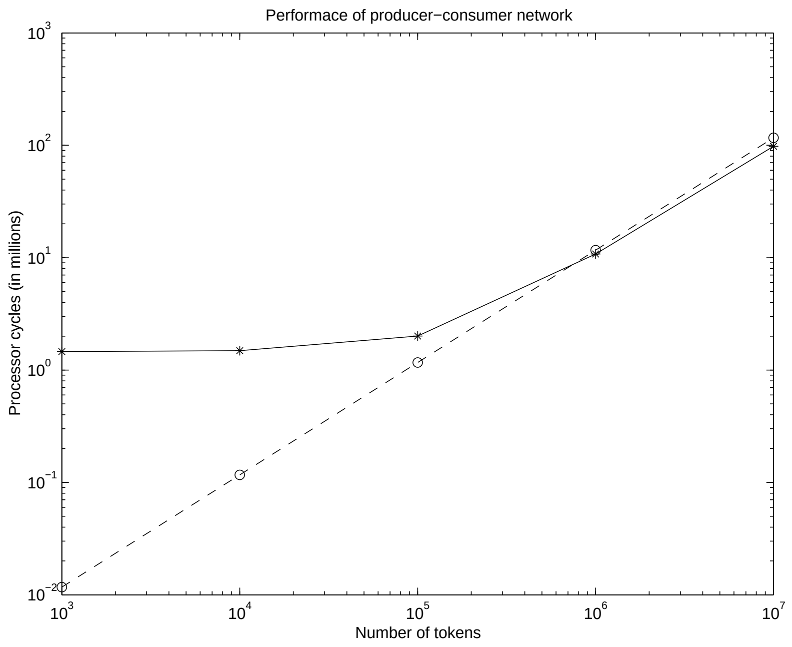


Figure 4.12: The Producer-Consumer performance numbers: `Wasabi` (solid) and our not-calibrated model of `Wasabi` (dashed)

The results show that our architecture model scales linearly with the total amount of the tokens being exchanged via the FIFO channel. This is clearly not the case for the `Wasabi` MPSoC.

After interviews with designers, we have reached the following conclusions (not immediately visible from [83]):

1. When used in the all-in-hardware mode, `Wasabi` engages Embedded MIPS to synchronize hardware accelerators. Its behavior varies and cannot be deterministically described (i.e. a statistical model is necessary),

2. When accelerators access the network (ICN) for writing, they buffer as many requests as possible. This became clear when we run different instances of PC application, with the producer having different write patterns: token-after-token, 10-tokens-after-10-tokens, and 100-tokens-after-100-tokens. The resulting performance numbers did not change.

3. Whenever accelerators access the network (ICN) for reading, they always take full

[16]The network given as the application model in Figure 4.1.

cache-lines (128 bytes), even when less than 128 bytes are available. In such cases, dummy data is used to fill up the cache line.

4. The network setup time should not be overlooked, since it takes up to several millions of PU cycles (depending on the application network size). This actually explains the horizontal part of the `Wasabi` characteristic shown in Figure 4.12. The network setup time can be measured when the source node (producer) does not send any data token to the destination node (consumer) - e.g., an unconnected FIFO.

5. All synchronization operations (*check-room/data*, *signal-data/room*) are executed by the Embedded MIPS. Rather than simply blocking (as we assumed in our architecture model), the synchronization operations take extra cycles.

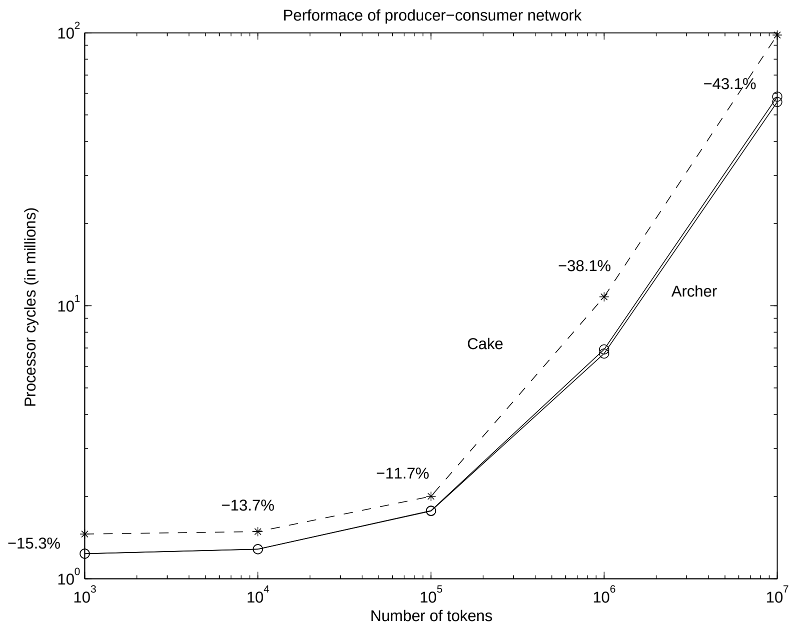


Figure 4.13: The Producer-Consumer performance numbers: `Wasabi` (dashed) and our calibrated model (solid)

The outcome of this calibration case was that our basic modeling approach remains the same - no compromise on that. Only new states have been added to basic modules of components from our architecture library. Thus, considering the insights listed above, we enriched the models of our interface components (FIFO In and Out CFSMs), arbiter's bus-related operations and FIFO synchronization operations (see Chapter 3, Section 3.5.6 and Appendix A, Figures A.18 and A.19). Finally, we added a setup delay in the bus-claim operation. This delay is now executed the first time a FIFO interface accesses the bus. During the setup phase no units can operate over the network. Thus, all units simply block and wait for the expiration of the setup delay.

The simulation results of the calibrated model are shown in Figure 4.13. The *x* and *y* axes maintain the same meaning as before. Note that unlike in Figure 4.12, the absence now of the 'setup time' gap makes the error seem bigger for larger numbers of tokens. However, this is not true, since both the *x* and *y* axes are logarithmic. It is sufficient to compare the deviation for a million cycles in Figure 4.12 and Figure 4.13. One can realize that the degree of error does not significantly change.

Architecture Specification & Mapping Description for All-in-Hardware

The architecture plus mapping specifications for the all-in-hardware experiment are established based on the calibration sub-case. Here follows the brief description of these specifications (for each mapping there is one architecture plus mapping specification):

1. **Binding:** We assumed a 1-on-1 mapping of SPs and unbounded channels of the JPEG application specification (see Figure 4.10) onto processors and FIFO components of the proper `Wasabi` architecture instance (similar to the one in Figure 4.11), respectively. This implies that the number of processor components in the architecture specification is equal to the number of processes in the application specification (i.e., 14), and that the number of FIFO components in the architecture specification is equal to the number of unbounded channels in the application specification (i.e., 27).

2. **Binding:** Computation resources are not shared, but the communication resources are shared. As determined by the calibrated case, the global FIFO memory buffers are accessed through a *burst-bus*. Consequently, the routing interfaces are modeled according to the example in Section 3.5.6.

3. **Matching:** FIFO buffers in the architecture are sized such that they provide enough space (in this case study, for the mappings shown in Figure 4.14 the FIFO buffer sizes are always a maximum 65536 tokens, where token equals one byte, which also matches the architecture specs given in [67, 83]).

4. **Matching:** The conditional synchronization protocol operations for the FIFO buffers (`check-room`/`check-data` and `signal-data`/`signal-room`) are 'executed in software' by Embedded MIPS. This creates a performance impact since these operations cannot run in parallel. Therefore, their cost is not negligible and they have to be explicitly modeled. (Based on the calibration case and also for the mappings shown in Figure 4.14, we estimated that the `check-room`/`check-data` operations consume 100 nanoseconds and that the `signal-data`/`signal-room` operations consume 50 nanoseconds).

5. **Refining:** The calibration sub-case shows that data-transfers happen through the burst-bus highway. The *burst* means that the data-transfers are packetized to utilize the bus. The *bus* means that the interconnection lines are shared. The *highway* means that a number of parallel data-transfers over interconnection lines are possible.

6. **Refining:** The simultaneous memory requests are also transferred over a bus. These requests are conveyed to Embedded MIPS, so they will ultimately be resolved as synchronization operations running in software.

7. **Matching:** The burst-line length is as in [67, 83] (i.e., 128 bytes), and the network setup time is as determined in the calibration sub-case (i.e., 12957510 nanoseconds). The depth of simultaneous memory requests is one, which means there is only one Embedded MIPS able to queue and process these requests.

8. **Matching:** The cost of the single read, execute, and write budgets (e.g, expressed in form of SIs: `read` M `(input,1)`, `execute` N `(input output,1)`, and `write` P `(ouput,1)`) for each accelerator is as determined in the calibration sub-case (2500 picoseconds).

9. **Refining:** For the read routing interface components, there is a significant switching delay due to the buffering of consecutive `read`s in order to utilize traffic over a burst-bus. This is not the case for the write routing interface components, so there is no significant switching cost there. Hence, the architecture and mapping specifications for the routing interface components are asymmetric in the sense of the aforementioned.

10. **Matching:** The cost of the read routing component switching is as determined in the calibration sub-case (9999 picoseconds). The costs of the `load-data` and `store-data` transfer operations when writing-to and reading-from FIFO components is determined as the same for both read and write routing interface components (209 picoseconds).

Results for All-in-Hardware

Once the architecture specification is sufficiently accurate to model the `Wasabi` communication behavior, the all-in-hardware modeling, mapping and simulation can be performed. The results are provided in Figures 4.14 and 4.15. The first chart reports the performance numbers for a set of eight JPEG images of various size. The *x* axis lists these images. The first row below the axis gives the amounts of raw input data for each image. The second row provides the spatial resolution (number of pixels) generated at the output. The third row simply lists the JPEG file names. The *y* axis represents the number of processor component cycles in millions, where each cycle corresponds to the previously estimated budget of the single `read`, `execute`, and `write`. In Figure 4.14 the left-side bars correspond to the default `Archer` [17] communication behavior, the middle bars correspond to the `spaceCake Wasabi` all-in-hardware execution, and the right-side bars correspond to the calibrated model of the communication behavior. Similarly, in Figure 4.15 the left-side bars correspond to the default model communication behavior, while the right-side bars correspond to the calibrated model communication behavior.

From Figure 4.14 one can see the improvement gained by the calibration. However, the amount of error is noticeable. This is illustrated in Figure 4.15. The cause is the unpredictable behavior of the Embedded MIPS, which loads and stores semaphores used for the synchronization in a manner not supported by the `Archer` models yet. However, one should also notice that the error slope is significantly reduced after the calibration (see Figure 4.15). The images are ordered with respect of the spatial resolution - so if an imaginary line is

[17] `Archer` is the name used for the authentic models and methods described in this thesis - see Chapter 5.

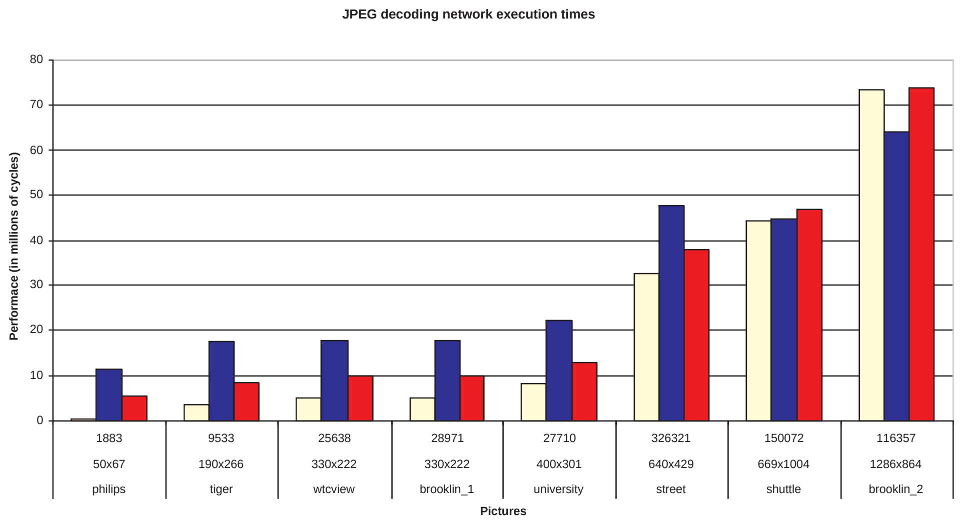


Figure 4.14: Performance of the JPEG decoder network: our model without calibration (left-side bars), `Wasabi` (middle bars) and our model with calibration (right-side bars)

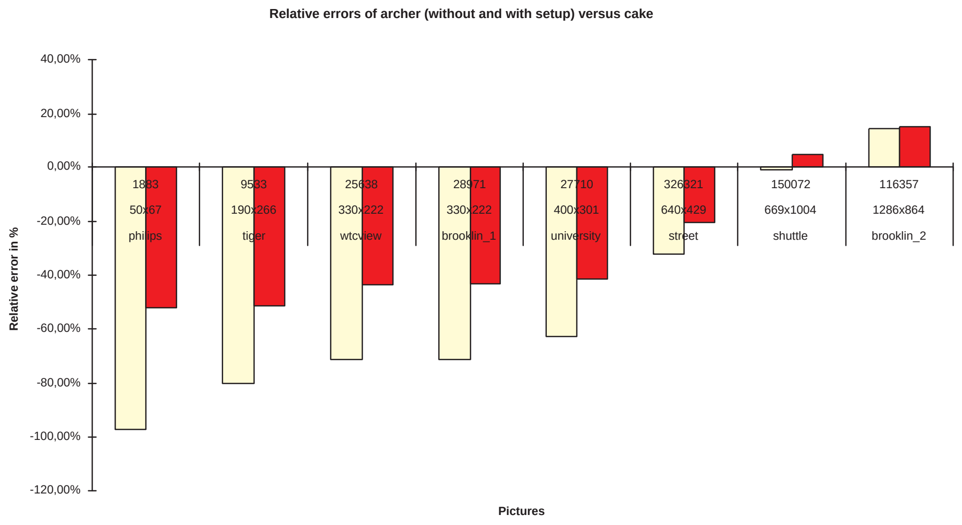


Figure 4.15: Simulation relative error: our model without calibration (left-side bars), and our model with calibration (right-side bars)

drawn over the top of the bars, the slope in the first case (non-calibrated) is steeper than in the last case (calibrated). Additionally, for the lower traffic volumes, it is obvious that the setup time parameter plays the main role. However, for the higher traffic volumes, the effect of this parameter disappears, and other effects (see *Producer-Consumer Calibration Sub-case*) start to compensate calibrated behavior towards the real `Wasabi` numbers. The conclusion is that the Embedded MIPS, which in `Wasabi` executes all synchronization operations for the all-in-hardware, introduces unpredictable behavior.

The All-in-Software Subcase

In this paragraph we proceed with the all-in-software case. The all-in-hardware case showed that `Wasabi` is a highly unpredictable architecture due to the specific caching and synchronization mechanisms. It should not come as a surprise that the all-in-software case experiences unpredictability, as a consequence of the operating system usage in the architecture, i.e., resource-sharing, scheduling and preemption of tasks ontop of the programmable homogeneous configuration of the `Wasabi` multiprocessor. One can argue whether or not `Wasabi` is a domain-specific embedded architecture or a GPP.

Using the results of the preceding case, we can state that the `Wasabi` communication network parameters are determined with an accuracy of about -50% for small JPEG pictures (on the left hand side of bar-charts Figures 4.14 and 4.15) and about +15% for large JPEG pictures (on the right hand side of bar-charts Figures 4.14 and 4.15). From the perspective of cycle-accurate simulations and synthesis-driven methodologies, this error is probably unacceptable. However, it is fair to say that 50% off the expectations in this case also comes from the fact that we modeled an existing architecture which did **not** emerge from system-level simulations and exploration-driven methodologies, but rather from a guru approach (see Section 1.2.2). On the contrary, in the cases where the underlying architecture has not been *finalized* yet, we believe that our models can lead to successful designs. The architecture that would result from DSE would be a composition of library components that do not need calibration and have much improved accuracy. This is the case shown in Section 4.5.1, where the architecture and mapping came as a result of DSE.

Here, we do not repeat the complete example from the all-in-hardware case. Rather than evaluating the case for the eight different JPEG pictures, we focus only on the worst case determined in the all-in-hardware case - the small JPEG picture (the picture data: 1. name *philips-logo.jpg*, 2. size on the file-system 1883 bytes, 3. resolution 50×67, 4. Y:C_b:C_r is 4:1:1). The architecture specification and mapping set-up are as follows:

1. The multiprocessor is now modeled differently as compared to the all-in-hardware mapping model. It is not necessary for the number of processor components to equal the number of processes in the application specification.
2. The network model is copied from the all-in-hardware mapping model. The number of FIFO components equals the number of unbounded application channels, as well. The behavior of the Network-on-chip (NoC) is almost the same as in the all-in-hardware mapping model, except for the following two differences: (1) the all-in-software mappings have increased the throughput for the simultaneous memory requests (i.e., it goes

from one-request-per-time in the all-in-hardware case, to the number-of-requests-per-time, where the *number* is the number of MIPS cores in the multiprocessor), and (2) the all-in-software mappings reduce network start-up delay proportionally to the number of MIPS cores in the multiprocessor.

3. The processor components of the MPSoC model are homogeneous and allow for *migration* of the execution threads of the mapped symbolic programs. This originates from the fact that the MIPS-es in `Wasabi` support migrations of the mapped application processes.

4. The control delays and function-call delays now have non-zero component parameter values because the symbolic programs are mapped to software, and thus processor instructions and data caches play important roles.

5. The operating system parameters, such as: migration delay [18], scheduling delay [19], interrupt delay [20], and context-switching delay [21] (see Chapter 3, Section 3.4.2) may be required [22]. It is worth noting that these delays are now at the level of tens or hundreds of nanoseconds, which illustrates the fact that instead of accelerators (all-in-hardware), we have relied on the many-on-many processor component. Similarly, nanoseconds are expected delays caused by the SI constructs: `read`, `write`, and `execute`, as well as `function call`, `condition`, etc.

6. When `Wasabi` is configured for all-in-software mappings, the synchronization operations (*check-room/data*, *signal-data/room*) are executed by the homogeneous MIPS-es, since there is no independent Embedded MIPS involved. Thus, there is no need for extra cost cycles for the synchronization operations.

The results of this sub-case are shown in Figure 4.16. It provides performance numbers in millions of cycles for the chosen JPEG image. The *x* axis lists the number of MIPS-es in the MPSoC. The *y* axis represents the number of processor component cycles in millions (as in the all-in-hardware sub-case). JPEG-Cake stands for the `spaceCake` `Wasabi` all-in-software execution. JPEG-1 and JPEG-2 correspond to two different architecture choices for our modeling of the `Wasabi` architecture.

As already mentioned, we simulated two mappings: the first one (JPEG-1) has inherited the NoC settings from the all-in-hardware case, and the second one (JPEG-2) has been calibrated regarding the set-up time (see the text above). Obviously, our model is able to simulate the trend of the MPSoC behavior, but the irregularities (or unpredictabilities) caused by the cache-plus-operating-system combination definitely affect the accuracy of the simulation. That is, the simulation errors in the JPEG-1 mapping case are definitely affected by the fluctuation in the results of the all-in-hardware case: in-between -50% and +20%. Based on these performance measures, the immediate assumption would be that the most optimistic

[18]A *migration delay* is the time needed to migrate a context of one symbolic program execution from one processor component to the other.

[19]A *scheduling delay* defines the time period between two consecutive OS scheduler invocations.

[20]An *interrupt delay* is the time needed to run an Interrupt Service Routine (ISR).

[21]A *context-switching delay* is the time needed to deliver the next symbolic instruction to the processor-component core.

[22]It is possible to ignore them as well, i.e., to assign zero-values.

error-margin (probably unrealistic) for the all-in-software case would be the same as or even larger (a more realistic assumption) than the error-margin for the all-in-hardware case.

The JPEG-2 mapping sub-case illustrates how calibration can improve architecture parameters such as the network set-up time. By reducing the set-up time by 40% (which is the error-margin determined for small JPEG pictures) we can approach the performance numbers of the cycle-accurate results (JPEG-Cake). Of course, this should be an iterative process, aimed at matching the performance numbers of the cycle-accurate simulation to the performance numbers of our model. After the desired accuracy is achieved [23] the case should be reproduced for all other data-sets.

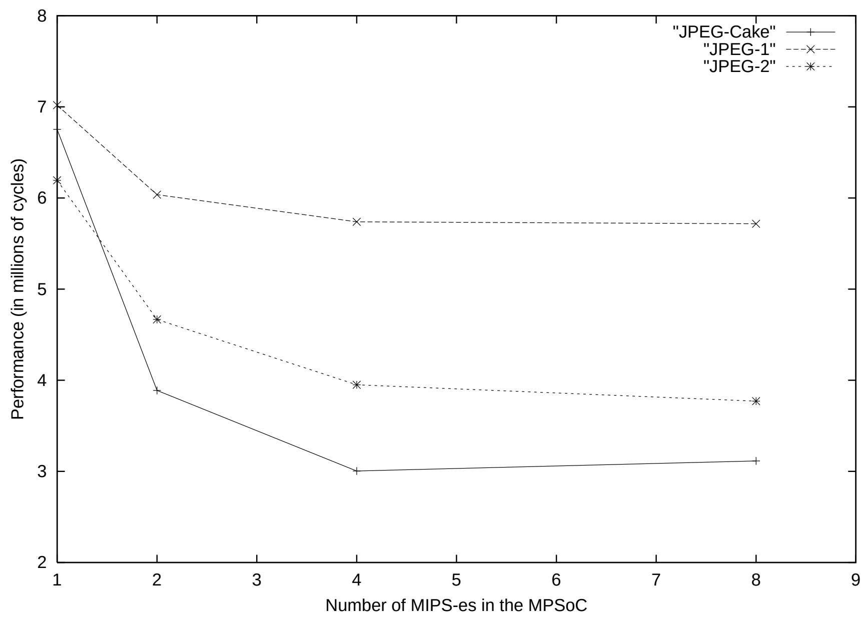


Figure 4.16: Performance of the all-in-software JPEG decoding network: (*JPEG-Cake*) the cycle-accurate execution model of the `Wasabi`, (*JPEG-1*) our model without-any-calibration, and (*JPEG-2*) our model with the initial calibration.

At this point we do not experiment further with this case, leaving additional all-in-software calibrations as part of future work. In this thesis, because we are aware of architecture model restrictions (such as non-supported cache modeling, etc.) we were more interested to show mapping capabilities . As a part of the future work, the repository of architecture components or the components themselves should be enlarged and enriched for memory-management and cache-models. With such enhancements we could definitely achieve highly accurate results for the mappings onto the resource-shared and resource-managed platforms. At this moment such platforms can only be modeled, which is nevertheless a remarkable result which other

[23] What the *desired accuracy* means is design-space exploration specific and it may differ from application to application.

exploration methodologies are unable to offer [36].

4.6 Contribution

The intention of this section is to underline the differences between our exploration method and other closely related exploration methods [7, 36], and at the same time, to point point out the added advantages of our mapping method.

Our method is a simulation based exploration method, and as such it relies on: i) high-level and abstract models (representations) of both applications and architectures, ii) a repository of non-functional architecture components, iii) discrete-event component implementations (e.g., `SystemC`) instead of cycle-accurate component implementations, and, finally, iv) the exploration-aimed Y-chart [14] (see Chapter 1, Section 1.2.2).

However, amongst other simulation based exploration methods mentioned earlier [7, 36], this is also a very distinctive method . The three major differences are:

1. Rather than using so-called trace-driven approach where an application process behavior is captured by the symbolic instruction trace, our method captures an application process behavior using a generic CDFG-like representation - symbolic program - and the corresponding sequential control trace.

2. Rather than relying on heuristic solutions for modeling OS [7], our method is able to model OS without any model-imposed restrictions.

3. As opposed to restricting the possible application domain or architecture set to only control-dependency-free applications with rudimentary or no multi-tasking software architectures at all [36], our method is not restricted in this way.

Our application model is able to separate concerns clearly: i) the data-set insensitive information is captured in symbolic programs, and ii) the data-set sensitive information is captured in sequential control traces. Changing the data-set changes only the control trace values. Yet, the symbolic programs do not need to change because they are data-set insensitive. The same single symbolic program can be reused for any data-set as long as it is produced by the application process from which the symbolic program originated. Remember that the annotated control points in the process are in 1-to-1 relation with control points within the symbolic program (see Chapter 2, Section 2.4). It is possible to establish the generality and reuse of the same single mapping for as many data-sets as desired (as long as the control structure is not changed).

Our architecture model is based on a library of generic, high-level, abstract architectural components. They represent a mix of various formalisms (Communicating Sequential Processes, Kahn PN, Synchronous Data-Flow, State-Charts, and ROOM-charts), ideas (Master-Slave Protocols, Trace-Transformations, Transactional Protocols), technologies (Data-Structures & Algorithms, Compilers & Translators, Design-Patterns, Parallel Programming by Multi-threading, Object-Orientation, and Component-Based Development), and implementations

(Open-Sources, Library-Reuse, and Discrete-Event Simulation). If, for instance, mutual-exclusion or condition-synchronization is needed, the appropriate formalisms and technologies inside or in between those components support the proper architecture model specification because the components are not CSP-only or SDF-only or Kahn-only - they are hybrids. Furthermore, the roles of each component are orthogonal to the role of the other components. Thus, computation-related mapping affects one kind of component, while communication-related mapping affects the other kind of component. Finally, components are easily extensible because, as explained earlier, they are able to clearly separate computation-only, communication-only, resource-sharing and resource-schedule concerns. Therefore, each enhancement can easily be added without affecting the entire repository of components.

The architecture components interpret (unroll) and refine mapped symbolic programs at run-time, making the modeling context close to the real architecture context [24]. Performing unrolling and refinement at run-time is particularly beneficial for the modeling of real system software . For run-time refinement, various synchronization issues remain open, and since they are open, they need to be closed as well. If we were to remove synchronization from the architecture, the model becomes poorer (or restricted) and, worse still, the omission of this significant component is simply overlooked. Moreover, synchronization is in general a costly operation from the performance point of view, and hence we do not exclude it from the architecture model.

Our method resembles the standard synthesis-driven mappings quite closely (see Section 4.1). Even-more, one can identify similarities with real-designs or even recode our abstract mappings to the real (synthesizable) code. As such, our method is both iterative and waterfall-like [61]. It is not able to "prune exploration space efficiently" as for instance the method described in [36] does. Yet, it connects well with real designs; as described above there are almost no restrictions when dealing with streaming application/architecture combinations and it supports reuse to a large extent. Therefore, we believe that it is a better (more complete) candidate for efficient architecture exploration than for instance the methods [7, 36]. The reason has its roots in the product-development of today's consumer-electronics industry; embedded designers are asked often to provide quick rough estimates of some real application standard (e.g., JPEG, MPEG-2, MPEG-4, etc.) running on some high-end MPSoC architecture (e.g., `Wasabi`-like MPSoC). In such cases, an embedded system designer usually performs an example-case which represents the implementation with the most challenging application specification and maps it on the architecture exploration simulator of the designer's choice. Now, if the simulator is too low level, this trial would require an unjustifiably greater expenditure of time and effort. This fact justifies the use of high-level abstract exploration methods such ours or any other similar [7, 36]. Nevertheless, if the exploration method cannot handle/model neither the application specification nor the architecture specification, then the method will simply be avoided. Moreover, if a designer can establish some correlation between: 1) the mapping steps he is asked to execute in the exploration method and 2) the mapping steps he has to perform in the real synthesis process, he will be in favor of this type of high-level exploration method. Our method allows for all the above:

- It is not selective with respect to mapping cases, neither from the aspect of the applica-

[24]`Sesame`, for example, really decouples trace refinement from the architecture, making the mapping layer responsible for the modeling of a system software

tion behavior nor from the aspect of the architecture features.

- It resembles the real synthesis flow, as in [85] as well as the real code transformations from [56], [86] and [55].
- It is easily extensible, since its components are orthogonalized - changing one component will not affect the sequencing in the others.
- It relies on discrete-event simulation, so it is fast enough (e.g., compared to cycle-accurate approaches).
- It can be easily further automated in two directions: by better pruning of design-space by generating Pareto curves [37], and by introducing an appropriate memory-hierarchy model (cache modeling).These two improvements will not affect its current robustness, generality and re-usability.

Chapter 5

Big Picture & Conclusion

What concerns me is not the way things are, but rather the way people think things are. [1]

5.1 Summary

The main aim of this chapter is to provide the complete view of relations between models, model representations and simulations for performance analysis as part of a Design Space Exploration (DSE) process.

Models and model representations are on a level of abstraction where flexibility, accuracy and cost of simulations are well balanced, see Figure 1.1. As shown in Figure 1.2, performance analysis is conducted on the association with each other of an application model and an architecture model. This association is in terms of model representations generated by the application model and interpreted by the architecture model, as well as in terms of transformations to better match the two representations.

Our application model representations are Symbolic Programs (SP), see Chapter 2, and our SP interpreting architecture model is the `Archer` [2] model, see Chapter 3. We call the performance analysis approach the `Archer` Symbolic Program approach, or simply the *SP approach*.

[1]The words of Epictetus - $E\pi\iota\kappa\tau\eta\tau\o\varsigma$ - (55 A.D.-ca.135 A.D.), Greek philosopher associated with the Stoics.
[2]The `Archer` stands for ARCHitecture ExploRation.

5.2 Big Picture

The Symbolic Program approach, which we introduced in this thesis, is positioned between the Symbolic Instruction Trace (SIT) approach, as in the `Spade` [7] and the `Sesame` [36] exploration driven approaches and the Control Data-Flow Graph (CDFG) approach, as in the `MTG-DF*` [25, 38] design-driven approach. The SP approach and its positioning has been shown in Figure 5.1. On the left-hand side, the typical sequence of activities using SIT is depicted. Similarly, on the right-hand side, the typical flow of activities using CDFG is shown. Finally, in the middle, the SP flow is shown; SPs allow designers (1) to perform design-steps as in the case of detailed design (indicated with dashed lines), (2) to run fast simulations of architectures being explored, and (3) to have more accurate numbers than in the case of TD simulations.

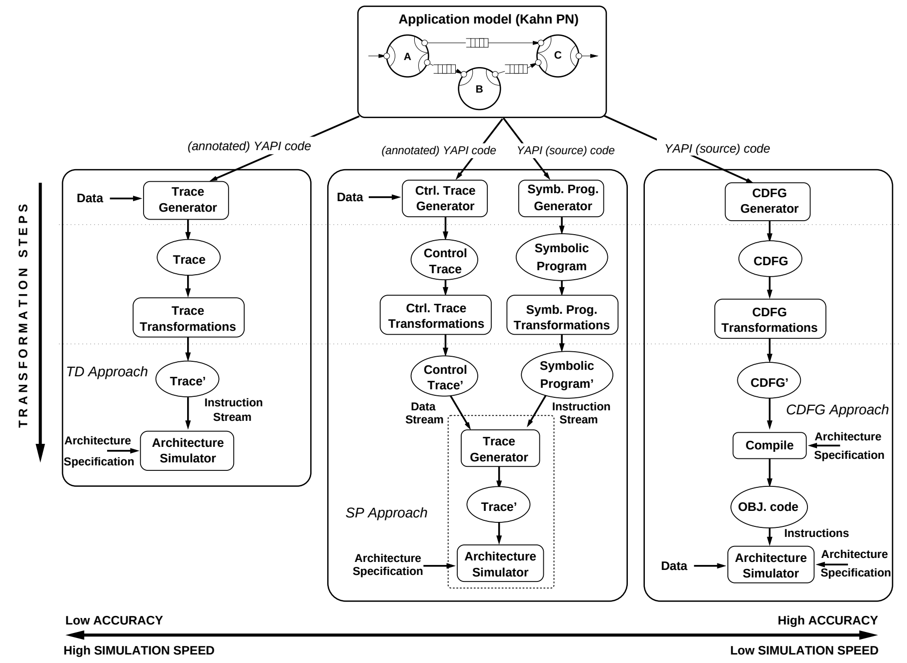


Figure 5.1: The Symbolic Program approach vs. SIT & CDFG approaches (repeated Figure 2.1).

Applications are modeled using Kahn's MoC. Process behaviors are captured in Symbolic Instruction Traces (left), Symbolic Programs plus separated Control Traces (center), or Control Data-Flow Graphs (right). The SP approach allows for a performance analysis with accepted as well as simulation speed, whereas the other two approaches either have low accuracy or low simulation speed, respectively. The SP approach is, thus, a hybrid approach. Applications processes are abstracted by means of (1) generic non-executable CDFG-like representations of a process and (2) a symbolic instruction trace that captures conditional construct outcomes that are the result of the execution of a single data-set. The behavioral models of architecture components are derived from SI traces which in turn are determined by interpreting transformed application SPs in the light of control outcomes which are derived from control traces. It is worth noting here that the SP representations are repeating separation of concerns between control-traces and symbolic program data-structures. In Figure 5.1 these are marked as *Data Stream* (data-dependent part) and *Instruction Stream* (data-independent part). This is exactly the reason why `Archer` can reuse application representations in mappings without changing underlying architecture specifications and why e.g. `Sesame` cannot guarantee the same. Moreover, this also ends-up in many other flexibilities for architecture, architecture and mapping modelings we presented in the preceding chapters.

The architecture models in the three approaches are very simple (left), very flexible (center), and very detailed (right). See also Chapter 3.

5.2.1 Symbolic Program Flow Details

Figure 5.2 shows the flow of the SP approach in more details. After a parallel application (process network) specification is obtained, each process is parsed and SP abstractions are derived. The SP abstractions assume that some basic blocks of the original application process are going to be substituted by symbolic instructions, and that some control points of the original application process are going to be annotated. A designer is the one who selects whether a basic block or a control point is going to be annotated or not. This is indicated with the label “Abstraction of details”.

Once the application sources and abstraction assumptions are ready, a tool based on the general purpose C/C++ parser [87] can translate them into the data-set independent SPs and control annotated application sources. The workstation symbol in Figure 5.2, with the label “Parser” above it, represents this tool. The control annotated application sources are then compiled and run on a single data set in order to check functional correctness. The resulting simulation is an untimed simulation, since no architecture restrictions are applied yet. The workstation symbol in Figure 5.2, with the label “Untimed Simulation” above it, represents the above. At this point the annotated control points generate control trace outcomes. These control traces represent the data-set dependent part of the SP representation.

In parallel with the previous steps, an architecture specification is obtained. The specification is given in terms of platform parameters, i.e., a number of units, types of units, properties of units such as delays, policies and the platform topology.

The key Y-chart part, the mapping, connects the SP application representation (both SPs and control traces) with the architecture specification, i.e., maps application processes onto architecture units. The product of this binding is a source code file which can be compiled and run on an architecture simulator. When it comes to symbolic programs, they are translated to the corresponding data structure. This data structure is often known as a parse tree. At the same time, control traces which correspond to symbolic programs are connected with these parse trees. These two inputs are forwarded to each architecture component on top of which a process from the process network is mapped.

Each architecture component is instantiated according to an architecture specification. Also, architecture components are generic Transaction Level Modeling (TLM) [3] building blocks.

They describe only a timing behavior, and not a functional behavior. A set of parameters that is given in the architecture specification is directly related to the timing behavior of a particular component.

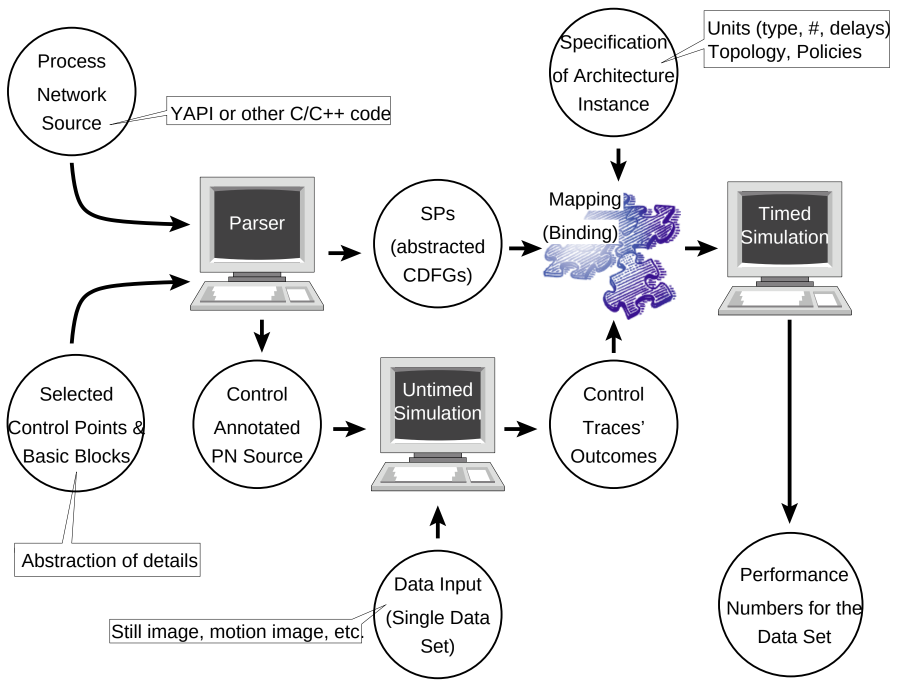

Process Network Source
YAPI or other C/C++ code
Specification of Architecture Instance
Units (type, #, delays) Topology, Policies
Parser
SPs (abstracted CDFGs)
Mapping (Binding)
Timed Simulation
Selected Control Points & Basic Blocks
Control Annotated PN Source
Untimed Simulation
Control Traces' Outcomes
Abstraction of details
Data Input (Single Data Set)
Still image, motion image, etc.
Performance Numbers for the Data Set

Figure 5.2: The flow of the Symbolic Program approach. The inputs are: (1) an application process network, (2) intra-process-level abstractions, (3) a data set, and (4) an architecture specification. The data-set independent parts of the application model are captured in (5) SPs, and the data-set dependent part of the application model are captured in (6) control trace outcomes.

After mapping is done, application and architecture are co-simulated, with functional behavior being captured by the non-timed application model, and non-functional behavior being captured by the timed architecture model. See also Chapter 1. The mapping step is not fully automated because DSE feedback loops are not taken into account here. Therefore, in Figure 5.2 the mapping step is shown as a puzzle - it requires a designer input.

After the simulation has completed one can choose to perform transformations and changes on SPs and control traces, architecture specifications, or data-sets (which again affects control traces) in order to see how different application-architecture combinations will behave when evaluated in subsequent simulations.

5.2.2 Directions for Improvements

The Symbolic Program approach has its pros compared with the approaches in the same field. We have emphasised them many times in this thesis. However, the approach has many points of improvement as well. The most critical one is the absence of both the front-end and back-end tools that interface to the designer. By the *front-end* tool we mean a tool that automates the following designer actions: (1) derivations of SP texts, and (2) annotations of control-points of interest. By the *back-end* tool we mean a tool that collects the performance numbers of N simulations of interests and then performs (semi-)automatic analysis of these numbers.

5.3 Conclusion

The main goal of this thesis was to produce a modeling approach suitable for efficient and accurate DSE of complex MPSoC systems. By *efficient* we mean that a fast system simulation model is derived from high-level representations of application(s) and architecture(s). By *accurate* we mean that the resulting simulation numbers are within a 20% error-margin with respect to the RTL model.

In Chapter 1 we showed complexities, issues, and directions of DSE of MPSoC systems. We recognized that the best way to deal with the DSE issues is to perform sufficient exploration at higher levels of abstraction before going to lower-levels and concrete designs (See Section 1.2.2). We identified that *higher* level of abstraction as the so-called ”Level of approximate performance models”. It is aligned with the Transaction Level Modeling paradigm, it requires the least level of platform details and it results in fast and accurate simulations (See Section 4.5.1 in Chapter 4).

In Chapter 2 we introduced a novel DSE application representation - Symbolic Programs. We defined symbolic programs as abstractions of behaviors which were earlier given either as control data-flow graphs or as source-code descriptions. The main idea behind them is the abstract execution [51]. We also defined a minimal set of expressions that is sufficient to expose hierarchy and partial-order and to preserve the information needed for the high-level architecture model later on. We pointed out that the symbolic program syntax may not express parallelism as easily as CDFGs do, but symbolic programs can be maintained more efficiently than CDFGs. The efficiency of maintenance of an application representation is directly proportional to the efficiency of a DSE method.

In Chapter 3 we presented our authentic architecture models for DSE of embedded systems. These models are built in a generic way, so they can be used to model various architecture characteristics. Further, the models capture most of the parallelism that designers express. We verified our model by regenerating the accurate results of the earlier case studies [33, 52, 58]. The models support both Intra-task level parallelism and Task-level parallelism. By *intra-task* level parallelism we mean compile-time and run-time partial-order executions. By *task-level* parallelism we mean multi-programming by means of multi-processing. The models have been implemented using `SystemC` [8], which makes it reusable in a wider DSE community.

In Chapter 4 we described the mapping approach using our application representation and our architecture models. We illustrated the mapping approach by means of a few case-studies. The results of these cases we use to quantify the level of accuracy and DSE efficiency of our DSE method.

5.3.1 Primary Contributions

- **A proper concurrent application representation for the simulation-based MPSoC DSE.** We have introduced *symbolic programs* as an application representation that can be used for design-space exploration of heterogeneous multiprocessor embedded multiprocessors without biases regarding mapping layers or ”virtual-processors”. The *SP* representation properly separates application characteristics from data-set dependencies which makes it possible to reuse symbolic programs for different data-sets. To our knowledge no representation in the related work that is used at the same level of abstraction and at the same level of the application complexity can achieve such reuse.

- **A basic repository of the generic architecture models for the simulation-based MPSoC DSE.** We have created an `Archer` library of components, where all architecture model components are divided into the four basic groups: *processors*, *interfaces*, *arbiters* and *memory buffers* (See Chapter 3). In our opinion, each embedded SoC architecture can be specified by components of these groups and within each group there is a diversification based on lower-level component characteristics. To our knowledge, no TLM DSE library in the related work is able to achieve such generality.

- **Mapping steps that closely mimic the activities of an MPSoC designer in real-life cases.** We have defined our mapping approach based on the Y-chart. However, we have used common design steps to define our mapping approach rather than inventing new steps (See Chapter 4). Therefore, our mapping approach is highly correlated with what a designer would and does in reality use, whilst some other approaches are not [7, 36].

5.3.2 Secondary Contributions

- **Introduction of level of approximate performance models.** We have introduced a new level of abstraction in the *Abstraction Pyramid*. This level is the level of approximate performance models and it is closely linked to the TLM approach (See Section 1.3.2). The TLM approach is important because it became part of the `SystemC` standard [62].

- **Positioning of the simulation-based MPSoC DSE approaches with respect to accuracy and simulation speed.** We have been first to rationalise the pros and cons of simulation-based DSE approaches. This reasoning has resulted in an approach allowing designers (1) to perform design-steps as in the case of detailed design, (2) to run fast simulations of architectures being explored, and (3) to have more accurate numbers than in the case of TD simulations (See [11]). This reasoning was the starting point of development of some other simulation-based DSE methods, such as `Sesame`. As a matter of fact, the *big-picture* of DSE approaches from Chapter 2 has been reused in [36]. Thus, `Sesame` is incorrectly regarded as the originator of such positioning.

- **Modeling of the specific code-transformations such as the Detection-of-variants transformation.** The result published in [55] represents the formal research of this transformation and it is more specific to the compiler technologies. In this thesis we showed our work that has been done independently of this formal research, more through the prism of Architecture Modeling for Design Space Exploration. Particularly, we identified the problem in [32] - as described in the modeling of the ’detection of variants’ SP transformation in Chapter 2, Section 2.5.1, and we reported on our case-study for the Proof-of-Concept (PoC) in [11]. We acknowledge the originality and contribution of the formal approach presented in [55], and we point that such designer’s decision can be modeled using our approach.

Bibliography

[1] F. Vahid. The softening of hardware. *IEEE Computer Magazine*, April 2003.

[2] C. Rowen. Reducing soc simulation and development time. *IEEE Computer Magazine*, December 2002.

[3] DATE panel. Transaction level modeling. *Design Automation and Test in Europe*, March 2003.

[4] P. Laplante. *Real-Time Systems Design and Analisys: An Engineer's Handbook*. IEEE Computer Society Press, 1993.

[5] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source code in C*. Willey, 1996.

[6] R. Lauwerenis (IMEC), I. Bolsens (Xilinx), C. Rowen (Tensilica), Y. Tanurhan (Actel), K. Vissers (Chameleon Systems), S. Wang (Axis Systems). Panel 6b: Reconfigurable computing - different perspectives. In *Proc. of Design Automation and Test in Europe (DATE)'03*, Munich, Germany, March 2003.

[7] P. Lieverse, P. van der Wolf, K. Vissers, and E. Deprettere. A methodology for architecture exploration of heterogeneous signal processing systems. *Journal of VLSI Signal Processing for Signal, Image and Video Technology*, 29(3):197–207, November 2001.

[8] Synopsys, Inc., CoWare Inc., Frontier Design, Inc. Functional specification for systemc 2.0 - final. January 17th 2001.

[9] A. van Gemund. *Performance Modeling of Parallel Systems*. PhD thesis, Delft University of Technology, Delft, the Netherlands, 1996.

[10] V. Živković, and P. Lieverse. An Overview of Methodologies and Tools in the Field of System-Level Design. *Lecture Notes In Computer Science*, 2268:74–88, 2002.

[11] V. Živković, et al. Fast and Accurate Multiprocessor Architecture Exploration with Symbolic Programs. In *Proc. of Design Automation and Test in Europe (DATE)'03*, pages 656–661, Munich, Germany, March 2003.

[12] L. Thiele, S. Chakraborty, M. Gries, A. Maxiaguine, J. Greutert. Embedded software in network processors - models and algorithms. *Lecture Notes in Computer Science*, 2211:416–434, 2001.

[13] Y. Le Moullec, N. B. Amor, J.-P. Diguet, M. Abid, J.-L. Philippe. Multi-granularity metrics for the era of strongly personalized SOCs. In *Proc. of Design Automation and Test in Europe (DATE)'03*, pages 674–679, Munich, Germany, March 2003.

[14] Bart Kienhuis. *Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools*. PhD thesis, Delft University of Technology, January 1999.

[15] J.D. Gajski, and R. Kuhn. Guest editors introduction: New vlsi tools. *IEEE Computer*, pages 11–14, 1983.

[16] National Institute of Standards and Technology. Model of computation.

[17] C. A. R. Hoare. Communicating sequential processes. (21):666–677, 1978.

[18] Gilles Kahn. The sementics of a simple language for parallel programming. *Proceedings IFIP congress 74*, July 1974.

[19] E.A. Lee and T.M. Parks. Dataflow process networks. In *Proceedings of the IEEE*, volume 83, pages 773–801, May 1995.

[20] K.M. Kavi, B.P. Buckles, and U.N. Bhat. A formal definition of data flow graph models. *IEEE Transactions on Computers*, C-35(11):940–948, November 1986.

[21] E. A. Lee and D. G. Messerschmitt. Pipeline Interleaved Programmable DSP's: Synchronous Dataflow Programming. *IEEE Transactions on Acoustics, Speech, Signal Processing, Vol.ASSP-35*, September 2001.

[22] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California at Berkeley, December 1995.

[23] J. T. Buck and E. A. Lee. Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model. In *International Conference on Acoustics, Speech, Signal Processing*, April 1993.

[24] J. T. Buck. Static Scheduling and Code Generation from Dynamic Dataflow Graphs with Integer Valued Control Streams. In *The 28th Asilomar Conference on Signals, Systems, and Compilers*, October 1994.

[25] N. Cossement, R. Lauwereins, and F. Catthoor. DF*: An extension of synchronous dataflow with data dependency and non-determinism. In *Forum on Design Languages*, September 2000.

[26] D. Harel. Statecharts: A visual fromalism for complex systems. pages 231–274, 1987.

[27] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. pages 244–263, 1986.

[28] B. Selic, G. Gullekson, and P. Ward. *Real-time Object-oriented Modeling*. Willey, 1994.

[29] F. Balarian, E. Sentovich, M. Chiodo, P. Guisto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli. *Hardware-Software Co-Design of Embedded Systems - The POLIS Approach*. Kluwer Academic, 1997.

[30] P. Lieverse, P. van der Wolf, and E. Deprettere. A trace transformation technique for communication refinement. *Proceedings of 9th Int. Symposium on Hardware/Software Codesign (CODES'01)*, pages 134–139, April 2001.

[31] J. Regehr, M. B. Jones, and J. A. Stankovic. Operating system support for multimedia: The programming model matters. *Technical Report MSR-TR-2000-89, Microsoft Research, Microsoft Corporation*, September 2000.

[32] V. Živković, et al. Design Space Exploration of Streaming Multiprocessor Architectures. In *Proc. of Signal Processing System (SiPS)'02*, San Diego, October 2002.

[33] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere. System level design with Spade: an M-JPEG case study. In *International Conference on Computer Aided Design (ICCAD'01)*, San Jose CA, USA, November 2001.

[34] A. D. Pimentel, P. Lieverse, P. van der Wolf, L.O. Hertzberger and E.F. Deprettere. Exploring Embedded-Systems Architectures with Artemis. *IEEE Computer*, 34(11):57–63, November 2001.

[35] A. D. Pimentel. The parallelisation of the object-oriented simulation language pearl. Master's thesis, Amsterdam, The Netherlands, August 1993.

[36] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2), February 2006.

[37] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multi-objective Optimization. In *Evolutionary Methods for Design, Optimization, and Control*, Barcelona, Spain, 2002.

[38] F. Thoen, J. Van Der Steen, G. de Jong, G. Goossens, and H De Man. Multi-thread graph: a system model for real-time embedded software synthesis. In *EDTC '97: Proceedings of the 1997 European conference on Design and Test*, page 476, Washington, DC, USA, 1997. IEEE Computer Society.

[39] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendor, Sonia and S. Yuhong. Taming heterogeneity - the Ptolemy approach. In *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, 2002.

[40] M. Gries. Methods for Evaluating and Covering the Design Space During Early Design Development. *Integration. the VLSI Journal*, 38(2):131–183, 2004.

[41] K. Lahiri, A. Raghunathan, S. Dey. Fast Performance Analysis of Bus-Based Systems-On-Chip Communication Architectures. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)'99*, San Jose, CA, USA, November 7-11 1999.

[42] K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *Proceedings of Design, Automation, and Test in Europe Conference (DATE'02)*, pages 506–513, Paris, France, March 2002.

[43] P. Paulin, C. Pilkington, and E. Bensoudane. StepNP: A System-Level Exploration Platform for Network Processors. *IEEE Design and Test of Computers*, 19(6):17–26, 2002.

[44] R. Morris, E. Kohler, J. Jannotti, and M.F. Kaashoek. The Click Modular Router. *SIGOPS Oper. Syst. Rev.*, 33(5):217–231, 1999.

[45] R.A. Bergamaschi, Y. Shin, N. Dhanwada, S. Bhattacharya, W.E. Dougherty, I. Nair, J. Darringer, and S. Paliwal. SEAS: a system for early analysis of SoCs. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 150–155, Newport Beach, CA, USA, 2003.

[46] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. *SIGPLAN Not.*, 37(7):18–27, 2002.

[47] A.S. Cassidy, J.M. Paul, and D.E. Thomas. Layered, Multi-Threaded, High-Level Performance Design. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE'03)*, pages 954–959, Munich, Germany, March 2003.

[48] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[49] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[50] G. K. Wallace. The jpeg still picture compression standard. *IEEE Transactions on Consumer Electronics*, page xxx, 1991.

[51] J. Larus. Abstract execution: A technique for efficiently tracing programs. *Software–Practice and Experience*, 20(12):1241–1258, December 1990.

[52] T. Harriss, R. Walke, B. Kienhuis, and E. Deprettere. Compilation from matlab to process networks realized in fpga. *Journal on Design Automation of Embedded Systems, Kluwer*, 7(4), 2002.

[53] A. Turjan, et al. The Compaan Tool Chain: Converting Matlab into Process Networks. In *Proc. DATE'02*, Paris, France, xxx 2002.

[54] J. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly, 1995.

[55] S. Derrien, A. Turjan, C. Zissulescu, B. Kienhuis and E. Deprettere. Deriving efficient control in process networks with compaan/laura. *accepted for publication in the International Journal of Embedded Systems Inderscience (IJES)*.

[56] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988.

[57] T. Stefanov and E. Deprettere. Deriving process networks from weakly dynamic applications in system-level design. *Proceedings of IEEE/ACM/IFIP Int. Conf. on HW/SW Codesign and System Synthesis (CODES-ISSS'03)*, pages 90–96, October 2003.

[58] E. de Kock. Multiprocessor mapping of process networks: A jpeg decoding case study. *Proceedings of 15th Int. Symposium on System Synthesis (ISSS'02)*, pages 68–73, 2002.

[59] Todor Stefanov. *Converting Weakly Dynamic Programs to Equivalent Process Network Specifications*. PhD thesis, Leiden University, December 2004.

[60] Alexandru Turjan. *Compiling nested loop programs to process networks*. PhD thesis, Leiden University, March 2007.

[61] I. Somerville. *Software Engineering, 6th Edition*. Addison-Wesley, August 11 2000.

[62] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, Boston, 2002.

[63] J. Carter. *Programming SQL, Computer Studies Series*. Blackwell Scientific Publications, 1992.

[64] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[65] J. L. Hennessy and D. A. Patterson. Computer architecture: a quantitative approach, 3rd edition. 2002.

[66] E. Dijkstra. Cooperating sequential processes. *Technical Report EWD-123, University of Eindhoven, The Netherlands*, 1965.

[67] P. Stravers and J. Hoogerbugge. Homogeneous multiprocessing and the future of silicon design paradigms. In *Proc. of VLSI-TSA'01*, 2001.

[68] M. Dewey. *A Classification and Subject Index for Cataloguing and Arranging the Books and Pamphlets of a Library*. Project Gutenberg - http://www.gutenberg.org/etext/12513, 2004 (originally from 1876).

[69] B. Lewis and D. Berg. *Multithreaded Programming With PThreads*. Sun Microsystems Press and Prentice Hall, 1998.

[70] Synopsys, Inc., CoWare Inc., Frontier Design, Inc. Systemc version 2.0 user's guide.

[71] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[72] P. Lieverse. Many-to-one mapping and scheduling. *Spade Project - Memo 12 (Internal Document)*, September 1999.

[73] B. R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. Wiley, September 1998.

[74] Sylvain Alliot. *Architecture Exploration for Large Scale Array Signal Processing Systems*. PhD thesis, Leiden University, March 2003.

[75] V. Živković, et al. Mapping Specification-level primitives to IP-primitives: A Case Study. In *Proc. of the Third International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)'03*, Samos, Greece, July 2003.

[76] R. A. Uhlig, T. N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys, Vol.29*, (2), June 1997.

[77] S. Bhattacharyya, P. Murthy, and E. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[78] E. de Kock, et al. Yapi: Application modeling for signal processing systems. In *Proc. of Design Automation Conference (DAC)'00*, Los Angeles, USA, June 2000.

[79] J. Proakis et al. *Algorithms for Statistical Signal Processing*. Prentice Hall, Inc., 2002.

[80] M. Goriss, et al. Adaptive beam-forming system for radio-frequency interference rejection. In *IEEE Proc. Rad.*, xxx, xxx 1999.

[81] W. A. Chen, C. Harrison, and S. C. Fralick. A fast computational algorithm for the discrete cosine transorm. *IEEE Transactions on Communications, Vol. COM-25, No. 9*, pages 1004–1011, 1977.

[82] C. Loeffler, et al. Practical fast 1-d dct algorithms with 11 multiplications. In *Proc. of ICASSP'89*, pages 988–991, 1989.

[83] P. Stravers et al. Coherent memory network for wasabi. *Technical Report ???, Philips Research Laboratories*, November 14 2003.

[84] Dong-Ik Ko and Shuvra S. Bhattacharyya. Modeling of block-based dsp systems. *J. VLSI Signal Process. Syst.*, 40(3):289–299, 2005.

[85] A. Gerstlauer and D. D. Gajski. System-level abstraction semantics. *Proceedings of 15th Int. Symposium on System Synthesis (ISSS'02)*, pages 231–236, 2002.

[86] W. Wolf. *Computers as Components - Principles of Embedded Computing System Design*. Morgan Kaufmann Publishers, 2001.

[87] Terence John Parr. *Language Translation Using PCCTS & C++: A Reference Guide*. 1996.

Appendix A

Implementation Details

A.1 Symbolic Porgram Definition Section

Figures A.1, A.2, A.3, and A.4 form the definition section of the Yacc notation of the SP text syntax. Figure A.2 shows the pattern matching rules for the lexical analyzer of the SP parser. The lexer takes an input stream (e.g., a file containing the SP-text) and *tokenizes* it, i.e., divides it up into lexical tokens. This tokens are combined (processed further) in the parser according to the syntax rules. The `% union` declaration in Figure A.1 identifies all of the possible C types that a symbol of SP can have. Some of these types are basic types, such as `int` (e.g., integer) or `text` (e.g., string). The other are more abstract (complex). Figure A.3 provides the input of what values are associated with *tokens* or terminal symbols. Concretely, only the `NUM` and `ID` tokens have values associated with themselves (an integer and a string, respectively). Figure A.4 declares the types of non-terminals. A type may be any of the types declared in `% union` in Figure A.1 and it is surrounded with `< >`, while the non-terminal is stated afterwards.

A.2 Symbolic Program Rules Section

Figure A.5 illustrates the grammar rules of the SP text. The rules shown resemble the SP abstractions from the most general one (i.e., the SP itself - Lines: 1-3) towards the leaf ones (e.g., an SI matches `trace_event`, or the parameters of an SI match `identifier` - Lines: 32-41). The actions (indicated inside of the C comments) are used to create the data-structure of the particular SP.

An SP representation must contain the `main` text-section[1], but may contain zero or more

[1]This is the main CDFG of a process.

```
%union
{
  text                text_field;
  int                 int_field;
  symbolic_program    program_field;
  main                main_field;
  function_list       function_list_field;
  function            function_field;
  compound_statement  compound_statement_field;
  statement_list      statement_list_field;
  statement           statement_field;
  selection_statement selection_statement_field;
  iteration_statement iteration_statement_field;
  jump_statement      jump_statement_field;
  flow_type           flow_type_field;
  label_var           label_var_field;
  program_call        program_call_field;
  condition           condition_field;
  variable            variable_field;
  trace_statement     trace_statement_field;
  parallel_trace      parallel_trace_field;
  trace_event         trace_event_field;
  event_type          event_type_field;
  identifiers         identifiers_field;
  identifier          identifier_field;
  identifier_type     identifier_type_field;
}
```

Figure A.1: The symbol values types used in the SP text syntax.

`function` text-sections[2] (see the Yacc code in Figure A.5 - Lines: 1-7). Each `main` or `function` is essentially a `compound_statement`, i.e., an ordered set of statements (Lines: 8-10).

A `statement` can be any of the procedural programming language constructs: a computation or communication load - `trace_statement`, the already explained compound statement - `compound_statement`, an if-construct - `selection_statement`, a loop construct - `iteration_statement`, a go-to construct - `jump_statement`, a label - `label_var`, and a sub-routine call - `program_call` (see Lines: 11-29).

Note that `trace_statement` is actually the extended SI; it is either a single symbolic instruction - `trace_event`, or a bundle of partially ordered symbolic instructions. A `trace_event` may be either of one of `READ`, `WRITE`, or `EXECUTE`, or it is an `SKIP` (empty) instruction (Lines: 32-35). The first value (`NUM`) either enumerates the application FIFO channels (in the cases of `READ` or `WRITE`) or determines the operation (in the case of `EXECUTE`). The non-terminal represents one or more arguments or `identifiers` (Lines: 36-37). An `identifier` has a unique name (string) in the SP scope. It may or may not be explicitly characterized as `INPUT` or `OUTPUT`(Lines: 38-41). Follows the load of a symbolic instruction (given by the third value-token of a symbolic instruction - `NUM`); it shows either the communication load in terms of words[3], or the computation load in terms of some abstract Worst-Case-Execution-Path[4].

[2]These are the abstracted CDFGs of the sub-routines of a process.

[3]This information will impact the real-platform communication. Thus, it is a number that can be securely translated into the multiples of the chosen platform communication-word size later-on.

[4]This information will impact the mapping stage; therefore this number must somehow depict the number of

```
"goto"       {lexReport("GOTO", (text) yytext);
              return GOTO;}
"break"      {lexReport("BREAK", (text) yytext);
              return BREAK;}
"continue"   {lexReport("CONTINUE", (text) yytext);
              return CONTINUE;}
"return"     {lexReport("RETURN", (text) yytext);
              return RETURN;}
"call"       {lexReport("CALL", (text) yytext);
              return CALL;}
"function"   {lexReport("FUNC", (text) yytext);
              return FUNC;}
"main"       {lexReport("MAIN", (text) yytext);
              return MAIN;}
"condition"  {lexReport("COND", (text) yytext);
              return COND;}
"loop"       {lexReport("LOOP", (text) yytext);
              return LOOP;}
"read"       {lexReport("READ", (text) yytext);
              return READ;}
"execute"    {lexReport("EXECUTE", (text) yytext);
              return EXECUTE;}
"write"      {lexReport("WRITE", (text) yytext);
              return WRITE;}
"skip"       {lexReport("SKIP", (text) yytext);
              return SKIP;}
"||"         {lexReport("OR", (text) yytext);
              return OR;}
"in"         {lexReport("IN", (text) yytext);
              return IN;}
"out"        {lexReport("OUT", (text) yytext);
              return OUT;}
[:();,]    {lexReport("char", (text) yytext);
              return yytext[0];}
[0-9]+       {lexReport("NUM", (text) yytext);
              yylval.int_field = (int)
              strtol((text) yytext, NULL, 0);
              return NUM;}
([a-zA-Z_0-9])* {lexReport("ID", (text) yytext, &yylval);
              return ID;}
.            {skip((text) yytext);}
n            {skip((text) yytext);}
```

Figure A.2: The regular expression rules used by the lexer to match the tokens.

A.3 Symbolic Program Interpretation

The interpretation of an SP by the Program Unit CFSM is done by means of a parse-tree traversal plus the particular control-trace.

A parse tree is a data structure used for storing the parsed text. The start symbol is stored at the top, as a root, while the terminals are stored as leaves. The tree body (branches and nodes) is formed from the other non-terminals available in the parsed text. In the case of SP shown in Figure 2.13, the `main` non-terminal is almost the root [5], and, e.g., `skip` at Line 20 is one of the leaves.

The tree-traversal we use is the *post-order depth-first traversal*. That means first the nodes

low-level operations required on the potential platforms.

[5]Actually, the root is an invisible `symbolic_program` non-terminal which wraps the `main` construct.

```
%token  GOTO
%token  BREAK
%token  CONTINUE
%token  RETURN
%token  CALL
%token  FUNC
%token  MAIN
%token  COND
%token  LOOP
%token  READ
%token  EXECUTE
%token  WRITE
%token  SKIP
%token  OR
%token  <int_field> NUM
%token  IN
%token  OUT
%token  <text_field> ID
```

Figure A.3: The token values used in the SP text syntax.

```
%type   <program_field>             symbolic_program
%type   <main_field>                main
%type   <function_list_field>       function_list
%type   <function_field>            function
%type   <compound_statement_field>  compound_statement
%type   <statement_list_field>      statement_list
%type   <statement_field>           statement
%type   <selection_statement_field> selection_statement
%type   <iteration_statement_field> iteration_statement
%type   <jump_statement_field>      jump_statement
%type   <flow_type_field>           flow_type
%type   <label_var_field>           label_var
%type   <program_call_field>        program_call
%type   <condition_field>           condition
%type   <variable_field>            variable
%type   <trace_statement_field>     trace_statement
%type   <parallel_trace_field>      parallel_trace
%type   <trace_event_field>         trace_event
%type   <event_type_field>          event_type
%type   <identifiers_field>         identifiers
%type   <identifier_field>          identifier
%type   <identifier_type_field>     identifier_type
```

Figure A.4: The non-terminal type values used in the SP text syntax.

of the left subtree are visited, then the nodes of the right subtree are visited, and in the end the root is visited. The control trace is always determined by two factors: the data set and the SP structure. Consequently, there may be many control traces because there may be many different data sets. However, all of them must resemble the SP structure, i.e., there must direct the tree-traversal correctly. That is, the value of the condition node (e.g., loop condition 1) must match the current control-trace event value (e.g., condition 1 equals true).

For example, given the following contents of the control-trace:

$(condition1 \equiv true) \prec (condition2 \equiv true) \prec (condition3 \equiv true) \prec (condition4 \equiv false)$

with the SP shown in Figure 2.13, the following partially ordered symbolic instructions are generated:

```
symbolic_program : main function_list {/* action */}
                 | main {/* action */}
                 | function_list main {/* action */};
main : MAIN compound_statement {/* action */};
function_list : function {/* action */}
              | function_list function {/* action */};
function : FUNC ID compound_statement {/* action */};
compound_statement : ’{’ statement_list ’}’ {/* action */};
statement_list : statement {/* action */}
               | statement_list statement {/* action */};
statement : trace_statement {/* action */}
          | compound_statement {/* action */}
          | selection_statement {/* action */}
          | iteration_statement {/* action */}
          | jump_statement {/* action */}
          | label_var {/* action */}
          | program_call {/* action */};
selection_statement : condition compound_statement {/* action */};
iteration_statement : LOOP condition compound_statement {/* action */};
jump_statement : GOTO label_var {/* action */}
               | flow_type {/* action */};
flow_type : BREAK {/* action */}
          | CONTINUE {/* action */}
          | RETURN {/* action */};
label_var : NUM ’:’ {/* action */};
program_call : CALL ID {/* action */};
condition : COND variable {/* action */};
variable : NUM ’(’ identifiers ’)’ {/* action */};
trace_statement : parallel_trace ’;’ {/* action */};
parallel_trace : trace_event {/* action */}
               | parallel_trace OR trace_event {/* action */};
trace_event : event_type NUM ’(’ identifiers ’,’ NUM ’)’ {/* action */}
            | SKIP {/* action */};
event_type : READ {/* action */};
           | EXECUTE {/* action */}
           | WRITE {/* action */};
identifiers : identifier {/* action */}
            | identifiers identifier {/* action */};
identifier : identifier_type ID {/* action */}
           | ID {/* action */};
identifier_type : IN {/* action */}
                | OUT {/* action */};
```

Figure A.5: The production rules of the SP text syntax.

`...; read 11 (a, 1) ‖ read 12 (b, 1) ; ...`

As a conclusion, all control information is disappeared and the partially ordered symbolic trace is passed further to the architecture processor model.

The SP parse tree plays the role of the process object code (i.e., program code). The control trace models the data-input and execution. By assigning appropriate delays (e.g., jump-delays, or loop-delays, etc.) to different parse-tree nodes various processor execution effects can be modeled. Additionally, the traversing algorithm may be constructed that it delays depending on the contents of the control trace; e.g., in the case of $(condition1 \equiv true)$ it delays while in the case of $(condition1 \equiv false)$ it does not delay. The last control trace execution scheme is co-related with the cache-hit and cache-miss modeling. Although we do not discuss modeling of the cache related behavior in this thesis, it is true that our model, in general, provides some features for modeling the same.

A.4 Processor Unit Threads

```
 *CFSM* **initialize** state ⟵ IDLE
**begin**
    **do forever**
        **if** state = IDLE
            *blocking acquire of a next symbolic instruction bundle*
            state ⟵ RUN
        **fi**
        **if** state = RUN
            *serialize the acquired bundle based on the number of RU, EU, WU using FCFS*
            *wait for a configured delay*
            **do until** *queue of serialized bundles = ∅*
                *blocking transmit of the topmost serialized bundle*
                *remove the transmitted bundle from the queue*
            **done**
            state ⟵ IDLE
        **fi**
    **done**
**end**
```

Figure A.6: Implementation of the FECTRL Concurrent Finite State Machine.

```
 *CFSM* **initialize** state ⟵ WAIT
**begin**
    **do forever**
        **if** state = WAIT
            *blocking acquire of a next symbolic instruction bundle*
            state ⟵ RUN
        **fi**
        **if** state = RUN
            *wait for a configured delay*
            *dispatch symbolic instructions from the bundle based on R, E, W attributes*
            state ⟵ IDLE
        **fi**
        **if** state = IDLE
            *block until all dispatched symbolic instructions finished*
            state ⟵ WAIT
        **fi**
    **done**
**end**
```

Figure A.7: Implementation of the BECTRL Concurrent Finite State Machine.

```
 CFSM initialize state ⟵ WAIT
begin
    do forever
        if state = WAIT
            if dispatched read instructions = finished
                check for the next read symbolic instructions
            fi
            if dispatched execute instructions = finished
                check for the next execute symbolic instructions
            fi
            if dispatched write instructions = finished
                check for the next write symbolic instructions
            fi
            blocking acquire of next symbolic instruction based on the above checks
            state ⟵ RUN
        fi
        if state = RUN
            if any new acquired
                wait for the configured delay
                do forall acquired read symbolic instructions
                    dispatch read instruction
                done
                do forall acquired execute symbolic instructions
                    dispatch execute instruction
                done
                do forall acquired write symbolic instructions
                    dispatch write instruction
                done
            fi
            state ⟵ IDLE
        fi
        if state = IDLE
            block until all dispatched symbolic read instructions finished ∨
            block until all dispatched symbolic execute instructions finished ∨
            block until all dispatched symbolic write instructions finished
            state ⟵ WAIT
        fi
    done
end
```

Figure A.8: Implementation of the Modified BECTRL Concurrent Finite State Machine.

```
CFSM initialize state ⟵ IDLE
begin
    do forever
        if state = IDLE
            get next execute symbolic instruction (operand, opcode, size)
            state ⟵ SETUP
        fi
        if state = SETUP
            blocking open connection to the physical FIFO based on port, RU index, data size
            state ⟵ STALL
        fi
        if state = STALL
            signal that read is ready
            blocking check whether write is ready
            state ⟵ RUN
        fi
        if state = RUN
            do until data size = 0
                load data unit
                wait for the configured data unit delay
                decrement size
            done
            signal close connection
            if run-time pipelining
                unblocking put output operand
            fi
            signal that symbolic instruction execution has been finished
            state ⟵ IDLE
        fi
    done
end
```

Figure A.9: Implementation of the RU Concurrent Finite State Machine.

A.5 Interface Component Unit Threads

```
 CFSM initialize state ⟵ IDLE
begin
    do forever
        if state = IDLE
             get next execute symbolic instruction (operand, opcode, size)
            if run-time pipelining
                blocking get operand
            fi
            state ⟵ SETUP
        fi
        if state = SETUP
             blocking open connection to the physical FIFO based on port, WU index, data size
            state ⟵ RUN
        fi
        if state = RUN
            do until data size = 0
                store data unit
                wait for the configured data unit delay
                decrement size
            done
            state ⟵ STALL
        fi
        if state = STALL
             signal that write is ready
             blocking check whether read is ready
             signal close connection
             signal that symbolic instruction execution has been finished
            state ⟵ IDLE
        fi
    done
end
```

Figure A.10: Implementation of the WU Concurrent Finite State Machine.

```
 CFSM initialize state ⟵ IDLE
begin
    do forever
        if state = IDLE
             get next execute symbolic instruction (operands, opcode, budget)
            if run-time pipelining
                do forall input operands in array of operands
                    blocking get input operand
                done
            fi
            state ⟵ RUN
        fi
        if state = RUN
            do until budget = 0
                wait for the configured unit delay
                decrement budget
            done
            if run-time pipelining
                do forall output operands in array of operands
                    unblocking put output operand
                done
            fi
             signal that symbolic instruction execution has been finished
            state ⟵ IDLE
        fi
    done
end
```

Figure A.11: Implementation of the EU Concurrent Finite State Machine.

```
 *CFSM* **initialize** state $\longleftarrow$ WAIT
**begin**
    **do forever**
        **if** state = WAIT
            **if** *this is Read Interface PIC*
                *acquire all read ports with available data from Read Interface*
            **fi**
            **if** *this is Write Interface PIC*
                *acquire all write ports with available room from Write Interface*
            **fi**
            state $\longleftarrow$ INTERRUPT
        **fi**
        **if** state = INTERRUPT
            **do forall** *acquired ports*
                *deliver port status to DOS through a master*
            **done**
            state $\longleftarrow$ WAIT
        **fi**
    **done**
**end**
```

Figure A.12: Implementation of the PIC Concurrent Finite State Machine.

```
CFSM initialize state ← INPUT
begin
    do forever
        if state = INPUT
            block until reschedule conditions appear
            state ← OUTPUT
        fi
        if state = OUTPUT
            do foreach SPU-core in set of all cores
                if set of all Symbolic Programs = $emptyset$
                    break
                fi
                if SPU-core $\neq$ busy
                    reschedule for this SPU-core
                    if rescheduling result in set of all cores
                        find Symbolic Program scheduled on this SPU-core
                        unblock Symbolic Instruction fetching for that Symbolic Program
                        unblock this SPU-core
                    fi
                fi
            done
            state ← INPUT
        fi
    done
end
$RIRQ_{slave}$()
begin
    queued as feasible Symbolic Programs for the read ports with data
end
$WIRQ_{slave}$()
begin
    queued as feasible Symbolic Programs for the write ports with room
    if room for any of postponed symbolic write instruction
        unblock CFSM (rescheduling condition appeared)
    fi
end
$FETCH_{slave}$( index )
begin
    if for indexed Symbolic Program exists an already fetched but postponed instruction
        block Symbolic Instruction fetching for indexed Symbolic Program
    fi
        acquire next Symbolic Instruction from the indexed SP Stream
    if this instruction may be feasible based on data/room availability
        unblock CFSM (rescheduling condition appeared)
    fi
end
$TRAP_{cores}$( index )
begin
    mark the indexed SPU-core as $\neq$ busy
    block the indexed SPU-core
end
```

Figure A.13: Implementation of the DOS composite channel.

```
1   CFSM initialize state ⟵ IDLE
2   begin
3       do forever
4           if state = IDLE
5               blocking acquire of a next symbolic instruction bundle
6               state ⟵ RUN
7           fi
8           if state = RUN
9               serialize the acquired bundle based on the number of RU, EU, WU using FCFS
10              wait for a configured delay
11              do until queue of serialized bundles = ∅
12                  deliver the topmost serialized bundle through a master
13                  remove the transmitted bundle from the queue
14              done
15              state ⟵ IDLE
16          fi
17      done
18  end
```

Figure A.14: Implementation of the OS-based FECTRL CFSM (see the modification at Line 12).

```
1   CFSM initialize state ⟵ WAIT
2   begin
3       do forever
4           if state = WAIT
5               acquire a next symbolic instruction bundle through a master
6               state ⟵ RUN
7           fi
8           if state = RUN
9               wait for a configured delay
10              dispatch symbolic instructions from the bundle based on R, E, W attributes
11              state ⟵ IDLE
12          fi
13          if state = IDLE
14              block until all dispatched symbolic instructions finished
15              state ⟵ WAIT
16          fi
17      done
18  end
```

Figure A.15: Implementation of the OS-based BECTRL CFSM (see the modification at Line 5).

```
 CFSM initialize state ⟵ IDLE
begin
    do forever
        if state = IDLE
             blocking wait for the connection opening request from the corresponding RU
            state ⟵ RUN
        fi
        if state = RUN
             establish the read connection between the RU and the FIFO
             wait for a configured delay
             acknowledge that the read connection is opened
            state ⟵ WAIT
        fi
        if state = WAIT
             block until close read connection request signaled
             close the read connection
            state ⟵ IDLE
        fi
    done
end
```

Figure A.16: Implementation of the FICTRL Concurrent Finite State Machine.

```
 CFSM initialize state ⟵ IDLE
begin
    do forever
        if state = IDLE
             blocking wait for the connection opening request from the corresponding WU
            state ⟵ RUN
        fi
        if state = RUN
             establish the write connection between the WU and the FIFO
             wait for a configured delay
             acknowledge that the write connection is opened
            state ⟵ WAIT
        fi
        if state = WAIT
             block until close write connection request signaled
             close the write connection
            state ⟵ IDLE
        fi
    done
end
```

Figure A.17: Implementation of the FOCTRL Concurrent Finite State Machine.

```
CFSM initialize state $\leftarrow$ IDLE
begin
    do forever
        if state = IDLE
            blocking wait until the next read connection is established
            state $\leftarrow$ HANDSHAKE
        fi
        if state = HANDSHAKE
            blocking check for read to become ready and register the data-size required
            state $\leftarrow$ STALL
        fi
        if state = STALL
            non-blocking test on the amount of data in the FIFO
            if configured for burst
                if data-size > cached-size
                    blocking wait for (data-size - cached-size) in the FIFO
                fi
            fi
            if not configured for burst
                blocking wait for data-size in the FIFO
            fi
            if bus configured $\vee$ data-size > cached-size
                blocking claim of the bus
            fi
            state $\leftarrow$ RUN
        fi
        if state = RUN
            if configured for burst
                if cached > data-size
                    get data-size data-units from cached data
                    cached, data-size $\leftarrow$ 0
                fi
                if cached > 0
                    get cached data-units
                    data-size $\leftarrow$ (data-size - cached)
                    cached $\leftarrow$ 0
                fi
                if data-size > 0
                    if burst-word > data-size
                        cached $\leftarrow$ (burst-word - data-size)
                    fi
                    do until burst-word $\neq$ 0
                        get data-unit from the FIFO
                        wait for a configured delay unit
                        decrement burst-word
                    done
                    data-size $\leftarrow$ 0
                fi
            fi
            if not configured for burst
                do until data-size $\neq$ 0
                    get data-unit from the FIFO
                    wait for a configured delay unit
                    decrement data-size
                done
            fi
            if configured bus $\wedge$ bus was claimed
                non blocking bus grant
            fi
            signal room appeared in the FIFO
            signal written data ready
            close the read connection
            state $\leftarrow$ IDLE
        fi
    done
end
```

Figure A.18: Implementation of the FIU Concurrent Finite State Machine.

```
 CFSM initialize state ⟵ IDLE
begin
    do forever
        if state = IDLE
          if not configured for burst
              blocking wait until the next write connection is established
              room-size ⟵ budget
          fi
          if configured for burst
              block until the buffering event appears
              room-size ⟵ budget
              if the room-size required < burst-word
                  counter = ( bust-word - room-size)
                  do until the room-size required < burst-word
                      wait until counter time units elapses
                      if counter > ( bust-word - room-size)
                          counter = ( bust-word - room-size)
                      fi
                      if counter ≤ ( bust-word - room-size)
                          decrement counter
                      fi
                      if counter = 0
                          break
                      fi
                  done
              fi
           fi
           state ⟵ HANDSHAKE
        fi
        if state = HANDSHAKE
           if not configured for burst
               blocking check for write to become ready and register the room-size required
           fi
           if configured for burst
               block until BUFFERING CFSM signals
           fi
           state ⟵ STALL
        fi
        if state = STALL
            blocking wait for room-size in the FIFO
           if bus configured
               blocking claim of the bus
           fi
           state ⟵ RUN
        fi
        if state = RUN
           if not configured for burst
               do until room-size ≠ 0
                   put data-unit to the FIFO
                   wait for a configured delay unit
                   decrement room-size
               done
               if configured bus
                   non blocking bus grant
               fi
               signal data appeared in the FIFO
               signal read data ready
               close the write connection
           fi
           if configured for burst
               if burst-word > room-size
                   do until burst-word ≠ room-size
                       wait for a configured delay unit
                       decrement burst-word
                   done
```

Figure A.19: Implementation of the FOU CFSM (cont. in Figure A.20).

```
                fi
                do until burst-word $\neq$ 0
                    put data-unit from the FIFO
                    wait for a configured delay unit
                    decrement burst-word
                done
                if configured bus
                    non blocking bus grant
                fi
                room-size $\leftarrow$ 0
                signal data appeared in the FIFO
            fi
            state $\leftarrow$ IDLE
        fi
    done
end
# The BUFFERING is an additional CFSM thread; exists only for the burst-bus!
BUFFERING initialize state $\leftarrow$ IDLE
begin
    do forever
        if state = IDLE
            non-blocking test on the amount of room in the FIFO (for DOS)
            blocking wait until the next write connection is established
            blocking check for write to become ready and register the room-size required
            state $\leftarrow$ HANDSHAKE
        fi
        if state = HANDSHAKE
            block access for setting up budget
            if requested room from RU $\geq$ burst-word
                budget $\leftarrow$ burst-word
            fi
            if requested room from RU < burst-word
                budget $\leftarrow$ requested room from RU
            fi
            requested room from RU $\leftarrow$ ( requested room from RU - budget)
            unblock after setting up budget
            if initialization $\neq$ true
                delay for a set-up delay
            fi
            state $\leftarrow$ STALL
        fi
        if state = STALL
            non-blocking test on the amount of room in the FIFO
            if ( budget > room tested) $\vee$
                ( budget $\geq$ burst-word)
                signal to main CFSM that buffering is done
                block until main CFSM signals that emptying is done
            fi
            if ( budget $\leq$ room tested) $\wedge$
                ( budget < burst-word)
                signal to main CFSM that buffering is done
            fi
            state $\leftarrow$ RUN
        fi
        if state = RUN
            if requested room from RU $\neq$ 0
                state $\leftarrow$ HANDSHAKE
            fi
            if requested room from RU = 0
                signal data appeared in the FIFO
                signal read data ready
                close the write connection
                state $\leftarrow$ IDLE
            fi
        fi
    done
end
```

Figure A.20: Continuation of the FOU Concurrent Finite State Machine.

Index

Samenvatting

Moderne embedded systemen zijn ontworpen om met veel complexere en reken-intensievere applicaties om te kunnen gaan dan 10 jaar geleden. Dit komt voornamelijk door de vooruitgang op het gebied van geavanceerde signal-verwerkings ICi's en de ontwikkelingen van geavanceerde applicaties. Door een toename in het aantal signaalverwerkingselementen die geïntegreerd kunnen worden op één IC maakt het systeem-level ontwerp uiterst complex en uitdagend.

Deze ontwerptaak wordt bereikt door nieuwe ontwerp-paradigmes, geaccumuleerde kennis en de expertise op het gebied van parallel-computing. Uit Object Oriented Technologies geiëvolueerde onwerpparadigmas worden steeds beter om de system level verkenning en onwerpmethodologien te ondersteunen. Even belangrijk, modelling applicaties en architecturen en de vertaling van taken naar parallele architecturen zijn versterkt door de solide en rijke methoden die in de laatste twee decennia door parallel-computing kringen waren onderzocht en voorgesteld.

Parallele computing vindt dus eigen weg uit de wetenschappelijke applicatie domeinen en gaat verder naar de andere domeinen, onder andere multimedia en (draadloze) communicatie domeinen. In deze domeinen, onder parallel computing system wordt bedoeld een heterogene systeem wiens onderdelen zijn communicatie eenheden van verschillende typen and meestal gedistribueerde geheugen eenheden. Een platform kan eigenlijk alles zijn: van multiprocessoren met taak-teogewijd processoren en een toegewijde communicatie netwerk, tot een (semi-) programmeerbare multiprocessor die meerdere processen in parallel kan draaien door het gebruik van beide interleaving en overlapping.

Ondanks dat de mogelijkheden voor het ontwerpen van de complexe en geavanceerde embedded systeem platformen zeer groot zijn, een passende methodologie die voldoet aan alle door de markt gesteld eisen is nog niet naar voren gekomen. Dat wil zeggen dat het specificaties opstellen, onderzoeken en ontwerpen van de applicatie multiprocessor systeem platformen gebaseerd op de gebruikers behoeftes is nog altijd qua tijd en hoeveelheid werk een kostbaar proces dat moet ingezet worden.

Ons antwoord op de bovengenoemde uitdagingen is de Archer aanpak - aanpak beinvloedt

door de ’Abstratie pyramide’ en Y-chart. Het hoofd doel van deze aanpak is om de performance analyse te ondersteunen door de executie modelling van parallel (streaming) applicaties op candidate multiprocessor architecturen. Onze aanpak heeft drie kernelementen: (1) representatie van de applicatie, (2) platform - gebaseerde bibliotheek van de (architecture) componenten, en (3) een mapping methodologie. Symbolic Program representatie dat was geiïntroduceerd door het onderzoek gepresenteerd in dit proefschrift is een brug tussen de twee werelden, die van de architecten en die van de ontwerpers. Dit Proefschrift brengt abstractie van functionele details maar toch zorgt voor het behoud van andere elementen zoals communicatie, computatie, control en dependencies. Archar architectuur representatie ondersteunt het modelleren van brede scala van systeem architecturen (bij voorbeeld all-in-hardware, all-in-software, hybrid multiprocessor, with dedicated network, shared-bus, or highway, burst-bus, or hybrid network). Ten slotte, onze mapping methodologie werkt met alle bovengenoemde representaties en toch vertaalt applicatie Symbolic Programs tot Archer architectuur executie threads. Het waardevol om te vermelden dat deze transfromaties lijken op de stappen dat de designers en de ontwerpers nemen in de process van mapping van hun applicatie specificaties op de echte-wereld architectures. Dit was aangetoond door een paar voorbeelden waar de verschillende digitale en beeld verwerking standaarden op de verschillende Multi-Processor (MP) Systems-on-Chip (SoC) zowel toegewijde (application specific) als programmeerbare (met embedded OS) architectures zijn geïmplementeerd.

Curriculum Vitae

Vladimir Dobrosav Živković was born on October 18th, 1970 in Aleksinac, Serbia (Federal Republic of Yugoslavia).

In 1989 he received his high school diploma at The Gymnasium of Mathematics and Natural Sciences “Svetozar Marković, in Niš, Serbia (Federal Republic of Yugoslavia).

Between years of 1989 and 1990 he served a mandatory service of the Yugoslav People’s Army.

In October 1990 he started his studies in the Faculty of Electronic Engineering at the University of Niš, Serbia.

In November 1995, Vladimir received his Dipl.Ing. (Graduated Engineer) degree in Electronics and Telecommunications from the Faculty of Electronic Engineering at the University of Niš.

In January 1996, he commenced post graduate studies. From 1996 till 2000, Vladimir took part in a Technology Innovation Projects run by the Serbian Ministry of Science and Technology. These projects were in the industrial control and automation and data acquisition fields.

During his post graduate studies, he worked at the same faculty as a Research and Teaching Assistant.

He received his M.Sc. (Magistar) degree from the Faculty of Electronic Engineering at the University of Niš in May 2000 after successfully defending his master thesis titled ”Graphical Functions in Real Time Operating Systems with Rigid Time Limitations”.

In August 2000, he joined the Leiden Embedded Research Center (LERC) which is a part of the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University, the Netherlands, where he was appointed to the post of Research Assistant.

Starting from March 2001 till June 2004, he was also a guest researcher at Philips Research (Natlab) Eindhoven where he participated in projects run by the Embedded Systems Group.

He was involved in the `Archer` project, a Philips Semiconductors (now NXP) sponsored project, which deals with ARCHitecture ExploRation of Embedded Multiprocessor Systems. As a member of the `Archer` project he conducted research in the context of application representations for stream-oriented media and DSP applications, modeling of the the parallel multiprocessor architectures, and mapping the former onto the later. In particular, he worked on replacing abstract symbolic instruction trace methods with symbolic programs and improving abstract and system-level architecture models. This research work ended in June 2004 and the results are documented in this Ph.D. thesis.

In June 2004, Vladimir became Principal Embedded System Engineer in Irdeto (formerly Irdeto Access) Hoofddorp, the Netherlands, a Conditional Access company. He worked on designing and developing Smartcard (DVB) and Softclient (IPTV) products, for which he was using advanced applied cryptography, code-protection, as well as software and hardware tools and standards.

In February 2008, Vladimir joined IBM Netherlands, as an Advisory Software Engineer and Event Source Architect, as a part of the IBM Software Tivoli Group.