

CONTEXT-FLOW SYSTEM-ON-CHIP PLATFORMS

by

Rami Beidas

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2004 by Rami Beidas

Abstract

Context-Flow System-On-Chip Platforms

Rami Beidas

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2004

Recent evolution in system-on-chip (SOC) design methodology demonstrates an important omission of a design principle that directly contributes to the success of conventional computer systems: the use of a simple programming model to separate application from architecture. In an effort to restore the powerful concept of programming in the context of system-on-chip, in this work we propose a new programming model, called context-flow, that is general-purpose, simple, safe, highly parallelizable yet transparent to the underlying architectural details. A high-performance SOC platform architecture is then designed to support this programming model, while fully exploiting the physical proximity between the processing elements. Using an architectural exploration framework with a multi-processor simulator, our case studies on real life applications demonstrate the feasibility of adapting imperative C programs to context-flow programs, as well as the performance efficiency of our context-flow architecture over bus-based and packet-switch-based alternatives. Finally, we propose an analytical performance model, based on queueing networks, for the new SOC platform architecture that is simple, synthesis-friendly, and as flexible and powerful as queueing theory.

Dedication

*To my parents,
who dedicated their life for their family*

Acknowledgements

First, I would like to thank my advisor Professor Jianwen Zhu for his advice, guidance, and support. He was always willing to set aside any amount of time to clarify issues relating to my work. His continual patience and encouragement were invaluable for the successful completion of this thesis. I am looking forward for the next few years of work under his wise guidance.

I express my appreciation to Prof. Tarek S. Abdelrahman, Prof. Gregory Steffan, and Prof. Wai Tung Ng, for their comments and advice, and for serving as members of the thesis committee.

To my family, thank you for your love, support, patience, and in believing in my ability to go the distance.

Last but not least, to my friends from lab EA306: Zhong, Fang, Linda, Dennis, and Silvian, thanks for all your help and support.

Contents

1	Introduction	1
1.1	Background	1
1.2	State-Of-The-Art	5
1.3	Contributions	6
1.4	Thesis Organization	8
2	Background and Related Work	9
2.1	Programming Model	9
2.2	Platforms	11
2.2.1	Platform Types	12
2.3	Network-On-Chip	15
2.4	Performance Estimation	19
3	Context-Flow Programming Model and Architecture	21
3.1	Context-Flow Programming Model	21
3.2	Context-Flow Architecture	25
3.2.1	Macro-Architecture Specifications	25
3.2.2	Possible Organizations of CFA On-Chip Network	27
3.3	Tunnel-based CFA	29
3.3.1	Memory System	30
3.3.2	Interconnect	33

3.3.3	Processing Elements	35
3.4	Context-Flow SOC Design Example	35
3.5	Discussion	36
4	Performance Evaluation Framework	42
4.1	SimpleScalar Tool Set	42
4.2	Sim-CFA	45
4.3	Architecture Configuration and Application Compilation	49
5	Queueing-Theoretic Performance Model	53
5.1	Queueing Networks	54
5.2	Analytical Performance Model	56
5.2.1	The Modelling Process	56
5.2.2	Stochastic Model	56
5.2.3	Derivation of Analytical Performance Metrics	58
5.3	Discussion	60
6	Case Studies	62
6.1	Target Applications	62
6.1.1	Overview of MPEG1-LayerIII Decoder	63
6.1.2	Cryptography Accelerator	64
6.2	Data Transformation	65
6.2.1	Data Transformation of MP3 Decoder	65
6.2.2	Data Transformation of the SSL Processor	68
6.3	Performance Experimental Results	69
6.3.1	Simulation Results of the SSL Accelerator	69
6.3.2	Simulation Results of MP3 Decoder	77
6.4	Evaluation of Queueing-Theoretic Performance Estimation Model	78

7 Conclusion and Future Work	84
7.1 Conclusions	84
7.2 Future Work	85
Bibliography	86

List of Tables

6.1	Detailed Workload Distribution of SSL Processor (%)	70
6.2	Average Processing Time Per Packet by Each Procedure (cycles)	71
6.3	Effective Processing Time Per Packet by Each Procedure (cycles)	72
6.4	SSL Mappings given as Target PE for Each Procedure	73
6.5	MP3 Decoder Results	79
6.6	SSL Accelerator Mappings for Performance Model Evaluation	80
6.7	MP3 Decoder Mappings for Performance Model Evaluation	80
6.8	Simulated and Estimated Residence Time for SSL Accelerator	82
6.9	Simulated and Estimated Residence Time for MP3 Decoder	83

List of Figures

1.1	Design Productivity Gap [64]	3
1.2	Current and Future On-Chip Interconnection Architectures	5
2.1	Platform Types, (a) Full Application, (b) Processor-Centric, (c) Communication-Centric	13
2.2	Philips Nexperia Multimedia Platform	14
2.3	Texas Instruments OMAP Wireless Platform	15
2.4	Sonics' SiliconBackplane Primary Components	16
2.5	Ring, Mesh, and Torus Topologies	16
2.6	On-Chip Interconnection Architecture in Raw Microprocessors	18
3.1	Context	22
3.2	Context-flow API.	23
3.3	Context-Flow Architecture Instruction Set.	27
3.4	Alternative Implementations of Context-Flow Architectures	28
3.5	Tunnel-based CFA Block Diagram	29
3.6	Memory Management Unit (B : Busy, F : Free)	30
3.7	First-In First-Out Queue (FIFO)	31
3.8	Memory Module	32
3.9	Crossbar	34
3.10	Self-routing Crossbar	35

3.11	A Simple Design Example	38
3.12	System Input	39
3.13	System Behavior to the Input Described in 3.12 (a)	39
3.14	System Behavior to the Input Described in 3.12 (b)	39
3.15	System Behavior to the Input Described in 3.12 (c)	40
3.16	System Behavior to the Input Described in 3.12 (d)	40
3.17	System Behavior to the Input Described in 3.12 (e)	40
3.18	System Behavior to the Input Described in 3.12 (f)	41
3.19	System Behavior to the Input Described in 3.12 (g)	41
3.20	System Behavior to the Input Described in 3.12 (h)	41
4.1	SimpleScalar Tool Set Overview	43
4.2	State Variables for the sim-safe Version of the Simulator: (a) regs (b) mem	44
4.3	The Original SimpleScalar Simulator Core	45
4.4	The Modified SimpleScalar Simulator Core	46
4.5	The Actual Simulated System	47
4.6	SimpleScalar PISA Instruction Format	48
4.7	Activity Monitoring in Sim-CFA	49
4.8	An Example Configuration File “config.dat”	51
4.9	A Context-Flow Version of the Simple Array Processor	52
4.10	Sim-CFlow Simulation Process	52
5.1	A Queueing Network	55
5.2	An Example of Total Propagation Time for Job Class is a CFA System	61
6.1	MP3 Decoder Stages	63
6.2	Crypto Accelerator Flow	64
6.3	MP3 frame format	65
6.4	Original Data Structures in the Reference MP3 Decoder Implementation	66

6.5	Transformed Data Structures in the Context-Flow Implementation of the MP3 Decoder	67
6.6	Simulation Results for SSL Acceleration Processor: Average Packet Processing Time	74
6.7	Simulation Results for SSL Acceleration Processor: Average PE Utilization . .	75
6.8	Performance Results of Single-PE and Multi-PE Implementations of the SSL Accelerator	76
6.9	Performance Results of Homogeneous and Heterogeneous Implementations of the SSL Accelerator	77
6.10	Mapping Effects on System Performance for the SSL Accelerator	78
6.11	Queueing Model Accuracy for SSL Accelerator and MP3 Decoder	81

Chapter 1

Introduction

1.1 Background

In the mid-1990's, the semiconductor fabrication process technology had achieved the scales of 0.35 and 0.25 μm , allowing the major components of an electronic product, traditionally implemented as chip sets on printed circuit boards (PCBs), to be placed on a single die. Although these designs were far from being complete systems, since it was not clear whether it would be profitable to integrate analog and passive subsystems, the term System-On-Chip (SOC) was used, arguably as a marketing term [44]. It was by the turn of the millennium, with 0.18, 0.15, and 0.13 μm technology in hand, when a single-chip implementation of real systems became possible. SOC technology is currently being utilized in small, increasingly complex consumer electronic devices, such as digital cameras and mobile phones, and making its way into higher-performance multimedia and telecommunication designs, such as image processors and network routers.

According to the 2001 International Technology Roadmap for Semiconductors (ITRS) [29], SOCs crafted in 45 nm technology will have a 4 billion transistor budget and run at 10 GHz. Such massive transistor budget and switching speed will enable the realization of complex applications previously impossible to implement on a single chip, and achieve a high degree of

integration, parallelization and performance measures.

Along with the great advantages promised by the SOC theme, mapping complex applications on a single die pose plenty of challenges to designers and EDA tools developers. These challenges can be classified into two major categories, *silicon complexities* and *functional complexities*. Silicon complexities refer to the difficulties associated with crafting synthesized functionality in a target advanced technologies. These challenges include deep submicron effects – such as interconnect delay, power density, mixed signal integration, etc. The second category, functional complexities, refers to the difficulties associated with the transformation of functional system specifications into efficient implementations within the overall design constraints. These challenges include rapidly-shrinking time-to-market (TTM) windows and short product life cycles, performance/area/power estimation, verification, automatic testing, etc.

The EDA industry has been striving to address silicon complexities with new methodologies and tool families such as the *Galaxy Design Platform* from Synopsys [68] and the *Encounter Digital IC Design Platform* from Cadence [12]. However, it is the functional complexities category that was much less defined and understood by the community. These complexities effects are already in realm, worsening what is known as *design gap* between what the industry is capable of manufacturing and what design teams are capable of designing (Figure 1.1). Such challenges could potentially make the development of multi-billion transistor SOCs infeasible.

To cope with this design productivity crisis, and in search of a scalable design methodology that can dramatically reduce design cost and time, the *intellectual property (IP) assembly paradigm* was proposed in the mid-1990's as *the* solution to overcome SOC functional complexities. IP assembly paradigm advocates the use of pre-designed, characterized, and verified components with a well-defined functionality and interface in the development of SOC applications.

Unfortunately, IP reuse alone did not achieve the dreamed revolution. Apart from processor

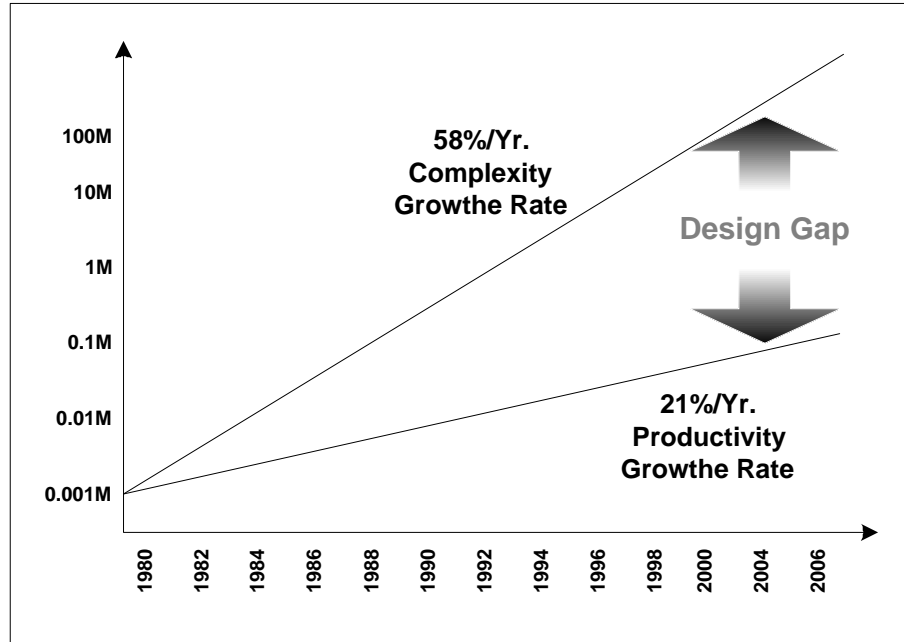


Figure 1.1: Design Productivity Gap [64]

core IPs – such as ARM, MIPS, and ARC, third-party IP reuse was, in general, declined by the industry for several reasons. These reasons include the lack of standard IP models, lack of standard interfaces, questionable design and verification quality, and most importantly, the lack of an overall integration methodology and design flow that supports this reuse vision. What we need is a complete well-defined design methodology based on the concept of design reuse at higher levels of abstraction, beyond that of IP reuse. As a result, the notion of *platform based design* emerged.

The *platform-based (PBD) design paradigm* has been attracting lot's of attention in the recent past. Although we do not have a universal definition of the concept of PBD, it is agreed that the main idea behind platform-based designs is the shift from from-scratch customized hardware designs towards flexible, stable, pre-defined hardware architectures that can support a variety of applications via programmability and/or configurability [20, 61].

The notion of PBD is expected to be the next trend in SOC design. Starting with a pre-defined, verified, charaterized architectural framework results in an increased likelihood of

first-time silicon success, elimination of many of the physical problems encountered in the ultra-deep submicron technology, and reduced verification cost, while sacrificing very little, if any, in the performance, power, and area metrics. In fact, several platforms, with various properties and configurability degrees, have already found their way to successful industrial applications [59, 28, 79, 2, 4, 67, 78].

One of the major concerns related to the design of future SOCs is the scalability of on-chip communication architectures for larger systems. The success of future SOC platforms rely on the ability to interconnect system components in an efficient manner. In fact, it is expected that the on-chip physical interconnections will present a limiting factor for performance and energy consumption [9]. So far, the industry has been using common busses and ad-hoc communication channels to interconnect system components [5, 27]. Such global-wiring communication architectures are unable to scale with the large dies fabricated in the near future technology [25]. To overcome this problem and accommodate the massive parallelism requirements of future applications, researchers proposed the use of interconnection networks-based hardware platforms, previously used to interconnect the components of high-performance parallel computers with multiple processors, to fulfill on-chip communication requirements.

Several critical issues were ignored in these propositions. In this thesis we propose solutions to essential key issues to provide gains in design productivity and system performance. In particular, we focus on the development of a programming abstraction and a common communication-centric performance-oriented SOC platform targeted at massive parallelism, integrity, and data movements, which will be of central interest for future synthesized applications.

The subsequent sections of this chapter will briefly describe recent propositions in the industry and academia to interconnect SOC components. Motivation of the work presented in this dissertation will be discussed before summarizing our main contributions to the field.

1.2 State-Of-The-Art

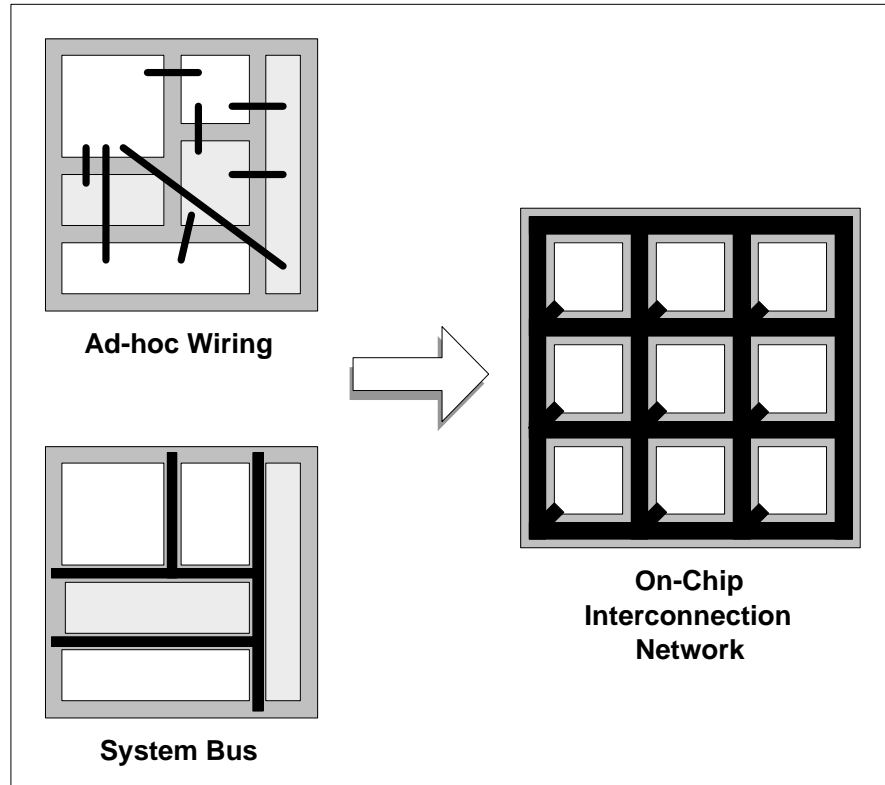


Figure 1.2: Current and Future On-Chip Interconnection Architectures

Today's trend in interconnecting SOC components is the use of system buses and custom wiring interconnects (Figure 1.2). Several industrial platforms based on this approach were proposed in the past few years [5, 27, 78, 1].

Researchers in the academia recognized the scalability limitations of today's on-chip communication architectures. First, a system bus is a communication bottleneck. Contention over variations of such communication media seriously limits the ability of reaching potential parallelisms of target applications. Second, technology scaling is rapidly making wires a critical performance limiting factor. As a result, contemporary platforms, that depend on global interconnects, will suffer with the larger wire delays [14].

To overcome these challenges, researchers proposed new hardware platforms that provide

high-bandwidth and physical locality with respect to data movement. These platforms were based on interconnection networks, previously used to interconnect supercomputer components [18]. These on-chip networks interconnect equally sized tiles, not necessarily homogeneous, using regular interconnection topologies, such as torus and mesh networks [73, 15, 41, 35]. Lot's of research effort is currently being spent on designing these network micro-architectures [58], developing simulation environments [75], and analyzing the systems power and performance [26].

When the whole design flow is defined, a designer would *ideally* plug in a set of processing elements and embedded processors in the interconnected tiles of these platforms to arrive at a stable, parallel, high performance design.

1.3 Contributions

While a burst of efforts have appeared under the banner of network-on-chip, we observe some common, yet important omissions.

First, while traditional computer architecture is well abstracted with a programming model, new SOC architectures have not made much progress on that front. An SOC platform is either modeled in system-level languages, such as SystemC [69] or SpecC [22], where a distinction between application, architecture and hardware does not exist, or using traditional parallel programming models, which are usually very complex. For example, the popular Message Passing Interface (MPI) programming model [51] defines an API with 127 C functions and there is no easy path to parallelize a sequential program into an MPI program other than the use of array-oriented scientific applications.

Second, while traditional networks in supercomputers are designed with the bandwidth limitation imposed by chip pin count, new SOC platforms do not take full advantage of the much relaxed physical constraints and almost unlimited on-chip bandwidth.

We propose a solution that is both old and new: old in the sense that it restores the impor-

tant principle that architectures should be equipped with a simple programming model; new in the sense that a unique programming model in which SOC applications can be developed, a unique architecture that delivers high-performance to this programming model, and collectively a complete SOC platform, are proposed for the first time. It is important to note that in the context of SOC, our application does not imply “software running in user mode”, as in traditional computer systems or as in [61]; instead, parts of our application can be mapped to hardware running below the operating system layer in final implementation.

More specifically, we make the following contributions. First, in contrast to the common practice of extending the C language with new syntax (thus difficult to accept) or APIs (thus difficult to analyze) with explicit parallelism, or introducing model of computations (MOCs) that depart dramatically from the imperative programming model, we introduce a new programming model revolving around a new concept designed to abstract autonomous data structures, called *context*. The model is *familiar*: its execution semantics add nothing more to the imperative, sequential programming model – any C program is a special-case of context-flow program and any context-flow program can be compiled by a conventional C compiler. It introduces nothing more than **two** additional C library functions for the replacement of `malloc`, as well as a discipline of memory accesses that can be enforced by a checker added to the compiler proper. The model is *simple* to program. It is significantly easier than parallel programming: there is no need for explicit parallelism. It is even easier than sequential C programming: there is no need for explicit memory management. The model is also *safe* in the sense of Java – it is free of problems such as free memory access and dangling pointers.

Second, we propose a new SOC platform architecture, called the *context-flow* architecture, revolving around an on-chip network infrastructure called a *tunnel*, which takes full advantage of the physical proximity of tightly coupled processing elements. The tunnel implements the on-chip remote procedure call (RPC) abstraction, therefore achieving the *transparency* of the programming model, since an application does not have to change with respect to the change in the underlying architecture, yet with a cost almost as cheap as local procedure calls, thereby

achieving performance efficiency.

Third, we build a development suite by extending the popular SimpleScalar environment, which was designed for single processor architecture evaluation, so that complex applications can be compiled and simulated on the multi-processor context-flow architecture platform. This environment enables the architectural exploration of real world applications.

Finally, we developed a static queueing-theoretic performance model for the tunnel-based on-chip network. The model is simple, fast, and synthesis friendly.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides the reader with the necessary background on concepts discussed in this thesis and reviews related work in this field. Chapter 3 describes the newly proposed context-flow programming model and architecture. The details of our performance evaluation framework are presented in Chapter 4 followed, by that of queueing-theoretic performance model in Chapter 5. Test cases and experiment results are described in Chapter 6. Finally, Chapter 7 concludes the thesis along with suggestions of future work directions.

Chapter 2

Background and Related Work

This chapter provides background information for the material presented in this thesis. It also summarizes related efforts and compares them to the work done in this study.

First, the concept of a programming model is presented to familiarize the reader with the ideas behind an important proposition in this work. Second, the recent propositions in the field of platform based design, with emphasis on communication-centric platforms, are summarized. And finally, previous work in performance estimation of SOC platforms is briefly outlined.

2.1 Programming Model

A programming model is an abstraction that separates application from architecture. This separation is important to allow applications be developed and reused across different architectures, and vice versa.

A programming model can be defined at different levels of abstraction, and a hardware/software infrastructure is usually needed to support such abstraction. For example, an instruction set is a programming model defined at the low level to abstract away architectural details such as pipelining and out-of-order issue, and a massive amount of hardware logic is used to realize this abstraction. A programming language is defined at the higher level to abstract away the differences between different instruction sets, and a compiler is used to realize such abstrac-

tions. For the same programming model, a middleware infrastructure, such as CORBA [53] or DCOM [48], can be used to abstract away architectural details of a distributed environment to implement a distributed application the same way as a sequential one.

The concept of programming model, however, has been ignored in the hardware-centric CAD community. Even though platform-based design is advocated to allow the reuse and customization of pre-aggregated components, the concept of platform has not been formalized with a programming model for applications. Recent interest in building the communication infrastructure on massive parallel SOC has led to the concept of network-on-chip (NOC) [15, 35, 24]. Building a programming model for network-on-chip either has to use explicit communication with send/receive system calls, a wide departure from the traditional imperative programming model, or has to build another middleware infrastructure, by cloning the traditional layered protocol stack concept, on top of the network, leading to performance degradation with the number of layers one communication session has to go through.

At the time of writing this thesis, many researchers in academia are raising the point of the need for a programming model for future SOC platforms. A recent article in EETimes [33] argued the need for such an SOC abstraction where an application can be developed, based on the abstraction, without referring to the lower level details.

In spite of all this attention, very little work has been reported regarding this issue. A well-known work from this perspective is associated with the MESCAL project, which stands for Modern Embedded Systems, Compilers, Architectures, and Languages [34] developed within the Gigascale Silicon Research Center (GSRC). The MESCAL project targets all aspects of the design of programmable, platform-based systems for specific application domains along with their software development tools. Kurt Keutzer et al. presented some important concepts of system design within the MESCAL framework in [34]. One of the most crucial concepts discussed in this article was that of programming model. The authors only gave a description and discussed an outline of the properties the programming model to be developed. Their approach in designing the model is to combine a top-down view and bottom-up view. Top-down view en-

ables the programmer pass his knowledge about bit-level and task-level parallelism along with some hints on task-level scheduling, binding, and synchronization, which were traditionally managed by the operating systems. While bottom-up view exports some of the architecture to enable the programmer to efficiently program the hardware platform. At the time of writing this thesis, no details are reported on the features of the developed programming model as the target platform is still under development. Although it is still early to comment on the proposed model, it is obvious, from the discussed features, that a considerable effort is asked from the system developer.

We do believe that there is a need for the adoption of programming model concept for future SOC platforms. This model should be high-level, to insure development productivity and flexibility, architecture-independent, to insure portability, and optimization friendly, to satisfy the performance requirements placed upon future systems.

2.2 Platforms

An architecture is an aggregate of components such that an application can be executed or implemented through a well defined programming model. A *micro-architecture* is an aggregate of components such as fetch stage, decode stage, execution stage and memory stage to implement a sequential application in C or other programming languages by its instruction set. On the other hand, a *macro-architecture* is an aggregate of components, such as processing elements (PEs) and memories, to implement a parallel application by a programming model. The composition of a macro-architecture in the case of SOC is often customized according to one application or one family of applications. In that case the generic macro-architecture is referred to as a *platform*, and its customization is called a *platform instance*, or *platform configuration*. A macro-architecture is said to be *homogeneous* if all PEs are of the same type, e.g., processors, and *heterogeneous* if PEs can be microprocessors, DSPs, ASIPs or custom hardware cores.

Platform based design is a design methodology proposed for adapting to the increased challenges in the semiconductor industry. There have been many definitions of the notion of platform, and the one listed above is what a platform is from our perspective. However, it is agreed that the main idea behind platform-based designs is the shift from from-scratch customized hardware designs towards flexible, stable, predefined hardware architectures that can support a variety of applications via programmability and/or configurability [20, 61]. In other words, a design project will start with a hardware architecture that is not fully specified, leaving some degrees of freedom to the system designer to choose/design/configure components for the final implementation.

With the absence of a universal definition of platforms, Frank Schirrmeister, Martin Meindl, and Stan Krolikoski identified different classes of platforms [62]. This classification helps in clearing up some of the confusion when discussing platforms currently under investigation by the industry and academia. In this work, we report a similar classification which, however, concentrates on the degrees of design freedoms associated with each category.

2.2.1 Platform Types

We identify three major types of SOC platforms, presented below in an increasing order in terms of flexibility, cost, design time, and risk (Figure 2.1).

- **Full Application Platforms:** In these platforms, complete applications can be developed on top of a fixed architecture using a comprehensive software development environment with a high-level programming entry, such as C/C++. These platforms usually consist of one or more embedded processors, such as ARM or MIPS, and a number of application-specific modules, such as MPEG decoders and peripheral interfaces, connected via a set of system busses. Examples of such platforms include Philips Nexperia [59] (Figure 2.2) and Texas Instruments OMAP [28] (Figure 2.3).

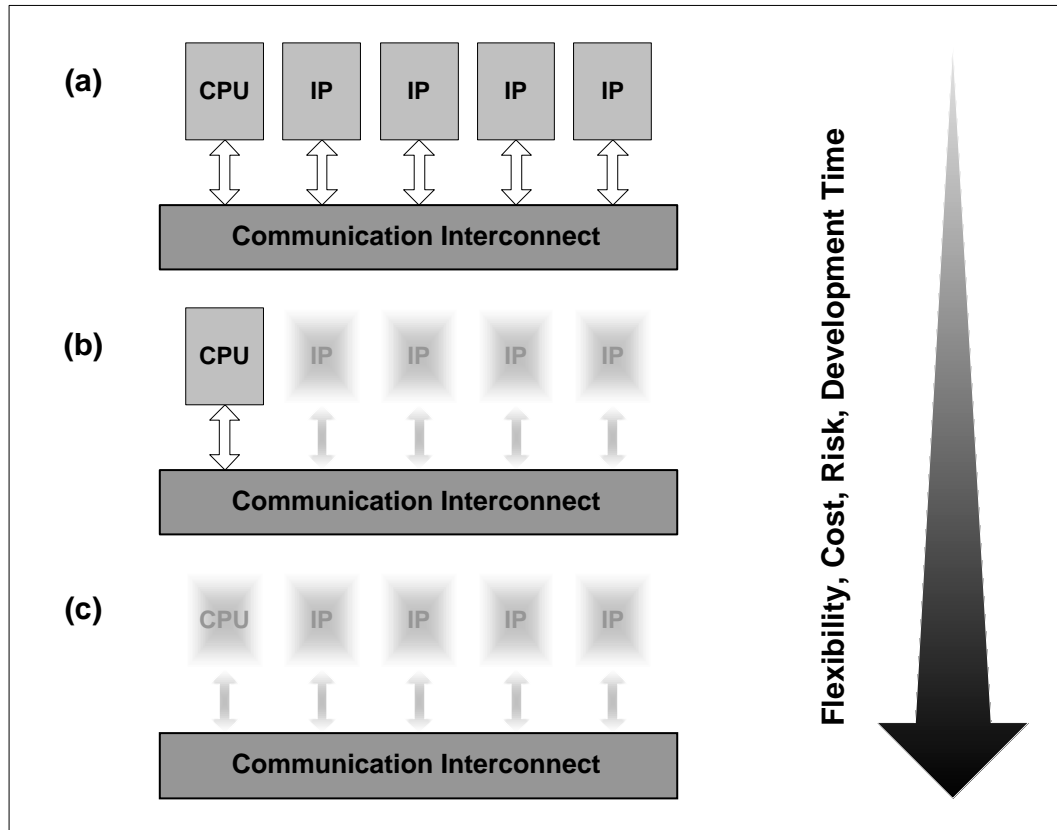


Figure 2.1: Platform Types, (a) Full Application, (b) Processor-Centric, (c) Communication-Centric

- Processor-Centric Platforms:** These platforms are centered around an embedded processor, focusing on software access to user-defined hardware modules, such as MPEG decoders and encryption engines, via system buses or specialized channels. Compared to full application platforms, these platforms provide the designer with the flexibility of specifying, and possibly in-house building, the exact type and number of system co-processors at the cost of development overhead. StarCore [67] and ARM PrimeXsys platforms [4] are examples of this platform category.
- Communication-Centric Platforms:** These platforms only define the communication fabric to interconnect the components of these platform instances. The system designer has to provide the storage and processing blocks to implement the target application.

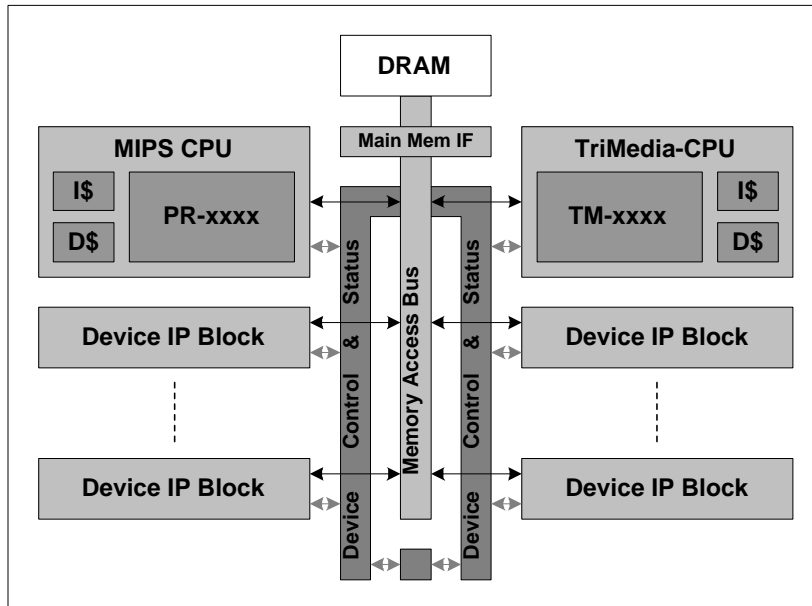


Figure 2.2: Philips Nexperia Multimedia Platform

The design does not even need to have a programmable processing core. Sonics' Smart Interconnect is an example of such platforms [78], which provides many advantages over traditional system busses.

One of the major difficulties associated with the design of future SOC platforms is the scalability of on-chip communication fabrics for larger systems. The success of future SOC platforms rely on the ability to interconnect system components in an efficient manner. In fact, it is expected that the on-chip physical interconnections will present a limiting factor for performance and energy consumption for chips manufactures in the ultra-deep submicron technologies [9]. This comes from the technology scaling fact that each time line widths halve, the devices get approximately twice as fast and a minimum width wire of constant length gets four times slower. State-of-the-art platforms use shared busses and ad-hoc communication channels to interconnect system components, as shown above. Such global-wiring communication architectures are unable to scale with the large dies fabricated few years from now [14].

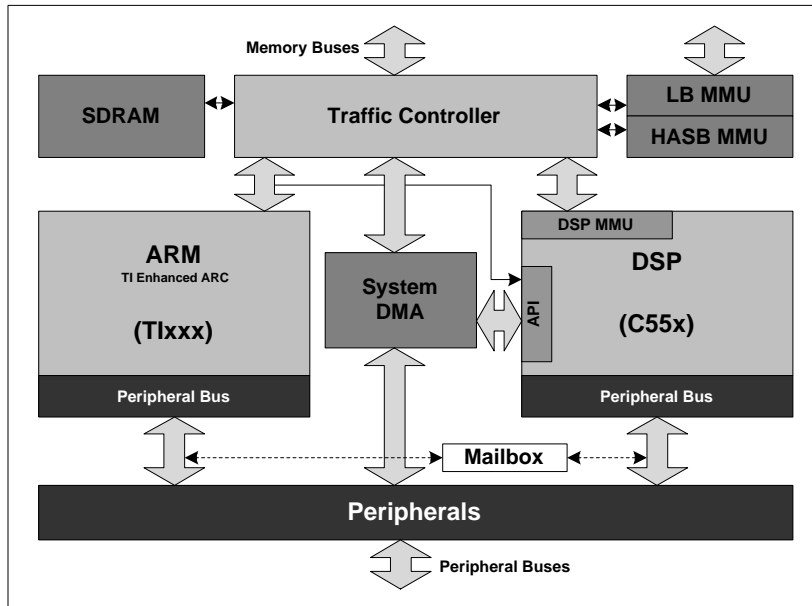


Figure 2.3: Texas Instruments OMAP Wireless Platform

To overcome this problem, and to accommodate the massive parallelism requirements of future applications, researchers started to investigate platforms that deal with the on-chip communication architecture as *the* design objective. Several platforms were proposed which use interconnection networks, traditionally used to interconnect supercomputer component, to fulfill communication requirements. Since then the term *Network-On-Chip* (NOC) emerged to become one of the hottest topics in the design of SOC platforms.

2.3 Network-On-Chip

An interconnection network consists of a set of nodes (or routers) and a set of links (or channels). Most interconnection networks are buffered, i.e., the routers contain storage for buffering messages when they are unable to obtain an outgoing channel. An interconnection network can be defined by four parameters – its topology, the routing algorithm governing it, the flow control protocol, and the router micro-architecture. First, the topology of a network concerns how the nodes and links are connected. It dictates the number of alternate paths between nodes, and

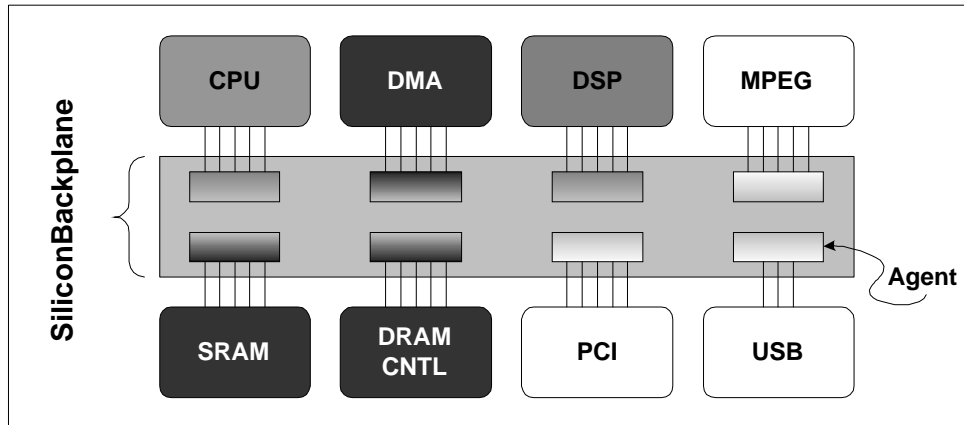


Figure 2.4: Sonics' SiliconBackplane Primary Components

thus how well the network can handle contention and different traffic patterns. Figure 2.5 shows various topologies which have been adopted commercially. The routing algorithm determines

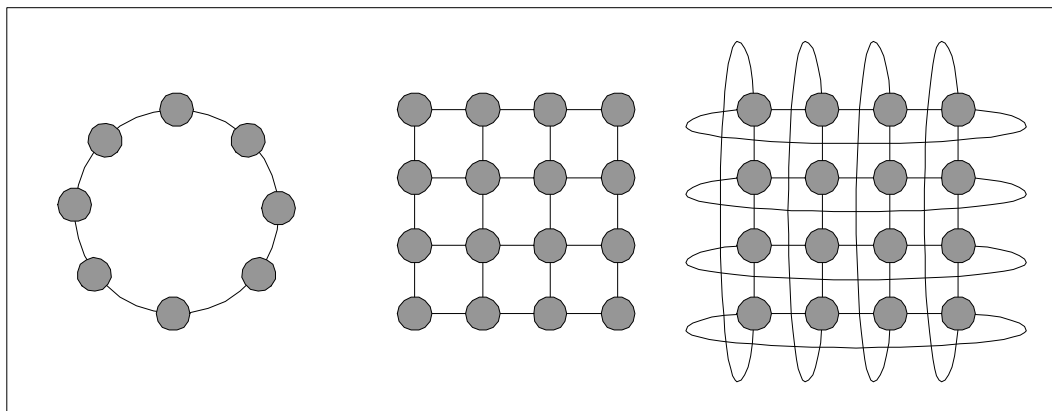


Figure 2.5: Ring, Mesh, and Torus Topologies

the actual path, among the ones possible within a given topology, traversed by a message. Flow control dictates when a message gets to leave a router through the desired outgoing channel, and when it gets buffered and has to wait. Finally, router micro-architecture specifies how a router is built. This parameter has great effect on the overall performance of the network. See [18, 16] for further details on interconnection networks.

Interconnection networks were designed to satisfy the high-performance and massive-parallelism

requirements of demanding supercomputers. These networks are starting to replace global-wiring architectures of SOC platforms, which pose the same demands, as the latter started to reach their bounds.

The MIT Raw machine, a fully programmable platform, was one of the earliest designs to utilize on-chip interconnection networks [73]. It uses synchronous 2-D mesh networks to connect an array of identical programmable tiles of RISC processing cores (Figure 2.6). Two logically distinct networks are multiplexed over the same set of physical wires – one static and one dynamic. The static network is programmable. It guarantees that the single-word communicated messages are available when needed, nearly the same speed as register reads, eliminating the need for explicit synchronization. The dynamic network handles traffic that cannot be accurately estimated at compile time. Messages routed on the dynamic network require the addition of message header, which contains flow control data. The Raw compiler tries to make the maximum utilization of the high-performance static network. The Raw design team claims the scalability of the proposed architecture to even 32x32 tiles. The basic idea was pushed further by replacing the RISC processing cores with custom logic processing elements using a very similar design flow [6].

Dally and Towles in [15] proposed a communication centric platform which used on-chip interconnection networks for future SOC where traditional interconnection techniques do not scale. They suggested the use of pipelined regular interconnection topologies, such as torus and mesh networks, as a means of communication between square tiles of identical dimensions, but not necessarily homogeneous. It was estimated that the networking overhead will consume a mere 6% of the overall system resources. Compared to Raw processors, only dynamic networks routing data packets/messages will be used, as opposed to the static networks on Raw. A similar architecture, which defines the layers of the protocol stack, was presented in [35, 49].

A different approach to on-chip communication is considered in the MESCAL project. It maintains a layered view of the custom SOC communication architecture similar to the OSI reference model to abstract low-level details of the interconnection configuration [65]. In fact,

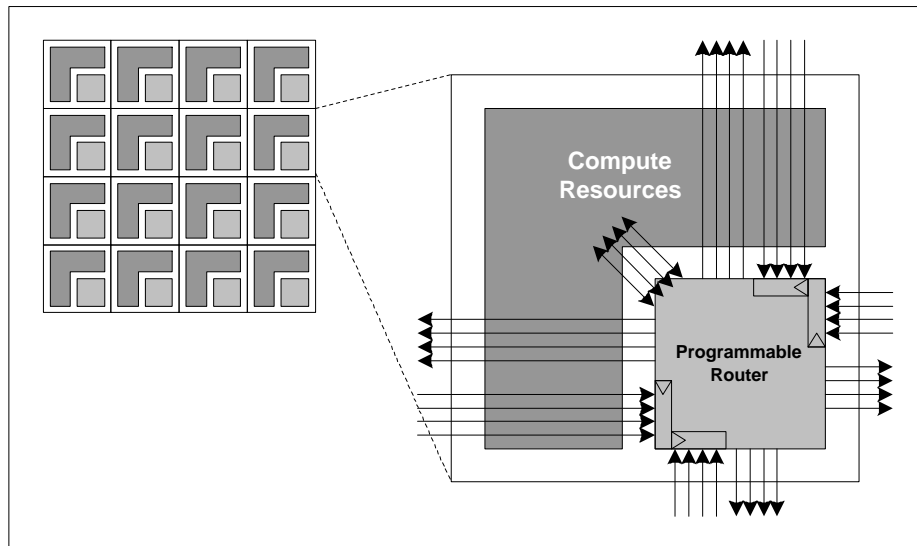


Figure 2.6: On-Chip Interconnection Architecture in Raw Microprocessors

it is one of the very few approaches that currently support the use of irregular interconnection topologies as on-chip communication architectures even at the system level.

As opposed to the previous efforts which adapt a flat on-chip networks, Hui Zhang et al. viewed the advantage of hierarchical interconnection networks for low-power DSP chips by illustrating the power savings of 2-level mesh networks when compared to a single level implementation of similar topologies [83]. A 2-level hierarchical network was also used on Smart Memories [41].

The adoption of NOC to achieve efficient communication on future SOC's has gained a universal acceptance, and massive research efforts have appeared under the banner of this notion. Peh proposed new flow control and micro-architectural mechanisms for extending the performance of on-chip interconnection networks [58]. Wang et al. proposed micro-architectural power optimizations for switching routers [74]. Ye and De Micheli proposed an automated physical floorplanning methodology suitable for various on-chip network topologies [80]. Jerjaya et al. defined a complete wrapper-based communication network design flow for SOC components integration in [31]. Hu and Marculescu investigated mapping algorithms that tar-

get the power/performance optimization problems for the regular communication architecture [26]. Dumitras et al. proposed fault-tolerance optimization algorithm through system configuration [19]. Murali and De Micheli proposed mapping algorithms for mesh NOCs under bandwidth constraints to minimize communication delay [52]. Jalabert et al. presented a tool for automatically instantiating an application-specific NOC to match communication patterns among components of heterogeneous SOCs [30]. Pande et al. proposed butterfly fat tree architecture that also targets the scalability problem of global wiring communication architectures, but also addresses multicasting requirements [56].

We do believe that the immediate migration of interconnection networks to SOC platforms ignores the characteristics of the new design environment. Using the available wiring resources and a better analysis of the applications running on heterogeneous architectures should result in a more suitable solution

2.4 Performance Estimation

While a rich literature on performance modeling in general has been reported [21, 42, 82, 81, 32, 45, 7], in the field of hardware-software codesign, very little work has been carried out focusing on the performance modeling of SOC architectures [60]. In this section, we give a brief review of those efforts. We start by first developing a taxonomy to help categorize these work.

- A performance model is *dynamic*, if it relies on the use of simulation. Otherwise, it is *static*, relying on some form of theoretical models. In general, a dynamic performance model is more accurate with respect to specific input trace. A static performance model is faster to evaluate.
- A performance model is *analytical*, or *architecture-aware*, if the result depends not only on the characteristics of the application, but also the architecture and how application

is mapped to the architecture. In general, an architecture-aware performance model is preferred for synthesis.

- A performance model is *automatic*, if it can be automatically constructed from the application and architectural mapping. It is *manual* otherwise.
- A performance model is *validated*, if its accuracy has been confirmed by detailed simulation.

Stochastic Automata Networks (SANs) were used in [43] to analyze application and derive probability distribution for various performance aspects of the target application. This model is static ¹, however, not architecture-aware. Furthermore, the construction of a SAN network from an application is not yet an automated process.

A static performance model for network packet processing architectures was derived in [70] using Network Calculus results. The proposed approach uses deterministic bounds to describe the arrival and service processes of the target system. The model is also analytical. However, the model is not yet complete in the sense that conflicts over communication resources are ignored, which could easily result in large errors of the estimated measures. As a result, estimation results of the test cases were not validated.

The work in [37, 38] proposes a hybrid static/dynamic performance analysis methodology for bus-based SOC communication architectures. Although the flow was validated and accurate estimates were reported, a speedup of only 2x over full hardware/software co-simulation was obtained.

We do believe that there is a great need for a static, architecture-aware, and automatically evaluated performance model that is usable in a system-level synthesis framework for future SOC platforms.

¹referred to as analytical by the authors.

Chapter 3

Context-Flow Programming Model and Architecture

A programming model is an abstraction that separates application from architecture. This separation is important to allow applications be developed and reused across different architectures, and vice versa. A programming model can be defined at different levels of abstraction, and a hardware/software infrastructure is usually needed to support such abstraction. In Section 2.1 we argued the need for a programming model for SOC platforms that is high-level, as opposed to explicit send/receive communication approach, but also efficient, as opposed to the OSI-like multi-layer protocol stacks. In this chapter we describe our new SOC programming model, called *context-flow programming model* along with a new SOC architectural platform that supports this programming model in an efficient manner, taking the drawbacks of recently proposed communication architectures into consideration.

3.1 Context-Flow Programming Model

Our programming model is based on a new data abstraction that we call a *context*. A context is defined as a set of dynamically allocated memory blocks closed under the point-to relation,

which captures the set of storage locations each pointer variable *may* point to. Figure 3.1 illustrates the invariant associated with the newly defined notion of context.

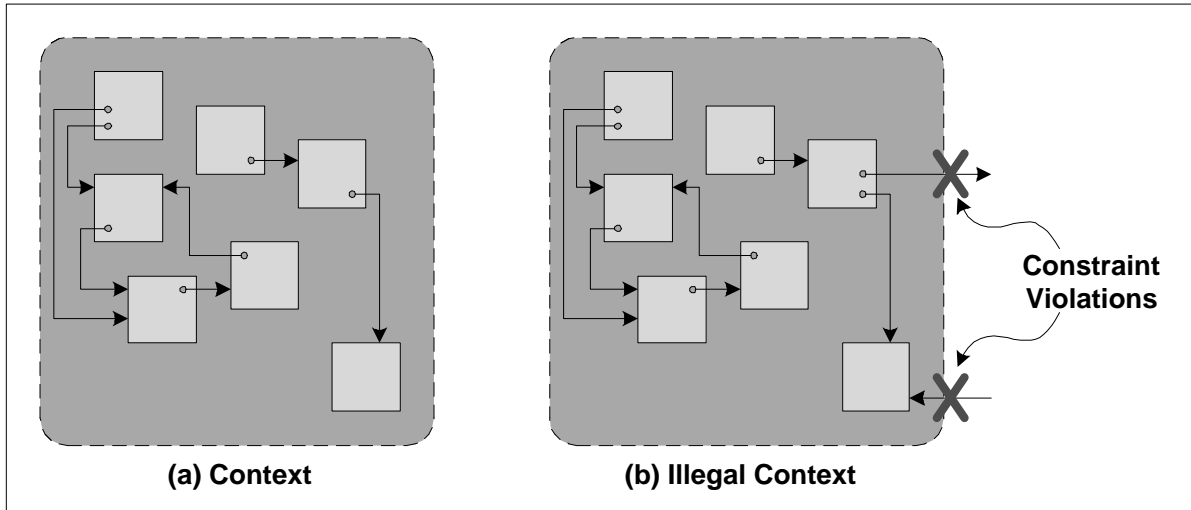


Figure 3.1: Context

The new programming model is formally defined in Definition 1.

Definition 1 Given a program with a set P of procedures¹, operating on a program state consisting of a set $B = G \cup L \cup H$ of memory blocks, where G corresponds to the set of global variables, L corresponds to the set of stack variables, and H corresponds to the set of heap objects. A block $b_i \in B$ is said to point to $b_j \in B$, if there exists a program point where the content of b_i contains the address of b_j . A context-flow program (CFP) is a tuple $\langle P, B, C, A \rangle$, where a set C of contexts forms a partition of the heap H such that any $c \in C$ is closed under the point-to relation, that is, any block b_i reachable from block $b_j \in c$ is also an element of c , and $\forall p \in P, A(p)$ gives the set of contexts p accesses. \square

Informally, we can view a context-flow program (CFP) as a set of procedures operating on a stream of dynamically allocated objects which satisfy the context closure property. And each context can be accessed by a single procedure at any point of time.

¹For convenience, here we distinguish between different invocations of the same procedure written in an imperative program.

The programming model is *general-purpose*. It adds nothing to the traditional imperative programming model other than an explicit partition of the heap into contexts, each of which is a data structure formed by dynamically allocated objects. These data structures can be anything ranging from arrays, linked lists, trees, graphs, or the combination of all. Any C program is therefore automatically a context-flow program with the trivial partition $C = \{H\}$. On the other hand, an application programmer can choose to refine this partition into multiple contexts, or multiple data structures. Thanks to the closure property, each of these data structures is *autonomous*, therefore any pair of data structures are also *disjoint*. The benefits of such partition shall become apparent later in the chapter.

Compared to other proposals, our context-flow programming model offers several important advantages. A CFP is extremely *simple*: it is simply a C program with the same sequential semantics. Therefore it can be compiled using any conventional compiler and executed on any conventional machine. Contexts can be implemented using the API shown in Figure 3.2. While the API consists of only two functions, it is the complete API seen by the application programmer. Here, `cfNewContext` creates a context and returns a unique identifier. `cfAlloc` allocates a memory block of certain size from the specified context. These API functions are intended as replacement of the standard `malloc` function in C for memory allocation.

<i>int</i>	<code>cfNewContext(void);</code>	1
<i>void*</i>	<code>cfAlloc(int c, int size);</code>	2

Figure 3.2: Context-flow API.

Given the invariant defined in our programming model, a *discipline* of context memory usage has to be imposed. First, an object in one context can only reference objects in the same context. Second, the context itself can only be referenced by the stack variables associated with that context. These constraints can be imposed by a program analyzer added to the compiler.

A discussion on how this analyzer can be constructed is outside the scope of this paper.

It is interesting to note that the API does not include any function to reclaim the memories allocated by `cfAlloc`, implying that a garbage collector is assumed. The availability of garbage collection not only simplifies programming, since the programmers are free of explicit memory management, but also leads to program *safety* in the same sense of Java. On the other hand, the closure property of contexts ensures that *cheap* implementations of memory allocation and garbage collection algorithms are available. In fact, our implementation of both algorithms confines the complexity to **constant runtime**.

- Garbage collection can be performed at the granularity of context, where all objects belonging to the same context will be reclaimed together. This preserves program safety – for example, there cannot be any references to freed memory blocks, since the memory containing the reference should belong to the same context, and therefore be freed already as well.
- Since CFP ensures that the contexts themselves are referenced only by stack variables, the test of whether a context becomes garbage can be performed at compile time by simple escape analysis, rather than at runtime. The invocations of context deallocation algorithm, which runs in constant time thanks to the fixed size of contexts, can be automatically inserted by the compiler.
- Since objects belonging to the same context will always be deallocated at the same time, the `cfAlloc` algorithm can again be implemented by a constant time algorithm: it simply increments a pointer, which points to the start of the current free space, by the requested amount.

Having described the general benefits of being general-purpose, simple, safe and efficient, we now discuss how our programming model satisfies the unique requirements and challenges of SOCs, which contain an unprecedented amount of silicon real-estate that can be exploited by applications for massive parallelism.

In contrast to a SpecC/SystemC specification, a CFP is highly *transparent* to the specific SOC architecture or circuit fabric chosen to implement the application. Therefore, the same application can be used for single processor, homogeneous multi-processor, or heterogeneous architectures. As the applications of SOC become increasingly complex, architecture transparency is key to reduce design cost and enable architectural exploration.

A CFP is considerably *easier to program* than parallel programming or multi-threading, since it does not need the explicit specification of parallelism – it just makes the job of identifying coarse-grained parallelism easier. A CFP is in fact highly *parallelizable* by a compiler: since data structures maintained by different contexts are disjoint, any procedures which access different sets of contexts can be run in parallel.

3.2 Context-Flow Architecture

3.2.1 Macro-Architecture Specifications

We consider the design of a macro-architecture, called the context-flow macro-architecture (CFA), formally defined in Definition 2.

Definition 2 *Given a context-flow program $\langle P, B, C, A \rangle$, a context-flow architecture (platform) is the set of all its platform instances $\{\langle E, M, F^P, F^C, F^N \rangle\}$, where E is a set of processing elements (PEs), M is a set of memory banks, and at any time t , $F_t^P : P \mapsto E$ binds a procedure to a processing element; $F_t^C : C \mapsto M$ binds a context to a memory bank, and $F_t^N : M \mapsto E$ connects a memory bank to a processing element; such that $\forall p \in P$ running at time t , and $\forall c \in A(p)$, we have $F_t^P(p) = F_t^N(F_t^C(c))$ to ensure that the running procedure can access its contexts. \square*

According to our definition, a CFA instance is a network of processing elements and memory banks. The set of processing elements P and memories M are always statically determined, or “customized”, whereas F^P, F^C, F^N can either be time-invariant, in which case they

are customized statically, or time-variant, in which case they are dynamically configured using an on-chip network. In this work, we consider that F^P , that is, the binding procedures to the processing elements, time-invariant. In other words, the design space of architecture exploration of CFA is constructed from different choices of P , M and F^P . This assumption is used in our experiments in Chapter 6.

The first requirement of CFA design is to deliver execution efficiency. This is achieved by the fact that different PEs can be used to run different procedures in parallel. This task is considerably easier to accomplish than the traditional shared or distributed memory architecture since CFP procedures accessing disjoint contexts can be run in parallel without the concern of dependency hazard or cache coherence.

The second requirement of CFA design is to deliver the context-flow programming model. This is achieved by the transparent implementation of procedure call abstraction across different PEs, commonly referred to as *remote procedure call* (RPC), as well as context management. This task is accomplished by the design of its on-chip network. We start by first defining a programming model, which abstracts how it interacts with the PEs that it connects. It is important to note that this programming model is the “assembly level” model seen by the compiler or synthesis tools, rather than application programmer. We therefore define the programming model in the form of an instruction set, as shown in Figure 3.3. The instruction set is simple enough to contain only 10 instructions. It is encoded by the values of the wires on each port that connects a PE to the network. From the perspective of the network, it encodes a command or request from a PE. From the perspective of a PE, the instruction set is a complement of its own, for which it can assume the availability of a co-processor for actual execution – effectively by driving the right wires in the corresponding ports.

It is easy to notice the close correspondence between the abstraction at the two levels of programming models. `cfiAllocBank` and `cfiMalloc` implement context allocation and objects allocation within a context, mapping to `cfNewContext` and `cfAlloc`, respectively. `cfiFreeBank` implements context deallocation. `cfiLoad` and `cfiStore` perform mem-

<i>int</i>	<i>cfiAllocBank(void);</i>	3
<i>void</i>	<i>cfiFreeBank(int bankid);</i>	4
<i>void*</i>	<i>cfiMalloc(int size);</i>	5
<i>word</i>	<i>cfiLoad(int addr, int size);</i>	6
<i>void</i>	<i>cfiStore(int addr, int size, word data);</i>	7
<i>void</i>	<i>cfiCnctBank(int bankid);</i>	8
<i>void</i>	<i>cfiRPC(int procid);</i>	9
<i>void</i>	<i>cfiRet(int procid);</i>	10
<i>word</i>	<i>cfiAckRPC(void);</i>	11
<i>word</i>	<i>cfiAckRet(void);</i>	12

Figure 3.3: Context-Flow Architecture Instruction Set.

ory access operations. The size parameter defines the accessed data size (8, 16, or 32 bits). *cfiCnctBank* is optional, depending on hardware implementation. The remaining instructions implement the RPC interface, *cfiAckRPC* and *cfiAckRet* are optional, dequeuing the next request available at the PE queue. We distinguish the two functions for the possible implementation of separate queues for procedure calls and returns.

3.2.2 Possible Organizations of CFA On-Chip Network

We now consider how to implement an on-chip network that can implement this instruction set efficiently. There are several alternatives, each employing a different network topology.

We first consider the cases where F^C is time-variant, in other words, the context can “flow” from memory to memory; whereas F^N is time-invariant, in other words, each memory is fixed to a particular PE. As shown in Figure 3.4 (a), a *bus-based* CFA maintains a private memory bank for each of its PEs. However, every time a RPC is invoked, the content of the corre-

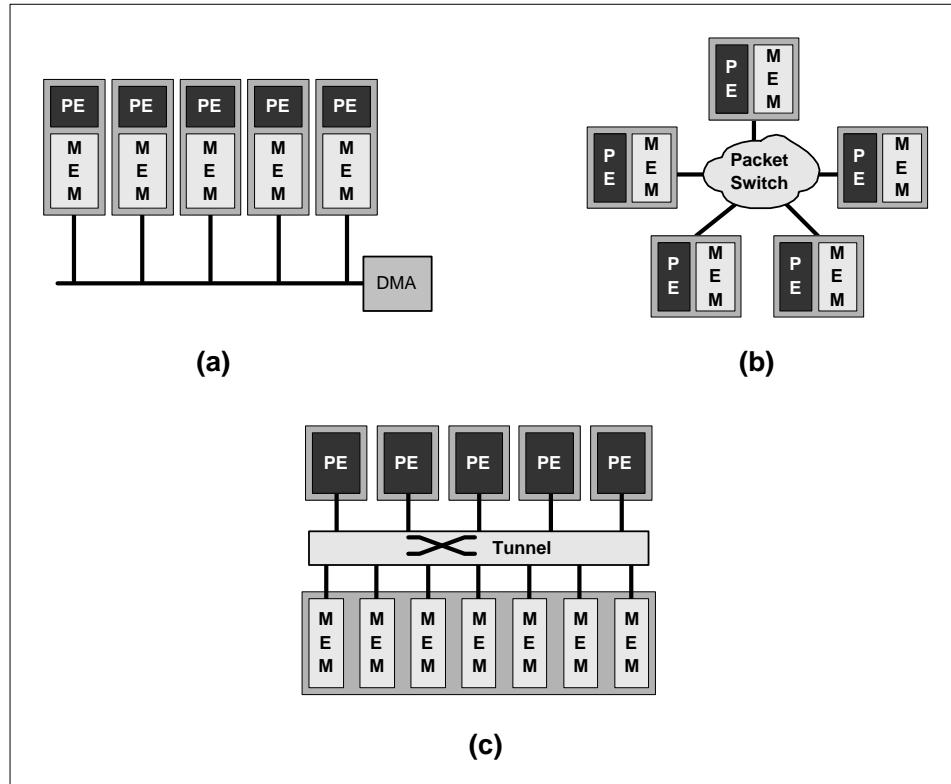


Figure 3.4: Alternative Implementations of Context-Flow Architectures

sponding context needs to be copied to the memory bank that belongs to the callee, and this data transfer is carried out by a shared bus. As shown in Figure 3.4 (b), a *packet-switch-based* CFA is the same as bus-based, except that data transfer can be performed more efficiently: while a shared bus may invite transfer congestion, a well designed packet-switched network can distribute the communication traffic evenly.

Like previous efforts, these two alternatives do not take full advantage of the fact that the network we are designing is on-chip, and the PEs are physically close to each other. We propose a new on-chip network, called a CFA *tunnel*, where F^C is time-invariant, whereas F^N is time-variant. As shown in Figure 3.4 (c), the tunnel maintains a pool of separate memory banks, as well as an intelligent crossbar switch. Each context is dynamically mapped to a single memory until it is deallocated, and the crossbar ensures the access to the memory is dynamically switched to the callee whenever an RPC occurs. Note that our crossbar should not

be confused with the crossbars in previous efforts, which is designed still for the purpose of data transfer. Instead, the goal of our crossbar is to provide the direct, wired access for memories. RPC, or the flow of contexts from one PE to another, can then be achieved at virtually no cost!

3.3 Tunnel-based CFA

A tunnel-based CFA system consists of a reconfigurable memory system, an interconnection architecture we refer to as a tunnel, and a set of processing elements. A block diagram of target architecture is shown in Figure 3.5.

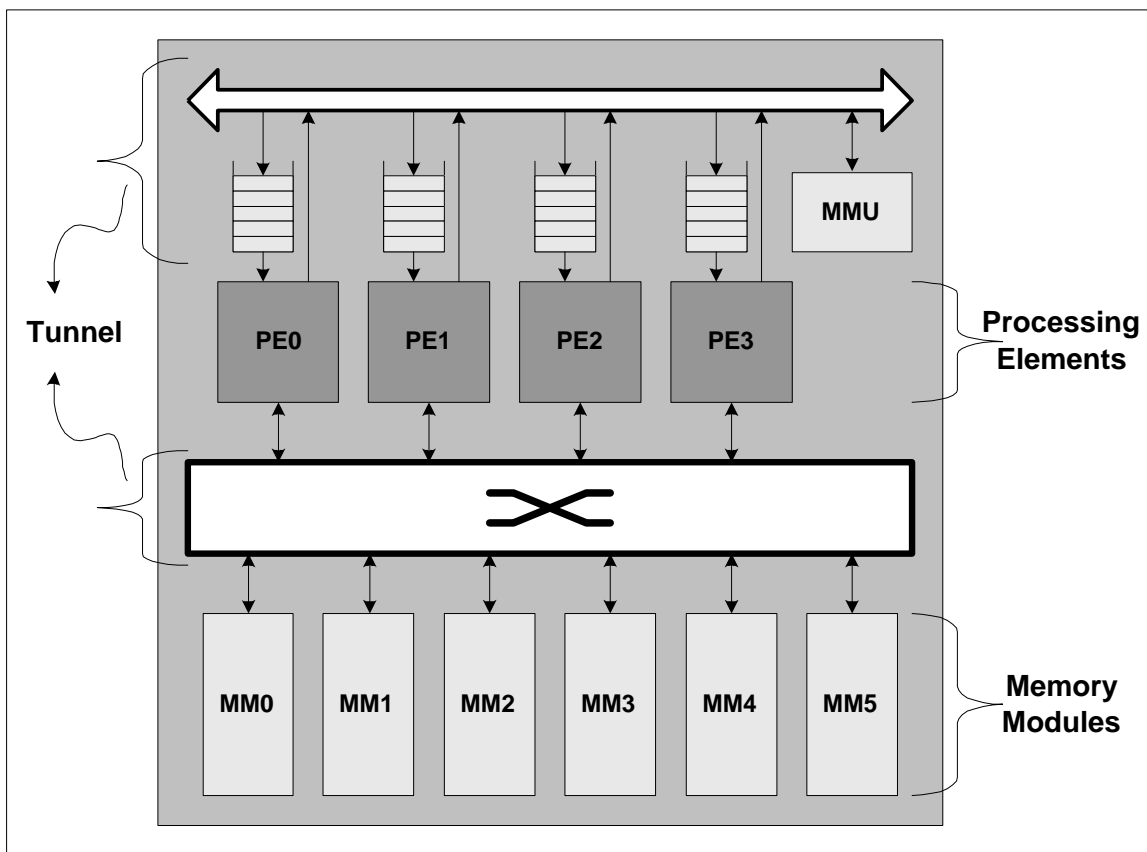


Figure 3.5: Tunnel-based CFA Block Diagram

3.3.1 Memory System

The memory system is one of the main design concerns for future SOC and multiprocessor designs [66]. Our architecture features a high-bandwidth, high-performance, flexible, SRAM-based memory system that can be optimized statically for the target application and dynamically for variable runtime workload.

Our memory system consists of a centralized memory management unit (MMU) and a pool of replicated SRAM-based memory modules (MM).

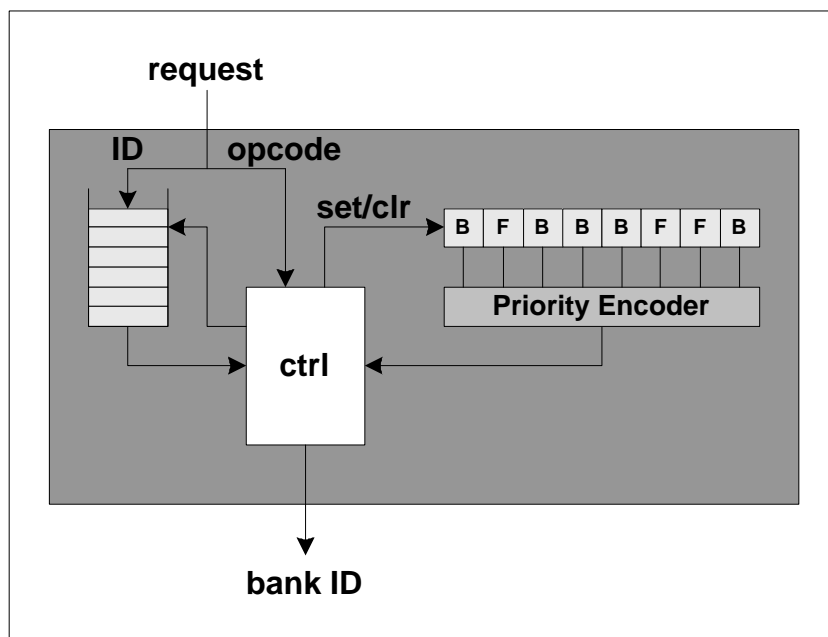


Figure 3.6: Memory Management Unit (**B**: Busy, **F**: Free)

The centralized memory management unit is responsible for context/bank allocation and deallocation. A block diagram of this unit is shown in Figure 3.6. The management unit keeps track of the available banks and allocation requests. If an allocation request is received and one of the context memory modules is available, the bank ID is returned, otherwise the request that contains the caller ID is queued waiting for the next free bank. This queue could be a simple first-in first-out (FIFO). Alternatively, a port-based priority scheme could be adopted, which statically assigns a priority level to each port a request may come from. Similarly,

a procedure-based priority scheme could be used. Assigning priorities to ports/procedures should be taken into account when synthesizing the target application. MMU is implemented using very simple components. The priority encoder circuitry is used to find the index of the first free “F” memory module. The queue is implemented as a simple 2-port register file with two wrap-around adders, one for write pointer (tail) and one for read pointer (head) as shown in Figure 3.7.

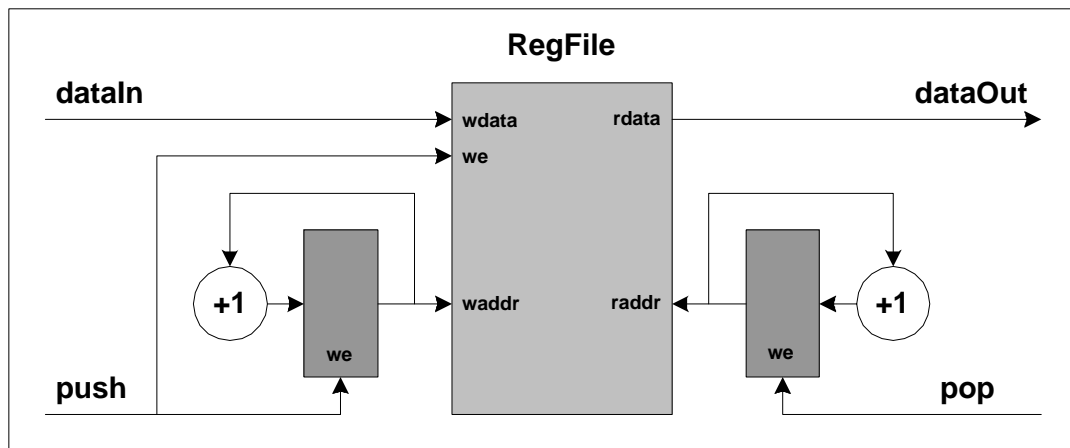


Figure 3.7: First-In First-Out Queue (FIFO)

Each memory module contains a single-port SRAM memory bank, a SP/HP management unit, and module controller (Figure 3.8). Regarding bank size, we use small memories, few KBytes, targeting the optimal speed and power consumption of SRAMs [3]. The basic SRAM bank size will be statically determined by the system designer based on the memory granularity needed by the synthesized application. The SP/HP management unit maintains the stack pointer (SP) of the calling stack and the heap pointer (HP) of the context mapped to the memory module. SP and HP have analogous semantics to those of traditional computer systems, pointing to the top of stack and heap within a memory segment. This unit is controlled by the module controller which decodes and executes the input instructions. The memory module can perform read, write, malloc operations. Special management of the SP, similar to that performed in traditional computer system, is needed when a RPC or Return takes place. For

example, when a RPC occurs, the old SP of the caller needs to be saved on the stack, and the SP should be updated with the new stack pointer for the callee after saving call parameters. This functionality can be optionally embedded within the memory module controller. However, we currently implement this feature as a macro-instruction running on the PEs.

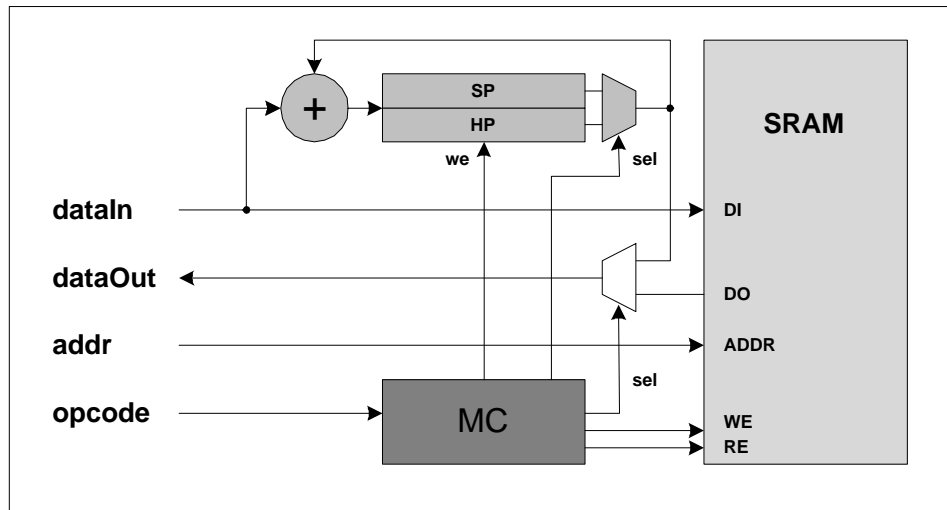


Figure 3.8: Memory Module

As mentioned before, each memory module or a context can be accessed by a single PE at any point in time. Some application may require access of data objects common across procedures/PEs. These objects may be mapped to preallocated memory modules that can be accessed by any PE. To ensure mutual exclusion of the accesses to the shared memory modules, a token is used to regulate these accesses. To access a common data object, a PE needs to acquire that token before the access, access the memory module, and release the token when done passing it to the next PE. If the procedure running on a PE does not need to access a common memory module, the PE passes the token immediately to the next module. We assume that the common data objects, usually some control/header information, will be accessed infrequently, which makes this solution an efficient and adequate one, especially at the scale of single-cluster designs.

One of the main advantages of the proposed memory system is its flexibility. There is no

static assignment of memory modules to PEs. So if the distribution of the traffic load changes at run time such that some PEs remain idle, while others become heavily loaded, more memory modules will be allocated for the busy PEs, making better utilization of resources and achieving better performance.

3.3.2 Interconnect

The interconnect architecture, which we refer to as the tunnel, performs two main tasks; it connects the PEs to the memory modules for data accesses, and it routes the RPCs>Returns to the target PEs as well as context allocation/deallocation requests to the memory management unit.

The interconnect fabric that routes the RPCs>Returns and context allocation/deallocation requests is a low-bandwidth shared bus. We use a single circulating token, the one mentioned in Section 3.3.1, to control access to the shared channel.

To service memory access requests, a non-blocking point-to-point crossbar is used to connect PEs to memory modules. Such a communication architecture can provide the required bandwidth to memory, allowing one memory access per clock cycle for each PE. In other words, if there are n PEs, each issuing one memory access per cycle, then the crossbar must have a peak bandwidth of n words per cycle.

In spite of their cost ($O(n^2)$, where n is the number of interconnected modules), crossbars are finding their way as an interface to multiple bank memory systems. For example, both IRAM, a vector processor [57], and Smart Memories tiles, a reconfigurable 64-bit processing engine [41], use a variation of crossbar to provide access for a single PE, basically a single programmable processor, to multiple bank memory system. It has also been proposed in [40] to use a crossbar to provide access for the processors of parallel systems to the L1 cache to reduce energy consumption of memory accesses.

There are several ways to build a fully-connected interconnection network. The traditional approach in constructing a non-blocking fully-connected crossbar is shown in Figure 3.9.

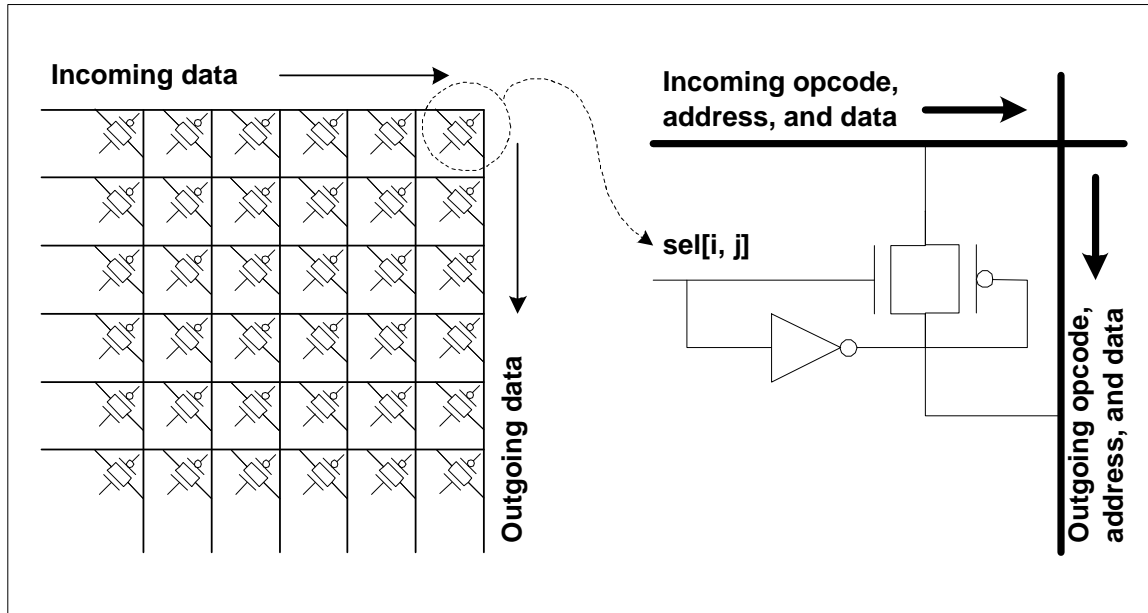


Figure 3.9: Crossbar

One of the problems with this traditional implementation is the need to route control signals to all intermediate switches from a centralized configuration unit. Besides, whenever a RPC/Return takes place, the parameter context ID needs to be sent to this configuration unit to reconfigure the interconnect.

An alternative implementation was proposed in [23], called self-routing crossbar. Self-routing crossbar is a decentralized implementation of a non-blocking point-to-point interconnect. The control signals are not routed from a centralized configuration unit. Instead, they are generated locally at each switch point by decoding the appropriate address bits and, if there is a match, activating a tristate buffer which routes the input data to target output lines (Figure 3.10). Simulation results showed an improvement in delay of 23% while consuming almost the same area and energy per transmission, when compared to traditional implementations.

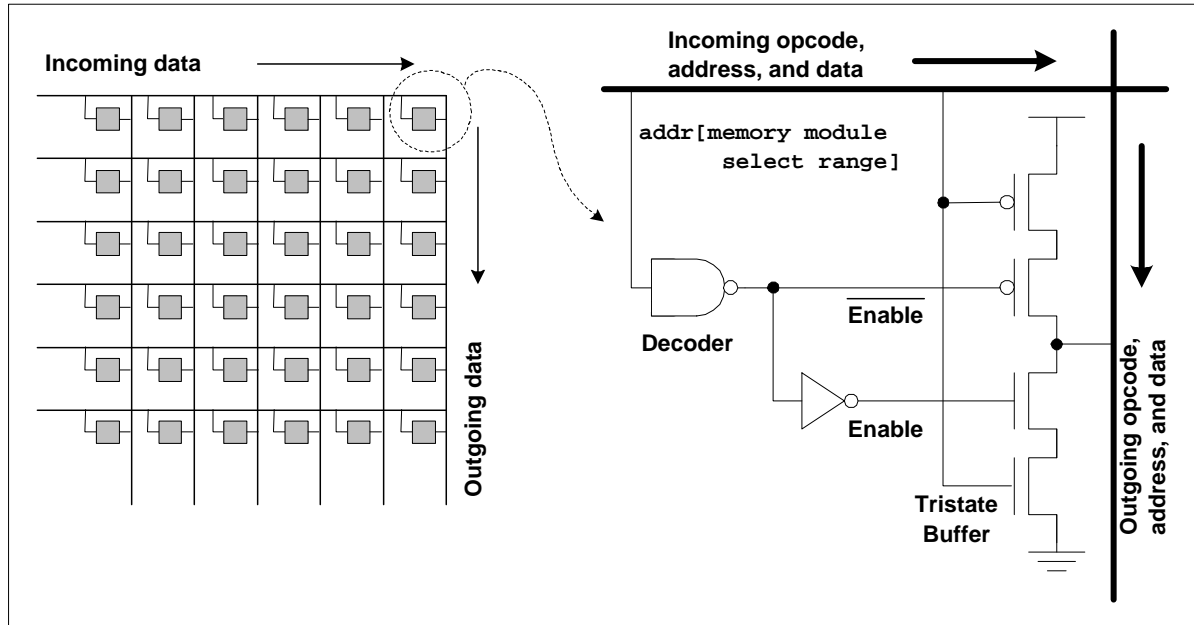


Figure 3.10: Self-routing Crossbar

3.3.3 Processing Elements

In this work, we target the design of heterogeneous SOCs which provide the system designer with the flexibility in choosing appropriate processing element for each application or even individual procedures. These processing elements could be anything, ranging from programmable embedded processing cores to ASIC, ASIP, or even programmable FPGA, as long as these PEs implement the interface defined in Figure 3.3. As a result, our platform is of a heterogeneous architecture with a homogeneous programming model.

3.4 Context-Flow SOC Design Example

To illustrate the operation of the tunnel-based context-flow SOC platform, we present a simple, yet complete design example transformed from its C description to the actual implementation.

Let us assume that we want to build a simple design that performs some arithmetic operations on a stream of data packets that arrive as inputs to our system. The packets are of variable

size, and the raw data is an array of floating point numbers to be updated. The input program is shown in Figure 3.11 (a).

In this design, identifying the contexts is a trivial task; we only have one context which consists of the data array A.

Let us assume that after some investigation, for example by profiling a typical input traffic, we came up with the following design decisions:

- Both `top` and `pow2` procedures are mapped to PE0, which is implemented as a simple RISC processing core – Figure 3.11 (b).
- The `sin` procedure is mapped to PE1, which is implemented as a custom ASIC – Figure 3.11 (c).
- We need three context memory modules to cope with input traffic intensity.

The resulting implementation is shown in Figure 3.11 (d).

To see how the data flows within the target application, let us assume we have the scheduled input traffic listed in Figure 3.12 (a). This behavior can be captured in the context-flow behavioral description listed in Figure 3.12 (b).

Figure 3.13 through Figure 3.20 show the system at several time stamps as it progresses in response to the input harness.

Assuming that no further packets arrive for a while, C0 will finish processing on PE1 and exit the system. And C1 will follow C0 in its behavior. The remaining stages until both packets exit the leave are similar to those listed above.

3.5 Discussion

At the end of this section, several points relative to the tunnel-based CFA are worth elaborating on:

- **Performance Efficiency:** Several factors participate in the performance enhancement of the tunnel implementation when compared to the alternatives mentioned in Section 3.2. First, data transfer time is saved, as only one clock cycle is needed for the context to flow from one PE to another using bank switching. Alternative implementations will require one clock cycle for each data word to be transferred. Second, no contention exists over shared communication, when compared to the bus-based implementation. And finally, non-blocking context flow allows requests be sent from caller to callee even if the latter is busy. Which gives the caller a chance for immediate processing of subsequent requests.
- **Energy Efficiency:** In the past few years, power/energy efficiency has become a major design constraints in addition to area and performance, especially for portable applications [8]. Experiments have shown that in data dominated applications, such as multimedia and network processors, a very large portion of the power consumed is due to data transfer and storage as each memory access consumes 8 – 30 time more energy than that of a primitive arithmetic/logic operation. Therefore, extensive research efforts were spent on the minimization of memory accesses [13]. Our architecture saves two memory accesses per word per context for each RPC/Return when compared to other implementations.
- **Scalability:** We do recognize that there is a physical limit for the scalability of the CFA tunnel. As the network gets larger, the delay of the crossbar grows quickly, thereby increasing the cost of each memory access. This can be contained by employing a two-layer strategy, where PEs are partitioned into clusters based on the communication traffic among them, and intra-cluster network is based on the tunnel connecting tightly coupled processing modules. Whereas the inter-cluster network is based on packet switch. In this work, we focus only on the study of the flat network, which is appropriate for the applications we are interested in.

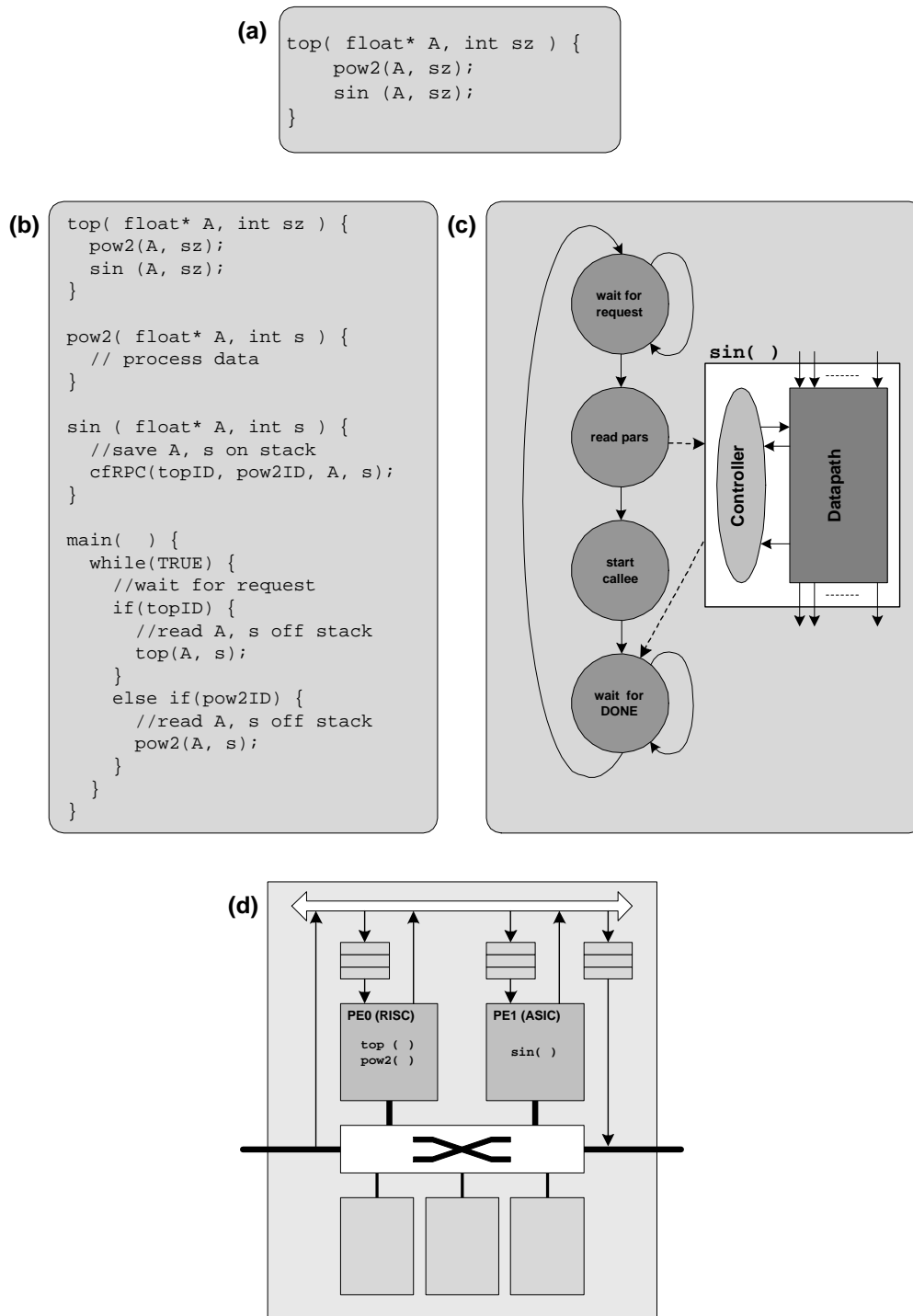


Figure 3.11: A Simple Design Example

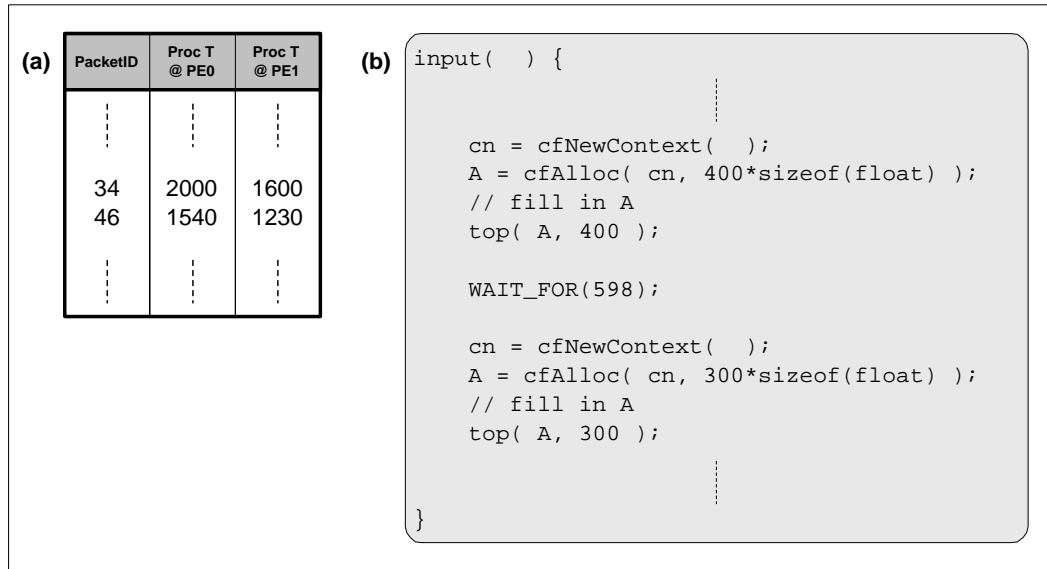


Figure 3.12: System Input

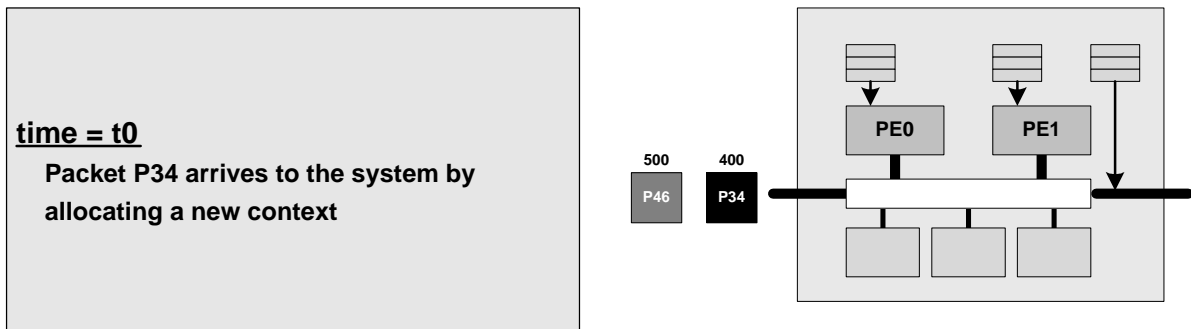


Figure 3.13: System Behavior to the Input Described in 3.12 (a)

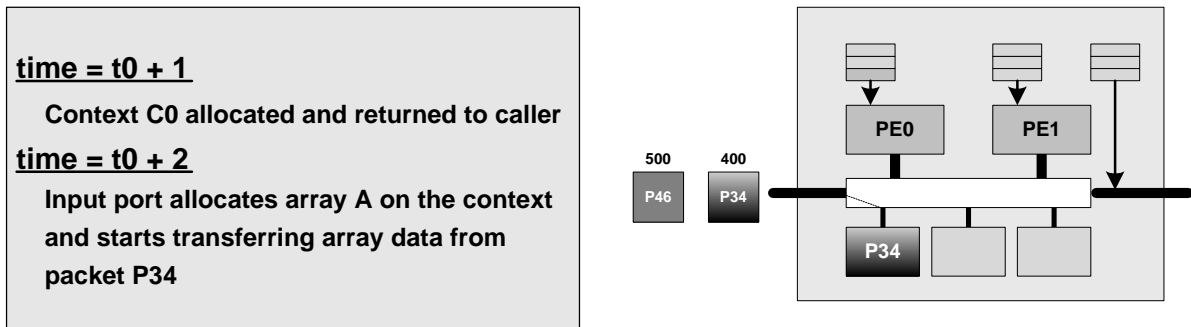


Figure 3.14: System Behavior to the Input Described in 3.12 (b)

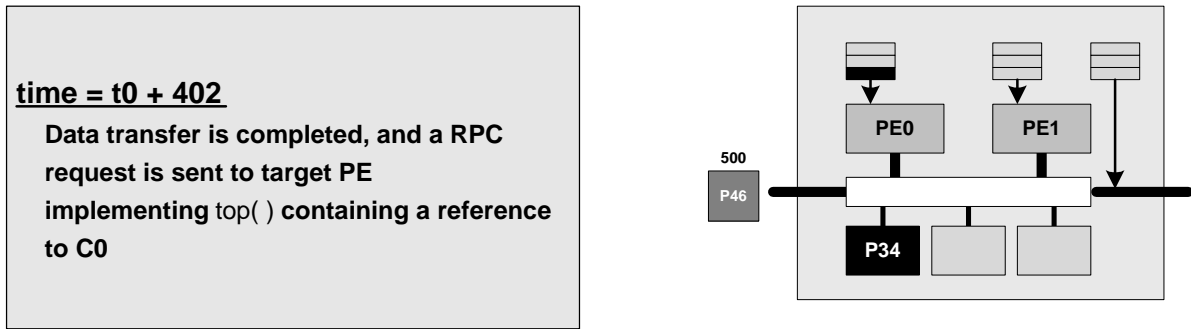


Figure 3.15: System Behavior to the Input Described in 3.12 (c)

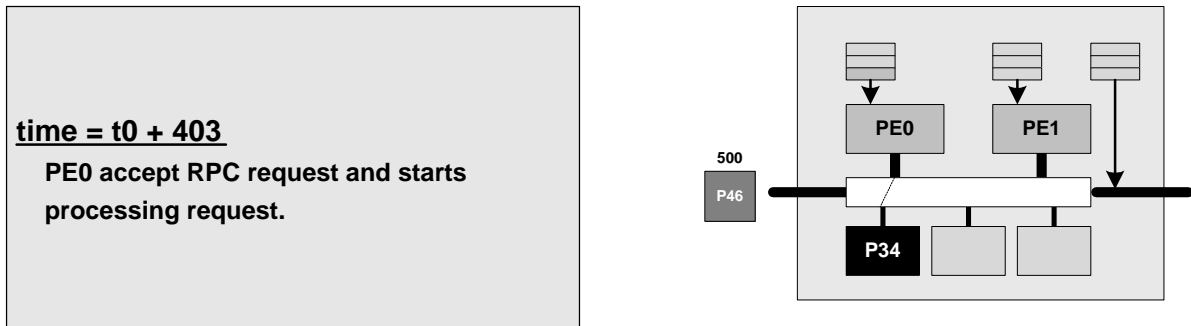


Figure 3.16: System Behavior to the Input Described in 3.12 (d)

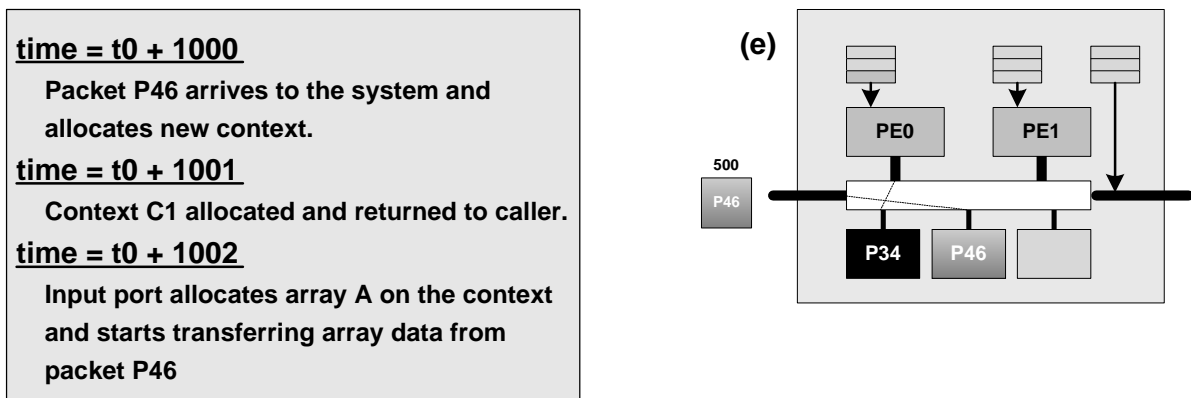


Figure 3.17: System Behavior to the Input Described in 3.12 (e)

time = t0 + 1302
 Data transfer is completed, and a RPC request is sent to target PE implementing top() containing a reference to C1
 Request will be blocked as PE0 is still running top()/pow2() on C0

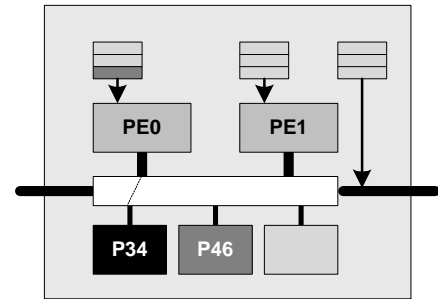


Figure 3.18: System Behavior to the Input Described in 3.12 (f)

time = t0 + 2404
 top()/pow2() running on PE0 finish processing C0 and call the the local implementation of sin(), a proxy, which sends a RPC request to PE1

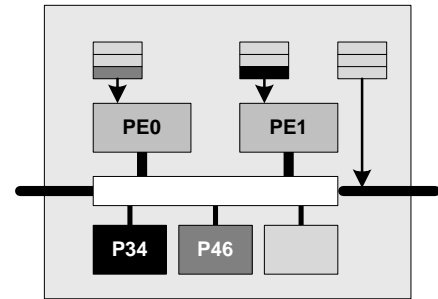


Figure 3.19: System Behavior to the Input Described in 3.12 (g)

time = t0 + 2405
 Both PEs are available, therefore both RPC requests will be accepted at the same clock cycle

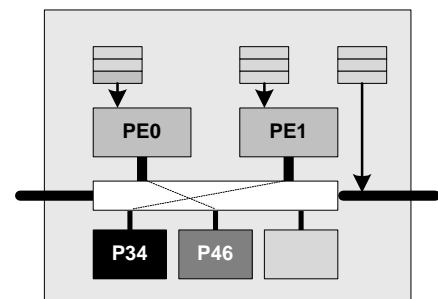


Figure 3.20: System Behavior to the Input Described in 3.12 (h)

Chapter 4

Performance Evaluation Framework

To evaluate new architectures and design methodologies in the field of SOC, we target complex applications which are usually described in C using high-level language features such as pointer references and complex data structures. The speculated performance advantage of new propositions can only be validated on such applications. A performance evaluation environment that can simulate CFA with reasonable architectural details for any CFP applications is therefore needed.

In this section, we introduce our context-flow performance evaluation framework. The developed environment is based on the SimpleScalar Tool Set [11]. We start by giving a brief overview of the SimpleScalar tool set. Then we introduce how the SimpleScalar infrastructure is extended into a multi-processor, CFA performance evaluation environment. Finally, we show how a C program is mapped onto a CFA in our environment by a simple, yet complete example.

4.1 SimpleScalar Tool Set

SimpleScalar tool set is a good example of an architectural evaluation environment [11]. It is designed to study new innovations in micro-architecture such as pipelining, branch prediction, out-of-order issue etc. The environment provides a complete compiler tool chain that can compile a C application into a binary in the PISA instruction set, MIPS-like ISA. An instruction

set simulator can then be used to simulate the binary executable, while collecting performance metric of interest.

The SimpleScalar tool set provides the user with several versions of simulators ranging in complexity from functional (*sim-safe*), which executes instructions serially assuming a perfect hazardless pipeline, to a detailed simulator (*sim-outorder*), which models a detailed pipeline with out-of-order issue and execution. An overview of the tool set is shown in Figure 4.1.

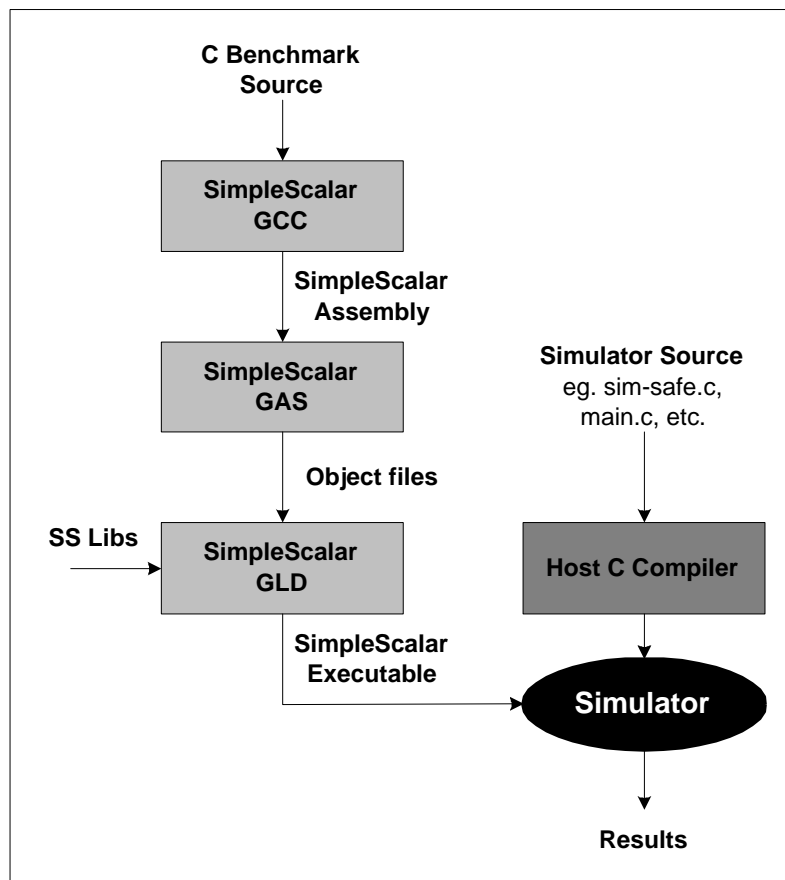


Figure 4.1: SimpleScalar Tool Set Overview

For the purpose of modeling processing elements in our systems, we only need a simulator that models the functionality of a PE, which could be a simple RISC processor as well as ASIC. From this perspective, we do not need the detailed simulation model of a possible micro-architecture of a superscalar processor, such as that implemented in *sim-outorder*. The simple

serial execution with a single cycle per instruction performance is all we need. Therefore, we decided to use the functional simulator, *sim-safe*, especially that it delivers several orders of magnitude in simulation speed when compared with *sim-outorder*.

Figure 4.3 shows the pseudo code of *sim-safe*, a fast simulator provided in SimpleScalar, which maintains the processor state by a simulated set of registers (*regs*) and memory system (*mem*). These two state variables are shown in Figure 4.2 (a) and (b), respectively.

Simulation starts by loading the application binary into the simulated memory, and then enters a loop which fetches an instruction from the simulated memory one at a time, decodes it, and performs an action that is consistent with the instruction semantics, while updating simulated registers and memory accordingly ¹.

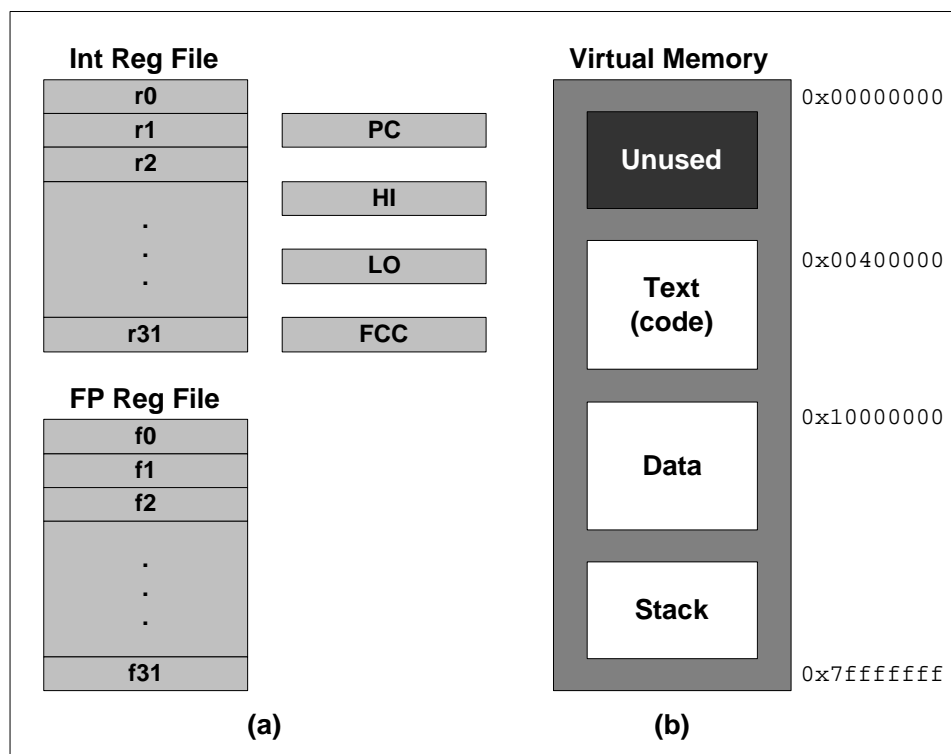


Figure 4.2: State Variables for the *sim-safe* Version of the Simulator: (a) regs (b) mem

¹Please note that the actual implementation spans several files, e.g. loading the application binary into a simulated memory takes place in `main` before calling the simulator core which implements the execution loop. However, for the sake of clarity, the code was presented as a single block.

```

/* memory space and register file (state variables)    */
RegsType      regs;
MemSpaceType  mem;

void simCore( )
{
    /* create memory space & load target program      */
    memCreate(mem);
    loadProg(prog, mem);

    while(TRUE)
    {

        /* fetch next instruction to execute          */
        inst = Fetch(mem, reg.PC);

        /* decode, execute, and commit the instr     */
        switch ( opcode(inst) )
        {
            case ADD: perform_add;
            case SUB: perform_sub;
                    .
                    .
                    .
        }

        /* go to next instr                            */
        reg.PC = reg.NPC;
        reg.NPC++;
    }
}

```

Figure 4.3: The Original SimpleScalar Simulator Core

4.2 Sim-CFA

We consider a homogeneous CFA where each PE is implemented by a processor equipped with the PISA instruction set complemented by the context-flow instruction set defined in Section 3.2.1.

The simulator was modified to run multiple SimpleScalar processors simultaneously, modeling the multiple threads executing in parallel on the system PEs. For this purpose, the processor state of memory space and register file in the single processor environment was replicated, one per PE, as shown in Figure 4.4. The main execution loop of the simulator was modified to execute one instruction from each PE code at each simulation cycle.

While each PE has its own private address space, the unused memory space segment of each PE, from address 0x00000000 to 0x03FFFFFF as shown in Figure 4.2 (b), was mapped

```

RegsType      regs[NUM_OF_PES];
MemSpaceType  mem [NUM_OF_PES];

void simCore( ) {
    /* create memory space and load target program for each PE */
    for( each PE p ) {
        memCreate(mem[p]);
        loadProg(prog[p], mem[p]);
    }

    while(TRUE) {
        for( each PE p ) {
            /* fetch ... */
            inst = Fetch(mem[p], reg.PC[p]);
            /* decode, execute, and commit */
            if( annotated(inst) ) {
                /* context-flow instruction */
                switch ( annotation(inst) ) {
                    case RPC:      perform RCP;
                    case AllocBank: perform alloc;
                    .
                }
            }
            else if (memAccess(inst) && addr<0x04000000) {
                access context flow memory banks;
            }
            else { /* normal code */
                switch ( opcode(inst) )
                    case ADD:      perform add;
                    case SUB:      perform sub;
                    .
            }

            /* go to the next instruction */
            reg.PC[p]=reg.NPC[p]; reg.NPC[p]++;
        }
        /* for none-tunnel implementations */
        execute one DMA cycle;
    }
}

```

Figure 4.4: The Modified SimpleScalar Simulator Core

to the context memory pool. For example, if the CFA contains 8 context banks each of which is 4 Kbytes, the memory space ranging from 0x00000000 to 0x0000FFFF is partitioned into 8 equal sub-segments, each representing a context memory bank. With this approach, high-level language features, such as array references, pointer indirection, and structure member references can still be used directly in the source code to access objects within the context.

The actual simulated environment is shown in Figure 4.5.

SimpleScalar suite provides a very useful annotation interface where unused bits in the instructions can be used to introduce new instructions without the change of compiler tool suite

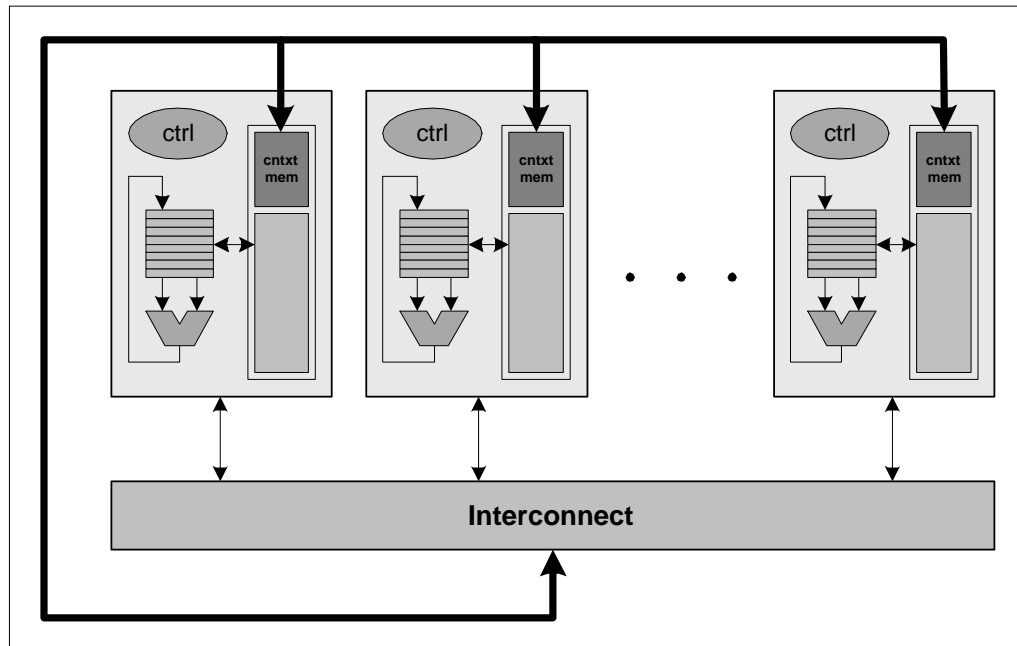


Figure 4.5: The Actual Simulated System

(Figure 4.6). A new instruction is defined by giving a non-zero annotation value to predefined instruction opcodes. This annotation value can be detected at runtime and executed by emulating the corresponding behavior. We use this feature to introduce the context-flow instruction set to each PE.

As shown in Figure 4.4, the simulation engine starts by loading the binary executables for each PE into the simulated memories. At each simulation cycle, for each PE, the simulator fetches an instruction from memory and decodes it. If its annotation field is non-zero, meaning that it is a context-flow instruction, it will invoke the corresponding on-chip network simulation to process a request on one of the ports of the network. If it is a memory access whose address falls into the range from $0x00000000$ to $0x03FFFFFFF$, the corresponding location inside the context memory pool will be accessed. Otherwise, it will interpret the instruction the same way as SimpleScalar does.

We implemented the networks defined in Section 3.2, namely bus-based, packet-switch-based, and tunnel-based. Note that at this stage of implementation, our packet switch network

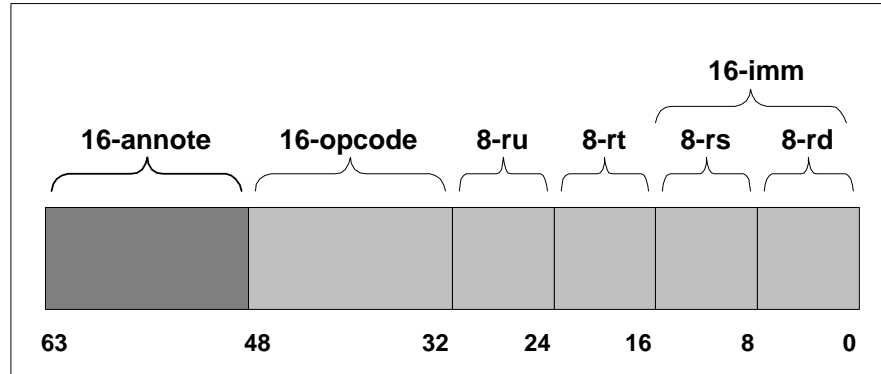


Figure 4.6: SimpleScalar PISA Instruction Format

is very preliminary: we assume a perfect network where no congestion can ever occur (equivalent to point-to-point), which can nevertheless give the performance upper bound. The actual implementation includes a parameterized direct memory access (DMA) engine. The main parameter is the number of DMA channels the engine can run simultaneously. This way we can provide a cycle accurate simulation for single-channel bus, multi-channel bus, and perfect-packet switch network.

Another simplification we use for now to obtain a first order approximation of heterogeneous CFA, where processing elements can be custom hardware, is to include a linear speedup number for a PE intended for non-RISC implementation, thereby getting an approximate execution time. In our experiments, we used some recursive training, by changing the speedup and using the simulation results as a feedback to improve the accuracy of that speedup for various job sizes. Future versions of the simulator will consider a better approximation for heterogeneous systems.

To collect performance statistics during simulation, each request that enters the system is assigned a unique identifier. The simulator keeps track of the movements of each context within the system by eavesdropping on the interconnect activities, recording the *begin* and *end* of queuing, in case of tunnel-based CFAs, and processing times at each port, as well as the overall system (Figure 4.7).

The collected performance figures are translated into useful performance statistics. These statistics include average, minimum, and maximum *processing/busy time* for each PE; average, minimum, and maximum *waiting/queuing time* for each PE, in case of tunnel-based network; *utilization*, which measures the percentage at which the PEs are busy computing rather than idling; and *throughput*, which measures the rate at which CFA can accept the top-level RPC.

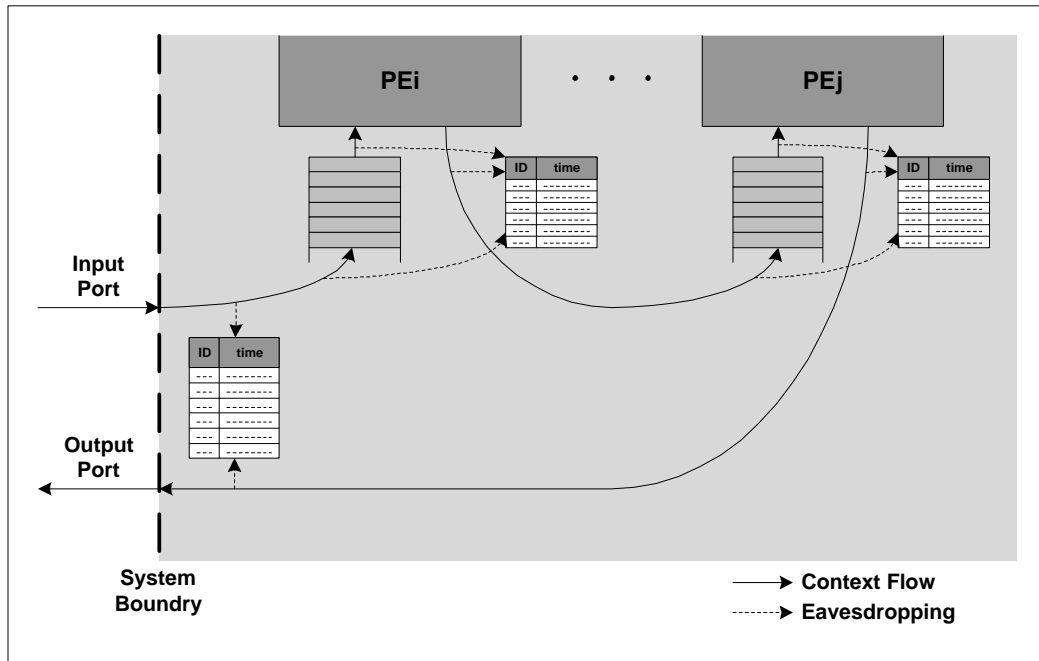


Figure 4.7: Activity Monitoring in Sim-CFA

4.3 Architecture Configuration and Application Compilation

To illustrate various aspects of our simulation environment, consider the simple design example presented in Figure 3.11, which calculates $f(A) = \sin(A^2) : A \in R^n$. As we did before, we will be running the application on two PEs, mapping `top` and `pow2` to PE0 and `sin` to PE1.

A description of the application procedures and various architectural parameters are defined in a procedure stamps file, "`pstamps.cfg`", and a configuration file "`config.cfg`". "`pstamps.cfg`" specifies the format of each procedure in our application simply through

procedure declarations. `config.cfg` needs to completely specify the target implementation. These specifications include:

- Number of PEs (`NUM_PORTS`)
- Size of each RPC/Return queue of each PE (`PORT_FIFO_SIZE`)
- Number of memory modules (`NUM_BANKS`)
- Bank size of each memory module (`BANK_SIZE`)
- Number of procedures to be mapped (`NUM_PROCEDURES`)
- Procedure mappings (`PROC.ID_PE`)
- Procedure speedup, relative to some custom implementation (`PROC.ID_SU`)
- Interconnect architecture (`INTERCONNECT`)
- Number of DMA channels in case of bus-based implementation (`NUM_DMA_CHNLS`)

An example of `config.cfg` is presented in Figure 4.8. It specifies that our implementation has 2 PEs, each of which has a 3 slot queue, and 3 memory modules, each of which has a 2K memory bank. We map procedures `top` and `pow2` to PE0 with no speedup, assuming a simple RISC processing core, and `top` to PE1 with a speedup of 20, which we got after comparing the performance of SimpleScalar with that of a custom hardware implementation. Finally, our CFA is tunnel-based.

Both `pstamps.cfg` and `config.cfg` are used by an automatic code generator that was developed to generate proxies or stubs for mapped procedures, to be used when a remote procedure call occurs in a manner similar to that used in middleware systems. The generator also returns a `main` function for each PE. An example of the returned code is shown in Figure 4.9, where part (a) belongs to PE0 and (b) belongs to PE1. Note that the `main` for each PE simply runs an infinite loop waiting for a call to the procedures it implements.

```
/* system configuration */
#define NUM_PORTS      2
#define PORT_FIFO_SIZE 3
#define NUM_BANKS     3
#define BANK_SIZE     2048
#define NUM_PROCEDURES 3

/* procedure ID definition */
#define TOP_FUNC      0
#define POW2_FUNC    1
#define SIN_FUNC      2

/* procedure mapping */
#define TOP_PE        0
#define POW2_PE       0
#define SIN_PE        1

/* procedure speedup */
#define TOP_SU        1
#define POW2_SU       1
#define SIN_SU        20

/* implementation */
#define INTERCONNECT  TUNNEL
```

Figure 4.8: An Example Configuration File “config.dat”

WAIT_FOR_RPC and READ_2_ARGS are simply macros that use `cfiAckRPC` and `cfiLoad`, respectively.

Once coded/generated, the source files of each PE along with proxies’ definition are compiled by the SimpleScalar gcc compiler `ss-gcc`. Sim-CFlow then can simulate the modeled system by running the generated binary files to return detailed performance reports. This flow is shown in Figure 4.10.

```

(a)
top ( float* A, int sz ) {
    pow2( A, sz );
    sin ( A, sz );
}

pow2( float* A, int sz ) {
    for(i=0; i<sz; i++)
        A[i] = (A[i]*A[i]);
}

sin ( float* A, int sz ) {
    //save parameters
    cfiRPC(SIN_ID);
}

main() {
    while(1) {
        WAIT_FOR_RPC();
        if(callee==TOP_ID) {
            READ_2_ARGS(A, n);
            top(A,n);
        }
        else if(callee==POW2_ID) {
            READ_2_ARGS(A, n);
            pow2(A,n);
        }
        else
            ERROR( );
    }
}

(b)
sin( float* A, int sz ) {
    for(i=0; i<sz; i++)
        A[i] = sin(A[i]);
}

output( A, sz );
}

main() {
    while(1) {
        WAIT_FOR_RPC();
        if(callee==SIN_ID) {
            READ_2_ARGS(A, n);
            sin(A,n);
        }
        else
            ERROR( );
    }
}

```

Figure 4.9: A Context-Flow Version of the Simple Array Processor

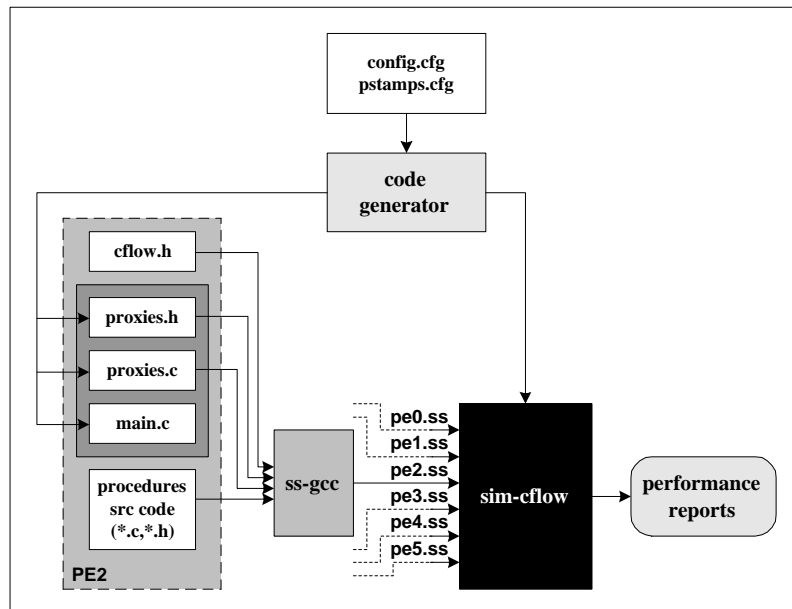


Figure 4.10: Sim-CFlow Simulation Process

Chapter 5

Queueing-Theoretic Performance Model

So far, simulation-based approaches have been the dominant choices by the industry for performance analysis of SOC designs. These approaches are highly accurate, but also prohibitively time consuming for large systems, which prevents the evaluation of a large number of possible system configurations. Intuition is usually used to decide what configurations to simulate out of the feasible ones. However, intuition-based decisions become less accurate the larger the problem, which could result in performance degradation when compared with the target system potentials. What is badly needed is a performance model that can give insight on how performance metric is related to implementation and architectural mapping decisions, commonly referred to as *analytical performance models*.

In recognition to the limitations of experimental evaluation and intuition approaches in the field of SOC design, we developed an analytical performance model to statically model systems implemented using our tunnel-based context-flow architecture. Our model is based on Queueing Networks, a field that received extensive research over many years, and whose models were used extensively in computer systems and networks modelling. Queueing network models were proved to be general, simple, and detailed, reporting various aspects of the target system and application performance measures.

In contrast to the previous work reported in the area, the following contributions are made.

First, the proposed performance model is extremely *simple*. In fact, the solution of important metrics involves only simple equations. Second, the proposed performance model is *synthesis-friendly*. An optimization procedure based on this model can be readily developed, in contrast to the manual “architectural exploration” approach commonly practiced. Third, the proposed model is *flexible*. In fact, our model is as flexible, accurate, and powerful as queueing theory itself. Fourth, our performance model is *validated* against real-life applications with a detailed multiprocessor simulator. The credibility and applicability of the proposed model is therefore guaranteed.

5.1 Queueing Networks

Queueing Networks are an efficient and accurate approach to computer system modelling. They have been used in the design of systems ranging from single network servers to wide area communication networks [39].

A queueing network consists of a set of communicating nodes of service providers. Jobs or customers arrive at a node, waits in the corresponding queue when all servers are busy, gets processed, and departs for another node or out of the system ¹. Figure 5.1 shows an example of a simple queueing system with some feedback flows.

A key feature and reason to the success of queueing network models is that they abstract away many of the low level details associated with the various modelled system. All it needs is a set timed parameters that affect the system performance.

The basic characterization entities of queueing network models are *service providers*, which represent the modeled system processing resources, and *jobs*, which represent the system jobs (contexts in our case). A typical set of inputs of a queueing model are [39]:

- λ , *arrival rate*, specifies the arrival intensity in customers per unit time.

¹Unless explicitly states otherwise, when we talk about Queueing Networks we always refer to Open Queueing Networks, as opposed to Closed Queueing Networks which does not interact with the outside world.

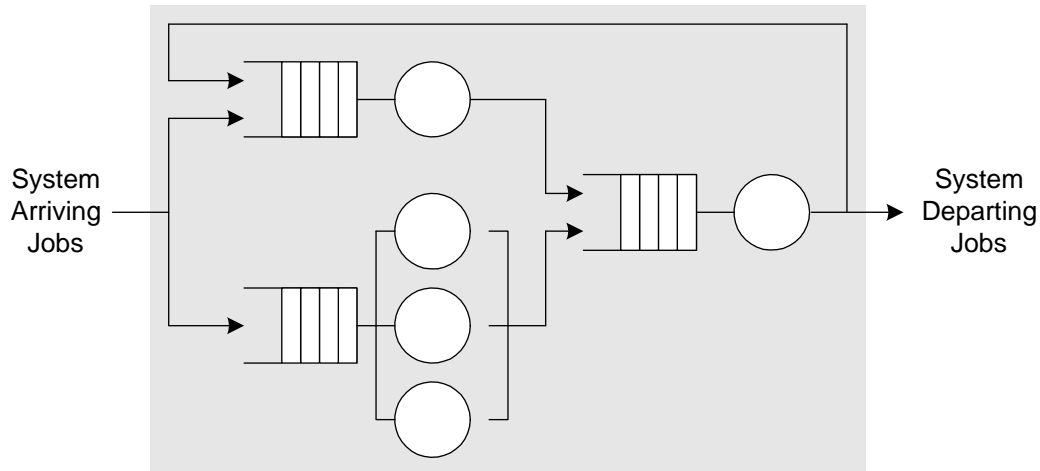


Figure 5.1: A Queueing Network

- Dem_i , *service demand* at server i , which specifies the time service time for each customer.

The outputs obtained by solving the system are:

- R , *average system response time*, which specifies the travel time between the system input and output.
- U_i , *utilization* of server i , i.e. the percentage of overall time the server is busy.
- Wq_i , *queueing time* of server i , which specifies the average waiting time at server i before a job gets serviced.
- Lq_i , *queue length* of server i .

If the jobs arriving to the system have some classification, usually referred to as *Multi-Class Systems*, the model's inputs need to specify the job mix and required services, and the outputs will be returned per class as well as overall system measures.

5.2 Analytical Performance Model

5.2.1 The Modelling Process

The close correspondence between the attributes of queueing networks and those of our CFA suggests that queueing networks could be ideal modeling tools to describe our system.

The modelling process could be viewed as a conversion from system specifications in the Context-Flow domain to those recognized by queueing systems. The output of this stage would be a fully specified queueing network that can be easily solved using simple equations. Whether the resulting system is single-class or multi-class depends on the application being mapped on a CFA.

The inputs of our modelling process are:

- *Workload Specification*, defining the arrival job mix and their corresponding arrival rates. This can be obtained by a process called *workload characterization*, which is a complex process of profiling to arrive at a typical workload. A second possibility is that a typical workload would be defined initially as part of the system specifications [39].
- *Procedure Frequency*, defining the number of calls made to each CF procedure per unit time. Again, this measure can be obtained by profiling of a typical workload, or by static prediction of the probability of edges of the application call graph for a typical workload.
- *Mapping*, describing the assignment of procedures to target system processing elements.

The output of our modelling is a fully characterized queueing network. Solving the model returns the performance estimates of various aspects of the system.

5.2.2 Stochastic Model

Traditional applications of queueing networks to model computer systems assumed the arrival of a Poisson process at the system inputs, and exponentially distributed service times at

the service centers [39]. These assumption implies that the resulting interconnection of our processing elements forms a *Jackson Network*. In this class of networks each queue can be analyzed separately as M/M/m queues. This model is parameterized only by the average arrival rate and average service rate, returning average waiting time, average queue length, and server utilization. This approach was proved quite successful in modelling such systems. For example, requests sent by users to a mainframe did have a random arrival pattern that was captured using a Poisson process. And the size of jobs to be serviced was also a randomized process. However, the immediate application of the same simplifying assumptions to model our architecture was unsuccessful. In a SOC, the arrival process and/or service times could easily be deterministic! For example, arrival rate for an MPEG decoder is usually deterministic, and service rate for ATM packet processing stages is also deterministic.

In [76], W. Whitt described the Queueing Network Analyzer (QNA), a software package developed at Bell Laboratories to analyze complex queueing networks. The package uses a GI/G/m approximation models to describe and analyze the given system. The arrival process is assumed to be a generalized inter-arrival (GI) process, and the service may have any general (G) distribution. The approximation made by this approach is that only the *mean* and *squared coefficient of variance* ($SQV = var / (mean)^2$) of the arrival and service processes are required for the our calculations (a two-moment model). In addition to the basic input parameters described in Section 5.1, we need to provide the SQV of inter-arrival time of the external arrival process to each node i , c_{0i}^2 , and the SQV of the service time, c_{si}^2 . The analysis process calculates the parameters of internal nodes, which enables the calculations of all required system measures. The model is capable of handling even more complicated system features, including superposition and splitting, which is outside the scope of this paper.

For our purpose, the proposed model seemed to be a suitable fit. The additional required parameters could easily be driven by workload characterization. The question left is the model accuracy, which will be reported in Chapter 6. In the sequel, we provide our approach to transform our CFA and application description into a fully described queueing network model.

5.2.3 Derivation of Analytical Performance Metrics

Let's assume that our system consists of N procedures with execution frequency $[f_0 f_1 \dots f_{N-1}]^T$, implemented on an M -port tunnel-based CFA. We define an $M \times N$ mapping matrix, MAP , where $MAP_{i,j}$ represents the mapping of procedure j to PE i .

$$MAP = \begin{pmatrix} m_{0,0} & \dots & m_{0,N-1} \\ \vdots & \ddots & \vdots \\ m_{M-1,0} & \dots & m_{M-1,N-1} \end{pmatrix} \quad (5.1)$$

For example, if we have an application realized in five procedures $[p_0 p_1 p_2 p_3 p_4]$ implemented on a 3-port CFA such that p_0 and p_2 run on PE0, p_1 and p_4 run on PE1, and p_3 runs on PE2, then the mapping matrix is:

$$MAP = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \quad (5.2)$$

Note that in this model $\sum_{i=0}^{M-1} m_{i,j}$ must add to 1. Values less than 1 imply logic/functionality replication and workload distribution. For example, if we want to replicate the procedure p_3 and divide the arrival requests such that one third of the requests go to PE4 and the rest to PE3, then the new mapping matrix will be:

$$MAP = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0.33 & 1 \\ 0 & 0 & 0 & 0.67 & 0 \end{pmatrix} \quad (5.3)$$

To force single instantiation of procedure's logic, we allow mapping figures to take only binary values, $\{0, 1\}$.

Using the summing rule, when two procedures are assigned to a single PE, the arrival rate will be the sum of their frequencies. This conversion from the abstract domain to the queueing

system domain can be captured using an mapping matrix, as shown in 5.4.

$$\begin{pmatrix} m_{0,0} & \dots & m_{0,N-1} \\ \vdots & \ddots & \vdots \\ m_{M-1,0} & \dots & m_{M-1,N-1} \end{pmatrix} \begin{pmatrix} f_0 \\ \vdots \\ f_{N-1} \end{pmatrix} = \begin{pmatrix} \lambda_0 \\ \vdots \\ \lambda_{M-1} \end{pmatrix} \quad (5.4)$$

After deriving the arrival rate for each PE/queue, we can calculate all performance measures that fully characterize the system behavior. The average execution time at each PE can easily be obtained using the equation:

$$D_i = \frac{\sum_{j=0}^{N-1} (m_{i,j} \cdot f_j \cdot Dp_j)}{\sum_{k=0}^{N-1} (m_{i,k} \cdot f_k)} \quad (5.5)$$

Where Dp_j is the average processing time of job by procedure j . Although Dp_j is assumed to be constant, the model can be easily extended to make procedure delays a function of the mapping. On heterogeneous systems, a single procedure could be mapped to different embedded processors with different micro-architectural features, or even to custom logic. To take that into account we define Dp_j in terms of $d_{i,j}$; the average processing time of job by procedure j when running on PE i , as follows:

$$\begin{pmatrix} m_{0,0} & \dots & m_{0,N-1} \\ \vdots & \ddots & \vdots \\ m_{M-1,0} & \dots & m_{M-1,N-1} \end{pmatrix} \begin{pmatrix} d_{0,0} & \dots & d_{0,M-1} \\ \vdots & \ddots & \vdots \\ d_{N-1,0} & \dots & d_{N-1,M-1} \end{pmatrix} = \begin{pmatrix} Dp_0 \\ \vdots \\ Dp_{N-1} \end{pmatrix} \quad (5.6)$$

In Queueing Theory it is a common practice to use service rate instead of service or processing time:

$$\mu_i = \frac{\sum_{k=0}^{N-1} (m_{i,k} \cdot f_k)}{\sum_{j=0}^{N-1} (m_{i,j} \cdot f_j \cdot Dp_j)} \quad (5.7)$$

Using these numbers we can derive major performance measures of processing elements using very simple formulas. The equation describing processing element utilization would be:

$$Utilization_i = \rho_i = \frac{\lambda_i}{\mu_i} = \sum_{j=0}^{N-1} (m_{i,j} \cdot f_j \cdot D_j) \quad (5.8)$$

Using equations 5.7 and 5.8, and c_{ai}^2 and c_{si}^2 for each node i , we can calculate further estimates of PE statistics. For example, the average waiting time at PE i is:

$$AveWaitingTime_i = W_{q_i} = \left(\frac{c_{ai}^2 + c_{si}^2}{2} \right) W_{q_i}^{M/M/1} \quad (5.9)$$

Where:

$$W_{q_i}^{M/M/1} = \frac{\rho_i}{\mu_i(1 - \rho_i)} \quad (5.10)$$

$$AveQueueLength_i = L_{q_i} = \lambda_i W_{q_i} \quad (5.11)$$

We can also derive performance estimates of the overall system. An average processing elements utilization is:

$$Utilization = \rho = \frac{\sum_{i=0}^{M-1} \rho_i}{M} \quad (5.12)$$

And the average service time for a request is:

$$AveServiceTime = D = \frac{\sum_{i=0}^{M-1} \lambda_i \cdot (D_i + W_{q_i})}{\lambda} \quad (5.13)$$

Further processing is needed if the more detailed probability distribution of the above quantities is required, which is outside the scope of this work.

5.3 Discussion

Using this simple model we can easily derive performance estimates for each procedure, for each processing element, each job class, as well as the overall system.

For each procedure running on each PE, the total residence time is the sum of average processing time for that procedure on the PE implementing the procedure, which is an input of the system model, and the average waiting or queueing time at that PE.

For each PE, the total residence time is the sum of average processing time and the average waiting or queueing time, both of which are explicitly calculated in our model.

We can also calculate performance measures for each job class, where a class can be defined

as the set of requests that traverse a specific path within our system. In this case, the total propagation time of each job class is the some of average residence time for all the procedures on its calling path. For example, let's assume that we have the system presented in Figure 5.2, and a certain percentage of the packets that enter the system traverse the path shown in dotted lines. The total propagation time of this class can be given by:

$$TotalPropagationTime = (d_{0,A} + Wq_0) + (d_{2,B} + Wq_2) + (d_{3,C} + Wq_3) + (d_{0,D} + Wq_0) \quad (5.14)$$

Finally, measures of the overall system performance are either explicitly calculated in our model, or can be calculated using the ones explained above.

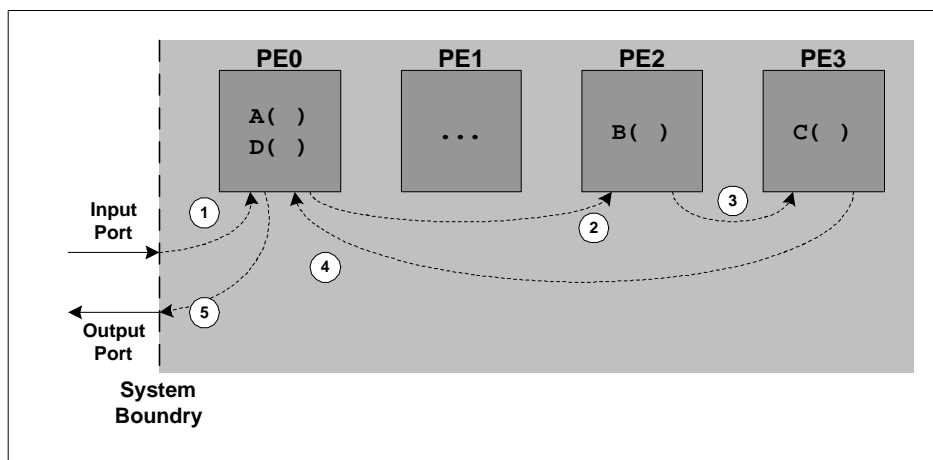


Figure 5.2: An Example of Total Propagation Time for Job Class is a CFA System

The model will enable the use of general-purpose optimizers, such as genetic programming, simulated annealing, linear programming for some constraints, and non-linear programming, in a system-level synthesis flow of single or multi-objective optimization problems.

Chapter 6

Case Studies

The goal of our empirical studies is three fold. First, to check whether the proposed model and data structure transformations can be applied to real-life applications. Second, to examine the feasibility of the process of program transformation into context-flow ones. Third, to compare the performance of various CFAs and quantify the advantages, if any, of tunnel-based CFAs. Finally, to study the strengths and weaknesses of the proposed queueing-theoretic performance model.

In this section, we carry out our experiments using two real-life applications, namely, an MPEG1-LayerIII decoder from the multimedia domain, and a cryptography acceleration co-processor from the networking domain. We start by describing each application in Section 6.1, followed by a description of the process of data and computation transformation in Section 6.2. Section 6.3 presents simulation results of various system configurations and parameters for each application. Finally, performance estimation results are compared with simulation figures in Section 6.4.

6.1 Target Applications

Although synthetic benchmarks were used in the intermediate stages of the design flow of the programming model and context-flow architectures, more realistic applications need to be

considered when targeting SOC designs.

6.1.1 Overview of MPEG1-LayerIII Decoder

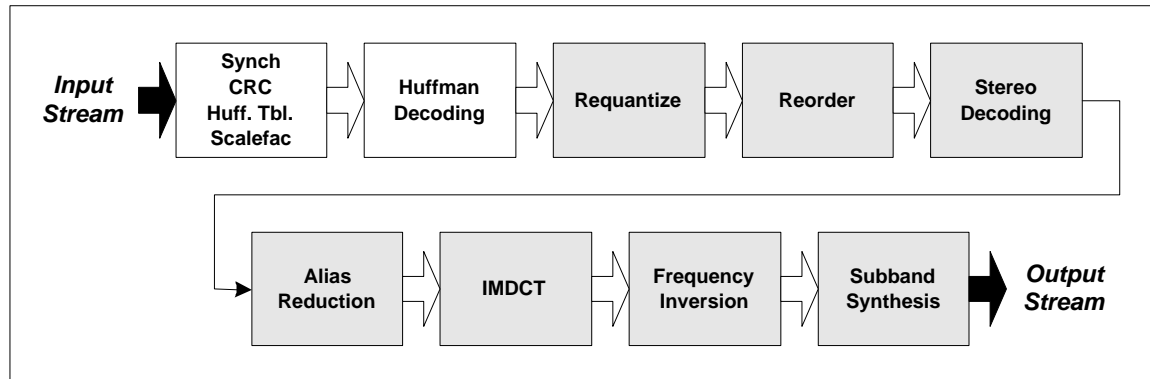


Figure 6.1: MP3 Decoder Stages

In the past few years, MPEG1-LayerIII, commonly referred to as MP3, has become the de-facto standard of high-quality high-compression of digital audio data streams. It emerged as the main tool for Internet audio delivery [10]. MP3 decoders became of central interest after their popular use in portable multimedia devices. The decoding process is computationally demanding, and power consumption is a critical constraint.

An overview of the decoder stages is presented in Figure 6.1. A detailed description of the algorithm and decoder stages was presented in [36], which implemented a portable reference MP3 decoder in C. This source code was used as the base for our pipelined implementation of the decoder. The highlighted stages were implemented in our test case. Each stage operates on data granules, one at a time, and passes these granules to the next stage in a sequential manner for further processing.

Each decoder stage is implemented in a single procedure processing one data granule at a time. Due to the absence of accurate hardware implementation performance numbers, the delay of each method is determined using the number of memory accesses per call, assuming a perfect pipeline implementation of the processors and that memory bandwidth is the primary

bottleneck. Current datapath synthesis tools, such as Module Compiler by Synopsys, can be used to pipeline computational parts of the target algorithms.

6.1.2 Cryptography Accelerator

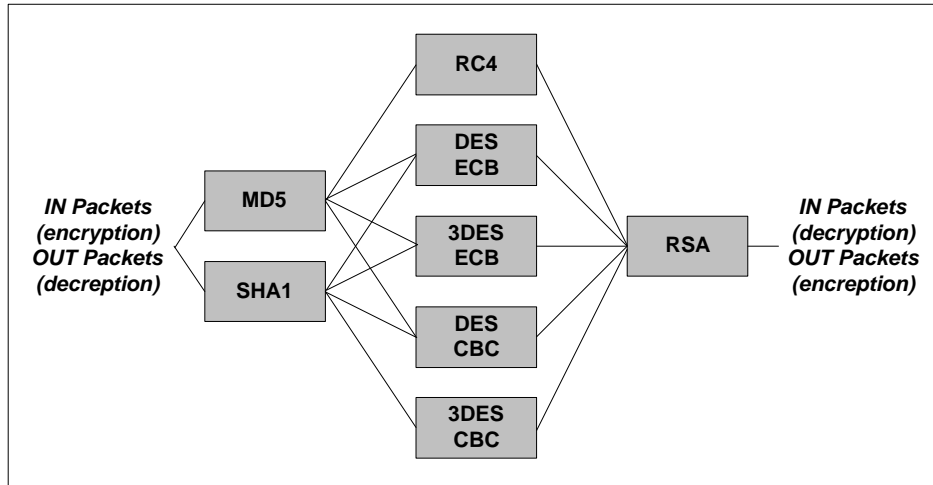


Figure 6.2: Crypto Accelerator Flow

Cryptography acceleration processors are becoming of central interest with the increase of SSL-based traffic over the Internet. Such traffic consumes a large percentage of the processing time of server CPUs to do the required mathematical/logical operations. The current trend, to avoid the expensive server replication, is to use dedicated coprocessors custom designed to accelerate such calculations and remove the load off the server processors.

In our test case, we implemented a number of symmetric and asymmetric algorithms commonly used in SSL and IPsec. A detailed description of each engine can be found in [46]. Our implementation was based on the OpenSSL project [55].

The implemented functions and the possible flows of packets are shown in Figure 6.2. Each engine was implemented as a single procedure. Delay of these processing procedures were obtained from actual RTL implementations [71] and comparison results [63]. The longest path of an input packet is to go through all three categories of processing, namely hashing (MD5 or SHA1), symmetric or private-key encryption (DES ECB, DES CBC, 3DES ECB, 3DES CBC,

or RC4), asymmetric or public-key encryption (RSA). Packets could skip hashing, public-key encryption, or both.

6.2 Data Transformation

6.2.1 Data Transformation of MP3 Decoder

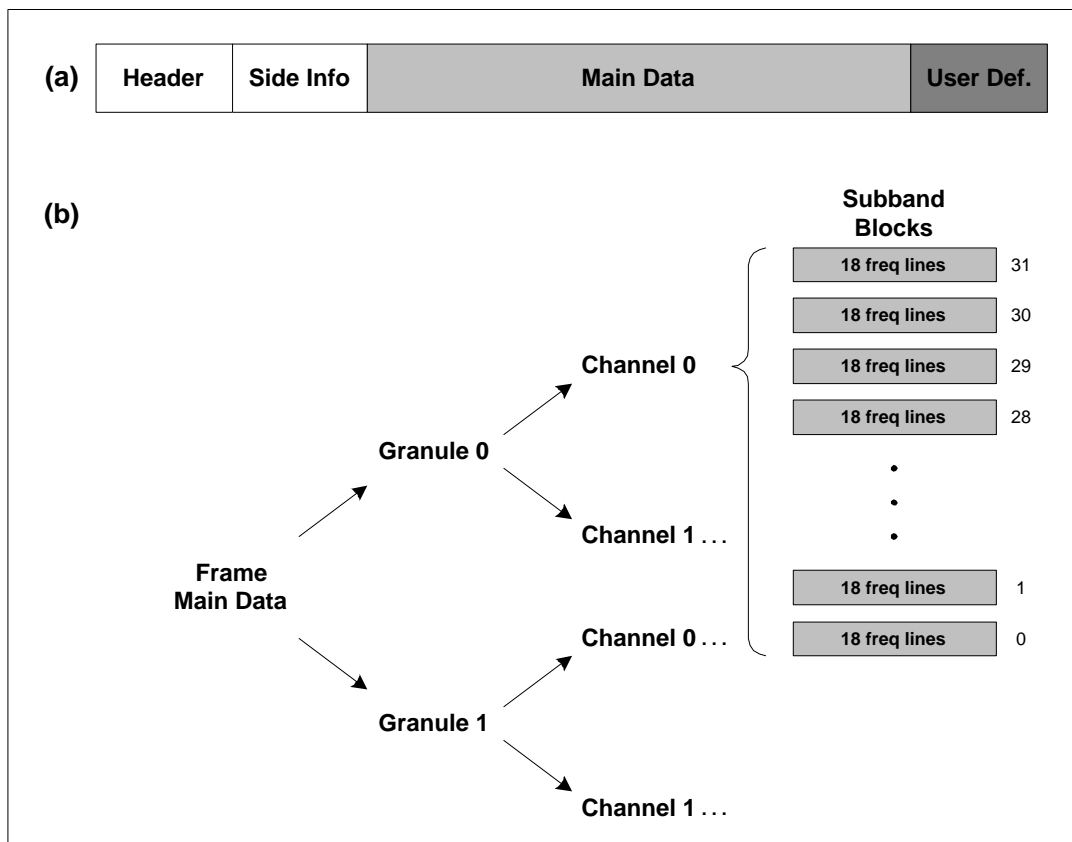


Figure 6.3: MP3 frame format

The frame is a central concept when decoding MP3 bitstreams. It consists of four parts; header, side information, main data, and some user-defined ancillary data, as shown in Figure 6.3 (a). Figure 6.3 (b) presents the logical partitioning of the frame main data. It consists of 1152 mono or stereo frequency-domain samples, divided into two granules of 576 samples each. Each granule is further divided into 32 subband blocks of 18 frequency lines apiece. The

reader is directed to [36] for a detailed definition and description of these components.

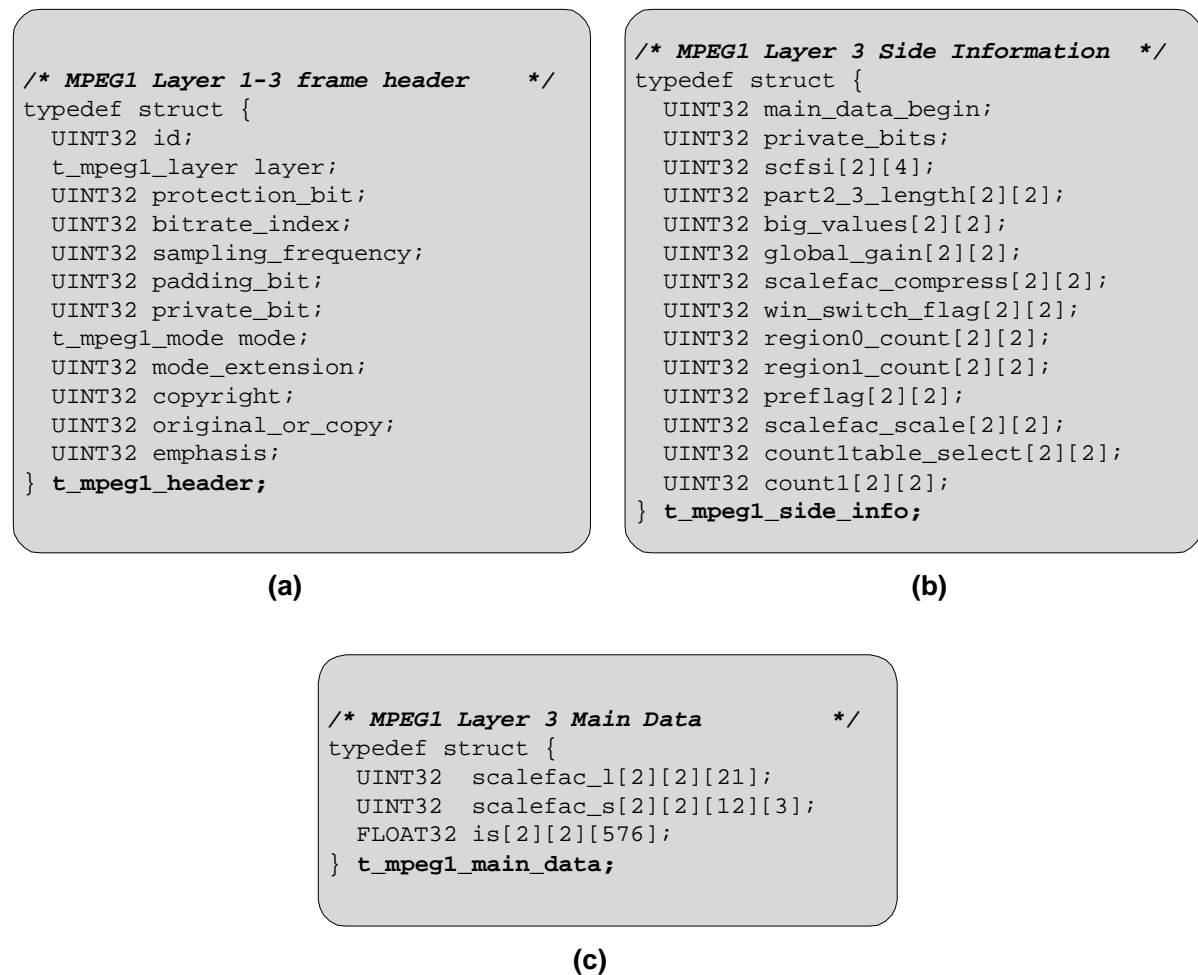


Figure 6.4: Original Data Structures in the Reference MP3 Decoder Implementation

The original data structures of the header, side information, and main data of the reference MP3 decoder implemented in [36] are listed in Figure 6.4 (a), (b), and (c), respectively. In the listed code, any [2][2] refers to [NUM OF GRANULES][NUM OF CHANNELS]. A single instance of each of these structures is statically allocated as a global variable in the reference implementation. To process an input audio file, frames are read into these objects, processed, and the decompressed audio data is returned before overwriting these objects with the next frame.

This programming methodology presented above is quite common in multimedia applications

and is suitable for sequential programming model of traditional computer systems.

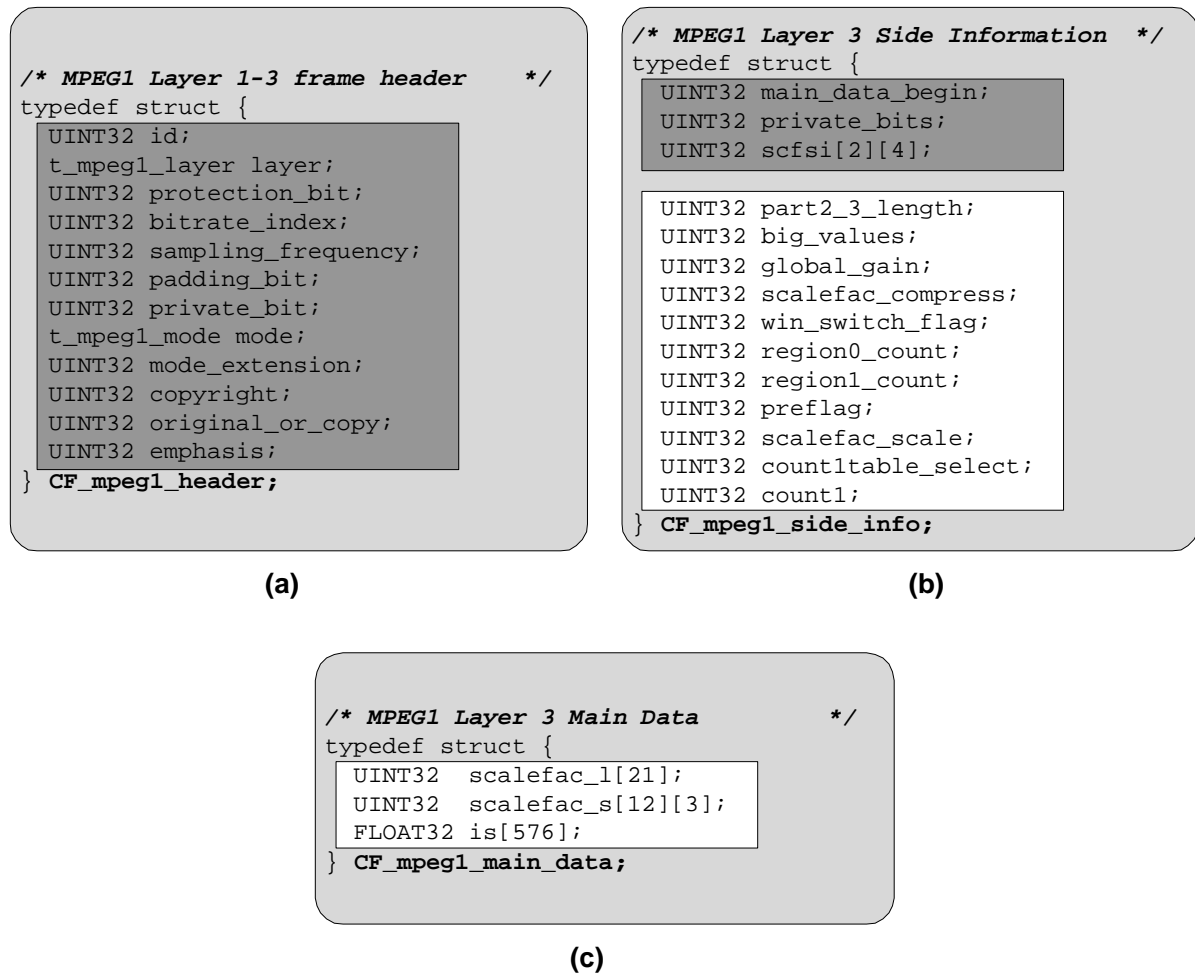


Figure 6.5: Transformed Data Structures in the Context-Flow Implementation of the MP3 Decoder

Obviously, the static objects used in the original implementation are not efficient for parallelization. We no longer have the simple sequential processing mode with a single thread of execution and non-distributed memory system. Instead, we aim to end up with a pipelined implementation at the system level running on a CFA. On a SOC, a stream of requests will be arriving at the input port of the system. Each request will be allocating a separate context, and objects of these requests, that correspond to frame components, need to be dynamically allocated on that context. Pointers to the allocated objects are passed as parameters.

The total size of the data corresponding to all channel in the side information and main data parts of the frame, shown inside white boxes in Figure 6.5, is more than 10 KBytes. This data can be partitioned into four parts, each corresponding to a separate channel, as in general each decoder stage processes one channel per granule per frame at a time. However, some read-only data shared between channels, shown inside dark grey boxes in Figure 6.5, need to be replicated. Shared data cannot, in this case, be placed in a global context with mutually exclusive access, as more than one frame may be present in the system at any point in time. The final data structures in our context-flow implementation are shown in Figure 6.5.

This data transformation results in a deeper implementation pipeline. Note that no performance is gained, as each request was replaced by four, each of which takes about one fourth of the processing time at each pipeline stage. However, smaller memory banks, almost one fourth in size, can be used to gain the same performance speedup when compared to sequential processing.

6.2.2 Data Transformation of the SSL Processor

Compared to that of multimedia applications, data transformation of network applications is in general a simpler process. The main data to be processed is usually already packetized in arrays a few KBytes in size.

In the case of our cryptography accelerator, the data to be encrypted/decrypted is an array of variable size dynamically allocated at the input port for each packet to be processed. The same applies to the encryption/decryption keys.

It is worth noting that network applications are evolving towards higher levels of abstraction, and networks routers are increasingly expected to do more than simple packet forwarding. Boundary routers, which lie on the borders between organizations, must often prioritize traffic, translate network addresses, tunnel or filter packets, or act as firewalls, among other things. In other words, network routers and processors need to handle more complex functionalities than simple data transfers and array manipulation [50]. Therefore, in addition to the other benefits

of our programming model such that safety and transparency, an efficient handling of complex data structures is still important in this domain.

6.3 Performance Experimental Results

After demonstrating the applicability and the feasibility of context-flow programming model on real-life applications, in this section we consider the performance efficiency of the proposed architecture. Several performance aspects of the proposed architecture will be illustrated:

- The importance of pipelining at the architectural level.
- Performance efficiency of tunnel-based CFA when compared with alternative implementations.
- Memory requirements of tunnel-based CFA when compared with alternative implementations.
- Performance efficiency of heterogeneous implementations when compared with homogeneous ones.
- The importance of procedural mapping/system configurations on the overall system performance.

These aspects will be illustrated separately on our test cases, starting with the SSL acceleration processor as more performance aspects could be discussed. We will arrive at similar but less diverse results for the MP3 decoder later in this section.

6.3.1 Simulation Results of the SSL Accelerator

In this application, different packets will be using different processing paths, where a path is the ordered set of procedures a specific packet follows, according to packet types. To carry out the experiment, we implemented a packet generator that generates a workload, or packet mix,

which uses various processing paths according to given distribution parameters. Table 6.1 and Table 6.2 lists the various workloads used to carry out our experiment.

In Table 6.1, each subcolumn under Workload ID presents the percentage of packets processed by each procedure in the Procedure column. For example, 52% of the packets in workload 1 are hashed using the MD5 procedure, while 47% are hashed by the SHA1 procedure. The remaining 1% skip this optional first stage of processing. Note that the figures for each workload corresponding to Stage 2 processing procedures must add up to 100%, while the other two stages are optional, and will sum up to $\leq 100\%$. Table 6.2 presents the average packet processing time, in cycles, for each procedure. For example, hashing each of the 52% of the packets in workload 1 MD5 takes, on average, 436.5 clock cycles.

Procedure		Workload ID			
		1	2	3	4
Stage 1	MD5	52	74	47	21
	SHA1	47	12	41	79
Stage 2	RC4	22	43	27	27
	DES ECB	11	9	32	10
	3DES ECB	33	19	6	29
	DES CBC	23	19	22	19
	3DES CBC	11	10	13	15
Stage 3	RSA	9	3	11	14

Table 6.1: Detailed Workload Distribution of SSL Processor (%)

For each workload distribution, we calculate what we call the *expected processing time* (EPT) for each procedure. This is defined as:

$$EPT_i = ProcessingTime_i \times Frequency_i \quad (6.1)$$

For example the EPT for procedure RSA in workload 1 is: $5056.2 \times 0.09 = 455.1$ cycles.

Procedure		Workload ID			
		1	2	3	4
Stage 1	MD5	436.5	382.9	391.0	353.4
	SHA1	438.4	350.0	324.5	414.9
Stage 2	RC4	1254.8	1225.4	1183.0	1154.2
	DES ECB	837.6	838.7	1046.5	1049.1
	3DES ECB	1063.8	764.7	747.6	1075.2
	DES CBC	2550.3	1922.5	1094.9	1516.8
	3DES CBC	6628.4	5946.7	3397.6	5462.2
Stage 3	RSA	5056.2	5051.3	5051.3	5051.3

Table 6.2: Average Processing Time Per Packet by Each Procedure (cycles)

This value represents the average processing time of procedure i on *every* input packet. The calculated effective processing times for each workload are listed in Table 6.3.

We use these EPT values to optimize our pipeline implementation. The pipelining process is similar to that of datapath pipelining in behavioral synthesis [47]. The main difference is that the order of procedure execution is not defined. Other than that, we use a similar technique to *chaining*, a commonly performed scheduling technique by considering the propagation delay, EPT in our case, of the datapath resources, procedures in our case. The corresponding mapping of each workload is presented in Table 6.4.

It is important to note, however, that this optimization process is only a heuristic for finding the architectural mapping that achieves efficient load balancing while minimizing implementation costs. It ignores the queuing time at each PE. It also considers a limited design space of two dimensions, average packet processing time, and design size. More advanced optimization, such as minimizing the processing and queuing time for a specific packet type, requires a more aggressive solution based on a detailed performance analytical model such as our queueing theoretic performance model described in Chapter 5.

Procedure		Workload ID			
		1	2	3	4
Stage 1	MD5	227.0	383.3	183.8	74.2
	SHA1	206.0	42.0	133.1	327.7
Stage 2	RC4	276.1	526.9	319.1	311.6
	DES ECB	92.1	75.5	334.9	104.9
	3DES ECB	351.1	145.3	44.9	311.8
	DES CBC	580.6	365.3	240.9	288.2
	3DES CBC	729.1	594.7	241.7	819.3
Stage 3	RSA	455.1	151.5	556.2	707.8

Table 6.3: Effective Processing Time Per Packet by Each Procedure (cycles)

The simulation results of our application for workloads specified in Table 6.1 and mappings described in Table 6.4 for various CFA implementations are shown in Figure 6.6 and Figure 6.7. The three network architectures, namely bus, perfect packet-switch (PPS), and tunnel, were evaluated. The basic configuration of each implementation has $(\#PEs + 2)$ memory modules. In case of bus-based and perfect packet-switch-based interconnects the extra two banks are used as input and output buffers of the system, as the input traffic has no direct access to memory modules at target PEs. We also varied the number of banks for each implementation to study the usage of extra memory modules. For bus-based and perfect packet-switch networks the added banks are used as extra input buffers which could be a bottleneck for the inbound traffic. For the tunnel implementation, banks are simply added/removed from the memory pool. The number of added/removed banks of each implementation is included between parenthesis in Figure 6.6. For example, BUS (+2) refers to a bus-based implementation of the target application with $(\#PEs + 4)$ memory banks, while Tunnel (-1) removes one bank from the original memory module pool.

From Figure 6.7 we can see that when using the same number of memory banks for all

Procedure		Workload ID			
		1	2	3	4
Stage 1	MD5	PE1	PE0	PE1	PE1
	SHA1	PE1	PE0	PE2	PE1
Stage 2	RC4	PE2	PE1	PE3	PE1
	DES ECB	PE2	PE2	PE1	PE2
	3DES ECB	PE2	PE2	PE1	PE2
	DES CBC	PE3	PE2	PE3	PE2
	3DES CBC	PE4	PE3	PE2	PE3
Stage 3	RSA	PE0	PE0	PE0	PE0

Table 6.4: SSL Mappings given as Target PE for Each Procedure

implementations, the tunnel-based implementation provides an average speedup of 2.08X when compared to bus-based implementation, and 1.80X when compared to packet-switch-based one. Adding more memory banks to the bus-based implementation was hardly beneficial, while adding 1 bank to the packet-switch-based implementation caused the speedup to drop to 1.53X, and adding 2 bank caused the speedup to drop to 1.17X. Even after removing 2 memory banks from the tunnel implementation and adding banks 2 to the bus-based one, we still obtain an average speedup of 1.62X. Removing 2 banks from the tunnel implementation and adding 2 banks to the packet-switch-based one, we obtain almost equally capable designs, i.e. in these experiments to achieve the performance of PPS (+2), we can save 4 memory banks simply by using the tunnel interconnect architecture.

To illustrate the importance of system-level pipelining of multi-PE implementations, we calculated an upper bound of the single PE performance. We assumed the single PE has all the processing power of custom hardware used in our heterogeneous implementations. Packets are always available for processing, and context switching happens in zero cycles. In short, the average packet processing time for the single PE implementation is simply the sum of the

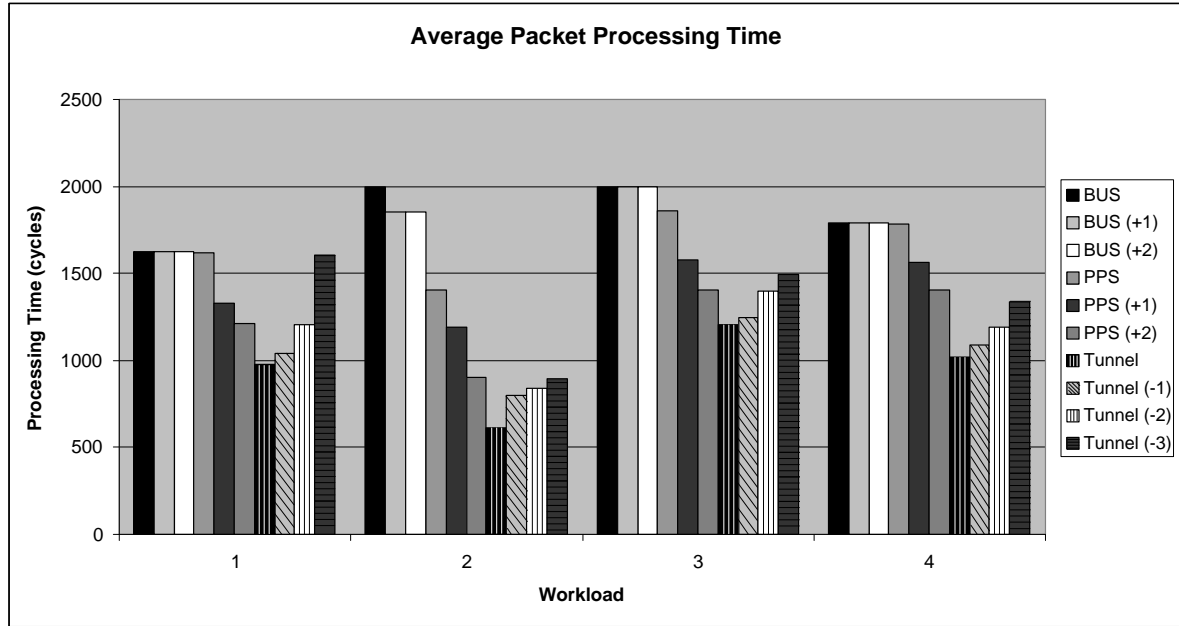


Figure 6.6: Simulation Results for SSL Acceleration Processor: Average Packet Processing Time

processing times of each packet by each procedure divided by the number of packets. A comparison between single-PE implementation and our tunnel-based context-flow implementation is illustrated in Figure 6.8. Results show the importance of parallel designs and system-level pipelining.

To illustrate the importance of custom-logic design of heterogeneous architecture, we simulated the same application running on a homogeneous architecture of identical superscalar processors, each of which is assumed have an instruction level parallelism (ILP) of 8 instructions per cycle, with perfect branch prediction, and infinite bandwidth to the memory system. One of these theoretical processors was dedicated to each procedure. Obviously, these assumptions provide a very loose upper bound for the possible performance of near-future embedded processors. To ensure the accuracy of our heterogeneous performance figures, we only run our experiments utilizing crypto engines with documented performance measures of actual RTL implementations [71, 54]. Figure 6.9 reports the results of both implementations with the same interconnect and memory resources for various workloads. Reported results with speedup rang-

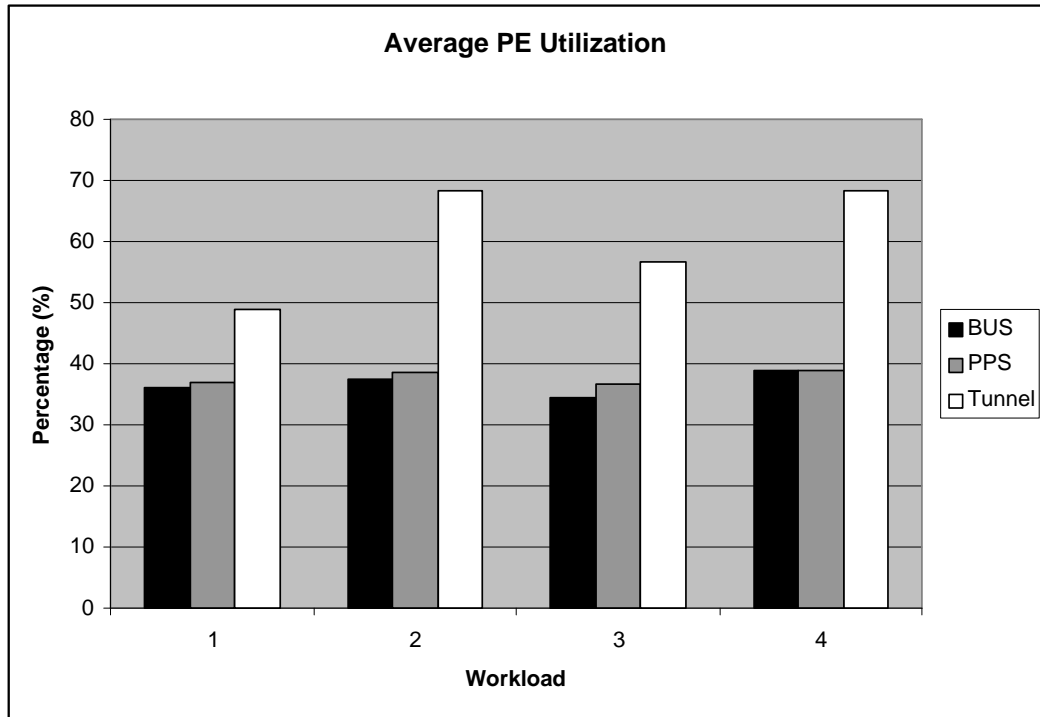


Figure 6.7: Simulation Results for SSL Acceleration Processor: Average PE Utilization

ing from 4.8X to 14.4X over an optimistic superscalar implementation reflects the importance of heterogeneous implementations

And finally, to illustrate the importance of procedural mappings on overall system performance, we simulated several tunnel-based configurations of the SSL accelerator, each of which is optimized for a specific workload; Mapping 1 was optimized for Workload 1, Mapping 2 for Workload 2, and Mapping 3 for Workload 3. The effects of workload rotation is shown in Figure 6.10, showing the result of ignoring a crucial design decision of using a non-optimal mapping for a given workload.

To conclude, we can derive several observations from simulation results:

- Our fastest pipelined implementation has an average of 2.68X speedup when compared to the single PE implementation with the same processing power. This gap shows the importance of pipelined implementations.
- Our tunnel-based implementation provides an average speedup of 43% when compared

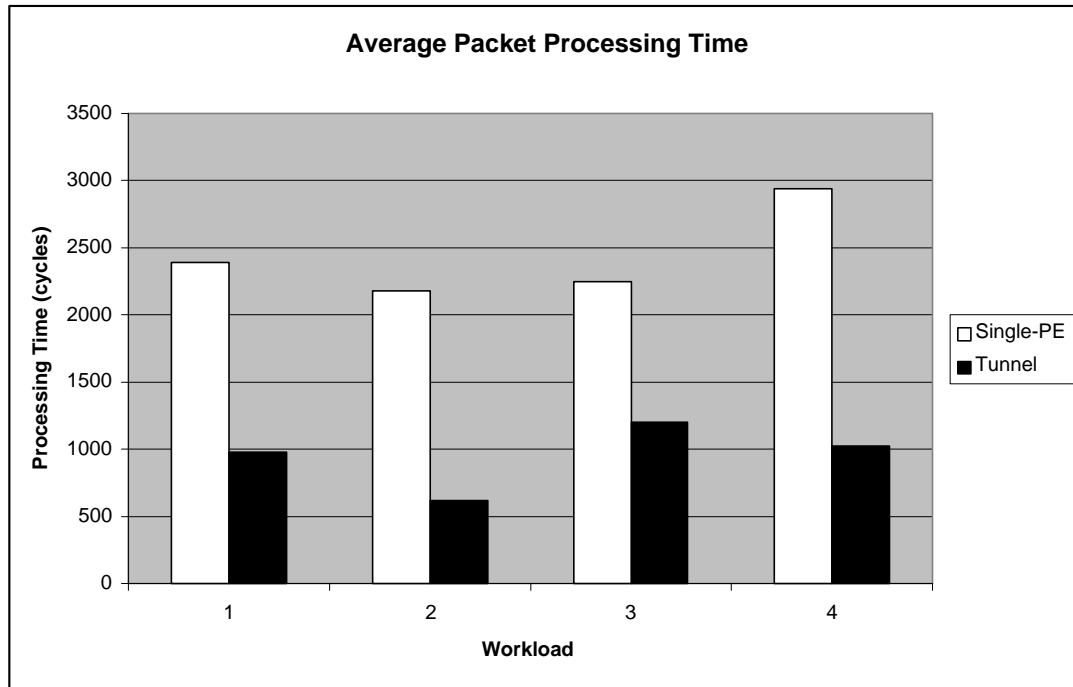


Figure 6.8: Performance Results of Single-PE and Multi-PE Implementations of the SSL Accelerator

to bus-based implementation, and 23% when compared to perfect packet-switch network implementation with the extra memory banks.

- Several memory banks can be saved when compared to bus-based and perfect packet-switch network implementations to arrive at a comparable or even better performance.
- The heterogeneous implementation is 11X faster than our optimistic homogeneous one, which shows that the programmable regular homogeneous solutions pose serious performance disadvantages.
- Mapping has a great effect on the overall system performance, which necessitates system-level architectural exploration for synthesis.

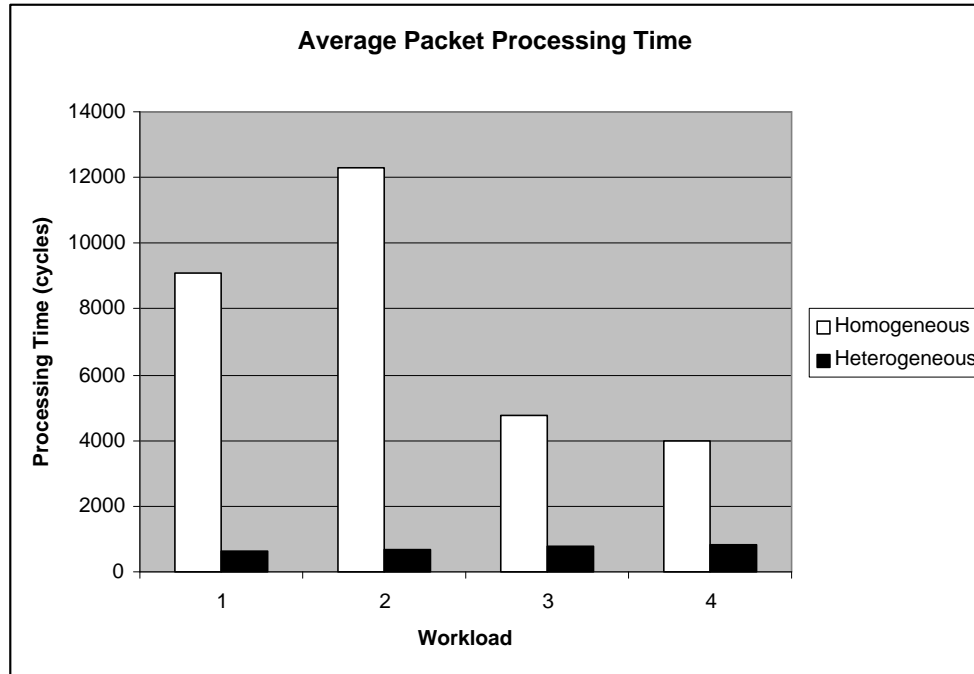


Figure 6.9: Performance Results of Homogeneous and Heterogeneous Implementations of the SSL Accelerator

6.3.2 Simulation Results of MP3 Decoder

Although it was a more interesting test case when considering context recognition and data transformations, the MP3 decoder is a simpler design when compared to the SSL processor in terms of configurability. There is only one well-defined linear path traversed by each frame that enters the system, starting with huffman decoding all the way to subband synthesis (Figure 6.1). Besides, no interesting variability in workload could be used to arrive at an optimum design mappings.

To carry out our experiment, we use some of the input files distributed along with the standard MP3 software. We use the same pipelining technique described in Section 6.3.1 to arrive at a suitable procedural mapping that optimizes average frame processing time.

Simulation results are shown in Table 6.5, where the second column reports the throughput in cycles per request. The third column reports the average PE utilization. Similar observations to those of the previous section could be drawn. For multi-PE configurations, PE0 implements

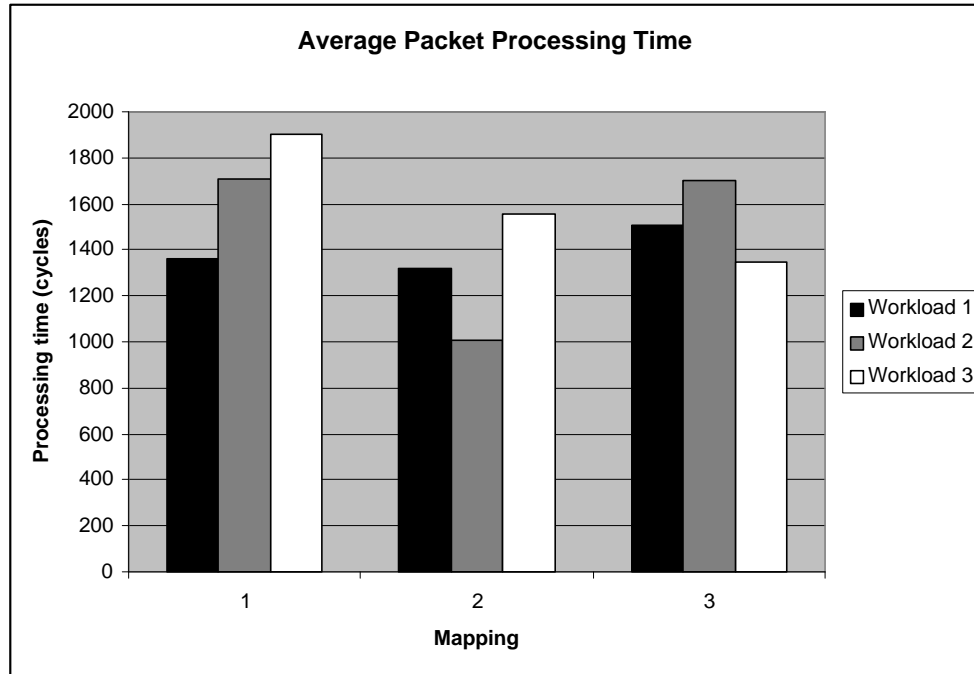


Figure 6.10: Mapping Effects on System Performance for the SSL Accelerator

stages 1, 2, and 3, PE1 implements 4 and 6, PE2 implements 5, and PE3 implements stage 7.

We did not carry out a homogeneous vs. heterogeneous experiment in this case due to the lack of accurate cycle count estimates of a custom hardware implementation of the decoder stages.

6.4 Evaluation of Queueing-Theoretic Performance Estimation Model

To evaluate our performance estimation model, we use experimental setups similar to those used in Section 6.3. The only difference is that we do not assume saturated input traffic to measure the maximum possible throughput, where a request is always available at the system input port for processing. Instead, we assume what we call *jittered-periodic traffic*, where requests arrive at the input port periodically with some variance specified by the user.

A GI/G/1 network analysis tool similar to that reported by W. Whitt in [76] was imple-

Architecture	Throughput	PE Util.
Tunnel	3439	71%
Tunnel (-1)	3515	70%
Tunnel (-2)	3537	69%
Tunnel (-3)	3655	67%
Single-PE	9800	100%
Shared-bus	5944	41%
Perf. Packet Switch	5043	48%

Table 6.5: MP3 Decoder Results

mented to become the core of our estimation model. We ran several experiments for each test case. For each of these experiments, we compare the residence time for each PE given by simulation to that of our estimation. Waiting/queueing time is the only basic performance measure where considerable errors in estimation could result. Other measures are either directly dependent on the accuracy of input statistics, or composite measures directly dependent on queueing time. We report the total residence time as it is the measure of most interest to us. Besides, large errors in queueing time estimated could be negligible when compared to the overall residence time. For example, if the actual average queueing time at a given PE is 10 cycles, and the average processing time is 5000 cycles, and the estimated queueing time is, say, 40 cycles, then the estimation error with respect to queueing time is 300%! However, this error is not really an issue when compared to the overall residence time of 5010 cycles. Other measures such as queue size, total residence time for each packet type, etc. are simply composite measures.

In case of the SSL accelerator, for a given workload we used the different mappings described in Table 6.6. The corresponding estimation results are reported in Table 6.8, and the average estimation errors for each mapping over all PEs are presented in Figure 6.11. Similarly, for the MP3 decoder we tried the mappings described in Table 6.7, and the corresponding estimation results are reported in Table 6.9 and Figure 6.11.

Mapping	Target PE							
	RSA	MD5	SHA1	RC4	DES ECB	3DES ECB	DES CBC	3DES CBC
1	PE0	PE1	PE2	PE3	PE4	PE5	PE6	PE7
2	PE0	PE1	PE1	PE2	PE3	PE3	PE4	PE5
3	PE0	PE1	PE1	PE2	PE2	PE2	PE1	PE3
4	PE0	PE1	PE1	PE0	PE2	PE2	PE1	PE3

Table 6.6: SSL Accelerator Mappings for Performance Model Evaluation

Mapping	Target PE						
	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Stage 6	Stage 7
1	PE0	PE0	PE0	PE1	PE2	PE1	PE3
2	PE0	PE1	PE2	PE3	PE3	PE4	PE5
3	PE0	PE1	PE1	PE1	PE2	PE3	PE4

Table 6.7: MP3 Decoder Mappings for Performance Model Evaluation

From the reported results, we can see that the estimation results were accurate in some cases, and varied (either high or low) in others, but correctly reported the relative waiting time values at different PEs with acceptable average error (Figure 6.11). It turned out that the way the solver handles multi-class networks through simple aggregation could potentially be improved. To illustrate this issue, mapping 3 of the SSL test case was intentionally configured such that procedures with largely different processing times were mapped to the same PEs. Also, the use of a single variability parameter to characterize the variability of an arrival process to a queue was not optimal. More advanced solutions were reported in [77], and further enhancements to queueing network solvers are being proposed in this active area of research, which is outside the scope of this work. However, as we observed in our experiments, the used solver still serves as a first order approximation of the queueing time at each PE. For example, the solver does not report a waiting time in thousands of cycles while the actual value is only in hundreds, or vice versa.

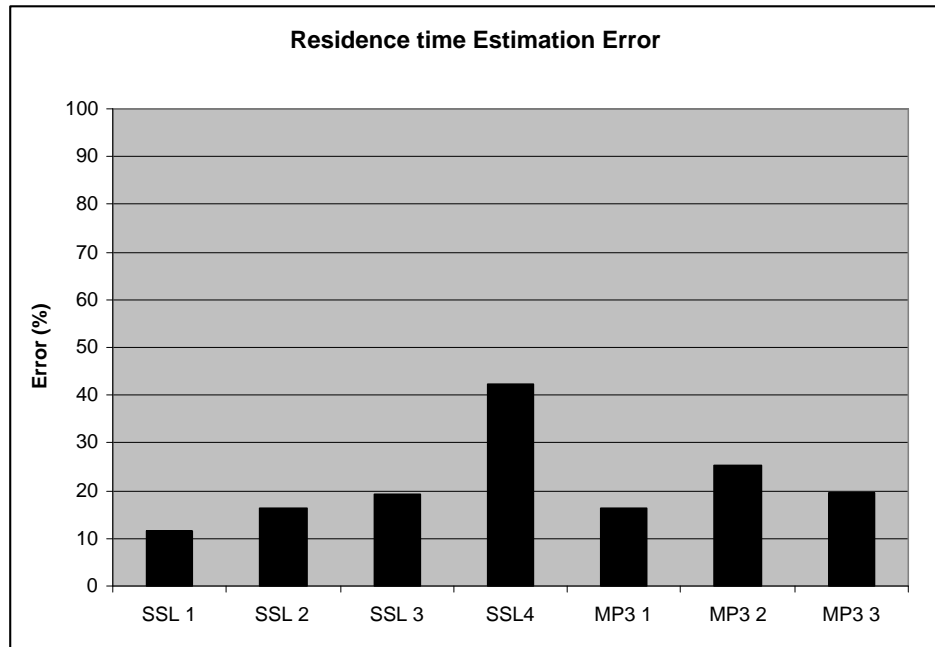


Figure 6.11: Queueing Model Accuracy for SSL Accelerator and MP3 Decoder

Although higher accuracy levels would have been appreciated, our proposed model is still valid, and it gets as accurate, flexible, and powerful as queueing theory itself. Even at the reported accuracy measures, the model will provide important optimization directions as part of a system-level optimization framework.

Mapping	Proc. Element	Sim. Residence Time	Est. Residence time	Error
1	PE0	10255.5	7027.6	31.5%
	PE1	462.1	413.6	10.5%
	PE2	547.8	510.5	6.8%
	PE3	1864.9	1844.9	1.1%
	PE4	1144.5	1270.2	11.0%
	PE5	1244.4	1360.1	9.3%
	PE6	3396.5	3968.6	16.8%
	PE7	7645.9	7209.9	5.7%
2	PE0	8360.8	7020.0	16.0%
	PE1	650.9	453.9	30.3%
	PE2	1843.5	1840.1	0.2%
	PE3	2450.6	1923.6	21.5%
	PE4	3561.0	3736.5	4.9%
	PE5	8189.6	6164.9	24.7%
3	PE0	5646.9	6360.2	12.6%
	PE1	2399.0	2119.0	11.7%
	PE2	4020.1	2890.8	28.1%
	PE3	6459.6	4850.8	24.9%
4	PE0	46772.2	84003.1	79.6%
	PE1	2147.8	1206.3	43.8%
	PE2	1349.3	1265.0	6.2%
	PE3	6150.7	3595.8	41.5%

Table 6.8: Simulated and Estimated Residence Time for SSL Accelerator

Mapping	Proc. Element	Sim. Residence Time	Est. Residence time	Error
1	PE0	3621.8	2544.7	29.7%
	PE1	1301.9	1781.6	36.8%
	PE2	9246.7	9891.0	7.0%
	PE3	2756.0	4925.3	78.7%
2	PE0	1271.4	1288.6	1.4%
	PE1	832.5	972.6	16.8%
	PE2	1417.4	1576.9	11.3%
	PE3	11519.9	5918.8	48.6%
	PE4	764.8	786.0	2.8%
	PE5	2868.6	3340.9	16.5%
3	PE0	1271.4	1288.6	1.4%
	PE1	2242.9	2340.8	4.4%
	PE2	11533.5	3000.9	74.0%
	PE3	764.8	785.9	2.8%
	PE4	2873.2	3337.8	16.2%

Table 6.9: Simulated and Estimated Residence Time for MP3 Decoder

Chapter 7

Conclusion and Future Work

7.1 Conclusions

This thesis offers three main contributions. First, it proposed a new programming model for SOC designs based on an abstraction of autonomous dynamic data structures called *context*. This programming model is *high-level*, not so much different from imperative programming models, *simple*, even simpler than traditional imperative programming models such as that of C, and *safe*, due to the simple garbage collection embedded within the model. Yet, the model is *inexpensive* to realize in hardware. The second contribution was the development of a new SOC architectural platform, called *context-flow architecture*, which is based on a new on-chip network called *tunnel*. The proposed platform makes a better utilization of the on-chip resources when compared to the recently proposed communication-centric platforms. Finally, a new queueing-theoretic performance estimation model was proposed for our communication-centric platform. This model is simple, flexible, synthesis-friendly, and bounded only by the capabilities of queueing theory.

Based on our study, we draw several main conclusions. First, the context-flow programming model is suitable for realizing SOC applications of interest, namely multi-media and network applications. Secondly, the proposed communication-centric platform is shown to be

performance efficient when compared to alternative solutions. Thirdly, the scalability limitation of the tunnel-based CFAs are not an issue when considering the target applications. It is only when more than one application needs to be mapped to a single die when the interconnect scalability becomes an issue. This is when a second-level interconnect architecture comes into play. And finally, although the proposed performance model seems to be the ideal solution for our platform, a more accurate network analyzer, probably with a solver less general than the GI/G/n one, but still based on our approach would be a powerful solution.

7.2 Future Work

The work in this thesis opens a lot of scope for future work in several areas in the field of SOC design. The major step would be the automatic transformation of sequential programs into context-flow ones. This involves automatic context-recognition and code partitioning, which is a challenging task when considering dynamically allocated objects of complex architectures commonly used in interesting application.

Once synthesizing a complete application on a single cluster of a complete system of two-level interconnect architecture is well defined, studying the behavior of traffic on that second level network becomes an important task. So far, only random traffic [17] and small applications like MPEG decoder [72], small enough to accommodate on a single cluster or even a single PE, were used to study the behavior of these networks, which are far from being representative of the projected traffic between various applications running on different tiles of a large SOC.

After having a better understanding of the behavior of on-chip traffic, the queueing model proposed in this work will be enhanced and used as an essential part of complete system-level synthesis tool.

Bibliography

- [1] L. Adams. Overview of the CoreFrame Architecture. Technical report, Palmchip, January 2002.
- [2] Altera. *Excalibur Devices*. <http://www.altera.com/products/devices/arm/arm-index.html>.
- [3] B. S. Amrutur. *Design and Analysis of Fast Low Power SRAMs*. PhD thesis, Stanford University, August 1999.
- [4] ARM. *PrimeXsys Platforms Overview*. <http://arm.com/products/solutions/PrimeXsysPlatforms.html>.
- [5] ARM. AMBA Specification (Rev 2.0). Technical report, Advanced RISC Machines (ARM), 1999.
- [6] J. Babb, M. Rinard, C. A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing Applications into Silicon. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 70–80, April 1999.
- [7] A. Baghdadi, N.-E. Zergainoh, W. O. Cesario, and A. A. Jerraya. Combining a Performance Estimation Methodology with a Hardware/Software Codesign Flow Supporting Multiprocessor Systems. *IEEE Transactions on Software Engineering*, 28(9), September 2002.
- [8] L. Benini and G. D. Micheli. System-Level Power Optimization: Techniques and Tools. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(2):115–192, April 2000.
- [9] L. Benini and G. D. Micheli. Networks on Chips: A New SOC Paradigm. *Computer*, 35:70–78, January 2002.
- [10] K. Brandenburg and H. Popp. An Introduction to MPEG Layer-3. *EBU Technical Review*, June 2000.
- [11] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical report, Computer Science Department, University of Wisconsin, 1997.

- [12] Cadence. *Encounter Digital IC Design Platform*. http://www.cadence.com/products/digital_ic/.
- [13] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.
- [14] W. J. Dally and S. Lacy. VLSI Architecture: Past, Present, and Future. In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, March 1999.
- [15] W. J. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Proceeding of the 38th Design Automation Conference*, pages 684–689, June 2001.
- [16] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, December 2003.
- [17] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: Task Graphs for Free. In *International Workshop on Hardware/Software Codesign*, March 1998.
- [18] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks*. Morgan Kaufmann, 2002.
- [19] T. Dumitras, S. Kerner, and R. Marculescu. Towards On-Chip Fault-Tolerant Communication. In *Asia and South Pacific Design Automation Conference*, January 2003.
- [20] A. Ferrari and A. Sangiovanni-Vincentelli. System Design: Traditional Concepts and New Paradigms. In *IEEE International Conference on Computer Design*, October 1999.
- [21] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.
- [22] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Boston, March 2000.
- [23] J. Golbus, B. Gribstad, C. Kozyrakakis, and K. Wang. Interconnection Issues Between Memory and Logic in IRAM Systems. Technical report, University of California, Berkeley, 1997.
- [24] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist. Network on a Chip: An Architecture for Billion Transistor Era. In *Proceedings of IEEE NorChip Conference*, November 2000.
- [25] M. Horowitz, R. Ho, and K. Mai. The Future of Wires. In *Proceedings of the IEEE*, pages 490–504, April 2001.

- [26] J. Hu and R. Marculescu. Exploiting the Routing Flexibility for Energy/Performance Aware Mapping of Regular NOC Architectures. In *Proceedings of the Design Automation and Test Conference in Europe*, March 2003.
- [27] IBM. The CoreConnect Bus Architecture. Technical report, IBM, 1999.
- [28] T. Instruments. *OMAP Processors for Wireless Devices*. http://focus.ti.com/graphics/omap/omap_020603.pdf.
- [29] ITRS. International Technology Roadmap for Semiconductors, 2001 Edition. Technical report, Semiconductor Industry Association, 2001.
- [30] A. Jalabert, S. Murali, L. Benini, and G. D. Micheli. pipesCompiler: A Tool for Instantiating Application Specific Networks on Chip. In *Proceedings of the Design Automation and Test Conference in Europe*, March 2004.
- [31] A. A. Jerraya, A. Baghdadi, W. Cesario, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, and S. Yoo. Application-Specific Multiprocessor Systems-on-Chip. *Microelectronics Journal, Elsevier Science*, 33(11):891–898, November 2002.
- [32] A. Kalavade and P. Moghe. A Tool for Performance Estimation of Networked Embedded End-Systems. In *Proceeding of the 35th Design Automation Conference*, June 1998.
- [33] K. Keutzer. Programmable Platforms Will Rule. *EETimes*, September 2002.
- [34] K. Keutzer, S. Malik, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design*, 19(12), December 2000.
- [35] S. Kumar, A. Jantsch, M. Millberg, J. Oberg, J.-P. Soininen, M. Forsell, K. Tiensyrja, and A. Hemani. A Network on Chip Architecture and Design Methodology. In *IEEE Computer Society Annual Symposium on VLSI*, April 2002.
- [36] K. Lagerstrom. Design and Implementation of an MPEG-1 Layer III Audio Decoder. Master's thesis, Chalmers University of Technology, May 2001.
- [37] K. Lahiri, A. Raghunathan, and S. Dey. Fast Performance Analysis of Bus-Based System-On-Chip Communication Architectures. In *Proceedings of the International Conference on Computer-Aided Design*, November 1999.

- [38] K. Lahiri, A. Raghunathan, and S. Dey. Performance Analysis of Systems with Multi-Channel Communication Architectures System-On-Chip Communication Architectures. In *13th International Conference on VLSI Design*, January 2000.
- [39] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance, Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., February 1984.
- [40] L. Li, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, and I. Kadayif. CCC: Crossbar Connected Caches for Reducing Energy Consumption of On-Chip Multiprocessors. In *EUROMICRO Symposium on Digital System Design, Architectures, Methods and Tools (DSD'03)*, September 2003.
- [41] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: a modular reconfigurable architecture. In *28th Annual International Symposium on Computer Architecture (ISCA)*, pages 161–171, June 2000.
- [42] S. Malik, M. Martonosi, and Y.-T. S. Li. Static Timing Analysis for Embedded Software. In *Proceeding of the 34th Design Automation Conference*, June 1997.
- [43] R. Marculescu and A. Nandi. Probabilistic Application Modeling for System-Level Performance Analysis. In *Proceedings of the Design Automation and Test Conference in Europe*, March 2001.
- [44] G. Martin and H. Chang. *Winning the SoC Revolution: Experience in Real Design*. Kluwer Academic Publishers, 2003.
- [45] A. Mathur, A. Dasdan, and R. K. Gupta. Rate Analysis of Embedded Systems. *ACM Transaction on Design Automation of Eletronic Systems*, 44(3), July 1998.
- [46] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 5 edition, October 1996.
- [47] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.
- [48] *Microsoft Web Site*. <http://www.microsoft.com/com/>.
- [49] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch. The Nostrum Backbone - A Communication Protocol Stack for Networks on Chip. In *Proceedings of the VLSI Design Conference*, January 2004.
- [50] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. In *17th ACM Symposium on Operating Systems Principles (SOSP)*, December 1999.
- [51] *Message Passing Interface (MPI) Web Site*. <http://www-unix.mcs.anl.gov/mpi>.

- [52] S. Murali and G. D. Micheli. Bandwidth-Constrained Mapping of Cores onto NoC Architectures. In *Proceedings of the Design Automation and Test Conference in Europe*, March 2004.
- [53] *OMG Web Site*. <http://www.omg.org/>.
- [54] *OpenCores Web Site*. <http://www.opencores.org>.
- [55] *OpenSSL Project Web Site*. <http://www.openssl.org>.
- [56] P. P. Pande, C. Grecu, A. Ivanov, and R. Saleh. Switch-Based Interconnect Architecture for Future Systems on Chip. In *Proceedings of SPIE, VLSI Circuits and Systems*, 2003.
- [57] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM: IRAM. *IEEE Micro*, 17(2):34–44, /1997.
- [58] L.-S. Peh. *Flow Control and Micro-Architectural Mechanisms for Extending the Performance of Interconnection Networks*. PhD thesis, Stanford University, August 2001.
- [59] Philips. *Nexperia: Streaming Media for Advanced Multimedia Applications*. <http://www.semiconductors.philips.com/products/nexperia/>.
- [60] J. Russell. Literature Survey: Software Performance Estimation. Technical report, University of Texas at Austin, June 2001.
- [61] A. Sangiovanni-Vincentelli. Defining Platform-Based Design. *EEdesign*, February 2002.
- [62] F. Schirrmester, M. Meindl, and S. Krolikoski. IP Authoring and Integration for HW/SW Co-Design and Reuse – Lessons Learned. In *Proceedings of the 9th IEEE/DATC Electronic Design Processes Workshop (EDP)*, April 2002.
- [63] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*. John Wiley & Sons, second edition, October 1995.
- [64] Sematech. *International Technology Roadmap for Semiconductor*. <http://public.itrs.net>.
- [65] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the System-on-a-Chip Interconnect Woes Through Communication-Based Design. In *Proceeding of the 38th Design Automation Conference*, June 2001.
- [66] R. L. Sites. It’s the memory, stupid! *Microprocessor Report*, 10(10):19–20, August 1996.
- [67] StarCore. *SC1000 Platforms*. <http://www.starcore-dsp.com/>.
- [68] Synopsys. *Galaxy Design Platform*. http://www.synopsys.com/products/solutions/galaxy_platform.html.

- [69] *SystemC Web Site*. <http://www.systemc.org>.
- [70] L. Thiele, S. Chakraborty, M. Gries, and S. Kunzli. A Framework for Evaluating Design Tradeoffs in Packet Processing Architectures. In *Proceeding of the 39th Design Automation Conference*, June 2002.
- [71] R. Usselmann. DES/Triple DES IP cores, September 2001.
- [72] G. Varatkar and R. Marculescu. On-Chip Communication Analysis for Multimedia Applications. In *IEEE International Conference on Multimedia and Expo*, August 2002.
- [73] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, September 1997.
- [74] H.-S. Wang, L.-S. Peh, and S. Malik. Power-Driven Design of Router Microarchitectures in On-Chip Networks. In *36th International Symposium on Microarchitecture (MICRO)*, November 2003.
- [75] H.-S. Wang, X.-P. Zhu, L.-S. Peh, and S. Malik. Orion: A Power-Performance Simulator for Interconnection Networks. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, January 2003.
- [76] W. Whitt. The Queueing Network Analyser. *The Bell System Technical Journal*, 62(9):2779–2815, November 1983.
- [77] W. Whitt. Towards Better Multi-Class Parametric-Decomposition Approximations For Open Queueing Networks. *Annals of Operations Research*, 48:221–248, 1994.
- [78] D. Wingard. MicroNetwork-Based Integration for SOCs. In *Proceeding of the 38th Design Automation Conference*, pages 673–677, June 2001.
- [79] Xilinx. *ASMBL: Revolutionary Platform FPGA Architecture Delivering Highest Value*. <http://www.xilinx.com/products/asmb1/index.htm>.
- [80] T. T. Ye and G. D. Micheli. Physical Planning for Multiprocessor Networks and Switch Fabrics. In *Proceeding of the 14th International Conference on Application-Specific Systems, Architectures and Processors*, June 2003.
- [81] W. Ye, R. Ernst, T. Benner, and J. Henkel. Fast Timing Analysis for Hardware-Software Cosynthesis. In *International Conference on Computer Design*, June 1993.

- [82] T.-Y. Yen and W. Wolf. Performance Estimation for Real-time Distributed Embedded Systems. In *International Conference on Computer Design*, June 1995.
- [83] H. Zhang, M. Wan, V. George, and J. Rabaey. Interconnect Architecture Exploration for Low-Energy Reconfigurable Single-Chip DSPs. In *IEEE Computer Society Workshop on VLSI*, pages 2–8, April 1999.