Reconfigurable Architectures for Embedded Systems

Henrik Svensson

Lund 2008

The Department of Electrical and Information Technology
Lund University
Box 118, S-221 00 LUND
SWEDEN

This thesis is set in Computer Modern 10pt,
with the LaTeX Documentation System
using Pontus Åströms thesis template.

Series of licentiate and doctoral thesis
No. 10
ISSN 1654-790X

Printed in Sweden by Tryckeriet E-huset, Lund.
September 2008

Abstract

Application-specific circuits are used to migrate computer systems from workstations to handheld devices that need real-time performance within the budget for physical size and energy dissipation. However, these circuits are inflexible as any modification requires redesign and refabrication, which is both expensive and time-consuming considering the complexity of recent embedded platforms. Therefore, reconfigurable architectures that can be dynamically reconfigured and reused over several platforms have been suggested, and they have proven to provide high performance in a wide range of applications.

This thesis focuses on two important topics when designing reconfigurable embedded systems: coarse-grained reconfigurable architectures and system level architectural exploration. It is argued that embedded systems that require programmable hardware acceleration of regular computation intensive kernels with word-level arithmetic should utilize coarse-grained reconfigurable architectures. It is also argued that design of these complex systems should be performed with tools for efficient modeling, simulation, and architectural exploration in order to analyze and tune design parameters before a chip is fabricated. This thesis presents two different coarse-grained reconfigurable architectures and a modeling and exploration environment to build and explore complete reconfigurable computing platforms.

The first reconfigurable architecture consists of a number of locally interconnected processing elements and memory banks. The processing elements are configured into customized datapaths and the memory banks are used to move data back and forth between datapath and memory at high data rates. The reconfigurable architecture was integrated as a coprocessor and used to accelerate the G.723.1 speech codec. It is shown that the number of used clock cycles is reduced with 83% compared to processor only execution. The second reconfigurable architecture is built as an array of small instruction set processors and memory blocks, which are interconnected with local dedicated wires and a global hierarchical routing network. To address efficient architectural exploration, a SystemC exploration environment with user-interactive control is presented. The reconfigurable architecture is described as a scalable and parameterizable SystemC transaction level model. To evaluate a complete system, models of instruction set processors, busses, and memories have been developed.

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.

T. S. Eliot (1888 - 1965)

Contents

Preface

This thesis summarizes my academic work in the digital ASIC group at the department of Electrical and Information Technology. The main contribution to the thesis is derived from the following publications:

- Henrik Svensson, Thomas Lenart, and Viktor Öwall, "System Level Modeling of a Reconfigurable Computing Platform," In preparation for ACM Transactions on Reconfigurable Technology and Systems.

- Thomas Lenart, Henrik Svensson, and Viktor Öwall, "Modeling and Exploration of a Reconfigurable Architecture for Digital Holographic Imaging," in *Proceedings of IEEE International Symposium on Circuits and Systems*, Seattle, USA, pp. 248-251, May 18-21, 2008.

- Henrik Svensson, Thomas Lenart, and Viktor Öwall, "Modelling and Exploration of a Reconfigurable Array using SystemC TLM," in *Proceedings of Reconfigurable Architectures Workshop*, Miami, Florida, USA, April 14-15, 2008.

- Thomas Lenart, Henrik Svensson, and Viktor Öwall, "A Hybrid Interconnect Network-on-Chip and a Transaction Level Modeling approach for Reconfigurable Computing," in *Proceedings of IEEE International Symposium on Electronic Design, Test and Applications*, Hong Kong, China, pp. 398–404, January 23-25, 2008.

- Henrik Svensson, Thomas Lenart, and Viktor Öwall, "Implementing the G.723.1 Speech CODEC using a Coarse-Grained Reconfigurable Coprocessor," in *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, USA, pp. 195–198, June 25-28, 2007.

- Henrik Svensson, Thomas Lenart, and Viktor Öwall, "Accelerating Vector Operations by Utilizing Reconfigurable Coprocessor Architectures," in *Proceedings of IEEE International Symposium on Circuits and Systems*, New Orleans, USA, pp. 3972–3975, May 27-30, 2007.

- Henrik Svensson, Viktor Öwall, and Krzysztof Kuchcinski, "Implementation Aspects of a Novel Speech Packet Loss Concealment Method," in *Proceedings of IEEE International Symposium on Circuits and Systems*, Kobe, Japan, pp. 2867–2870, May 23-26, 2005.

The following papers are also published, but not considered part of this thesis:

- Hugo Hedberg, Thomas Lenart, Henrik Svensson, "A Complete MP3 Decoder on a Chip," in *Proceedings of Microelectronic Systems Education, MSE'05*, pp. 103–104 June 12-14, 2005, Anaheim, California, USA.
- Hugo Hedberg, Joachim Neves Rodrigues, Fredrik Kristensen, Henrik Svensson, Matthias Kamuf, and Viktor Öwall, "Teaching Digital ASIC Design to Students with Heterogeneous Previous Knowledge," in *Proceedings of Microelectronic Systems Education, MSE'05*, pp. 15–16, June 12-14, 2005, Anaheim, California, USA.
- Hugo Hedberg, Thomas Lenart, Henrik Svensson, Peter Nilsson and Viktor Öwall, "Teaching Digital HW-Design by Implementing a Complete MP3 Decoder," in *Proceedings of Microelectronic Systems Education, MSE'03*, pp. 31–32, June 1-2, 2003, Anaheim, California, USA.

Acknowledgment

The years at the department have given me opportunities to meet many inspiring people and visit many exciting places. It has indeed been a pleasant and challenging time from which I will have many good memories.

First, I would like to express my gratitude to my advisor, associate professor Viktor Öwall. He has guided me on my studies throughout these years, always encouraged me to try my ideas, and showed me how research may be balanced with other pleasures. I would also like to show my appreciation to my co-advisor professor Krzysztof Kuchcinski for all his guidance and discussions and for his inspiring enthusiasm for academic research and embedded systems. I would like to thank in particular PhD Thomas Lenart for valuable input to this thesis, for inspiring discussions regarding reconfigurable computing and system level design, and for encouraging me to do an internship at Xilinx. It has really been a pleasure working with you.

I would like to thank present and former colleagues at the department. I am especially grateful to Matthias Kamuf for being a good friend and for reading and commenting parts of this thesis; Joachim Rodrigues for his coffee and for reading and commenting parts of this thesis; Hugo, Fredrik, Johan, Hongtu, Deepak, Erik, Peter, Martin, and my former colleague graduate students in the research group for friendship and support during the years.

I wish to express my gratitude to PhD Adam Donlin for giving me the opportunity to visit Xilinx Research Labs during a 6 months internship. I really enjoyed our discussions, appreciate all the support I was given, and am grateful that I was introduced to the field of abstract system modeling. The time in San Jose was a very inspiring and valuable time for me and I met many new friends. I will never forget the ski tours, the nights in San Jose and San Francisco, or the trap shooting.

I would like to thank all my wonderful friends, because you make everyday life entertaining. Finally, for more than words express I show my gratitude to my mother because you always care for me, my father because you were always there when I needed you, my brother and sister because you are wonderful and funny to be with, and my love Vanja because you are who you are and because you make me a happier man.

Lund, September 3, 2008

Henrik Svensson

List of Acronyms

ACELP	Algebraic Code Excited Linear Prediction
ACF	Autocorrelation Function
ACS	Add-Compare-Select
ADL	Architecture Description Language
AG	Address Generation
AGU	Address Generation Unit
AHB	Advanced High-performance Bus
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction Set Processor
AV	Architects View
CGRA	Coarse-Grained Reconfigurable Architecture
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal Oxide Semiconductor
CODEC	Coder-Decoder
CPI	Cycles Per Instruction
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DFG	Data-Flow Graph
DMA	Direct Memory Access
DRAM	Dynamic Random-Access Memory

DSOCM	Data-Side On-Chip Memory
DSP	Digital Signal Processor or Digital Signal Processing
ESL	Electronic System Level
FER	Frame Erasure Rate
FFT	Fast Fourier Transform
FIFO	First In, First Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GDB	GNU Debugger
GPP	General Purpose Processor
GPR	General Purpose Register
GUI	Graphical User Interface
HDL	Hardware Description Language
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
ITU	International Telecommunication Union
LUT	Lookup Table
MAC	Multiply-Accumulate
MB	Memory Bank
MC	Memory Cell or Memory Controller
MIMD	Multiple Instructions Multiple Data
MPMC	Multi-Port Memory Controller
MP-MLQ	Multipulse Maximum Likelihood Quantization
NoC	Network-on-Chip
OCP	Open Source Protocol

OPB	On-chip Peripheral Bus
OSCI	Open SystemC Initiative
PA	Parallel Architecture
PE	Processing Element
PC	Processing Cell
PCM	Pulse Code Modulation
PCI	Peripheral Component Interconnect
PLB	Processor Local Bus
PLC	Packet Loss Concealment
PLD	Programmable Logic Device
PPC	PowerPC
PPE	Pitch Period Estimation
PV	Programmers View
PVT	Programmers View with Time
RA	Reconfigurable Architecture
RAM	Random-Access Memory
RC	Resource Cell
RCP	Reconfigurable Computing Platform
RISC	Reduced Instruction Set Computer
RM	Reconfigurable Module
ROM	Read-Only Memory
RT	Register Transfer
RTL	Register Transfer Level
RTR	Run-time reconfiguration
SCENIC	SystemC Environment with Interactive Control

SDRAM	Synchronous DRAM
SIMD	Single Instruction Multiple Data
SMC	Stream Memory Controller
SNR	Signal-to-Noise Ratio
SoC	System on Chip
SPB	System Processor Bus
SRAM	Static Random-Access Memory
SystemACE	System Advanced Configuration Environment
TL	Transaction Level
TLM	Transaction Level Modeling
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
VDR	Vector Definition Register
VHDL	Very high-speed integrated circuit HDL
VLIW	Very Long Instruction Word
VRF	Vector Register File
XML	eXtensible Markup Language

Chapter 1

Introduction

In the year 1960, in a landmark paper by Estrin, a new type of computing machine was first suggested [1]. It dynamically adapted its computational resources to different algorithms so that each algorithm could be executed very fast. The machine was named *fixed plus variable (F+V) structure computer*, and it was built using a conventional microprocessor and reconfigurable hardware. The reconfigurable hardware was used to construct high performance computational structures, whereas the microprocessor controlled when and how often a new computational structure was initiated. At the time, the F+V structure computer was intended for scientific computational problems that were out of reach for conventional microprocessors. However, it did not reach into commercial products as reconfigurable hardware was a hardly explored field and as the performance of conventional microprocessors was constantly growing and simply considered sufficient.

Today, the field of reconfigurable computing is compelling as a complement to embedded systems that contain numerous application specific integrated circuits (ASICs). Since 1960, reconfigurable architectures have evolved and there is a consensus that they are a promising solution to bridge the gap between *ASIC performance* and *processor flexibility*. However, customary architectures such as field programmable gate arrays (FPGAs) can not match energy and area efficiency of ASICs. It is because FPGAs are built as arrays of interconnected functional blocks operating at bit-level, whereas many signal processing algorithms utilize word-level arithmetics. Word-level arithmetics implemented on architectures that have bit-level functional blocks, cause an overhead in required hardware area and energy consumed during operation. In order to reduce this weakness, researchers have suggested new types of reconfigurable

architectures with *coarse-grained* functional blocks. The design space for these coarse-grained architectures is neither sufficiently explored nor well understood when it comes to implement their micro-architecture, to integrate them into an embedded systems, and to efficiently map programs to them. Advanced design tools are required to explore design parameters and establish their relationship to performance, area, and power. That *modeling methods* and *computer-aided tools* are keys in development and analysis of reconfigurable computers, was recognized already by the pioneers [2]. Still, many recently presented reconfigurable architectures have been developed without tools for efficient modeling and analysis of the embedded system. New reconfigurable architectures, to be used in increasingly complex embedded systems, require development of modeling and exploration tools.

This thesis discusses coarse-grained reconfigurable architectures as a complement to ASICs in embedded systems. It presents two different coarse-grained reconfigurable architectures and a modeling and exploration environment to build and explore complete reconfigurable computing platforms. Embedded systems that require programmable hardware acceleration of regular computation intensive kernels with word-level arithmetic should utilize coarse-grained reconfigurable architectures. Design of these complex systems should be performed with tools for efficient modeling, simulation, and architectural exploration in order to analyze and tune design parameters before a chip is fabricated.

Thesis Contribution

The main contributions of this thesis are the following:

- An exploration methodology, is introduced, based on tool support to construct, simulate, analyze, and tune models. Transaction level models are consistently used to compose and simulate complete reconfigurable computing systems.

- A coarse-grained reconfigurable coprocessor implemented in VHDL is presented. The coprocessor provides over 30 times speedup of kernels that accounts for nearly 90% of the processing time on a RISC processor that executes a signal processing intensive speech codec.

- A reconfigurable array that consists of memory and processing cells is introduced and it is suggested how these cells should be constructed. Processing cells are instruction set processors with enhanced performance for communication-intensive inner loops. Inter-processor communication is performed using a self-synchronizing protocol that simplifies algorithm

mapping and manages unpredictable time variations. Memory cells are shared resources that are distributed throughout the array to reduce memory bottlenecks.

- A hybrid interconnect network is presented, and it consists of dedicated local links and a global hierarchical network. The hierarchical network enhances routing flexibility, allows communication links to be dynamically created, and provides a shared connection to external memory and processor. It is shown that the hybrid interconnect maintains high data rates assuming realistic localization of communicating blocks in the array.

Thesis Outline and Included Papers

This section summarizes the purpose and content of each of four chapters and four parts in this thesis. Included parts are based on merged and revised versions of published papers. The material presented in Part II and III is a result of a joint project between the author and PhD Thomas Lenart. For published material the main contributor is indicated by the first named author, and for unpublished material the contribution is indicated in this section.

Chapter 2 introduces embedded systems and is intended to give a short background to better understand subsequent chapters and parts. It describes the embedded system components and relates architectural techniques to design goals such as performance, area, energy, and flexibility.

Chapter 3 is on reconfigurable architectures. After a short introduction on reconfigurable computing platforms, the chapter discusses reconfigurable architectures. The latter is intended to provide related work to Part I–III, and it focuses on coarse-grained reconfigurable architectures.

Chapter 4 discusses modeling and exploration of embedded systems. It describes advantages with abstract models, and it presents transaction level models as a complement to register transfer level models. It also deals with architectural exploration and claims the importance of tool-support to build, observe, evaluate, and tune a system.

Part I: A Coarse-grained Reconfigurable Coprocessor Targeting DSP Kernels

This part investigates the applicability of reconfigurable coprocessors targeting processing kernels in multimedia applications. Many multimedia applications contain a large portion of regular and computation intensive signal processing kernels that are suitable for acceleration in reconfigurable fabrics. With dynamic reconfigurability, the hardware resources can be reused across these kernels and across applications. A generic coarse-grained reconfigurable coprocessor has been developed in VHDL, and its efficiency accelerating signal

processing kernels and the G.723.1 speech codec is evaluated. Speedups in the range of 2 to 46 compared to general purpose processor execution is achieved for vector operations and larger kernels such as filtering and fast fourier transform. These kernels utilize 86% of the G.723.1 processing time on the RISC based target architecture. With our approach the average used clock cycles are reduced by 83% compared to processor-only execution.

The content in this part is based on modified versions of the following publications:

- Henrik Svensson, Thomas Lenart, and Viktor Öwall, "Implementing the G.723.1 Speech CODEC using a Coarse-Grained Reconfigurable Coprocessor," in *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, USA, pp. 195–198, June 25-28, 2007.
- Henrik Svensson, Thomas Lenart, and Viktor Öwall, "Accelerating Vector Operations by Utilizing Reconfigurable Coprocessor Architectures," in *Proceedings of IEEE International Symposium on Circuits and Systems*, New Orleans, USA, pp. 3972–3975, May 27-30, 2007.

Part II: System Level Exploration of Reconfigurable Computing Platforms

This part presents a SystemC environment and transactions level models in order to address efficient design exploration of reconfigurable computing platforms. System level evaluation of reconfigurable platforms is necessary to predict performance, and find and cure bottlenecks before a chip is fabricated. Evaluation of a complete system at register transfer level becomes unattractive because small architectural changes require a considerable design effort. Electronic system level design has been a driving factor for modeling languages that support multiple abstraction levels. Standard SystemC is one such language that supports modeling abstractions from register transfer level to transaction level. A set of flexible transaction level models has been developed so that a complete system with instruction set processors, buses, and memories is evaluated together with the reconfigurable architecture itself. Architectural exploration is addressed using our proposed SystemC environment with interactive control (SCENIC).

The content in this part is based on modified versions of the following publications:

- Henrik Svensson, Thomas Lenart, and Viktor Öwall, "System Level Modeling of a Reconfigurable Computing Platform," In preparation for ACM Transactions on Reconfigurable Technology and Systems.
- Thomas Lenart, Henrik Svensson, and Viktor Öwall, "Modeling and Exploration of a Reconfigurable Architecture for Digital Holographic Imaging," in

Proceedings of IEEE International Symposium on Circuits and Systems, Seattle, USA, pp. 248-251, May 18-21, 2008.

- Henrik Svensson, Thomas Lenart, and Viktor Öwall, "Modelling and Exploration of a Reconfigurable Array using SystemC TLM," in *Proceedings of Reconfigurable Architectures Workshop*, Miami, Florida, USA, April 14-15, 2008.
- Thomas Lenart, Henrik Svensson, and Viktor Öwall, "A Hybrid Interconnect Network-on-Chip and a Transaction Level Modeling approach for Reconfigurable Computing," in *Proceedings of IEEE International Symposium on Electronic Design, Test and Applications*, Hong Kong, China, pp. 398–404, January 23-25, 2008.

The author's contributions to the SCENIC exploration environment are new features, basic building blocks, porting the environment to operating systems supporting POSIX, and integrating ArchC instruction set simulators. The SCENIC core functionality was mainly developed by PhD Thomas Lenart.

Part III: Modeling and Exploration of a Reconfigurable Processor Array

This part presents a coarse-grained reconfigurable architecture built as an array of interconnected processing and memory cells. A hybrid interconnect network that consists of local communication with dedicated wires and a global hierarchical routing network is proposed. Memory cells are distributed and placed close to processing cells to reduce memory bottlenecks. Processing cells are instruction set processors with enhanced performance for communication-intensive inner loops. Inter-processor communication is performed using a self-synchronizing protocol that simplifies algorithm mapping and manages unpredictable time variations. The reconfigurable architecture is described as a scalable and parameterizable SystemC transaction level model, which allows rapid architectural exploration. Our exploration environment SCENIC is used to setup scenarios, control the simulation models and to extract performance data during simulation.

The content in this part is based on modified versions of the following publications:

- Henrik Svensson, Thomas Lenart, and Viktor Öwall, "Modelling and Exploration of a Reconfigurable Array using SystemC TLM," in *Proceedings of Reconfigurable Architectures Workshop*, Miami, Florida, USA, April 14-15, 2008.
- Thomas Lenart, Henrik Svensson, and Viktor Öwall, "A Hybrid Interconnect Network-on-Chip and a Transaction Level Modeling approach for Reconfigurable Computing," in *Proceedings of IEEE International Symposium on Electronic Design, Test and Applications*, Hong Kong, China, pp. 398–404, January 23-25, 2008.

The author's contributions to the presented hybrid network are performance simulations and the transaction level modeling approach. The VHDL models used for synthesis have been developed in the IC project and verification course. The VHDL models were integrated and synthesized by PhD Thomas Lenart.

Part IV: Algorithm and Coprocessor Implementation of a Speech Packet Loss Concealment Method

This part presents a speech data packet loss concealment algorithm and its implementation in an embedded system. The presented algorithm is based on pitch period repetition and a novel low complexity method to refine a pitch period estimate. It is shown that the pitch refinement improves the quality of the original concealment method. Hardware-software co-design techniques have been investigated to implement the algorithm. With a coprocessor acceleration unit a processing delay of 0.9 ms and a overall speedup of 3.3 was achieved as compared to processor execution.

The content in this part is based on a modified version of the following publication:

Henrik Svensson, Viktor Öwall, and Krzysztof Kuchcinski, "Implementation Aspects of a Novel Speech Packet Loss Concealment Method," in *Proceedings of IEEE International Symposium on Circuits and Systems*, Kobe, Japan, pp. 2867–2870, May 23-26, 2005.

Chapter 2

Embedded Systems

Embedded systems is a broad definition of computer systems that are designed for specific tasks. Their architectures are optimized to conform to requirements in their intended application field. In application fields without specific requirements, standard processor systems are used. In contrast, battery operated systems with real-time performance constraints, require customized architectures to achieve the desired performance with constrained power consumption. This characterizes a segment of embedded systems as for example digital cameras, mobile phones, and portable music players. The architectural optimizations that are applied, are trade-offs between competing design goals such as: low development cost, low production cost, high performance, low energy dissipation, high flexibility, and short development time. Extensive research results on processing machines and techniques, which address different combinations of design goals and balance trade-offs differently, have been presented. Within this research there are multiple research disciplines that concern processor architectures, interconnect structures, compilers, and design automation tools. This chapter gives an overview of processing components, interconnect structures, and data storage devices found in embedded systems.

2.1 Processing Machines

In this section processing architectures, from general purpose processors to dedicated hardware, are briefly described. It starts with an overview on parallel architectures and then a selected number of commonly used architectures is discussed in more detail. The discussed architectures are: general purpose

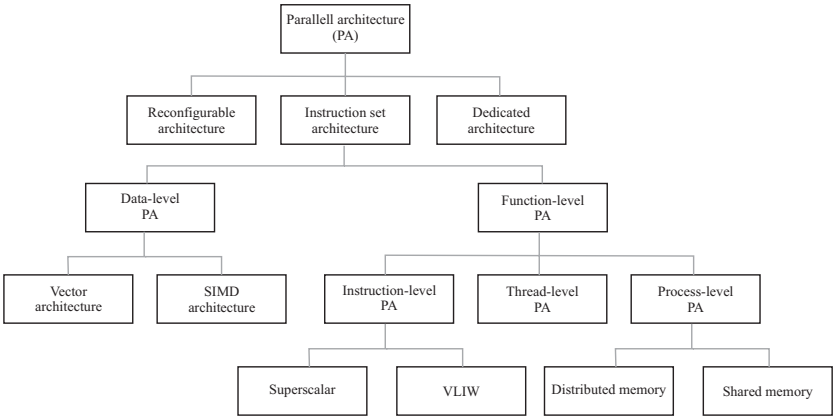
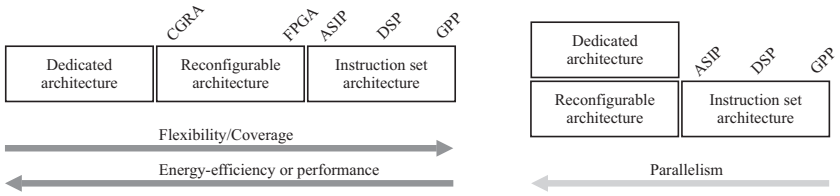


Figure 2.1: Classification of architectural techniques that exploit parallelism ([3] with modifications).

processors, digital signal processors, application-specific instruction set processors, vector processors, and dedicated architectures. These architectures provide different amount of parallel computing and domain-specialization. Reconfigurable architectures are only briefly discussed in this section, as they are discussed separately in Chapter 3.

2.1.1 Overview of Parallel Architectures

Figure 2.1 shows how *parallel architectures* (PAs) are classified into *instruction set architectures*, *reconfigurable architectures*, and *dedicated architectures*. Instruction set architectures are characterized by a stored sequence of instructions that are executed in the order given by a program counter. This class is further categorized into data- and function-level PAs. Data-level PAs apply the same operator to different data operands, whereas function-level PAs use multiple functional units or multiple processor cores to parallelize the computation. Dedicated architectures are developed for one specific application and hence parallelism can be tailored for this application's specific requirements. Reconfigurable architectures have a large number of basic functional blocks that can be programmed to realize a custom hardware structure. A specific processor or embedded system is often implemented by combining several architectural techniques shown in Figure 2.1. Therefore this classification often fails when existing machines or platforms are to be placed into one single category. There are many open research questions on how to combine these techniques for optimized performance in different application fields. One example that will be



(a) (b)

Figure 2.2: Illustration of trade-off between flexibility and energy-efficiency or performance when different architectural techniques are used ([4] with modifications).

discussed in more detail in this thesis is how to combine reconfigurable and instruction set architectures in order to achieve an architecture with *flexibility* and *high performance*.

Figure 2.2(a) illustrates, and makes a generalization, how flexibility, performance, and energy-efficiency are balanced within each of the categories: instruction set architecture, reconfigurable architecture, and dedicated architecture. The first fundamental aspect is the amount of parallelism that different architectural techniques can exploit. In general, dedicated or reconfigurable architectures are able to exploit more parallelism than instruction set processors, as illustrated in Figure 2.2(b). Within instruction set architectures, an application specific instruction set processor can generally exploit more parallelism than a general purpose processor. With parallel architectures, performance is increased or energy dissipation is reduced. Performance is gained with parallel architectures, as more operations per second are performed. Energy dissipation is reduced as parallel architectures can exploit voltage scaling and use low leakage cell libraries. Voltage scaling is an effective way to increase energy-efficiency, because the dynamic energy consumption of CMOS drops quadratically with the supply voltage [5, 6]. However, reducing the supply voltage or using low leakage libraries increases gate delay and hence, parallelism needs to be exploited to compensate for the degraded performance.

The second fundamental aspect is that *specialization*, or reduced flexibility, generally gives the same performance with less energy and area cost. For example, a reconfigurable architecture excludes energy and area overhead associated with instruction caching, fetching, and decoding, which are required in instruction set processors. Furthermore, a dedicated architecture excludes energy and area overhead consumed in routing resources, which are required in reconfigurable architectures. As much as 76% of the total energy consumed on a processor that executes typical digital signal processing (DSP) applications

may be in instruction fetch and decode [4]. Some FPGAs use up to 90% of the total core area for routing resources [7]. Hence, reduced flexibility leads to positive gains in chip area and energy dissipation.

2.1.2 General Purpose Processors

General purpose processors found in embedded systems cover a wide range of implementations from 8-bit low cost microcontrollers, to 32-bit RISC microprocessors, to high performance embedded processors with DSP enhancements. This section biases the discussion to the latter two, as they provide the most flexible platform in that they support development of real-time operating systems, high-level application code, user interfaces, and signal processing. This makes them ideal for implementations where cost and power consumption constraints may be relaxed in favor of short development time and flexibility.

Embedded processors are often build as single-issue reduced instruction set computers (RISCs) that have relatively few, simple, and orthogonal instructions. Examples of instruction set architectures (ISAs) found in this group are ARM [8], MIPS [9], and PowerPC [10]. These fall into the *load-store* category, which means that operands are references to general purpose registers (GPRs). To move data between memory and GPR a set of data transfer instructions and memory addressing modes is supported. Compared to desktop computers, which utilize two cache levels and a large main memory, the memory hierarchy is often reduced to one level cache and a size optimized main memory. The cache memory is generally organized as a Harvard architecture that has separate memories for instruction and data.

Compared to processors found in desktop computers, embedded processors distinguish themselves in a few more aspects. What follows is a brief summary of these aspects. The first two items will be discussed in more detail as they concern performance and power aspects.

- DSP enhancements – The ISA is extended with commonly used DSP operations, as ARM [11] or MIPS [12] DSP extensions.
- Low Power Implementations – Gate and task level techniques to reduce power dissipation.
- System integration – Processors can be delivered as intellectual property and integrated in a customized system on chip (SoC).
- Code size reduction – Many processors have support to mix 32-bit and 16-bit wide instructions.
- Coprocessor interfaces – Fast coprocessor interfaces for communication

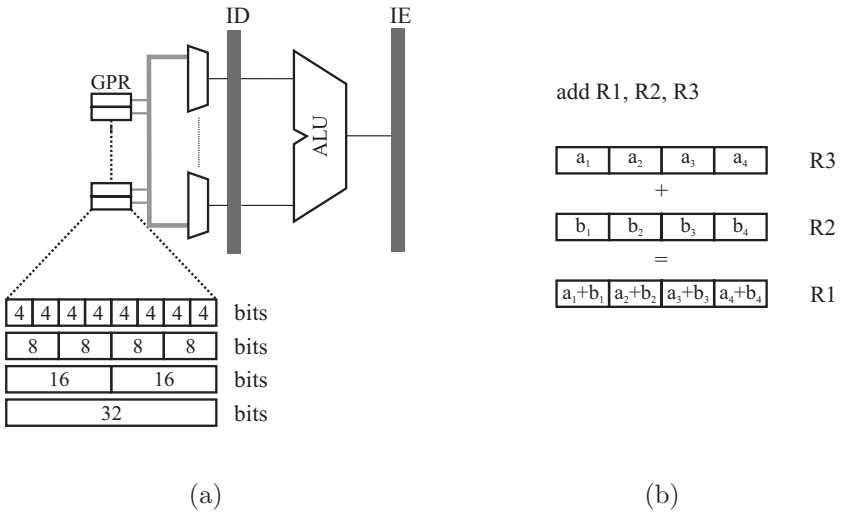


Figure 2.3: (a) SIMD instructions is implemented with functional units that allow parallel operations to 4/8/16/32-bit wide sub-words. In (b), an example of an addition instruction that operates on 8-bit wide operands are shown.

with customized hardware extensions and specialized instructions to communicate with these interfaces.

- Tools – An extensive set of design tools for compilation, simulation, profiling, and real-time operating systems.

The increased amount of signal processing incorporated into embedded systems, has led to hybrid processors that unify control and signal processing in a single processor core. It is accomplished by extending a RISC processor with DSP instructions, as in ARM [11] or MIPS [12] DSP extensions. A RISC processor with DSP enhancements can deliver very high performance and still allows efficient implementations of control dominated functionality and advanced real-time operating systems. In addition, DSP enhancements can be a cost-efficient alternative to an implementation that otherwise would use both a general purpose and a digital signal processor [13]. With these extensions processor companies claim that performance is increased by 30% to 400% in a range of applications [11, 12]. The DSP extensions found target a broad range of applications and include multiply-accumulate (MAC) instructions, single instruction multiple data (SIMD) instructions, saturation support, and sub-word register references. The MAC enhancements increase performance in signal

Table 2.1: Energy efficiencies and silicon area for PowerPC [10], ARM [8], and AMD Athlon 64 FX [15] processors. The ARM and PowerPC core are fabricated in 90 nm CMOS technology and have 16 kB separate instruction and data cache.

Processor	**Power Consumption** (mW/MHz)	**Max Clock Frequency** (MHz)	**Area** (mm^2)
PowerPC 405	0.19 @ 1.1V	400	2.0
ARM1136J-S	0.24 @ 1V	320	1.6 (area)
	0.45 @ 1V	620	2.5 (speed)
AMD Athlon 64 FX	40 @ 1.5V	2200	NA

processing where they are frequently used to calculate the vector dot product. Processors may provide a set of different MAC instructions, including those that allow half word register references with on-the-fly sign extensions. Hence, two half-word operands fetched in one memory access can be used in the following MAC instruction. SIMD extensions allow simultaneous operations to be applied to 4/8/16/32-bit wide sub-words in the operands, as illustrated in Figure 2.3. For example, experiments performed in [14] show that execution time for an MPEG-4 encoder is reduced a factor of 2 if only two SIMD instructions are added to a MIPS processor. Although DSP enhancements provide improved performance, the applicability of this strategy becomes a trade-off between the expected common case and performance gained in a specific application domain. If more specific DSP functionality is included it might lead to reduced performance and increased power consumption for a majority of the applications, which sacrifices general purpose processing. For example, specialized image processing instructions may lead to increased performance for image processing applications, but decreased performance for other application domains that can not utilize the specialized resources. Consequently, more signal processing specific instruction set architectures are found in another segment of processors referred to as digital signal processors.

A comparison with desktop computers reveals that embedded processors operate on lower supply voltage and operating frequency to reduce power consumption. Energy efficiency is a key parameter for these processors and as an example, energy consumption of an ARM7TDMI has dropped by a factor of 30 between 0.35 and 0.13 μm CMOS technology [16]. Table 2.1 shows power consumption for ARM, PowerPC, and AMD Athlon processors. The ARM

and PowerPC processors target low power embedded applications, whereas the AMD processor is used for desktop computers. The ARM core is presented for two different cell library implementations, one implementation optimized for *speed* and one optimized for *area*. As seen in Table 2.1, the AMD processor has a power consumption that is more than two orders of magnitude higher than the power consumption reported for ARM and PowerPC processors.

Power reduction at gate level is accomplished by partitioning the processor core into a set of resources with dedicated clock signals that can be activated/deactivated separately. When a specific resource is unused, the clock signal is deactivated to reduce switching activity and thereby also dynamic power dissipation. Another power reduction technique, performed at task level, is dynamic voltage and frequency scaling that adapt performance to current workload. Power management units that support a discrete range of supply voltages and clock frequencies are implemented in hardware and controlled from software. The power savings between sleep mode and highest speed are several orders of magnitude [17]. Clearly, this technique is interesting in embedded systems that expect a time varying workload. To allow scheduling for low energy, real-time scheduling algorithms that incorporate the characteristics of the underlying frequency and voltage scaling mechanism are required [17]. This includes to take into account task deadlines and overhead related to a change of supply voltage. A change in supply voltage has an overhead that may vary between 30 μs and 200 μs for different implementations [17].

2.1.3 Digital Signal Processors

Digital signal processors (DSPs) have incorporated multiple techniques to exploit data and instruction-level parallelism in signal processing algorithms. There is a wide spread of DSPs, from low power fixed-point single-issue processors, to high performance floating-point multi-issue processors [18, 19]. As fixed-point arithmetic is more energy and area efficient than floating-point arithmetic, fixed-point DSPs with saturated arithmetic dominate in devices with low power constraints.

In contrast to general purpose processing where RISC is commonly used, many DSPs are implemented as complex instruction set computers (CISC). These are characterized by non-orthogonal ISAs where special purpose instructions are used to increase parallelism for common operations in the application domain. Consequently, implementations have deep pipelines and latencies that differ between instructions. For example, instruction latencies found in TMS320C64x varies between 6 and 10 clock cycles [20]. A typical CISC example is special purpose instructions to support filtering, which is a key operation in many signal processing applications. In a DSP this is supported with in-

structions that do a MAC operation in parallel with two data fetches from data memory, modulo update of two address registers, zero-overhead looping, and an instruction fetch. Hence, multiple atomic operations are performed in a single clock cycle. The result is that a finite impulse response (FIR) filter can be implemented with a throughput of one clock cycle per filter tap.

Similar to complex instructions to accelerate filtering, there are DSPs that have other specialized media operations to execute critical loops with relatively few instructions. The instructions differ depending on the expected application field, as for example instructions to support color space conversion are found in DSPs that target video applications [21]. Drawbacks experienced with this approach are the lack of compiler support for many of these specialized instructions in combination with the complex trade-off between the common case and the gain for a selected number of applications. However, this approach to exploit parallelism is driven even further with application specific instruction set processors, which have instructions that are added based on performance improvement for a single application.

Instruction level parallelism is a well explored technique to utilize parallelism in instruction set processors. With N functional units in the processor core, N instructions can be executed simultaneously which means that execution time can potentially be reduced N times. However, an execution time reduction of a factor N is under assumption that instructions executed simultaneously are independent. If there are dependencies, then scheduling of instructions onto functional units needs to account for both dependencies and pipeline latencies. Figure 2.4 shows how the execution pipeline is organized in a multi-issue processor. Scheduling can be done dynamically with hardware support as in superscalar processors or statically by compilers as in very long instruction word (VLIW) processors. Static scheduling is less flexible than dynamic scheduling, however the repetitive nature of signal processing has proven to be well-suited for this approach [22]. Compared to superscalar processors, which require extensive hardware support, VLIW processors reduce hardware cost and power consumption. Hence, the VLIW technique has been favored to implement multi-issue DSPs. In VLIW processors a compiler determines exactly what operation that should be performed on each functional unit. Functional units may implement a heterogenous set of supported instructions in order to reduce hardware area. For example, the TMS320C6x has eight functional units organized as two identical groups of four different functional units [22]. An instruction contains control information to each functional unit in the processor core and therefore instruction words are long. Design of VLIW architectures is concerned with finding a balance between number of functional units and available parallelism inherent in targeted application domains. It also involves exploring compiler techniques that can make schedules to increase utilization of

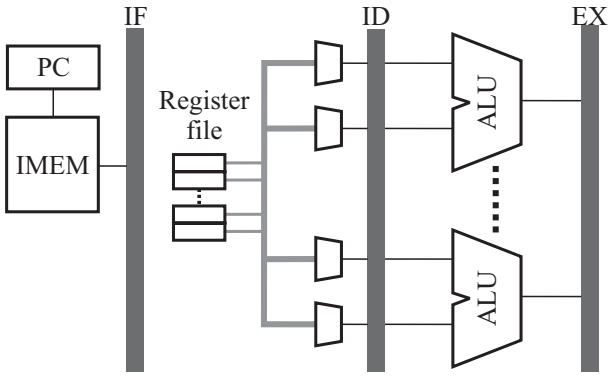


Figure 2.4: A multi-issue processor has a number of functional units which operate in parallel and communicate through a centralized register file. Scheduling operations onto the functional unit can be done statically as in VLIW, or dynamically as in superscalar.

the functional units. The parallelism that compilers and designers can exploit, as the number of functional units are increased, is a limiting factor for VLIW architectures.

As the number of functional units is increased and specialized instructions are added, the major bottleneck is moved from computations to the memory system. Therefore DSPs often have more than one data cache port in addition to the instruction cache. This is implemented using either a single multi-port memory, or using multiple single port memories. The latter approach suffers from pipeline stalls, to allow serialization of data fetches, if simultaneous load/-store requests are issued to the same memory [21]. This shortcoming is removed using a single memory with multiple ports, which allow simultaneous accesses. Memory indexing is supported by multiple address generation units that maintain memory address pointers in special purpose registers. These address generation units operate in parallel with functional units and they have address increment with modulus operation to support circular buffers commonly used in signal processing. This effectively accelerates index calculations that would require several instructions if implemented on a RISC. There are also more specialized address generation units that support bit-reversed addressing, which is used in fast fourier transform computations.

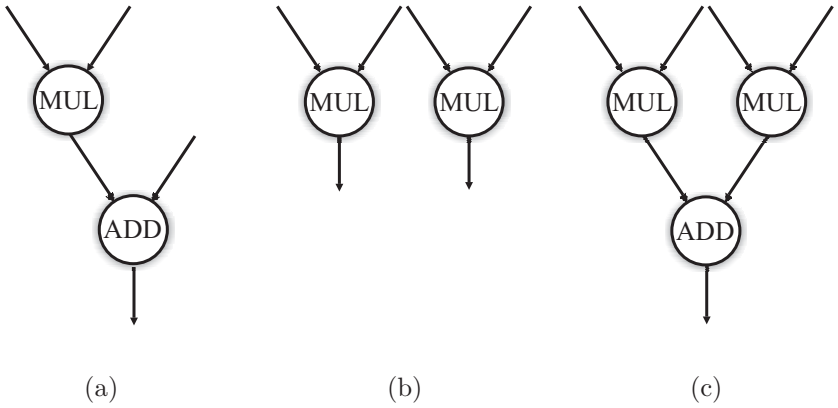


Figure 2.5: Customized instructions are implemented by (a) chaining operators, (b) parallelize operators, or (c) combine parallelized and chained operators.

2.1.4 Application-Specific Instruction set Processors

Whereas digital signal processors target a broad range of applications, application-specific instruction set processors (ASIPs) are optimized for a single application, or a small group of applications [3]. Optimizations may be performed at the microarchitecture level, so that functional units and memory system are tuned to the specific application. Furthermore, it may include exploring instruction- or data-level parallel architectures. However, the most characterizing for an ASIP is the instruction set customization. The generalized design flow is that a baseline processor, which is either a single- or multi-issue architecture, is extended with application-specific instructions. Customized instructions are designed by parallelizing and chaining operators in the application [3, 23] and this is illustrated in Figure 2.5. As compared to traditional instruction-parallel architectures, such as VLIW or superscalar, ASIPs trade flexibility to increase energy- and cost-efficiency.

To determine what parts of the functionality that should be implemented as specialized instructions is a difficult problem, which requires advanced design tools and systematic design-flows. The ASIP methodology presented in [3], uses a high level language (LISA) to allow designers to rapidly evaluate a customized instruction set. There are also automated instruction customization approaches. These are based on methods applied to formal representations such as data-flow graph (DFG) as used in [23], or hierarchical conditional dependency graph (HCDG) as used in [24–26]. The methods have as objective to identify customized instructions that fulfill external constraint parameters

such as area and execution time. The former methodology [23] uses a heuristic guide function to search for candidate subgraphs that can be implemented as customized instructions. A candidate subgraph is required to fulfill constraints such as maximum number of registers that are used for input and output communication. After candidate subgraphs have been identified, similar subgraphs are grouped into a customized functional unit in a combination step. Finally, functional units are selected based on an area constraint and the estimated performance improvement. The latter approach [24, 25], uses constraint programming techniques to find computational patterns that may be implemented as instructions. The explorer selects patterns in order to minimize the execution time on a given baseline processor. By running these explorers repeatedly with different area constraints, performance versus area can be investigated. Within the exploration of sixteen benchmarks in encryption, audio, network and image processing, an average speedup of 1.69 is reported for the DFG exploration approach [23]. Results from exploring fifteen multimedia benchmarks with the constraint driven approach, show that the customized instructions identified are able to provide speedups in the range of 1.17 to 3.5, with an average of 1.9 [25].

As ASIPs sometimes target a group of applications in a domain, aspects that concern cross-application applicability of customized instructions need to be considered. A natural approach is to do customized instructions for each application. However, this strategy somewhat limits flexibility to design time, and hence it omits new applications that need to be supported by an already fabricated ASIP. It is shown in [23], that exact subgraph matching typically does not occur across applications in a domain. This means that customized instructions discovered by identifying subgraphs in one application can not likely be applied to accelerate another application. To address this shortcoming *generalization techniques* are applied on customized instructions. In [23], two generalization techniques are described: *subsumed subgraph* and *wildcarding*. Subsumed subgraphs refers to the ability for operands to pass through an atomic operation unaltered. Hence, a customized instruction that performs the operation `AND-XOR-AND` could, after this technique has been applied, be used to perform `AND-AND`, `AND-XOR`, `XOR-AND`, `AND`, and `XOR`. Wildcarding refers to changing node operations in a subgraph, so that instructions with similar shape but with different operations, can execute on the customized instruction. With these generalization techniques, an ASIP designed for one particular application may provide increased performance also for other applications implemented on it. As an example, if a GSM *decoder* is implemented on an ASIP designed for GSM *encoding* a speedup of 1.3 is achieved without any of these generalization techniques. If subsumed subgraph and wildcarding are applied, a speedup of 1.7 is achieved. As a comparison, an ASIP designed for GSM decoding pro-

Table 2.2: Example of vector instructions.

Instruction		Operation	Comment
ADDV	V1,V2,V3	V1[i]=V2[i]+V3[i]	vector + vector
ADDS	V1,F0,V2	V1[i]=F0+V2[i]	scalar + vector
LV	V1,R1	V1[i]=M[R1+i]	load, stride=1
LVWS	V1,R1,R2	V1[i]=M[R1+i*R2]	load, stride=R2
LVI	V1,R1,V2	V1[i]=M[R1+V2[i]]	indexed

vides a speedup of 1.9 for that particular application. Clearly, generalization techniques improve flexibility. However, flexibility may be further improved if instruction customization approaches are combined with reconfigurable architectures to implement customized instructions.

2.1.5 Vector Processors

Vector instruction set architectures have been used in scientific and high performance computing for three decades [27]. Vector ISAs are different from scalar processors, or SIMD extensions previously discussed. In scalar processors, arithmetic instructions perform operations on scalar operands to produce a scalar result. SIMD extensions exploit data-level parallelism through a set of instructions that perform simultaneous operations on short vectors with narrow data. However, the degree of available data-level parallelism in multimedia is only partially utilized with SIMD extensions. In contrast, vector instructions perform element-wise operations on variable sized linear arrays of operands in order to produce the result array. Table 2.1 shows two vector instructions for addition. As compared to RISC, CISC, VLIW, and superscalar processors, code size, instruction fetch, and decode rates are reduced with vector instruction sets [27, 28]. A vector instruction set often contains vector instructions for integer, logical, and floating-point operations. As with RISC, all vector processors since late 1980s are load-store processors that have vector operands in vector register files (VRFs) and special instructions to move data between memory and VRFs [27]. Processors commonly have three addressing modes: unit stride, strided, and indexed, see Table 2.1. As memory access patterns are deterministic, prefetch techniques can be used to increase performance of the implementation.

If SIMD extensions are excluded, there have been few studies or commercial products with vector architectures for embedded low power multimedia applica-

tions. In [28] the VIRAM instruction set is proposed and it extends traditional vector architectures with multimedia enhancements. These enhancements include fixed-point arithmetic, narrow data type, and support for conditional execution in the main loop body. It is shown that most benchmarks in a multimedia benchmark suite do 90% of their operations using VIRAM vector instructions. The average vector length for benchmarks varies between 13 and the maximum vector length, which were 128 in the experiments. This shows that SIMD extensions that have 2/4/8 simultaneous operations fail to fully exploit the available data-level parallelism in these benchmarks. Two microarchitecture implementations of the VIRAM instruction set are presented, VIRAM-I [29] and CODE [30]. VIRAM-I uses a centralized VRF and multiple vector lanes to allow parallel execution, whereas CODE uses multiple cores with local VRFs and its own instruction stream in addition to multiple lanes.

2.1.6 Dedicated Architectures

Dedicated architectures are designed to compute specific tasks. Therefore memories, datapaths, and controllers may be optimized to meet design goals. This gives dedicated architectures by far the highest performance and energy-efficiency, as compared to other architectural techniques. The discussion in this section will be on dedicated architectures implemented as a fixed network of logic cells. However, tasks statically mapped to any programmable logic also fall into the category dedicated architectures.

A design flow for dedicated architectures contains several levels, the main ones are [31]:

1. **Algorithm** – On the algorithm level, number of operations, memory sizes, and numeric precision are explored and selected to fulfill the design constraints. The design entry is generally a model with floating-point arithmetic and therefore powerful signal processing functions, available in tools such as Matlab, are used to increase productivity. The algorithm is gradually refined to use fixed-point or floating-point operators and standard constructs such as `for-loops` and `if-then-else`. Fixed-point algorithms are most common due to area and power constraints in many application fields in which dedicated architectures are used. However, any floating-point algorithm may also be implemented with dedicated hardware. The final model contains information about wordlengths, operations, order of operations, memory sizes, and memory addressing. This information is passed to the architectural level. The model is also used as a reference for other derived refinements when traversing the design flow.

2. **Architecture** – The architectural level is concerned with time-sharing, parallelism, and pipelining. The design is divided into submodules and hierarchy is imposed as a method to reduce design complexity. The architecture is described at register transfer level with hardware description languages (HDL) such as VHDL, Verilog, or SystemC, which all have mechanisms to describe hierarchy, concurrency, and arbitrary wordlengths.

3. **Netlist** – A netlist is generated from an architectural description with a synthesis tool. Synthesis tools automatically determine the hardware implementation to meet design constraints such as speed, area, and power. For any given register transfer level (RTL) description, area and performance depend on cell library, choice of cells from that cell library, and also how HDL operators are mapped to specific architectures. The synthesis tool comes with pre-built synthetic components that implement the built-in HDL operators. Synthetic components are technology independent and several components to implement the same operator exist. For example, an addition operator could be implemented as ripple-carry or carry lookahead. As different implementations have different properties when it comes to power, area, and delay, designers provide power, area, or delay constraints to the synthesis tool, which selects the most suitable implementation. After the initial step, which is to determine the architecture of arithmetic operators, the design is mapped to standard cells in order to generate the final netlist. With the netlist, estimates on the final area, speed, and power consumption are generated. Figure 2.6 shows the area-delay design space for a 16-bits addition operator. The figure has been generated by performing synthesis repeatedly, with different delay constraints. As seen in the figure, optimal area-delay curve spans through different types of adder architectures. For example, a required delay in the range of 1.7 to 2.0 ns would be implemented with a ripple carry adder (rpl), and a delay in the range of 0.4 to 0.75 ns would be implemented with a brent-kung adder (bk). The infeasible region says that there is no adder with delay smaller than 0.4 ns.

4. **Physical layout** – Place and route tools place cells on a defined floorplan and route wires between them. In addition, clock and power wires are automatically routed. Accurate timing and area can be extracted from this level.

Development time is a major bottleneck using traditional design flows. Especially going from an algorithm to an architectural description is time consuming, because of the complex design space. This is addressed by high level synthesis tools that automate transformation from an algorithm description

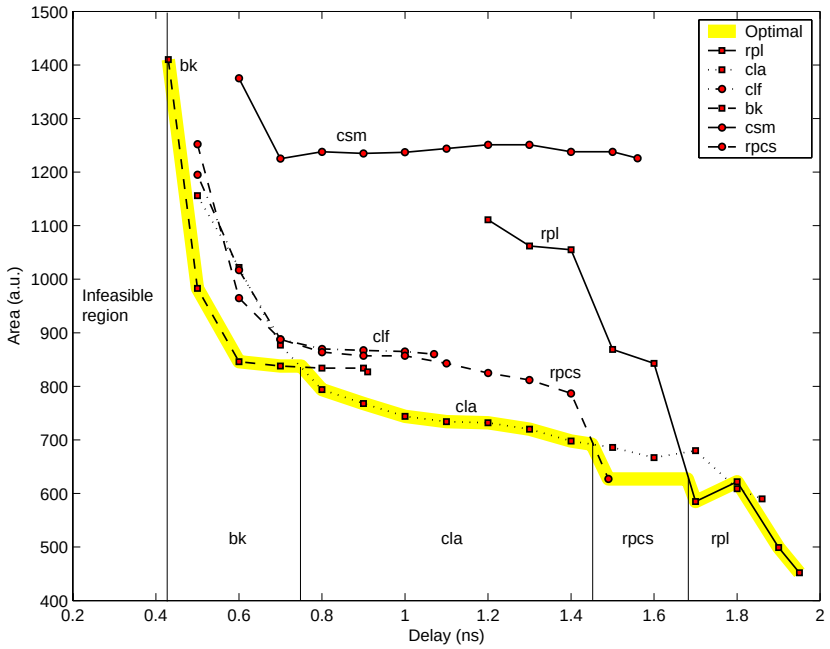


Figure 2.6: Area versus delay for 16-bits addition, which is implemented with a 0.13 μm CMOS cell library. The plot is generated by running synthesis repeatedly for each type of architecture and for different delay constraints. The adder architectures explored are: ripple carry (rpl), ripple carry select (rpcs), carry look-ahead (cla), fast carry look-ahead (clf), brent-kung (bk), and conditional sum (csm).

to an RTL architectural description [32]. With designer provided constraints these tools perform resource allocation, scheduling, and mapping in order to generate the RTL description. The implementation is supposed to fulfill given design constraints and minimize a designer provided cost function, such as execution time, area, or latency [33]. With automated approaches a wider part of the available design space is explored.

If properly addressed, a dedicated architecture may be designed with limited amount of flexibility. For example [34, 35], proposes an ASIC accelerator for holographic image reconstruction, in which a streaming accelerator XSTREAM exposes configuration registers to software. These registers are used to control the operation of individual processing blocks, such as CORDIC, FFT, and DMA interfaces. In [36], a flexible architecture for Trellis decoding is proposed.

The architecture adapts decoding resources to the channel's signal-to-noise-ratio. However, flexibility in these approaches is highly application-specific, and subordinated other design parameters that require a dedicated architecture.

2.2 Interconnect Structures

Traditional system on chips use bus-based interconnect, derived from a conventional microprocessor bus, to allow communication between various functional blocks. Each functional block is attached to the bus as a *slave*, a *master*, or both. Slaves on the bus are assigned an address range to which they respond. A master may initiate a transfer, whereas a slave remains passive and only responds when it is addressed. A typical interface contains *address*, *data in*, and *data out* in addition to control signals to make a request, acknowledge a response, etcetera. When masters make simultaneous requests, the bus performs arbitration in order to select a master based on some priority selection algorithm. Existing bus architectures range from simple *or*-gate networks to pipelined implementations with separate phases for address and data buses. Most processors have their own specific bus interconnect system such as AMBA for ARM, and CoreConnect for PowerPC.

To determine the actual bus performance is complex as it depends on dynamic behavior of the application. However, theoretical single word and burst throughput are easily determined from bus specification. Features that determine the actual performance are:

- **Data bus wordlength** – Typically 32-, 64-, or 128-bit wide buses. However, dynamic bus sizing allows devices with narrower data interfaces to be attached to the bus.

- **Decoupled data buses** – Allows concurrent read and write operations.

- **Burst support** – Fixed and variable length bursts allow several data words, usually stored in consecutive addresses, to be transferred with a single request. This reduces the address phase overhead and allows slaves to pipeline subsequent data words.

- **Cache line support** – Allows a cache line to be transferred with a single request. To reduce processors idle time, a cache line request responds with critical word first and wrap to the lowest byte address at the high address boundary. For example, a 16 byte cache line request with address `0x4`, will result in data transfers from addresses in the following order: `0x4`, `0x8`, `0xC`, and `0x0`.

- **Split transactions** – Allows splitting the read request from the data transfer, so that time between the request and the transfer can be used to send more requests. This may effectively hide long latencies associated with external memory accesses.

- **Priority scheme** – Priorities are used by an arbitration algorithm such as round robin, least recently used, or fixed priority to resolve bus contention. This ensures that time critical requests are handled before non-critical. The priorities can be fixed priority per master, dynamically configurable master priority, or priority per request.

- **Address Pipelining** – A new transfer can be initiated while there is an ongoing transfer in the same direction.

Bus systems provide limited scalability in number of interconnected components. When the bus lengths increase or when number of components increase, the propagation delay grows and can eventually exceed the required clock period [37]. Consequently, this limits the number of components that can be attached to a bus and thereby limits scalability of bus systems. As a single bus do not scale well with increased number of components, these interconnect structures contain different buses and bridges, so that bus hierarchies can be built. Hence, communication between two units can include several buses and bridges. Typically, peripheral devices with moderate data rate are attached to a slow and low complexity bus (CoreConnect OPB, AMBA APB), whereas memories and DMA devices are attached to a high performance bus (CoreConnect PLB, AMBA AHB).

Another approach to scalable systems is network on chip interconnects [37]. In this research field, flexible networks are built from short wires and routers to share the wires using circuit switching, packet switching, or wormhole switching. All components have a common interface to the network and data communication between components are carried within packets, which also carry information to control the network. Popular network topologies are mesh, torus and folded torus.

2.3 Memory and Storage

As new architectures for parallel processing evolve, fast transfers of data back and forth between memories and datapaths, are as important as the processing itself. However, the performance of memories has not improved at the same pace as the performance for processing [27]. This is referred to as the *processor-memory performance gap*. To overcome this gap several techniques have been deployed and here will be mentioned a few.

2.3.1 Memory Hierarchy

As there is not an unlimited amount of fast, small, and low-power memories the *principle of locality* is used. This principle states that accesses to memory are not uniformly distributed, but generally have a strong temporal correlation. Data recently used in processing has high probability to be used soon again. Hence, a hierarchy of memories is built, so that recently used data is stored in fast and low-power memories referred to as *caches*, whereas other data is stored in small but slower and less power-efficient memories. This is used in processors in which general purpose registers are at the highest level followed by cache memory, off-chip main memory, and disk memory. In general, on-chip memory is implemented with static random access memory (SRAM), or registers, and off-chip memory is implemented with dynamic random access memory (DRAM).

2.3.2 Distributed Memories

For algorithms that can be divided into partitions that operate on different memory segments, multiple memory banks can be utilized to improve data delivery. This is a well known technique, used to improve performance for digital signal processors, which often have a multi-bank or multi-port memory system. For example, in an algorithm that consists of two different filters F_1 and F_2, filter coefficients are only accessed by the corresponding filter algorithm. Hence, the filter coefficients for F_1 and F_2 can be stored in two different memories that can be accessed simultaneously. Memory distribution may operate in conjunction with a memory hierarchy, so that each memory bank is loaded from a larger cache prior to execution.

2.3.3 Access Pattern

Algorithms that operate on large chunks of data with a single reference to each data element can not utilize the benefits provided by caches. Typically, such algorithms better access data directly in the off-chip main memory. Modern off-chip memories, such as DRAM or synchronous DRAM (SDRAM), are burst oriented and the achieved data rate depends on the order of accessed addresses. If data is randomly accessed each transfer is associated with a high latency, which significantly degrades overall performance. To cure such bottlenecks, designers need to change the access pattern by reorganizing how data is stored or by changing in what order data is accessed. A DRAM is organized as multiple memory banks that are indexed with row and column address. To perform a read or write operation, a bank and a row must be activated, with an initial latency. In an activated row, read and write operations are performed by

altering the column address with higher latency for randomly selected columns as compared to subsequent columns. To access a different row, the current row needs to be deactivated, and the new row activated.

2.4 Discussion

Many different types of architectures are combined in order to build an embedded system. This chapter has briefly described five classes of architectures: general purpose processors, digital signal processors, application-specific instruction set processors, vector processors, and dedicated hardware. These architectures balance competing design goals differently, so that one type of architecture may provide fast development time and flexibility, but with reduced performance and higher energy dissipation. The next chapter will introduce reconfigurable architectures and it discusses how reconfigurable architectures and instruction set architectures are combined into a reconfigurable computing platform.

Chapter 3

Reconfigurable Architectures

Reconfigurable architectures (RAs) are devices that contain *programmable functional blocks* and *programmable interconnects* between functional blocks, as illustrated in Figure 3.1. Spatial distribution of functional blocks in conjunction with a flexible interconnect of them, allows exploiting various forms of parallelism inherent in the application. In comparison with the programmability provided by instruction set architectures, the programmability provided by RAs allows substantial changes to the datapath itself. Hence, as with dedicated architectures, RAs can implement application-specific computing structures, but without sacrificing flexibility. That property makes reconfigurable architectures a promising solution to bridge the gap between the programmability of instruction set architectures and the performance of dedicated architectures.

The most mature class of reconfigurable architectures is field programmable gate arrays (FPGAs), which are considered to be *fine-grained reconfigurable architectures*. Recent advances in the field of reconfigurable computing are development of *coarse-grained reconfigurable architectures* and *reconfigurable computing platforms*. Reconfigurable computing platforms combine a reconfigurable architecture with other architectural techniques such as instruction set architectures. This idea originates from the work presented by Estrin already 1960 [2]. In this work he considers a new type of computing system, which combines a microprocessor with a reconfigurable structure, that can be temporarily distorted into a special purpose computer. Whereas the reconfigurable architecture is used to accelerate regular and computation intensive functionality, the microprocessor executes functionality that can not be efficiently accelerated. The word *temporarily* here means that the reconfigurable architecture

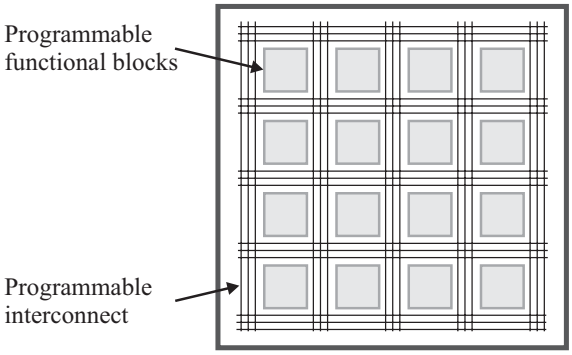


Figure 3.1: A reconfigurable architecture is built from programmable functional blocks and interconnect.

is dynamically adapted to match the current computational problem. This is referred to as run-time reconfiguration (RTR) and it is an important concept to reuse the reconfigurable hardware between applications or between functions within applications. Research on reconfigurable computing platforms addresses issues concerning system level integration, configuration management, and automatic tools to do partitioning, placement, routing, and scheduling.

Traditional RAs, such as FPGAs, provide interconnect structures and functional blocks that operate at bit-level. Hence, they are able to realize datapaths and controllers with arbitrary wordlengths. For signal processing algorithms that are based on operands represented with multiple bits, bit-level reconfigurability gives a large *overhead* in area, delay, energy, and configuration time. As a majority of multimedia applications are based on multi-bit arithmetic, fine-grained reconfigurable architectures have not been successfully applied to battery operated embedded systems. Within the research of coarse-grained reconfigurable architectures (CGRAs), interconnect structures and functional blocks with increased granularity are suggested in order to reduce the *overhead*.

3.1 Reconfigurable Computing Platforms

A reconfigurable computing platform (RCP) is constructed by combining reconfigurable fabrics with other processing machines. A common configuration is one or more processors, one or more reconfigurable units, and one or more memories [38]. It is argued that such a system is a more area-efficient alternative to higher performance than increasing number of processors. Integration of an additional processor doubles the area and gives at best a speedup of 2, whereas using that same area for a reconfigurable fabric provide means to

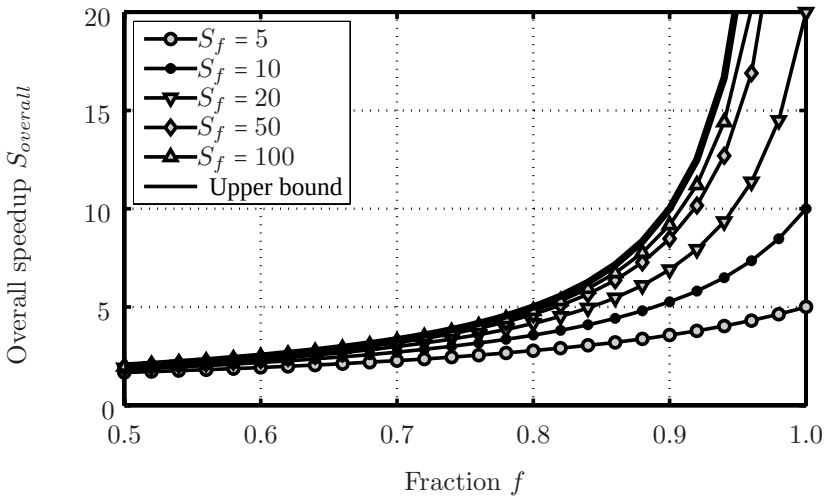


Figure 3.2: Overall speedup as the fraction (f) and the fractional speedup (S_f) are varied.

reach speedups close to theoretical upper bounds [39]. This is according to Almdahl's law, which states that the total speedup has an upper bound of $1/(1-f)$, where f is the fraction of the total execution time to which the improvement is applied. If S_f is the speedup of the fraction, the overall speedup is written as

$$S_{overall} = \frac{1}{1 - f + \frac{f}{S_f}}.$$

Figure 3.2 shows the effect of Almdahl's law as the fraction (f) and the fractional speedup (S_f) are varied. From the figure it is seen that even moderate fractional speedups, $50 \leq S_f \leq 100$, bring the overall speedup close to the upper bound if it is assumed that functionality that can be accelerated in the reconfigurable fabric stands for a fraction $0.9 \leq f \leq 0.94$ of the total execution time. Numerous implementations of signal processing algorithms on reconfigurable architectures have been reported to reach these fractional speedups over a single embedded processor [40–42]. Under these assumptions, with $f \geq 0.9$ and $S_f \geq 50$, $S_{overall} \geq 9.2$ is reachable.

The programmability of early reconfigurable devices were either *mask programmable*, or *field programmable*. Mask programming is performed by the semiconductor company by using a program table from the customer. Field

programming referred to the ability for users to program the devices after fabrication, or *in the field*. With the introduction of reconfigurable computing platforms a new programming technique was required as the device was to be reprogrammed during run-time, referred to as *run-time programmable*, or *dynamically reconfigurable*. This technique allows timesharing available hardware, so that large designs can be implemented on a limited amount of resources. It is also used to adapt the reconfigurable architecture based on information only available during run-time. Dynamic reconfiguration has raised issues related to both design-automation tools in order to analyze applications and specify when and how often the reconfigurable architecture should be reconfigured, and to the support required in the underlying micro-architecture. A reconfiguration refers to that a stream of *configuration data* is downloaded to the *configuration memory*. The configuration memory controls functionality in communication and computational resources. There are three main categories of architectures in terms of the underlying mechanism to reconfigure the system.

- **Single context** – A reconfigurable architecture that has a single *configuration memory* bit to hold each *configuration data* bit. Consequently, downloading a new configuration stream to the device, erases any previously downloaded configuration. Additionally, any context switch includes downloading a new configuration stream. Hence, dynamic reconfiguration can result in a significant configuration time overhead in the total execution time.

- **Multi-context** – A reconfigurable architecture that has multiple *configuration memory* bits for each *configuration data* bit. Consequently, a multi-context reconfigurable architecture can switch very fast between two configurations. In addition, new configurations can be downloaded in parallel with an actively used configuration, and thereby effectively hide configuration time when context is switched.

- **Partial reconfiguration** – Is the ability to reconfigure only parts of the reconfigurable fabric at a time. A new context only requires the resources allocated by that context to be configured, whereas other resources are left unaltered. The configuration memories operate as random access memories, so that addresses can be used to selectively change their content.

The first two, single and multi-context, are mutually exclusive, whereas partial reconfiguration may be added to any of these two mechanisms.

Implementation of an application on a RCP are summarized in four phases. First, *hardware-software partitioning* are performed, which means that an ap-

plication is divided into partitions that are either implemented in the processor (software), or on the reconfigurable architecture (hardware). Hardware-software partitioning is carried out either by analyzing software profiling results followed by manual partitioning, or by applying automatic partitioning methods that operate on formal models, as suggested for example in [25]. In this thesis, functionality that are implemented in the reconfigurable fabric is referred to as *functional kernels*, *kernels*, or *acceleration objects*.

In the second phase, each kernel is subject to *spatial partitioning*, which means that each kernel is divided into smaller pieces that will execute in different functional blocks. In fine-grained reconfigurable architectures, such as FPGAs, parallelism is described by using HDL at RT-level, whereas many proposed coarse-grained architectures have adopted a more software-centric programming approach. Examples of software-centric approaches are VLIW [43] or vectorizing [44] compiler technology, customized compilers [45], data-flow languages [46, 47], and custom description formats that aid designers to manually perform spatial partitioning [48]. Although, a software-centric programming approach is one of the key elements for reconfigurable computing platforms to succeed, there is no existing generalized programming framework and many presented architectures require manual efforts to be efficiently programmed.

In the third phase, the resulting description is mapped to the reconfigurable architecture. During mapping it is determined onto which resource each sub-kernel partition should be executed (placement) and onto which interconnect resource communication should be performed (routing). If the kernel does not fit onto the reconfigurable fabric, *time-sharing* or *temporal partitioning* is applied. Time-sharing is performed by applying folding techniques to the kernel and then redoing the spatial partitioning. Temporal partitioning is a form of *hardware virtualization*, which is accomplished by dividing the kernel that does not fit onto the reconfigurable fabric into smaller configurations. The smaller configurations may then be executed at different periods in time, however the actual schedule has to consider communication dependencies between temporal partitions. A special form of temporal partitioning is *pipeline partitioning*, which has been suggested in PipeRench [47, 49]. It is accomplished by breaking up the configuration into smaller pieces, each describing one pipeline stage in the overall computational structure. A pipeline configuration is loaded to the reconfigurable fabric when a previous pipeline stage is finished and hardware is deallocated. Typically, a new pipeline configuration is loaded in a single clock cycle.

In the last phase, the whole system must be tested to verify that configurations are loaded correctly, that data communication is functional, and that performance related constraints are satisfied. Typically, a processor is used to download new configurations. Hence, software drivers for the reconfigurable

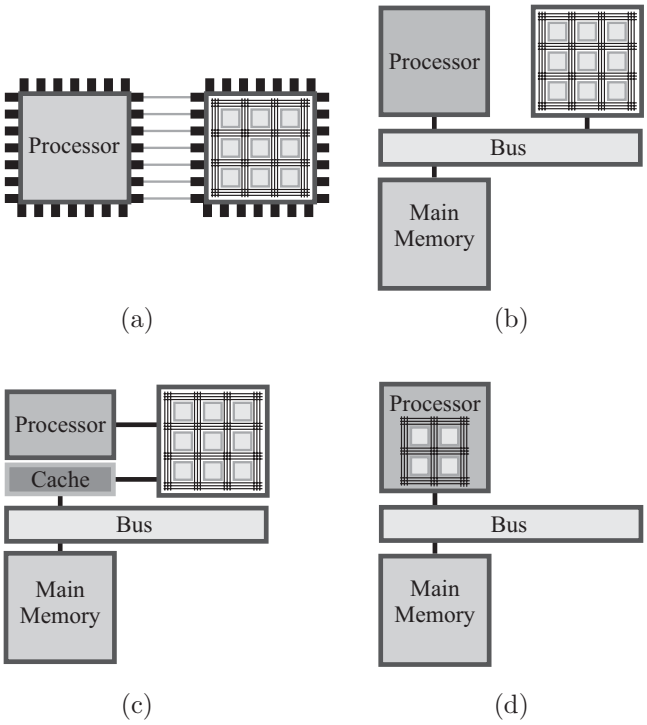


Figure 3.3: A reconfigurable computing platform is classified according to the coupling between reconfigurable architecture and the processor. The classes are: (a) external, (b) attached to bus, (c) coprocessor, or (d) functional unit.

device need to be developed and tested. The coupling and communication mechanism between processors and reconfigurable devices is used to classify a reconfigurable computing system. In [7] four classifications are suggested as shown in Figure 3.3, and summarized below.

(a) External – The processor and the reconfigurable fabric are placed on separate chips. Communication between them is mapped to standard communication protocols such as peripheral component interconnect (PCI), or universal serial bus (USB). As the off-chip interconnect speed is relatively low and has high initial latency, this type of system is intended for applications in which complex kernels are accelerated with limited intervenience from the processor.

(b) Attached to processor bus – The reconfigurable fabric is placed on the same chip as the processor and attached to the processor bus, to act as an additional processor. The reconfigurable architecture typically has both master and slave interfaces. The slave interface is used to reconfigure the fabric from the processor and the master interface is used for direct memory access during kernel execution. The processor's cache is not visible to the reconfigurable architecture. Consequently, data communication is handled mainly through a shared main memory. This means an initial high communication latency and slower transfer rates as compared to cache accesses. This coupling mechanism is used to accelerate larger kernels that execute thousands of clock cycles and operate on larger data arrays. As a consequence, this class of coupling often needs a substantial amount of memory banks distributed within the reconfigurable architecture in order to store intermediate results and to reorder data.

(c) Coprocessor – The reconfigurable fabric is attached as a coprocessor. Communication is handled either by allowing the reconfigurable device to access the processor cache memory or by allocating the processor for data shuffling using any special-purpose coprocessor instructions. After some initialization, the reconfigurable fabric can operate without processor supervision for a relatively large number of clock cycles and return the result after completion. This coupling mechanism is suitable to accelerate kernels that operate on data that fits inside the cache. Preferably, data already resides in the cache memory when needed by the coprocessor. This direct cache access allows high-speed transfer rate back and forth between reconfigurable device and memory. However, kernels that have high degree of temporal data locality and that allow data to be partitioned and placed into different memories, benefit from a multi-bank memory system embedded in the reconfigurable architecture.

(d) Functional unit – The reconfigurable fabric is embedded as a functional unit within the processor and occupies a range of *opcodes* to implement application-specific instructions. The customized instructions implemented in the fabric are typically reconfigured between applications. Hence, a customized instruction set can be selected based on performance improvement for a specific application. Communication is explicitly expressed within the instruction-format, which constrains number of input-output variables per clock cycle. As communication is fast and has low initial latency, this coupling mechanism is suitable for pipelined and parallel operations found in inner loops. Typically, there is no memory system embedded into the reconfigurable architecture, but only registers to store data between pipeline stages.

Because these coupling mechanisms provide different peak data rates, latencies, and degrees of time uncertainty due to contention in shared resources, each one has its preferable range of *functional granularities* that will benefit from execution in the reconfigurable architecture. In general this is summarized as: The tighter coupling between processor and reconfigurable architecture, the smaller kernels will benefit from acceleration. For example, a functional unit can be used to accelerate inner loop operations, whereas this would actually give longer execution time if applied with an external reconfigurable architecture.

3.2 Reconfigurable Functional Blocks

An early type of reconfigurable architecture is the programmable logic device (PLD), which realizes combinational functions with an `AND-OR`-array. By adding flip-flops and allowing outputs to be taken either from the `OR`-gates or the flip-flops, sequential logic is implemented. These early PLDs were small and several of them needed to implement a digital system. The solution was to collect several PLDs on a single chip and connect them through a programmable switch, and thereby the complex PLD (CPLD) was born. Advancement in the field have led to new types of reconfigurable architectures and this section discusses three of them: *field programmable gate arrays*, *ALU arrays*, and *processor arrays*. The complexity of functional blocks, referred to as the *granularity*, is used to classify an architecture. The FPGA is considered *fine-grained*, whereas ALU arrays and processor arrays are *medium-grained* to *coarse-grained*. Fine-grained functional blocks are useful for bit-level manipulations, while coarse-grained blocks are better for standard DSP applications. Some reconfigurable architectures have a mixture of fine- and coarse-grained functional blocks in order to support different application domains.

3.2.1 Field Programmable Gate Arrays

Field programmable gate arrays are the most mature type of reconfigurable architecture and most commercially available devices are placed in this category. Field programmable gate arrays were first introduced 1986 by Xilinx [50] and since then many new architectures have evolved. An FPGA offers fine-grained reconfigurability. As the name *gate array* indicate, any digital circuit can be realized with these devices. However, the actual implementation is not an array of gates. Instead, the most common type of functional blocks in these architectures are small lookup tables. Commercial FPGAs contain a large number of lookup tables, which have up to six inputs. These lookup tables are typically included in a basic computational unit, which also contains registers, multiplexors, and I/O ports to the interconnect. For example Altera has their

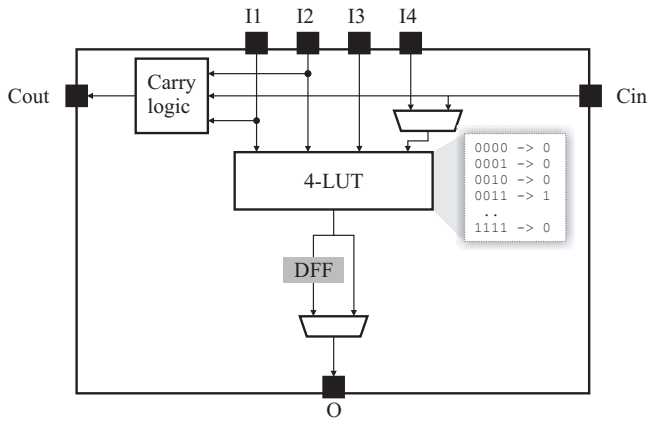


Figure 3.4: Functional blocks in field programmable gate arrays is fine-grained. The figure illustrates a functional block based on a 4 input lookup table.

architecture with *logic array blocks*, and Xilinx has *configurable logic blocks*. An example of a basic computational unit is shown in Figure 3.4.

FPGAs are programmed with hardware description languages that support RT-level modeling. Although, as with dedicated architectures there are tools which automate the step from algorithm description to RT-level description.

3.2.2 ALU arrays

In ALU arrays each basic functional block consists of an ALU that supports a set of word-level operations. A typical functional block used in ALU arrays are seen in Figure 3.5. Wordlengths of ALUs may be smaller than the required algorithm wordlength and more coarse-grained operations are then implemented with several ALUs. For example, eight 4-bit ALUs are allocated to implement a 32-bit ALU. This is implemented in medium-grained fabrics such as Garp [48], which is built out of 2-bit wide functional blocks. There are also architectures that have fixed wordlenghts, which are statically adapted to the application domain, as for example DReAM [51], RaPiD [52], and RSVP [46].

Significant for ALU arrays is that there is no locally controlled sequential execution flow. It means that typical functionality associated with an execution flow such as local program memory, execution pipeline, and branch instructions are excluded in the functional blocks. Instead each ALU operation is part of the configuration that is loaded into the reconfigurable fabric under management of a global controller. Each ALU operation and all connections between ALUs are

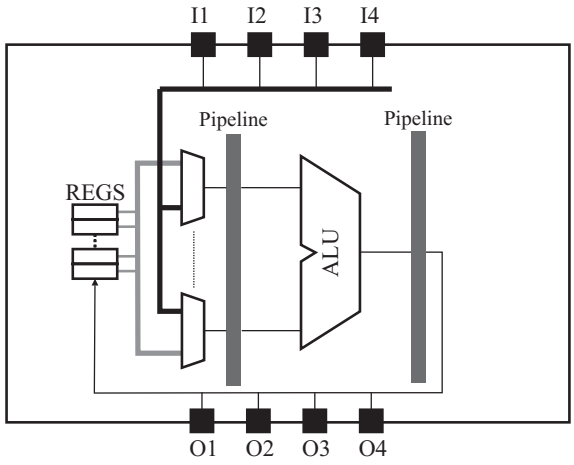


Figure 3.5: An ALU array is build from functional blocks that contain an ALU, internal registers, pipeline registers, and I/O ports to the interconnect.

set up by the global controller and remain static until the next reconfiguration. With that property ALU arrays are sometimes categorized as single instruction multiple data (SIMD) machines. This is typical for architectures such as ADRES [43], Garp [48], ARRIVE [53], XPP [44], RSVP [46], RaPiD [52]. The global controller is generally implemented in an attached processor, with some additional hardware support in the reconfigurable architecture to interface with the attached processor.

There are two major drawbacks with statically configured operators and connections. First, implementations can not exploit time-shared structures within a single context. Secondly, such architectures are not suitable for more control-oriented functionality that tends to generate complex computational structures with low utilization if they are parallelized. For example, if there is an `if-then-else` branch, static mapping requires that both branches are performed in parallel and that the appropriate result is selected with a multiplexer operation.

Several techniques have been deployed to address control flow in ALU arrays. Some architectures, which are intended to be used as functional units, rely completely upon the host processor to handle the control flow. Typical examples of this are ADRES [43] and ARRIVE [53]. Other architectures, intended to operate more autonomous, have incorporated various strategies to handle control flow in the reconfigurable fabric itself. In REMARC [54], each

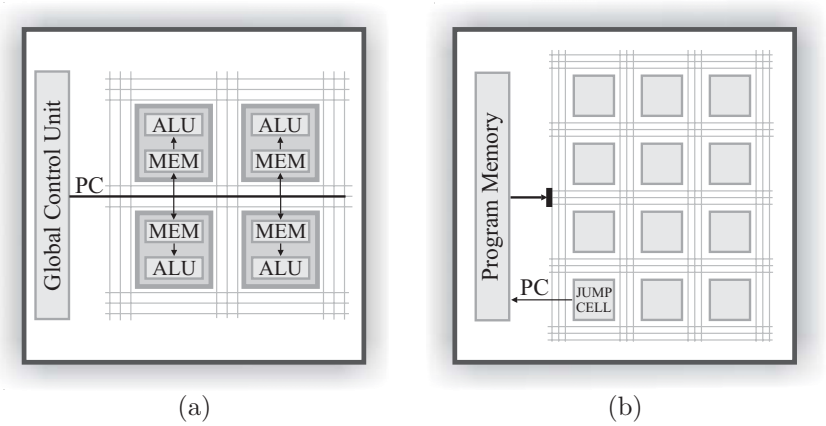


Figure 3.6: Some ALU arrays suggest hardware mechanisms to support a control flow as in (a) REMARC that uses a globally distributed program counter, or (b) RICA that uses special *JUMP* cells to change execution flow.

ALU has a local instruction memory, but there is no locally controlled program counter. Instead, one single program counter is distributed to all ALUs, as shown in Figure 3.6(a). An instruction controls which operation to perform and also to which I/O ports operands and results should be associated. Hence, the interconnect network is controlled by the configuration, which allows different data streams to time-share a single ALU resource. As this type of control flow mechanism requires static schedules, REMARC reminds of a VLIW architecture. However, compared to VLIW machines, which distribute an instruction to every functional unit, only the program counter is globally distributed. Additionally, operations and connections can remain static for a time longer than a clock period. Hence, instruction rate and program size is effectively reduced for regular data-flow computations. One weakness with this mechanism is that if only a few ALUs require an instruction flow, remaining ALUs will need to replicate the same instruction in order to have an unaltered functionality when the program counter is updated. Another weakness is that a local condition can not change the control flow locally. Instead, the condition must be communicated back to the global controller, which alters the program counter and changes the context globally.

In RICA [45] there are specialized cells, which control the next configuration, embedded in the array. These cells are called *JUMP cells* and they allow conditional loading of a new configuration by altering the program counter.

When the program counter is altered a new configuration is loaded into the reconfigurable fabric. The RICA control flow mechanism is shown in Figure 3.6(b). With this technique conditional execution is supported, however the context is changed globally.

A more general approach is suggested in XPP-III [55] where sequential execution is supported by small embedded processor blocks. With an instruction flow to the ALU control functions such as `if-then-else` and loop-constructs can execute within a single functional block to allow time-sharing to be efficiently implemented.

3.2.3 Processor arrays

Significant for processor arrays is that they allow operations to change on a cycle-by-cycle basis without any reconfiguration. Hence, all functional blocks have their own program flow and therefore processor arrays are classified as multiple instructions multiple data (MIMD) machines. A typical processor array have functional blocks that contain an ALU, execution pipeline, instruction memory, and a program counter, as seen in Figure 3.7. The same type of pipelined and parallel implementations supported by ALU arrays are also supported by processor arrays. In addition, implementations that require sequential execution are supported. Typical architectures in this family are MATRIX [56], RAW [41], CHESS [57], PicoArray [58], WPPA [59], MPPAs from Ambric [60], and FPOAs from MathStar [61]. Early presented architectures such as MATRIX and CHESS had 4-bit and 8-bit ALUs and hence wider wordlengths required several functional blocks to be interconnected. Recently suggested architectures have wider ALUs with 16 or 32 bits. What follows is a brief summary of main features in MATRIX, CHESS, RAW, and PicoArray.

- **MATRIX** – The basic block consists of an 8-bit ALU and a 256×8-bit memory block. The basic block can operate as either instruction memory, data memory, independent ALU operation, or a byte slice of a bundled ALU operation. Although memory blocks are included in functional blocks, they are shared resources as address and data ports can be connected to other blocks through the interconnect. The interconnect contains three levels: dedicated links to 4 nearest neighbors, bypass connections, and row and column buses.

- **CHESS** – The basic block consists of a 4-bit ALU and a 16×4-bit memory block. As in MATRIX, basic blocks can operate as either instruction memory, data memory, independent ALU operation, or a 4-bit bundled ALU operation. In addition to memory blocks embedded into the basic blocks, there are dedicated 256×8-bit memory blocks distributed across

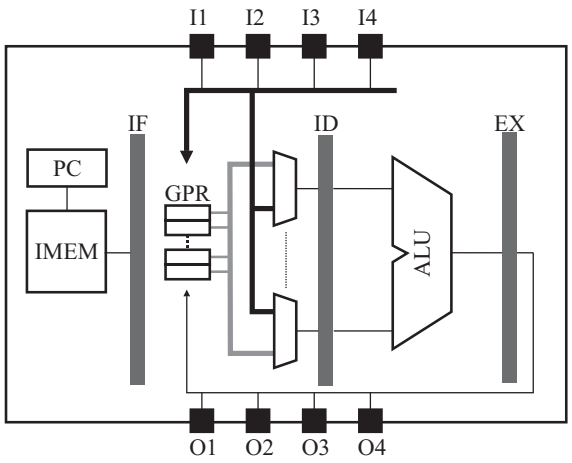


Figure 3.7: A processor array is made of functional blocks that contain an ALU, execution pipeline, instruction memory, program counter, general purpose registers, and I/O ports to the interconnect.

the array and they cover whole columns. The interconnect is build as 16 segmented buses connected to switches across each row and column.

- **RAW** – The basic block consists of an 8-bit ALU, a small amount of configurable logic, an 16 kB instruction memory, a 32 kB data memory and a switch and its associated 16kB instruction memory. The configurable logic is used to construct operations uniquely adapted to specific applications, which gives another level of adaption. The memories inside the basic block are dedicated to that specific block and cannot be accessed by any other block. However, the processor can operate as a address generator that feds data from memory to any other basic block. The interconnect consist of the switches that are integrated into the processor pipeline. These switches support statically defined interconnect channels by using the switch instruction memory, but also allow messages to be dynamically routed from source to destination.

- **PicoArray** – The basic block consists of a 16-bit processor, an instruction memory, and a data memory. Amount of memory and supported instructions are varied in three predefined basic blocks: *standard*, *memory*, and *control*. The *memory block* has 8704 bytes of memory, whereas the *standard block* has 768 bytes. As in RAW, memories are dedicated to each basic block and not part of the network as shared resources. The

interconnect is build of 32-bit row and column buses and programmable switches. Each port to the interconnect contains a buffer so that synchronization is performed when ports are accessed with special-purpose read-write instructions.

Although these architectures remind of multi-processor systems, they have unique features that make them more efficient when fine-grained parallelism is exploited. Compared to conventional multi-processor systems, processor arrays allow communication with less latency and higher throughput. This is accomplished by a reconfigurable interconnect with fast communication between processor registers, whereas multi-processor systems implement communication through the memory system. In order to unify a local instruction flow with efficient inter-processor communication, processor arrays need some mechanism to synchronize communication. Although there are static approaches, the dominating mechanism is based on handshaking channels. A typical protocol is to have a control bit from sender to receiver, which signals that the sending unit has valid data. Furthermore, there is a control bit that flows from receiving unit back to the sending unit, which signals that the receiving unit needs new data. Although dynamic approaches result in an area overhead, they simplify programming and mapping and they can effectively handle nondeterministic effects such as unknown execution time or contention in shared resources [58–60]. Typically, a handshaking channel is a word-level, unidirectional, point-to-point link from one functional block to another. Sending or receiving a word on these channels is carried out either by special purpose move instructions [58], or by allowing ports to be used as any local register in any instruction [60]. When a word is sent through a channel it is both a communication and a synchronization event. Local execution is stalled if inputs can not be received, or outputs can not be transmitted. A handshaking channel can be implemented with a two-phase protocol with *request* and *acknowledge* signals or with a FIFO that generates a *back-pressure* signal when it can not receive more data. With FIFO buffers between communicating blocks *globally asynchronous locally synchronous* architectures may be implemented, which eliminates the complexity involved with clock distribution networks in large processor arrays.

3.2.4 Heterogenous architectures

To obtain better performance or reduced hardware area, different types of functional blocks are used to build the reconfigurable array. A typical example is memory blocks that are used to provide improved storage capacity in both fine- and coarse-grained arrays. Heterogeneity introduces an extra design aspect, which is how to spatially distribute different functional blocks within

the array. It also increases complexity of spatial partitioning, placement, and routing as compared to a homogeneous set of functional blocks.

The current trend in commercially available FPGAs is that more coarse-grained functional blocks are embedded into the reconfigurable architecture. Compared to utilizing the fine-grained blocks, kernels that utilize coarse-grained blocks have improved area utilization, power consumption, and performance. Typical coarse-grained blocks found in commercially available FPGAs are listed below in increasing granularity.

- **Memory blocks** – A memory block allows a set of memory widths and depths to be implemented. In addition, memory blocks can be cascaded to implement deeper or wider memory types. The memory blocks are typically implemented with synchronous static RAM (SRAM). For example Altera Excalibur has 2 kbit memory blocks and Xilinx Virtex II has 18 kbit memory blocks.
- **Multiplier blocks** – Multiplications are common in DSP algorithms and they are not efficiently implemented with lookup tables. Consequently, FPGAs embed hardwired multipliers to improve performance for typical signal processing algorithms. For example, a Virtex-II multiplier block is an 18-bit by 18-bit wide multiplication.
- **DSP blocks** – In addition to multipliers these blocks typically contains large accumulators, pipeline registers, and multiplexors. This is used for efficient implementation of multiply-accumulate operations, which are commonly used in digital signal processing.
- **Processor blocks** – Hardwired processor blocks with caches such as ARM922T inside the Altera Excalibur family and the PowerPC 405 inside the Xilinx Virtex-II Pro and Virtex-4 families. This allows embedded systems, with custom acceleration units implemented in the reconfigurable architecture, to be implemented. With a configuration interface attached to the embedded processor this allows reconfigurable computing platforms to be realized.
- **Embedded blocks** – A recent trend is to hardwire common building blocks that are used to implement embedded systems. These blocks contain, in addition to processor blocks, more advanced embedded components such as buses and frequently used peripherals.

Heterogeneity is also exploited in many ALU and processor arrays, which have different functional blocks and also memory blocks embedded into the reconfigurable architecture. Heterogenous functional blocks are used to improve

performance in a constrained chip area. That is, hardware complexity of individual functional blocks are reduced and the released area utilized to increase number of functional blocks. For example, PicoArray [58] has three types of processors: a *standard*, a *memory*, and a *control* processor. Another example is WPPA [59], which allows the instruction set of individual processors to be decided prior to implementation.

The advantages with embedded memory blocks are that data may be partitioned into different memories, in order to improve data delivery. Furthermore, they may operate as scratchpad memories during computation to avoid slow external interfaces. In FPGAs, memory blocks are *shared resources* embedded into the interconnect, which means that they are not included and dedicated to a specific functional block. Shared memory blocks is also used in architectures such as RaPiD [52] and Pleiades [62]. These architectures typically have local address generation units embedded into the memory block itself. Processor arrays, generally have memories *dedicated* and included in the functional block. In contrast to architectures that have dedicated address generation units, processor arrays can use the processor to generate addresses, as in RAW [41] and PicoArray [58].

Memory blocks are excluded in some coarse-grained arrays that are intended to be tightly coupled to a processor. In these systems, the reconfigurable units use the processor cache memory or access data in main memory using direct memory access. This is significant for MorphoSys [63], PipeRench [47, 49], Garp [48], XPP [44], ARRIVE [53], and ADRES [43].

3.3 Reconfigurable Interconnect

The reconfigurable interconnect includes resources such as programmable switches and wires organized so that specialized communication networks can be created. As with functional units, reconfigurable interconnects can be either fine-grained or coarse-grained. A fine-grained interconnect allows individual wires to be switched, to route bit-level connections between functional blocks. On the other hand, a coarse-grained interconnect considers a group of wires as a bus that is switched as one unit. In most architectures the granularity of the interconnect and functional blocks are matched, as for example 4-bit ALU arrays have 4-bit buses.

Traditional FPGAs, typically have vertical and horizontal channels that consist of programmable *switch blocks* and *connection blocks*, as seen in Figure 3.8 [50]. The logic block is connected to the channel through the connection block. Switches are used to connect horizontal and vertical channels, to allow signals to change direction. To provide both local and global routing two primary methods have been presented [7]. The first, *segmental routing*, is to use

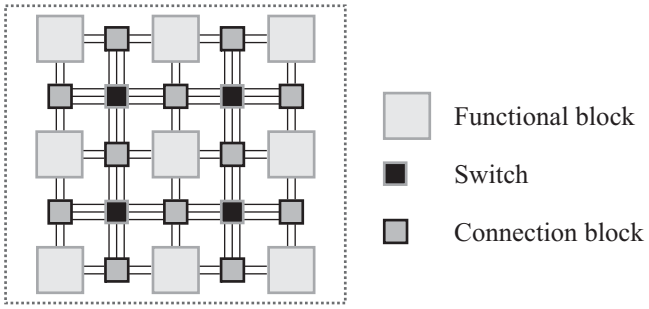


Figure 3.8: Traditional FPGAs contain interconnect based on vertical and horizontal channels with programmable switches and connection blocks.

longer wires that span multiple functional blocks without being switched. The second, *hierarchical routing*, provides denser wiring within a group of functional blocks and these groups are then connected with a hierarchical organization of switches.

In contrast to FPGAs that have horizontal and vertical channels, many coarse-grained arrays combine dedicated local communication with another level of interconnect through shared buses. The idea with dedicated local interconnect is that, assuming efficient placement, communication is mainly local and only a fraction of the communication requires to be routed longer distances. REMARC [54], for example, has dedicated wires between adjacent four neighbors and shared buses that span each row and column. Functional blocks in MorphoSys [63] have dedicated connections between four nearest neighbors, a second level of connections for a group of 4×4 functional blocks, and horizontal and vertical buses that span each row and column. A denser nearest neighbor interconnect is suggested in MATRIX, which has dedicated connectivity between the 12 nearest neighbors, a second level of 8 bypass connections, and row and column buses. In order to obtain a clock frequency that are known prior to placement and routing, these interconnects typically have deterministic delay by including pipeline registers within the interconnect.

To reduce routing overhead, some coarse-grained architectures utilize one dimensional routing. A typical result of this strategy is RaPiD [52], which is a one dimensional array in which functional blocks are connected with ten segmented buses. Two other examples are PipeRench [47, 49] and ARRIVE [53], which are both two dimensional arrays, but with limited routing in one dimension. ARRIVE uses switches for horizontal routing, whereas vertical routing is one directional and must be handled by a functional block. PipeRench has

a crossbar to connect functional blocks horizontally, whereas vertical routing is carried out in one direction only. The idea behind this is that in a good placement of regular functionality, data flows in a pipelined fashion from one stage to another.

3.4 Nomenclature

In the field of reconfigurable computing there are many terms and here is a brief summary of the terminology used in this chapter and in included parts:

- **Configuration** – Is the information stream used to configure or reconfigure the reconfigurable architecture.
- **Context** – A task with its own configuration.
- **Dynamic reconfiguration** – Is the ability to reconfigure the fabric during execution by downloading a new configuration to the configuration memories. Also referred to as run-time reconfiguration.
- **Hardware-software partitioning** – Division of an application into partitions that are either implemented in the processor (software) or on the reconfigurable fabric (hardware).
- **Mapping** – Is to assign functionality to different functional blocks. It can include some or all of spatial partitioning, temporal partitioning, placement, and routing.
- **Multi-context** – A reconfigurable architecture that has multiple memory bits per configuration bit. A multi-context reconfigurable architecture can switch very fast between two configurations. In addition, new configurations can be downloaded in parallel with an actively used configuration.
- **Partial reconfiguration** – Is the ability to reconfigure only parts of the reconfigurable fabric at a time. The configuration memories operates like random access memories, so that addresses can be used to selectively change their content.
- **Placement** – To bind functionality to a specific functional block in the reconfigurable fabric.
- **Routing** – To bind communication between functional blocks to specific channels in the interconnect.

- **Single context** – A reconfigurable architecture that has a single memory bit per configuration bit. Significant for these architectures is that loading new configurations may result in a significant overhead in dynamically reconfigurable systems.
- **Spatial partitioning** – Division of a computational kernel into smaller pieces that are mapped onto different functional blocks in order to exploit parallelism. Also referred to as functional partitioning.
- **Temporal partitioning** – Division of a computational kernel into smaller pieces that each have their own configuration and can execute in the reconfigurable fabric at different periods in time.

3.5 Discussion

Reconfigurable computing systems is still a maturing field and there are many open research questions. Over the last ten years numerous new reconfigurable architectures have been proposed [7, 38, 64]. However, there is not an existing framework to allow quantitative analysis of these architectures and hence it is not straightforward to relate impact of innovations to previously presented techniques. This is a major challenge and to allow such an analysis, the design parameters needs to be well understood. They need to be related to physical entities such as power-delay-area and the constraints imposed by the application domain [62]. Typical design parameters are number of resources, resource types, area-balance between resources, interconnect, and integration into a system. The complexity of these systems requires advanced modeling, simulation, and analysis tools. This was recognized as a key issue already by the pioneers in the field, which 1976 developed system architects apprentice (SARA) to discover and cure problems before the system was physically realized [2].

Another major challenge is to find tools that can map an application to a reconfigurable architecture and make use of all innovative mechanisms embedded in the reconfigurable architecture. Without automatic tools, the mapping step becomes a bottleneck both during design of an application, and in the evaluation of a specific reconfigurable architecture, as mapping is an integral part of a design exploration loop.

Part I presents a coarse-grained reconfigurable *ALU array* and evaluates it's performance in executing a signal processing intensive speech codec. Part III presents a *processor array*, and suggests how system level modeling tools may be used to evaluate and tune system parameters before a chip is fabricated. In Part II, it is described how a complete reconfigurable computing platform is constructed, simulated, evaluated, and tuned using a system level design approach.

Chapter 4

System-Level Modeling and Exploration

Development of embedded systems requires *modeling concepts* that allow designers to reason about different aspects of the system. These concepts include behavior, parallelism, time, structure, hierarchy, and communication. At different phases of development there are different views of the system with varying importance of these modeling concepts. For example, software development requires hardware to be modeled with precise register and memory map and accurate behavior, but may exclude timing information. System architects carrying out performance exploration to find bottlenecks require an estimated time metric, but may exclude detailed hardware modeling. In order to evaluate the design, an executable modeling language is needed [65]. Traditional modeling languages have emphasized the hardware design flow, whereas recent languages have incorporated concepts for efficient modeling at system level. With traditional modeling abstraction, models take too much time to develop, have poor simulation performance, and are not available early enough in the design flow to allow architectural exploration and software development. In this chapter, *transaction level modeling* is introduced as an efficient abstraction for system level *architectural exploration*. Transaction level modeling is supported in languages such as SystemC and SpecC by abstract *channels* that connect communicating modules. The discussion in this chapter will bias the discussion towards the transaction level approach suggested by the SystemC transaction level model working group. After discussing transaction level mod-

els, this chapter continues with a brief discussion on architectural exploration.

4.1 Transaction Level Modeling

There are two major abstraction levels used to model hardware systems: *register transfer level* and *transaction level* (TL). The former is the traditional modeling abstraction and the term *register transfer* refers to that models describe signal transitions between synchronously clocked registers. With design tools, an RT-level description is translated into a netlist of logic gates. Transaction level modeling is an abstraction adopted by system companies to address limitations of register transfer level design when system complexity increases. This section first gives a brief overview of transaction level modeling, and then the SystemC TLM library is introduced as it is currently the most widely spread language for simulating at the transaction level. After introducing the SystemC TLM library, this section describes the abstraction levels commonly used in system simulations, followed by a short modeling example of a bus system.

4.1.1 Overview

Transaction level models use abstract *channels* to model communication between concurrent processes in the system using function calls [66–69]. Consequently, an interface that is modeled with several ports in a RT-level model, is reduced to one single port in a TL model. In addition, a series of signal assignments used to model the timing behavior are replaced with function calls that either send or receive a transaction. A transaction is the atomic information that modules exchange in the simulation. It may represent a complex structure that contains control information and burst of data, or it may represent a single word of data. How a transaction is defined depends on how accurate the system is modeled. One transaction may be represented by multiple function calls in the model in order to accurately capture the time behavior. Time within TLM components are modeled as *untimed*, *estimated timing* or *cycle accurate* [70]. It is possible to make a transaction model *cycle accurate* and still achieve better simulation performance than offered by an RT-level model [68].

Transaction level modeling is intended as a complement to RT-level modeling, in order to perform activities such as [71]:

- Hardware micro-architecture exploration and starting point for more detailed hardware modeling.
- System level architectural exploration, such as selecting communication and processing components and HW/SW partitioning.

- Virtual platform for software development.
- Reference model for hardware functional verification.

Transaction level modeling is supported by languages such as SystemC [71, 72] and SpecC [65], through the channel concept. A channel connects communicating modules, which are either initiators or targets. An initiator can actively start a transaction whereas a target is only allowed to respond to an initiator request. When multiple initiators and targets need to communicate, the channel is required to implement *arbitration* and *routing* algorithms. The arbitration algorithm selects which initiator that should be given access to the channel, and the routing algorithm assures that the correct target is addressed by using information annotated in the transaction. With these concepts, interconnect structures found in SoCs can be modeled with adequate accuracy.

4.1.2 SystemC TLM Library

The open SystemC initiative (OSCI) [73] maintains a simulation library for SystemC. OSCI SystemC is a C++ library that contains routines and macros to simulate concurrent processes using a HDL like semantic. Systems are constructed from SystemC *modules*, which are connected to form a design hierarchy. A SystemC module encapsulates processes, which describe behavior, and communicates through *ports* and *channels* with other SystemC modules. Processes are used to describe concurrency and *wait-statements* are used to halt process execution for a specific time or until an *event* occurs.

The OSCI SystemC TLM library contains ports, interfaces, channels, and also data structures used to represent *request* and *response* in an initiator-to-target communication scenario. These are TLM primitives that increase model interoperability and allows rapid development of customized TL models, and it is not ready-to-use models of SoC components. SystemC TLM channels provide an inter-process communication mechanism based on *unidirectional* or *bidirectional* function calls. The unidirectional mechanism is based on blocking and non-blocking *put* and *get* functions. The *get* function moves transactions from the channel to the module, while the *put* function moves transactions from the module to the channel, as illustrated in Figure 4.1(a). The bidirectional mechanism is a single call to a *transport* function which sends a *request* and returns with or without delay with a *response*, as illustrated in Figure 4.1(b).

As seen in Figure 4.1(b), the bidirectional mechanism is based on that initiators call the transport function from a process, whereas targets do not contain processes. Consequently, targets implement the *transport* function call, which will be invoked in the initiator-process. The unidirectional mechanism,

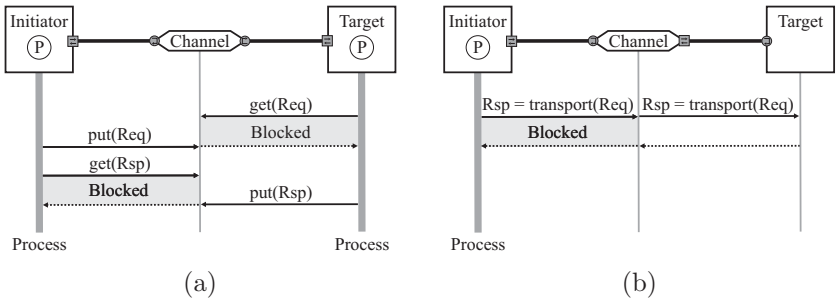


Figure 4.1: The OSCI TLM standard defines (a) unidirectional and (b) bidirectional function calls to model communication between modules. The figure shows sequences of function calls necessary to model a transaction. Function calls to *put*, blocks the calling process when the channel FIFO is full and function calls to *get* blocks the calling process if the channel FIFO is empty. With the bidirectional interface, function calls to *transport* is blocked if the target calls wait.

on the other hand, requires both initiators and targets to call their functions from a process, as seen in Figure 4.1(a). Hence, context switching will increase with a unidirectional interface. The advantage with the unidirectional interface is that it increases temporal accuracy, as many features implemented in modern bus systems require initiators, targets, and the bus itself to be modeled with separate processes. For example, consider overlapped arbitration, which is modeled as parallel activities, that is arbitration on the bus in parallel with an ongoing transfer from the currently addressed target. However, for untimed models, the bidirectional mechanism is preferred as simulation performance is increased, and model complexity is reduced.

The unidirectional channel, shown in Figure 4.1(a), are implemented as first-in-first-out (FIFO) queues with a configurable buffer depth. Blocking functions will block the calling process until the condition for getting or putting a transaction to the channel is fulfilled. The non-blocking versions return a boolean value to indicate if the call succeeded, but return immediately even if the condition is not satisfied. There are also functions to find out if a transaction can be moved without doing the transfer, events emitted when a new transaction enters or leaves the channel, and functions to get a transaction without consuming it from the channel.

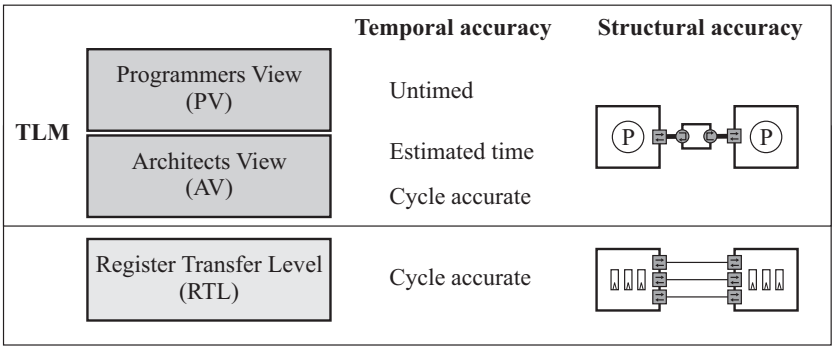


Figure 4.2: Transaction level models refer to a set of abstraction levels with different structural and temporal accuracy. One definition of TLM abstraction levels according to OSCI TLM standard are programmers view and architects view.

4.1.3 Levels of Abstraction

Even though the word *transaction level* may indicate a single level of detail, it refers to a set of abstraction levels with varying degree of functional and temporal detail [74]. Although terminology and definitions of specific abstraction levels within this set is still under development, two important abstraction levels are singled out here according to the OSCI TLM standard [74], see Figure 4.2:

- **Programmers view (PV)** – This abstraction level is intended to be used for software development and functional verification of the application. It contains adequate structure to represent the different components such as buses, processors, memories, and other hardware components. In addition, registers and memory map are accurately modeled so that software drivers can be developed and verified. This abstraction level provides an untimed model in which all transactions occur at a single instance in time. Although time is not modeled, communication channels may use some simple arbitration algorithm to resolve bus contention.

- **Programmers view with time (PVT)** – This abstraction level is also referred to as *architects view* (AV) as it is intended to be used for performance analysis and architectural exploration. It contains the same structural accuracy as the PV model, but has increased temporal accuracy to model *cycle accurate* or *estimated time* models. Time information is annotated into models with wait-statements inside processes.

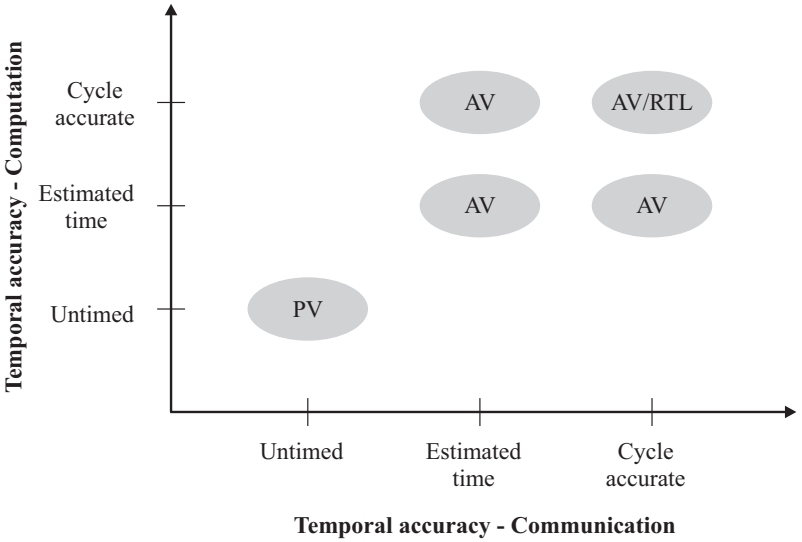


Figure 4.3: Temporal accuracy of TL-models in two dimensions: communication and computation. With these abstractions communication and computation can be developed, evaluated, and refined independently.

It is suggested in [67] that temporal accuracy in communication and computation can be developed and evaluated independently. This leads to an extended view of TLM abstraction levels, as shown in Figure 4.3. If, for example, the bus system should be explored, a *cycle accurate* bus model is used while masters and slaves are *estimated time* models. Mixed abstraction levels are also useful for micro-architecture exploration of new hardware components, which then use cycle accuracy while remaining components are modeled with *estimated time*. Temporal and structural accuracy should be carefully selected to match design activities and required prediction accuracy.

4.1.4 Modeling Example

With the TLM primitives in the OSCI TLM standard, more complex models can be developed. This section gives a brief description on how to model a generic bus system. Buses are one of the most required components in system simulations and several proposals of generic TLM bus models are found in [75, 76]. The main objectives for these models are to increase productivity by reusing the same models for different bus protocols, increase interoperability between models, and allow mixed abstractions. In addition, base classes with

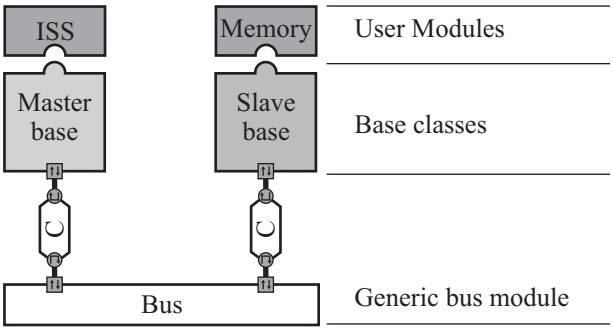


Figure 4.4: User modules inherits from the master or slave base class and expand functionality to model the specific behavior, timing, or register map.

common functionality for bus slaves or masters can be developed, so that new bus components can be developed reusing these base classes, as illustrated in Figure 4.4. The typical design process to develop bus-based transaction level models are:

1. Decide if the model should be used for software development, architectural exploration, or both.

2. Extract from bus specification what needs to be communicated in parallel. For example, data and address buses might operate in parallel, and hence it needs to be decided if this should be modeled as separate channels in order to reach desired temporal accuracy.

3. Define *request* and *response* structures and write interface classes. It is advised that information in *request* and *response* have a one-to-one correspondence to the information transferred in the pin accurate model. For example, if there are three transfer types, single, burst, and cache line, there should be a data-type called transfer-type in the *request* data structure.

4. Implement bus functionality, *arbitration* and *routing*, as untimed model.

5. Implement slave and master *base classes* as untimed models.

6. Verify functionality of the bus protocol by implementing a system consisting of traffic generators derived from the master base class, and memories derived from the slave base class.

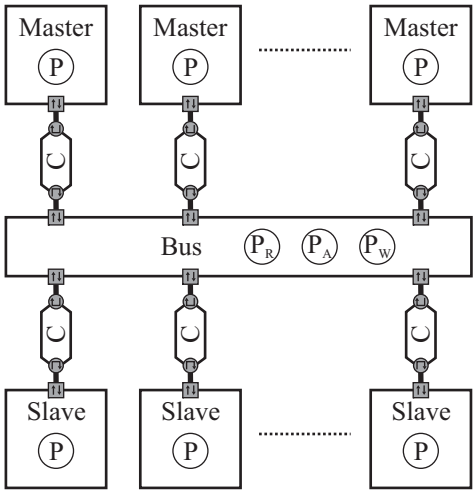


Figure 4.5: A generic bus model that uses the unidirectional TLM channels. The bus implements arbitration and routing algorithms. A transaction can be modeled with several *request* and *response* pairs in order to adequately model different phase in the bus protocol. The bus contains three processes to model parallel address (P_A), data read (P_R), and data write (P_W) interfaces.

7. Refine timing by annotating wait-statements in the processes according to the protocol. Guard wait-statements with a condition that can be switched on and off in order to improve simulation performance when temporal accuracy can be sacrificed.

8. New user modules are developed from the base classes.

Figure 4.5 shows how unidirectional channels is used to model the CoreConnect processor local bus (PLB) [77] at the *Architects View*. Each master and slave is connected to the bus module through a unidirectional channel. The type of channel used to model buses, uni- or bi-directional, depends on the level of temporal accuracy required in the model. If the model is intended only for software development, the bidirectional channel is preferred. The unidirectional channel is used for models intended for software development as well as architectural exploration.

The model illustrated in Figure 4.5 uses separate channels for address and data to reflect the PLB bus specification, which has decoupled address and data buses to support overlapped arbitration and address pipelining. The bus model

has 3 processes, P_R, P_W, P_A, to represent parallel read, write, and address interfaces. A transaction is initiated by a master which sends an address *request* to start an *address cycle*. The address-process inside the bus-module is then activated and performs arbitration using priority in the *request* combined with round robin arbitration on masters' fixed priority numbers. After arbitration, routing is performed by matching the *request* address to any of the slaves memory map. Finally, the *request* is transferred to the correct slave. After the address cycle, read or write *data cycle* starts, as the slave acknowledges the address *request*. The data cycle consist of a single word, a burst, or a cache line. Burst and cache line transactions are modeled either returning all words in a single *response*, or by modeling a data exchange for each word. The latter is used for higher temporal accuracy. The model is *cycle accurate* if enabling all wait-statements and if burst and cache line *requests* are modeled with one word per *response*. Hence, bus features such as data bus size, address pipeline depth, arbitration policy, burst size, etcetera can be evaluated for different application scenarios.

4.2 Architectural Exploration

A major task in a system level design flow is to systematically search for an architecture that balances competing design goals, such as performance and area, referred to as *architectural exploration*. In some design scenarios this activity can be as simple as tuning soft parameters, such as arbitration policy on a fixed hardware platform, to ensure real-time performance. In other design scenarios, such as development of new architectures, it might be required to evaluate how various design parameters are related to performance and area. The design metric is estimated as a combination of static and simulation-based analysis [65]. For example, area is extracted from the number of allocated resources such as memories and arithmetic operations, whereas performance metrics are extracted from simulation. A typical architectural exploration process is summarized as [78]:

I. A target architecture is selected and functionality mapped to this architecture. Bottlenecks are identified by *observing* performance data, such as execution time and utilization, extracted during the simulation.

II. Modifications to the architecture are suggested and *architecture tuning* is carried out by *controlling* different architectural parameters.

III. One of the architectures is finally selected based on the performance and other properties such as area and energy consumption.

Architectural exploration requires performance metrics to be collected during the simulation. This metric needs to be defined and annotated into models as it is module-specific. In a bus-based system, the bus itself is a natural module for communication based metric, and hence arbitration and slave *latencies*, *utilization*, *throughput*, *number of requests*, *burst size*, etcetera, are collected for each master. For some performance metrics, such as latency, it is natural to keep *average*, *maximum*, and *minimum* value. As these statistical values depend on simulation time, the time-frame during which they should be calculated should be controllable, in order to carry out both short-term and long-term analysis.

An important aspect for efficient design exploration and performance analysis is the design methodology. It involves *construction* and *configuration* of the system to be simulated, as well as *controllability* and *observability* of simulation modules. All the simulation modules are instantiated, connected and configured prior to the simulation step. During simulation, the modules are controlled to setup a simulation scenario, while performance and simulation data is observed and used for performance analysis. As there might be several thousands of systems that need to be evaluated, it is also important that *exploration scenarios* are expressed in a formal language that can step through the systems, post-process gathered metric, and organize the result for visualization in existing graphical tools. For example, if 8 processors, 4 arbitration policies, 4 memory configurations, and 2 different buses should be evaluated, there are in total 256 different systems that should be explored. Hence, manually constructing each system, observing performance metric, and post-processing gathered data, would be to time-consuming and error-prone.

4.3 Discussion

Due to the continuous increase in design complexity, system level exploration tools and methodologies are required to rapidly evaluate system functionality and performance. SystemC has shown to be a powerful system level exploration and modeling language. The advantages with SystemC, besides the wellknown C++ syntax, include modeling at different abstraction levels, simplified hardware/software co-simulation, and a high simulation performance. The abstraction levels range from *cycle accurate* to transaction level modeling, where higher abstractions trade simulation accuracy for a higher simulation speed. To address the discussed aspect for efficient design exploration, a SystemC environment with interactive control (SCENIC) has been developed and it is presented in Part II.

Part I

A Coarse-grained Reconfigurable Coprocessor Targeting DSP Kernels

Abstract

This part investigates the applicability of reconfigurable coprocessors that target processing kernels in multimedia applications. A coarse-grained reconfigurable coprocessor is presented and we evaluate its efficiency with respect to accelerating signal processing kernels and the G.723.1 speech codec. Speedups in the range of 2 to 46 compared to processor execution are achieved for vector operations and larger kernels such as filtering and fast fourier transform. These kernels utilize 86% of the G.723.1 processing time on the RISC based target architecture. With our approach the average used clock cycles are reduced by 83% compared to processor-only execution.

Based on: Henrik Svensson, Thomas Lenart, and Viktor Öwall, "Accelerating Vector Operations by Utilizing Reconfigurable Coprocessor Architectures ," in *Proceedings of IEEE International Symposium on Circuits and Systems*, New Orleans, USA, May 2007, pp 3972–3975.
and: Henrik Svensson, Thomas Lenart, and Viktor Öwall, "Implementing the G.723.1 Speech CODEC using a Coarse-Grained Reconfigurable Coprocessor," in *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, USA, June 2007, pp 195–198.

1 Introduction

Hardware platforms that target speech and audio signal processing are often composed of digital signal processors (DSPs), general purpose processors, and hardware accelerators. Hardware accelerators found in these platforms are dedicated hardware, single-instruction multiple-data (SIMD) units, vector processing units, and reconfigurable architectures (RAs). Among these approaches, reconfigurable architectures become increasingly interesting because they combine high performance and flexibility. They have proven to be adaptable and effectively exploit parallelism in a variety of application domains [7]. Current trends in embedded multimedia processing are coarse-grained reconfigurable architectures (CGRAs) compiled from a heterogeneous set of hardware elements such as processing elements, memories, address generators, and interconnect [7, 46, 47, 79, 80]. Coarse-grained architectures provide datapaths, which are build from word-level interconnect switches and processing elements. Compared to traditional field programmable gate arrays (FPGAs), a CGRA trades mapping flexibility to reduce delay, area, power consumption, and configuration time.

This part is concerned with design and system integration of a coarse-grained reconfigurable coprocessor that is used to accelerate kernel functions in digital signal processing (DSP) applications. It introduces our coprocessor accelerator Cpac, and proposes a hybrid computational model that allows it to operate as a *functional unit* extension, *vector accelerator*, or *DSP kernel accelerator*. Cpac contains programmable processing elements, a reconfigurable interconnect, and a memory system with multiple memory banks. Applications with one or more signal processing kernels that require some amount of acceleration beyond a basic RISC processor can utilize Cpac. Performance is enhanced by dynamically configuring processing elements into specialized datapaths that compute operations in inner loops. Furthermore, a multi-bank memory system operates as scratchpad memory to reduce the amount of external communication. Multiple memory banks allow data to be communicated in parallel. Cpac is designed as a parameterizable architecture in register transfer level VHDL. A parameterizable description facilitates rapid optimization of resources, wordlenghts, and interconnect to different application domains.

For a case study, we choose to accelerate the G.723.1 [81] fixed-point speech codec, to evaluate Cpac in the speech processing domain. The G.723.1 codec is part of the H.324 standard and it is used to transmit and receive speech at 6.3 and 5.3 kbps using multipulse maximum likelihood quantization (MP-MLQ) and algebraic code excited linear prediction (ACELP), respectively. Acceleration is carried out by extending a PowerPC system with Cpac and move time consuming DSP kernels from PowerPC to Cpac. Profiling results gathered

from a Xilinx Virtex-II Pro PowerPC show that 86% of the 6.3 kbps and 77% of the 5.3 kbps encoder execution time is spent in such DSP kernels. If these kernels are accelerated, Amdahl's law states that overall speedups have upper bounds that are 7.1 and 4.3 for the 6.3 kbps and 5.3 kbps codec, respectively. It is expected that similar figures may be extracted from profile investigations of other speech codecs, such as GSM enhanced full-rate and adaptive multi-rate. Experimental results demonstrate a significant performance improvement when these kernels are accelerated in CPAC. The implementation achieves an overall speedup of 5.9 for the high rate encoder and 4.0 for the low rate encoder. Compared with the target architecture without any coprocessor, this platform can operate with reduced clock frequency and voltage in order to conserve energy and still be able to maintain real time constraints.

2 Related Work

Several approaches have been proposed to find architectures that meet the challenging requirements on flexibility, performance, and energy dissipation in multimedia applications. Processor designers have suggested to use vector processing or SIMD extensions. In [29] it is shown that 8 out of 10 benchmarks in a multimedia benchmark suite have a degree of vectorization higher than 90%. Although vector and SIMD architectures result in programmable solutions, their performance is limited by their *centralized memory system* and their *hardwired datapaths*.

Reconfigurable architectures can dynamically change the datapath operation itself. Hence, they can explore pipelined and parallel computational structures that are customized for the specific application. Additionally, memories embedded into reconfigurable architectures may be distributed, so that memory bottlenecks are reduced. Several systems that extends a processor system with a reconfigurable architecture have been presented [7,46,80]. Proposed architectures differ in several aspects, among others mechanisms by which to interface and communicate with the processor, granularity of processing elements (PEs) and memory management system. These are all key issues that require more investigation and need to be evaluated for specific application domains.

In the case study, CPAC is used to accelerate speech processing kernels found in the G.723.1 speech codec. Several implementations of the G.723.1 codec have been reported in the research community. In the approach presented in [82], single *for-loops* are accelerated in a dedicated coprocessor that operates in tandem with an MIPS R3000 processor. The implementation reported in [83] utilizes a functional unit extension to a standard RISC in order to accelerate basic operators that are used in several fixed-point speech codecs from the international telecommunication union (ITU) [84]. Both approaches [82,83] utilize

fine-grained parallelism. There are also approaches which utilize coarse-grained parallelism, such as in [85] where an DSP operates in parallel with an ASIC accelerator that implements perceptual weighting filter, pitch estimator, and noise shaping filter. There are also pure hand-optimized DSP implementations, as presented in [86, 87]. In [79, 88] a reconfigurable architecture that targets the speech coding domain is described. Authors report a 44.6% reduction of clock cycles for the GSM coder and 98.2% clock cycle reduction for kernels they accelerate.

Compared with the dedicated coprocessor architectures reported in [82, 83] our approach is able to provide better performance. There are two other advantages with the our proposed architecture. First, these implementations lack embedded memories. Consequently, data access speed is limited by the memory interface provided by the host processor. Hence, they have poor scalability because increasing datapath resources in these implementations might not affect the execution time since memory communication has become the major bottleneck. Secondly, our approach based on a reconfigurable architecture provides hardware that can be reused in the application domain. Compared to the reconfigurable implementation reported in [79, 88], our architecture provides a more flexible computational model. This allows the reconfigurable datapath to be used either as a kernel accelerator where the coprocessor memory system and address generators are utilized during computation, or a functional unit extension where memory operations and loop variables are kept in the processor. These models enhance the flexibility as it allows design automation tools, such as reported in [25], to find acceleration objects with different granularities.

3 Architecture

This section introduces the Cpac architecture, which has been developed as a *parameterizable* VHDL description. A parameterizable description here refers to the ability to adapt machine parameters to different application domains prior to synthesis, contrary to *reconfiguration*, which refers to the ability to dynamically change the functionality after fabrication. A Cpac configuration is derived from a data-flow graph that describes parallel and pipelined computations. The nodes in the data-flow graph represent fundamental operations, such as addition or multiplication, which are mapped to processing elements. A configuration is held in registers and controls the current operation in processing elements, interconnection of processing elements, and access patterns to memory banks. A Cpac configuration is *single-context* and remains static during a computation, which means that no operations or connections are changed. Figure 1 shows a 4-way add-compare-select (ACS) datapath, which is used in Viterbi decoders to evaluate path metric, configured onto the processing ele-

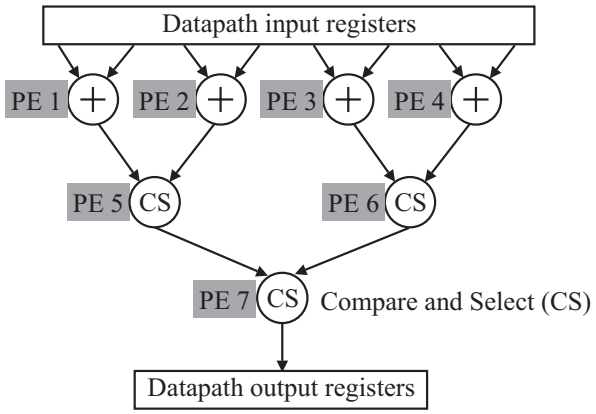


Figure 1: The figure shows how to map a 4-way add-compare-select data-flow graph to the coarse-grained processing elements in CPAC .

ments. There are five system parts used to compose the architecture:

- processing elements,
- memory system with memory banks and address generators,
- reconfigurable interconnect of memory banks and PEs,
- controller with scheduler and configuration manager, and
- host independent FIFO interface.

Figure 2 shows an overview of CPAC connected to an embedded processor by using the host independent first-in first-out (FIFO) interface. This section describes the different system parts and introduces a computational model to interact with a host processor.

3.1 Host Interface

The host interface contains three FIFO blocks, one for instructions or configurations and two for bidirectional data communication, see Figure 2. The wordlengths of data FIFOs are adaptable to the wordlentgh as provided by the host interface, whereas the instructions are 32-bits wide. Communication between CPAC and an external device is synchronized by using FIFOs' *blocking* read and write. In addition, FIFOs are used to form a queue of instructions and data.

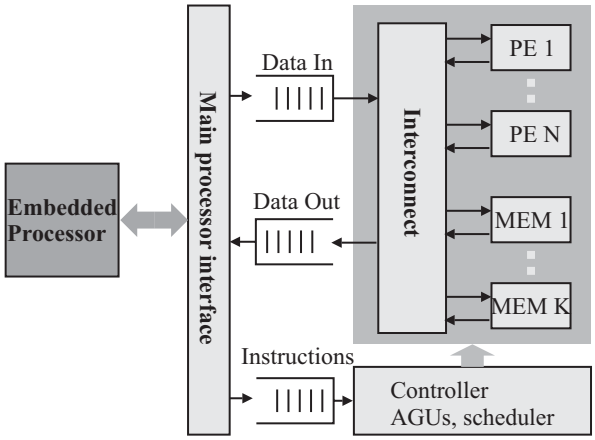


Figure 2: The reconfigurable computing system: processor core and reconfigurable coprocessor

3.2 Processing Elements

To target reconfigurable architectures for the DSP domain, coarse-grained functional blocks have been proposed and proven more efficient in terms of area and speed as compared to fine-grained functional blocks [7]. This is because DSP computations in general are characterized by arithmetic operations on multi-bit words of data. Hence, fine-grained logic blocks, such as 4-input look-up tables, give a large overhead when implementing word-level arithmetic. The processing elements developed for CPAC are customized ALUs that support a set of different operations. In addition, post- and pre-processing steps such as data slicing, scaling, and truncation is included as these operations are commonly used in fixed-point datapaths. To make the RTL implementation flexible, the VHDL *generic* operator was used in order to control implemented features such as supported operations, number of pipeline registers, wordlengths, etcetera. Hence, instantiated PEs can be made a heterogeneous set by configuring these parameters prior to synthesis.

Figure 3 shows an example of the processing element. The stages marked at the top of the figure are controlled through a set of software controlled 32-bit wide configuration registers, which are shown in the lower part of the figure. These registers are written by software in order to route the datapath and configure the ALU operation. Only PEs used in the computation need to be configured. What follows is a brief summary of the pipeline stages of the developed processing elements.

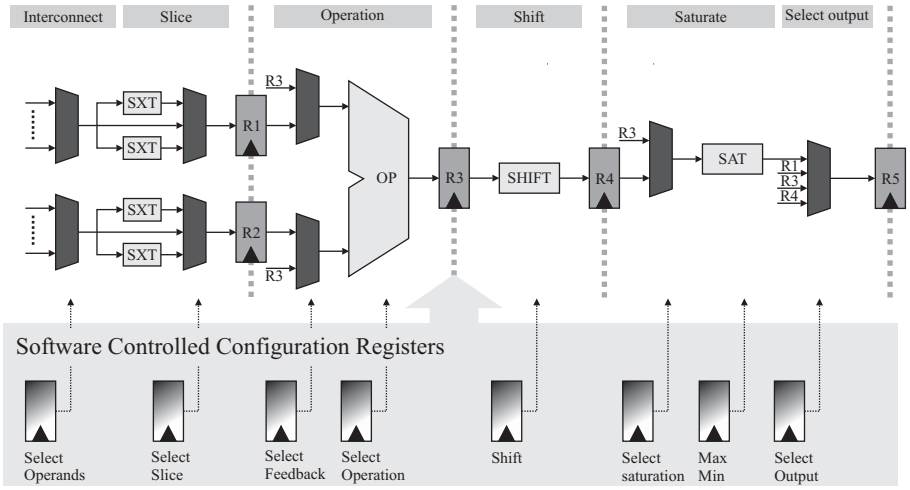


Figure 3: An example of processing element (PE) used as a datapath resource.

Interconnect Stage

The interconnect stage is used to select source operands from other *processing elements*, *memory system*, or *FIFO*. It is implemented with multiplexers and prior to synthesis, a setup is used to select types of interconnect, ranging from cross-bar to partial interconnects. A cross-bar interconnect provides a connection from every processing elements output to every other processing element input. Partial interconnects, such as nearest neighbor, reduce the interconnect cost but also mapping flexibility.

Slice Stage

The slice stage is used to split a word operand into sub-word operands and do sign extension to the operational wordlength. This is used to support wide accesses to memory and split operands to different processing elements which do operations in parallel. In the experimental studies, memories are 32 bits wide and slicing to 16 bits is supported.

Operation Stage

The operation stage supports a set of fundamental arithmetic and logic operations. In particular, basic operators used in standard codecs from ITU were studied in order to find suitable arithmetic operations. The basic operators are

currently used in the standards for G723.1, G729, GSM enhanced full-rate, and adaptive multi-rate speech codecs [84] and use 16 and 32 bits numbers. These operators are sufficient to build computational structures such as radix-2 FFT butterfly, complex multiplication, dual and quad multiply and accumulate, accumulated mean squared error, sum of absolute differences, and basic vector operations. To support accumulate operations with short latency, a feedback is connected to the *operation stage* from the result register R3, found in Figure 3.

Shift Stage

The shift stage is used to scale the result after multiplication, addition, or substraction to avoid overflow. As many processing elements are envisioned to use shifting by a fixed set of numbers $\{1, 15, 16\}$, implementation of the shifter can be selected either as partial shifter, or a barrel shifter that supports all shift operations.

Saturate Stage

The saturation stage is used to saturate the result from the shifter (R4) or from the operation (R3). Saturation values are software controlled through the registers *min* and *max* found in Figure 3.

Select Output Stage

The final stage is used to select results that are propagated to the output register and moved to next processing element, memory system, or FIFO. Data can be moved to the output register directly from any of the pipeline stages.

3.3 Memory System

Integration of CPAC into an embedded system is based on the assumption that requested data reside in the processor cache or registers. With this assumption, the memory hierarchy provided by the host processor is utilized rather than to directly access the main memory. The processor is responsible to load and store data and configurations before the computation starts. This coupling mechanism has been used in similar approaches [80] and is referred to as a tightly coupled coprocessor. The advantage over direct memory access is that cache incoherency is avoided, and that fast specialized coprocessor interfaces can be used to move data back and forth between processor and CPAC.

In kernels that reference data more than once, performance is improved if the CPAC internal memory system is used to store operands and result. First, it relieves the processor from data shuffling. Secondly, it reduces effects of slow interfaces between the processor and CPAC. Third, it allows operands and results to be partitioned and distributed into different memory banks, which

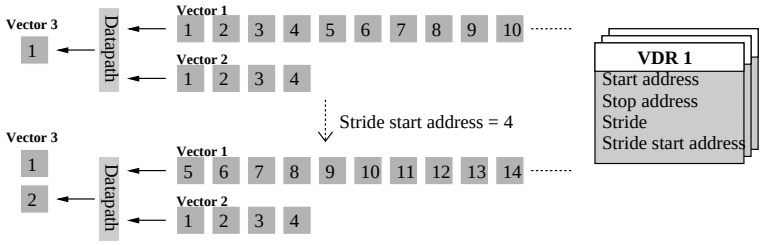


Figure 4: Data stored in internal Memory System is described with vector definition registers (VDRs). The figure shows an example in which *Vector 1* and *Vector 2* are streamed through the datapath in two passes to generate the result *Vector 3*. Within a pass, the *stride* value is used to index the next element. Between each pass, the *stride start address* value is used to index the first element.

parallelizes memory accesses to increase data delivery. These are key parameters to gain performance in kernels where temporal locality in data accesses can be utilized. When CPAC operates from the internal memory system, memory indexing, and data fetches are done in parallel with the actual computation. Consequently, loop-control, memory accesses, and datapath operations found in the DSP kernels are effectively accelerated.

The memory system is a structural design that instantiates a number of single-port SRAM memory blocks. Number of memory blocks, wordlengths, and size of the memory blocks are generic parameters. The memory wordlength is an important parameter to increase data delivery in DSP algorithms that have linear access patterns. For example, a 64-bit wide memory provides the datapath PEs with four 16 bits linearly accessed operands in a single cycle. In our proposed architecture, input register are used to *align*, *buffer*, and *sequence* data during such parallel memory fetches.

Data operands and results stored in the internal memory system are described using vector definition registers (VDRs) attached to each memory bank. The size and shape of vectors are described using *start address, stop address, stride*, and *stride start address*. During computation, these values are used by address generation units (AGUs) to compute next address and determine end of computation. Figure 4 shows how the first two elements in *Vector 3* are generated by streaming *Vector 1* and *Vector 2* through the datapath. For our current investigation, this way of describing data have been sufficient. However, it is clear that the concept would benefit from a more general approach of address generation, such as described in [89], where distributed address generators and a loop unit are proposed. This architecture supports a generalized

set of loop constructs and array indexing techniques.

3.4 Computational Model

The computational model is the mechanism by which operands and results are moved to and from the reconfigurable datapath. Figure 5 shows a conceptual model of sources and destinations during computation. Different computational models are used in order to provide support for different acceleration objects from inner-loop blocks, to vector operations, to nested loop constructs. The proposed computational models are:

M1 Load and store of coprocessor datapath registers. This is used when the acceleration object is a basic block of functionality which appears inside a loop-construct and the loop-control and memory operations are handled by the main processor.
Example C-code:

```
result = (a - b) * c;
```

M2 Stream operation where data operands are taken from the data input FIFO and/or the memory system and the result is stored in memory system, or redirected to output FIFO. This has the advantage to hide communication time when vector operations are performed.
Example C-code:

```
for ( j = 0; j < 100; j ++ )
    result = (a[j] - b[j]) * c;
```

M3 Load and store of coprocessor memory system. Used when temporal locality could be explored to increase performance in acceleration objects that contains nested loop constructs.
Example C-code:

```
for ( i = 0; i < 100; i ++ ) {
    diff = 0;
    for ( j = i; j < 100; j ++ )
        diff = (a[j] - b[j-i]) * c;
    result[i] = diff;
}
```

The first model (M1) is used if control and memory operations are kept in the processor and only the inner-loop data operations are mapped to the coprocessor, which then works as a functional unit extension. This is essential

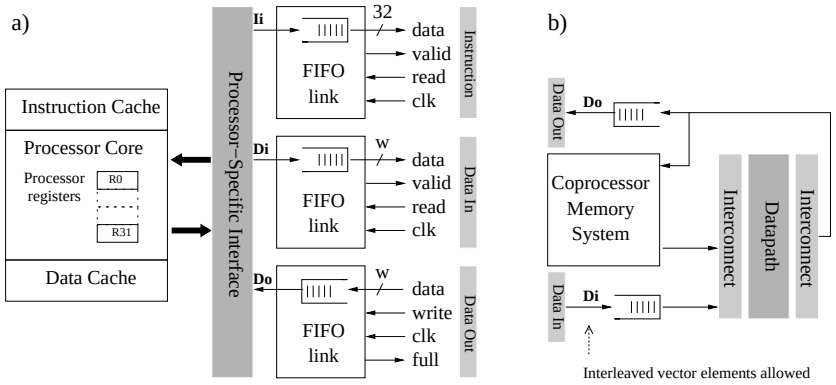


Figure 5: a) Operands and result are communicated from processor registers or from data cache. b) Sources and destinations of operands and results during computation. The configured datapath is fed with operands from the Memory System or directly from the data input FIFO. Result data is stored fed to the memory system or fed to the output FIFO.

in order to utilize the reconfigurable datapath in functions with access patterns unsupported in the coprocessor. Operands are received sequentially from the input FIFO and results sent sequentially to the output FIFO. The configuration controls to which processing element the sequenced data should be fed. If this is not a repetitive pattern, each data operand can be proceeded by an address word used to index a specific processing element.

The second model (M2) is useful when vector operations are accelerated, as it preserves linear access patterns. A vector operation is performed by loading the first vector to the memory system, and then stream the second vector while executing the operation in parallel. This is also beneficial in computations where performance is enhanced if the processor gets access to results on sample- instead of block-basis. For example, the result from a block filter computation is streamed to the output FIFO and then accessed by the processor sample by sample.

In the third model (M3) the processor downloads all data to the coprocessor memory system before execution starts. The processor is responsible for data partitioning over the different memory banks. Data stored in the memory system are described using VDRs used by address generation units (AGU) attached to each memory port. When the VDRs and the datapath have been configured, the processor issues an instruction that starts the execution.

3.5 Configuration

Reduced configuration time is one of the major advantages with a coarse-grained instead of a fine-grained architecture. Thus, even kernels with short execution time may benefit from acceleration. This means that the reconfigurable resources can be shared within tasks as the penalty from swapping kernels in and out is small. For Cpac, with 12 processing elements and 3 memories, configurations are about 100-200 bytes and can be downloaded in less than 1 μs using a 100 MHz system clock. In contrast, to partially configure an FPGA takes several milliseconds. For example, to configure a tenth of a Virtex XC2V6000 FPGA with 260 KB of configuration data requires 4 ms [90].

An example of a Cpac programming scenario is described as:

1. Configure vector definition registers.
2. Download data vectors.
3. Configure processing elements.
4. Start computation.
5. Synchronize by performing a blocking read operation to the result FIFO.

Configurations of processing elements or VDRs are sent to the instruction FIFO in Figure 5a. Configuration words are read out from the FIFO until the *start command* is found. When start command is found, new configurations will be buffered in the FIFO until current context is completed. However, new configurations can be written to hide communication time. In addition to VDR and PE configurations, there are instructions to read and write from the memory system and to set up data streams.

4 System Level Integration

We assume that an embedded processor is present in the system where the coprocessor should be integrated [7, 46, 79]. The embedded processor runs the control-oriented part of the application, whereas time consuming dataflow computations are mapped to the reconfigurable coprocessor. In particular, the embedded processor is used to *configure*, *communicate data* and *schedule tasks* in the coprocessor. This section discusses integration of Cpac into a embedded processor system using a general purpose bus, or a dedicated coprocessor interface. In addition we describe how acceleration objects are identified and mapped to Cpac.

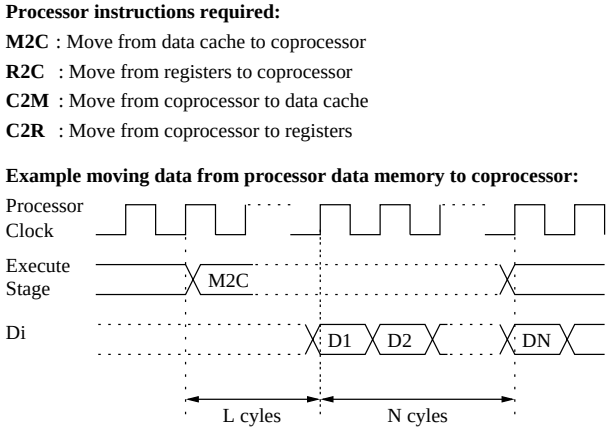


Figure 6: Specialized coprocessor instructions are used to move data and configurations back and forth from the reconfigurable architecture

4.1 Processor Interface

Figure 5a) shows the CPAC main interface with three FIFOs and a bus wrapper. These FIFOs are used to communicate data, instructions and configurations. Wordlengths in the data FIFOs are adapted to the provided host processor interface. Wordlengths of 32 or 64 bits are supported in the studied host processors Virtex-II Pro PowerPC 405, Virtex-4 FX PowerPC 405, ARM9, and ARM11 [8, 91]. The instruction FIFO has a fixed 32-bit wide wordlength in order to allow integration of configuration data into the processor instruction flow.

The interface provides synchronization primitives for parallel tasks executed on the processor and coprocessor through blocking read and blocking write. Hence, when an execution has been programmed in CPAC, the processor continues with its work and synchronizes with a blocking read operation when the result from CPAC is needed.

CPAC is attached to the processor subsystem using a general purpose bus (AMBA, OPB, DSOCM) or a dedicated coprocessor interface. Dedicated coprocessor interfaces are found in ARM9, ARM11 and PowerPC 405 Virtex4 embedded processors. A dedicated coprocessor interface provides a higher data rate between processor data cache and the coprocessor, as well as special purpose instructions to handle communication with the coprocessor. Figure 6 lists a set of typical instructions and shows the timing for an instruction which moves data from data cache to the coprocessor interface. When requested data resides in the cache, these instructions can move a burst of linearly spaced data

in $N + L$ cycles, where N is the number of words and L is the initial transfer latency. When CPAC is integrated as a bus component, software drivers with similar functionality as the instructions found in Figure 6, are developed.

4.2 Mapping Functionality

An application is initiated as a software implementation and functionality is migrated to CPAC based on analysis of software profiling metrics. After software profiling, functions are explored and mapped to CPAC in the order they affect the overall speedup according to Amdahl's law [27]. We refer to these functions as acceleration objects. An acceleration object is a computation that use a single loop, nested loops, or a block of operations in an inner loop. The loops describe repetitive computations with operands and result indexed by the loop variables commonly found in DSP applications.

An acceleration objects is made into a separate function and then transformed into CPAC configurations and communication routines by using the computational models M1, M2, or M3. In some cases this means that larger loops are split into several smaller loops. Conversion from C-code to coprocessor routines is currently carried out in a pre-compilation step by replacing code recognized as a macro with hand-optimized coprocessor configurations using the CPAC programming interface. A typical acceleration object is depicted below and it is taken from the G.723.1 codec:

```
for ( i = 0; i < SubFrLen; i ++ ) {
    Acc0 = (Word32) 0 ;
    for ( j = i; j < SubFrLen; j ++ )
        Acc0 = L_mac(Acc0,Tv[j],Imr[j-i]);
    ErrBlk[i] = L_shl(Acc0, Exp);
}
```

The operations, *L_mac* and *L_shl*, used in the inner and outer *for*-loops are basic fixed-point operators. In the example, operands are referenced more than once during computation, which means that performance is enhanced if computational model M3 is used. This means that all of this functionality are moved from processor to coprocessor. If there is other useful work for the processor after the coprocessor has been configured, the processor may continue execution. If the processor operates on sample basis, output is redirected to FIFO. This is part of the designer's decisions when programming the coprocessor.

Figure 7 shows an example of how this functionality can be mapped to the coprocessor resources. In this case it is implemented to calculate two taps and two outputs in parallel, with a throughput of four multiply and accumulate (MAC) operations per cycle. The operand and result vectors have been

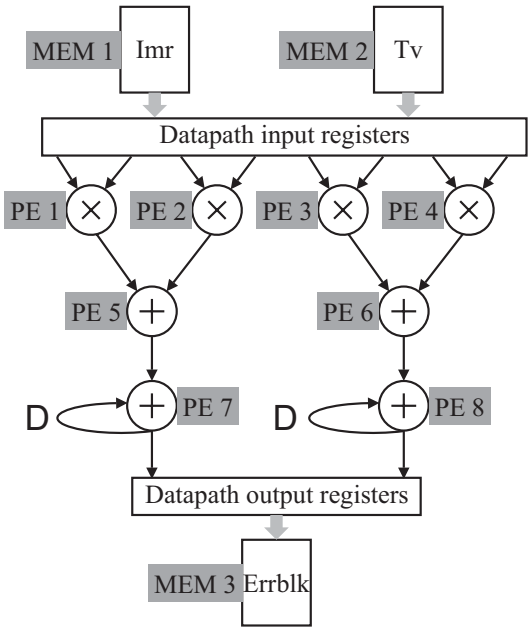


Figure 7: An example of resource allocation and mapping to implement a quad multiply and accumulate.

partitioned into different 32-bit wide memories in order to provide the datapath with operands at datapath throughput. To allow higher throughput, memory wordlength and number of PEs are increased. The example does not show the detailed implementation of the arithmetics used. However, each PE is programmed to comply with the arithmetic used in the reference code.

5 Experiments and Results

This section shows performance results from an FPGA implementation of an embedded system with Cpac. In particular, a version of Cpac that targets speech processing were synthesized and the G.723.1 speech codec accelerated. We also show how the processor interface affects the performance as size of acceleration objects are varied from vector operations to DSP kernels.

5.1 Experimental setup

For the sake of rapid prototyping, a Virtex-II Pro development system was used. The Virtex-II Pro contains two on-chip hardwired PowerPC (PPC) 405

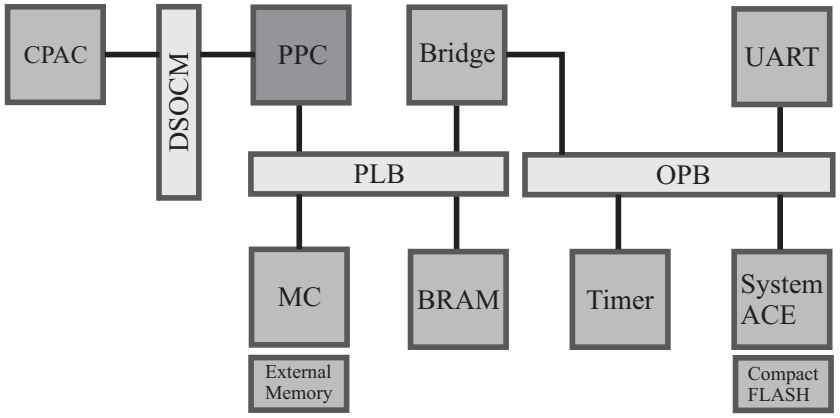


Figure 8: Embedded System with CPAC, PowerPC (PPC), data-side on-chip memory (DSOCM) bus, processor local bus (PLB), on-chip peripheral bus (OPB), block random access memory (BRAM), memory controller (MC), PLB to OPB Bridge, Timer, universal asynchronous receiver/transmitter (UART), and system advanced configuration environment (SystemACE).

processors. Figure 8 shows the embedded system used in the case study. The coprocessor was attached to the PowerPC using the data-side on-chip memory bus (DSOCM), since the processor had no dedicated coprocessor interface. Between processor cache and coprocessor, the DSCOM bus provides a maximum throughput of 32 bits every third clock cycle.

Figure 9 shows the CPAC configuration used in the experiments. The setup consists of three 256x32-bit single-ported SRAMs, six 32-bit ALU PEs, four 32-bit ALU+MULT PEs, two 40-bit ALU PEs, and three 16x32-bit FIFOs. The reconfigurable array is organized as 3×4 processing elements with nearest neighbor interconnect. The memory sizes were selected to target speech processing kernels that operate on frames of data with up to 256 samples. Supported operations and datapath wordlengths were also adapted to speech processing domain by adding support for 16-bit and 32-bit basic operations.

The PowerPC has 16 kB separate instruction and data cache and SDRAM, used as main memory, attached to the processor local bus (PLB). The profiling measurements were performed on target hardware using the GNU statistical profiler *gprof*, and a custom made profiler used to measure execution time of loop constructs. Hardware synthesis and software compilation were carried using *XST* and *gcc*, respectively [92]. The CPAC implementation has a maximum

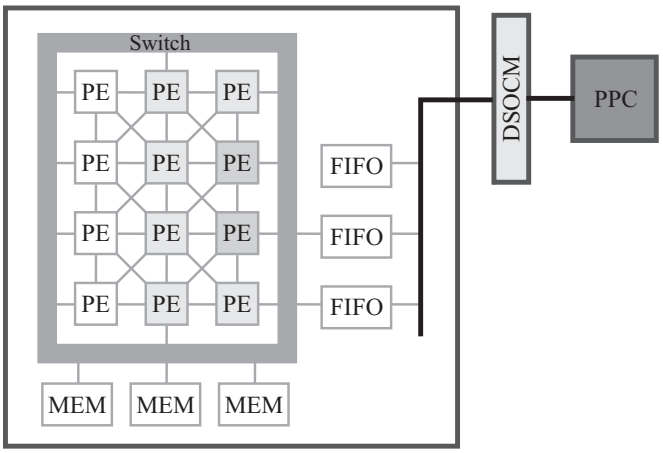


Figure 9: Resource Configuration of CPAC with processing elements organized as a 3×4 heterogeneous structure with nearest neighbor interconnect.

clock frequency of 135 MHz when mapped to the FPGA and utilizes 7500 slices. The interconnect system is implemented with multiplexers and utilize 2000 of these slices.

Test vectors were accessed reading and writing to compact flash using system advanced configuration environment (SystemACE) peripheral, as seen in Figure 8. Communication with a desktop computer to control simulation and receive printouts was handled through a UART peripheral. The timer in Figure 8 is used to measure execution time using *gprof* and the loop-level profiler.

5.2 **Accelerating DSP kernels**

We found that 86% of the execution time for the G723.1 6.3 kbps speech codec was spent in 23 acceleration objects. The kernels found in these acceleration objects were block filtering, autocorrelation, cross-correlation, etcetera. This section shows the speedup achieved when mapping these kernel to CPAC.

Figure 10 shows the speedup for a selected set of fixed-point *vector operations* and *DSP kernels*. The *DSP kernels* are marked with *) on the horizontal axes of Figure 10, whereas the other benchmarks are *vector operations*. *DSP kernels* use the memory system to access operands and results using computational model M3, whereas vector operations use computational model M2. When *DSP kernels* are executed, all three memories in Figure 9 are used. The number at the top of each bar specifies the number of output data elements

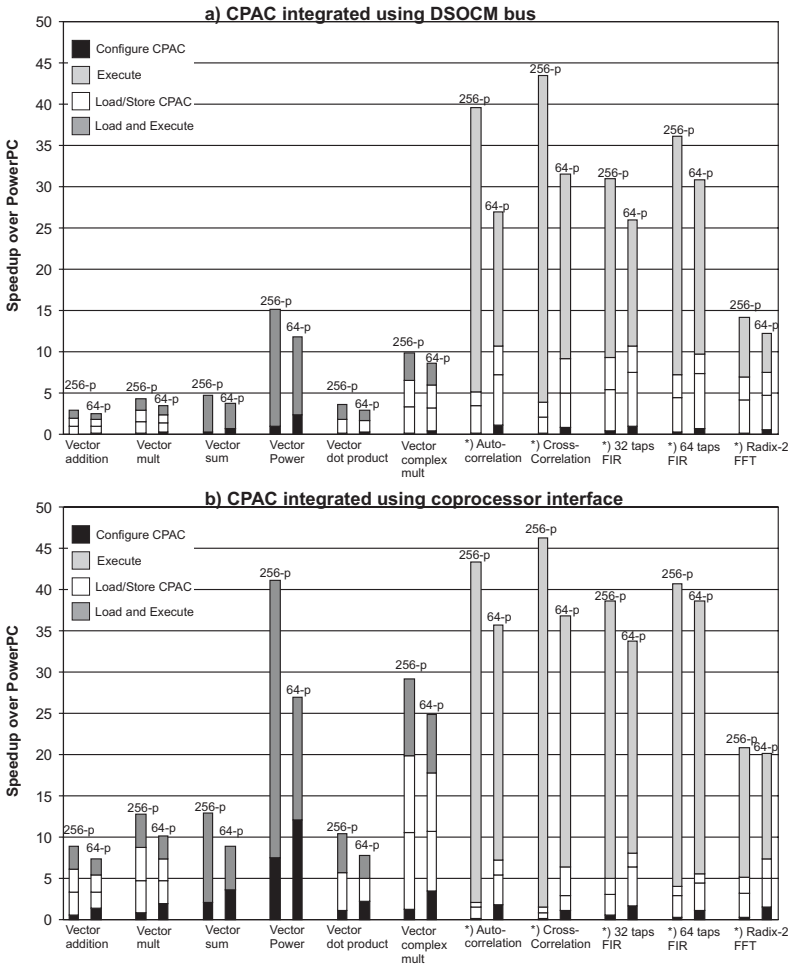


Figure 10: Speedup over PowerPC for vector operations and DSP kernels. a) Measured for the DSCOM bus. b) Estimated for a dedicated coprocessor interface.

for the FIR filter kernels and the number of input elements in the remaining kernels. Each bar shows the distribution of time between:

- configuration of CPAC,
- load and store,
- execute, and
- load and execute in parallel.

The speedups were measured for the DSOCM bus, Figure 10a), and estimated for a dedicated coprocessor interface, Figure 10b). The speedups vary between 2 and 46. The speedup gained over a processor implementation is due to several factors. First, inputs are 16-bit wide and the output is either two 16-bit numbers or a single 32-bit number. Hence, several result elements are computed in parallel, which is similar to techniques used in SIMD architectures. Secondly, the computations require several instructions to be executed in a serial fashion on the processor, but are implemented as pipelined operations in the coprocessor. Finally, overhead caused by *loop-control* and *array indexing* is eliminated when accelerating *DSP kernels*.

The main part of the time in the *DSP kernels* is spent in the execution stage in which the processor is free to do other useful work. If speedups for *DSP kernels* in Figure 10a) and 10b) are compared, it is seen that there are only small differences between a fast coprocessor interface and a 3 times slower interface. Consequently, when CPAC operates from the internal memory system slow speed interfaces are not critical. The datapath operations configured onto CPAC for the *DSP kernels* are a radix-2 butterfly for the FFT kernel and a quad multiply and accumulate (MAC) for the other four kernels. The quad MAC computes 2 accumulated outputs and 2 taps in parallel. Hence, it provides a throughput of 4 MACs per cycle. During execution, datapath latency is effectively hidden when filter computations are performed. It is because a *new* vector dot product is started as soon as all operands from *previous* vector dot product have been fetched from memory. This feature is not utilized during the FFT computation since the result from the previous pass is needed in the current pass.

Speedups for *vector operations* presented in Figure 10 are completely limited by the transfer rate between cache memory and CPAC, as described in Section 3.4. This is because each operand is only referenced once. Hence, changing from the DSOCM interface to a 32-bit dedicated coprocessor interface directly affects the speedup. The difference in transfer rate between the presented interfaces is about a factor 3. It is clear *vector operations* would benefit from a 64-bit or 128-bit coprocessor interface in which a complete cache line is transferred

Table 1: Average and worst case per frame clock cycle count for G.723.1 encoders executing on Virtex-II Pro PowerPC

Encoder Rate	Average	Worst case
6.3 kbps	$10.5 \cdot 10^6$	$11.6 \cdot 10^6$
5.3 kbps	$8.63 \cdot 10^6$	$9.28 \cdot 10^6$

in a single cycle. The difference in speedup for the different vector lengths is caused by overhead due to function call, reconfiguration time, and latency in interface and datapath.

If we compare the speedup for the 64-point *vector dot product* with the 64 taps *block filtering*, it is clear that operating from memory system enhances performance. If *block filtering* is performed by accelerating only of the *vector dot product*, the speed of the interface limits the overall speedup. When CPAC operates from the coprocessor memory system, different performance points may be explored by changing memory organization and datapath resources.

5.3 Accelerating G.723.1

The low bandwidth required to transfer speech signals has made the G.723.1 codec popular in voice over internet protocol. It encodes speech or other audio signals in frames using linear predictive analysis-by-synthesis coding. The encoder operates on 30 ms frames and encodes them into 10 or 12 code-words for the 5.3 kbps and 6.3 kbps channels, respectively. The pitch period and a differential value are sent to the decoder. When a frame is encoded, signal processing kernels is applied to the 30 ms frame and on subframes that are 7.5 ms and 15 ms. A 30 ms frame consist of 240 16-bit PCM samples. The fixed-point implementation found in [81] was used in experiments presented in this section.

Table 1 shows the average and worst case per frame encoding clock cycle count when the codec is executed on the PowerPC without any acceleration in CPAC. In order to perform real-time mixed-rate encoding, a minimum clock frequency of 11.6/0.03 = 390 MHz is required. Decoding only consumes a fraction of processing time compared to the encoding, and requires an average of $0.6 \cdot 10^6$ clock cycles per frame. Input test vectors used to generate Table 1 are `DTX63.TIN` and `DTX53MIX.TIN`, which contain 864 and 120 frames, respectively.

Figure 11 shows the profile graphs for the 6.3 kbps and 5.3 kbps encoder. The profile graph shows how many percent of the execution time is spent in each function. It also shows how many percent of the execution time is

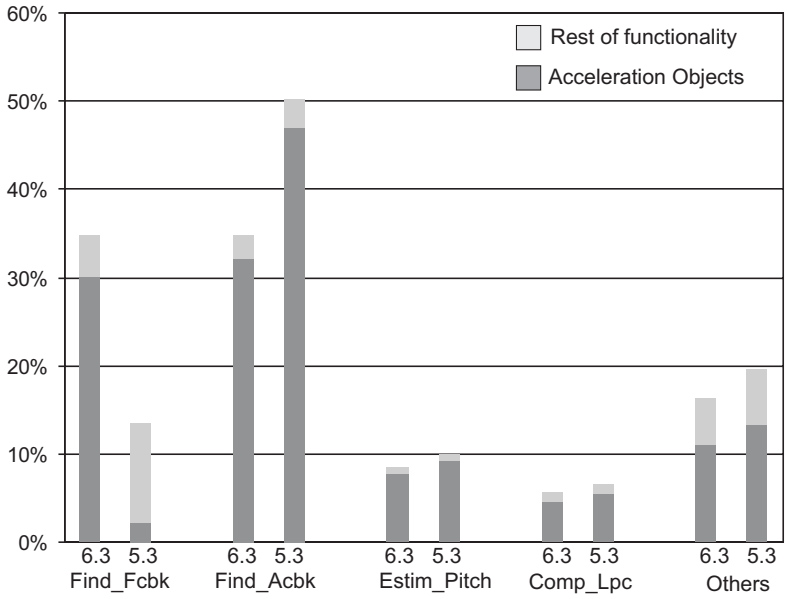


Figure 11: Profile graph for the 6.3 and 5.3 kbps encoder when mapped to the target architecture without accelerator

spent in functionality that are targeted for acceleration Cpac. The acceleration objects constitute in average 86% and 77% of the encoding time for the 6.3 kbps and 5.3 kbps, respectively. The acceleration objects isolated are filtering, autocorrelation, cross-correla-tion, and vector operations. Among the 18000 lines of C-code shipped with the G.723.1 speech codec, the acceleration objects are about 400 lines, or 23 functions. This indicates the complexity involved to convert this part of the code into library routines for the coprocessor. Some functionality recurrently appears in the source code and can be recognized as a macro and substituted with the proper coprocessor routine in a pre-compilation step. The effort to find and map this part of the code into Cpac binaries is estimated to two weeks.

The generated version of the coprocessor provides an overall speedup of 5.9 and 4.0 for the 6.3 and the 5.3 kbps encoder, respectively. When Cpac is utilized, 6% of the time is spent in configuration and 26% in data communication. Consequently, the processor is free to do other work during 68% of the time. The fast reconfiguration time is mainly due to the coarse-grained processing elements and the data descriptor programming technique accomplished using

Table 2: Comparison of average per frame clock cycle count for different implementations of the G.723.1 encoders.

Label	Description	6.3 kbps	5.3 kbps
Proposed	PowerPC and Cpac	$1.8 \cdot 10^6$	$2.2 \cdot 10^6$
CP-1	MIPS R3000 and coprocessor [82]	$3.3 \cdot 10^6$	$3.5 \cdot 10^6$
CP-2	Simplescalar and coprocessor [83]	$\geq 4.9 \cdot 10^6$	$\geq 2.1 \cdot 10^6$
DSP-1	TMS320C54x [86]	NA	$2.1 \cdot 10^6$
DSP-2	TMS320C62x [87]	$0.74 \cdot 10^6$	$0.28 \cdot 10^6$

VDRs. Implementations based on fine-grained blocks and without programming model would significantly increase the time spent in the configuration mode. The data communication time is affected only by the rate provided by the host interface and many processors have interfaces that reduce data communication time significantly.

Table 2 shows a performance comparison between the proposed architecture, two coprocessor approaches, and two hand-optimized implementations on Texas Instruments digital signal processors (DSP). The table shows the *number of clock cycles* required to encode one frame. It is assumed that these architectures may operate at the same clock frequency if proper pipelining techniques are applied to each architecture. Hence, the number of clock cycles gives a fair comparison of the relative performance.

CP-1 accelerates a set of manually chosen single-loops and CP-2 accelerates scalar fixed-point operations. Hence, CP-1 provides acceleration only for this specific coder, whereas CP-2 can be reused in the application domain. The performance reported for CP-2 is in number of *instructions per frame* and hence we use *greater or equal* to the reported number of clock cycles, as it is assumed that cycles per instruction (CPI) CPI ≥ 1. DSP-1 shows a hand-optimized implementation on a *power efficient* fixed-point single-issue DSP. DSP-2 shows a hand-optimized implementation on a *high performance* fixed-point DSP that has eight functional units, including two multipliers and six arithmetic units. The time required to derive the DSP-1 implementation is reported to 2.5 man-month.

Compared to implementations CP-1 and CP-2, the proposed architecture has better performance if the reported instructions per frame is scaled with a realistic CPI for CP-2. Hence, a reconfigurable coprocessor can provide better performance than customized coprocessors. In addition, a reconfigurable coprocessor has a wider application space and can be scaled or configured to

adopt new application domains. This is a key property for embedded platforms that target multimedia and general purpose computing.

As seen in Table 2, best performance is achieved with the high performance DSP. To be able to deliver the same performance, CPAC needs to target functionality that consumes at least 93% of the execution time on the PowerPC. As our current implementation with 23 acceleration objects that are mainly DSP kernels, we believe that the reported 86% can be increased. Either by hand-optimizing code for PowerPC or by mapping more functionality to CPAC. However, as mapping is currently a manual step, the design effort is high. Design tools that automate isolation and mapping of acceleration objects are required to overcome this limitation. Such tools are currently researched to be used in reconfigurable architectures [25] and in customization of instruction set architectures [23].

6 Conclusions

This part presented a coarse-grained reconfigurable coprocessor and evaluated its performance for vector operations, DSP kernels, and the G.723.1 speech codec. The merits of reconfigurable implementations are the ability to combine flexibility and performance. This is accomplished by a heterogeneous set of hardware units such as memories, address generators, processing elements, programmable interconnect, and a flexible computational model. A case study was carried out to evaluate the performance of our proposed architecture for speech processing applications. In the case study, an embedded system was implemented on a Virtex-II Pro and CPAC was attached to a hardwired PowerPC processor. Standard tools for compilation, synthesis and profiling were used in our design flow. The case study demonstrated that up to 86% of the execution time for the G.723.1 speech codec is spent in data driven kernels that can be accelerated. We showed that these kernels can be accelerated 2 to 41 times when mapped to the reconfigurable coprocessor. For a specific implementation of our coprocessor, the overall clock cycle count for the 6.3 kbps G.723.1 encoder was reduced by 83%. The implementation occupies 7500 slices and operates up to 135 MHz on a Virtex II Pro.

A comparison with two application-specific coprocessors that target the G.723.1 standard showed that the proposed architecture has better performance. In addition, the proposed reconfigurable architecture provides a more general solution to acceleration as compared to application-specific coprocessors. As automatic mapping tools mature more functionality may be accelerated, because these tools automatically find functionality and reduce time spent in manual programming. These tools should use the provided computational models to find and map basic blocks, vector operations, and kernels.

Part II

System Level Exploration of Reconfigurable Computing Platforms

Abstract

System level evaluation of reconfigurable platforms is necessary to predict performance, and find and cure bottlenecks before a chip is fabricated. To evaluate a complete system at register transfer level becomes unattractive because time spent in programming and simulation is too long. Therefore, Electronic system level design has been a driving factor for modeling languages that support multiple abstraction levels. Standard SystemC is one such language that supports modeling abstractions from register transfer level to transaction level. Although abstraction is important, another key challenge is interactive and automated approaches to architectural exploration. This part presents a SystemC environment with interactive control (SCENIC), which addresses several aspects necessary for efficient design exploration. Furthermore, it presents a set of flexible transaction level models, which allows a complete reconfigurable computing platform to be explored with SCENIC. Developed models include a generic coarse-grained reconfigurable architecture and a flexible instruction set simulator.

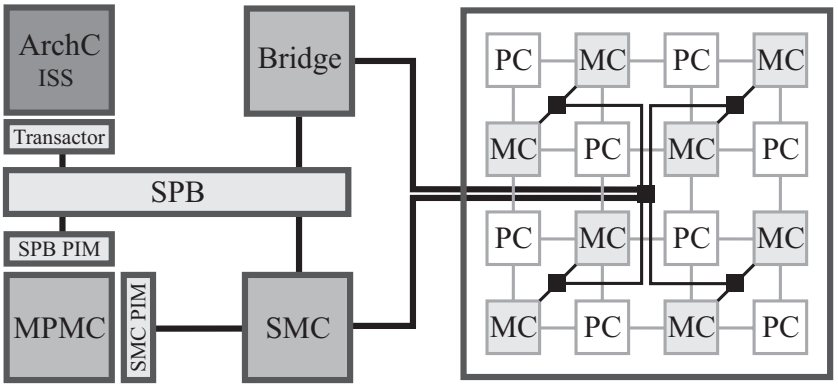
Based on: Henrik Svensson, Thomas Lenart, and Viktor Öwall "System Level Modeling of a Reconfigurable Computing Platform," in preparation for *ACM Transactions on Reconfigurable Technology and Systems*.

1 Introduction

The last few years have seen numerous reconfigurable computing platforms that combine a coarse-grained dynamically reconfigurable architecture with one or more processors on a single chip [7, 38, 64]. One major challenge in this maturing field is to allow quantitative analysis of these platforms, so that their design parameters can be understood and adjusted. This requires systematic methods that allow architectural refinement based on profiling data gathered from different components in the system. Based on gathered performance metric, architectural modifications can be suggested and explored. To perform architectural exploration in simulations at register transfer (RT) level becomes unattractive as small architectural changes require a considerable design effort. Furthermore, simulation speed offered at RT-level is too slow to allow efficient software development or architectural exploration. These limitations are addressed when developing a *virtual platform*. A virtual platform is used to provide early estimates of platform performance, or used for software development prior to RT-level, or existing silicon.

When reconfigurable computing platforms are developed, a virtual platform is used to perform micro-architecture exploration of the reconfigurable architecture, either as a stand alone component, or integrated into the embedded system. As a virtual platform has many different purposes, with varying degrees of temporal and structural accuracy, a modeling language that supports multiple abstraction levels is best suited. Standard SystemC [93] is one such language that supports modeling abstractions from register transfer to transaction level [66–70]. With abstraction levels comes the ability to handle increased complexity, allow gradual refinement, and provide a system perspective early in the design flow. In addition, abstract models significantly increases simulation speed so that complete systems can be efficiently explored with a simulation-based methodology.

This part presents a set of flexible models intended for architectural exploration of reconfigurable computing platforms. In addition to the coarse-grained reconfigurable architecture itself, there are models for instruction set simulators, system buses, and external memories. Abstraction is addressed by using the OSCI TLM standard [66], which provides a communication mechanism based on function calls. Although abstraction is important, another key challenge is *interactive* and *automated* approaches to architectural exploration. Interactivity here refers to the ability to *control* and *observe* components during simulation to gather profiling data for performance analysis, tune system parameters, or debug the system. Automation refers to the ability to explore multiple systems in batch mode and systematically change system parameters and gather performance metric. These concerns are addressed with our



(a) (b)

Figure 1: A set of flexible transaction level models have been developed for the SCENIC environment, so that an reconfigurable platform can be evaluated and tuned. The models include a reconfigurable array of processing and memory cells, a flexible instruction set simulator, a multiport memory controller (MPMC), a bus (SPB), and a stream-memory controller (SMC).

SystemC environment with interactive control (SCENIC). SCENIC is based on OSCI SystemC 2.2 [94] and runs under Windows, Cygwin and Linux operating systems.

This part has the following outline. Section 2 presents the SCENIC library models that have been developed to allow exploration of reconfigurable computing platforms. Section 3 describes how the SCENIC exploration environment is used to construct modules and systems, and support interactive simulations. Section 4 discusses different aspects of tool support for exploration, debugging, and trade-offs between simulation performance and accuracy.

2 Scenic Library Components

A set of flexible transaction level models have been developed, so that a virtual reconfigurable platform can be composed, evaluated, and tuned. Each model uses SCENIC features, so that performance metric can be collected, and configuration parameters tuned during an interactive simulation. Figure 1 shows the different components arranged as a reconfigurable computing platform, which

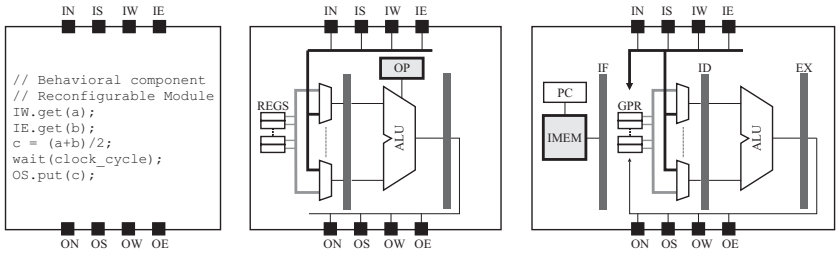


Figure 2: The developed processing cell allows a range of functional blocks to be modeled. Either the processing cell contains a computational kernel described with behavioral code (a), an ALU (b), or a small processor (c). The structural description of processor/ALU has customizable instruction set and SCENIC parameters to control execution time for instructions, pipeline latencies, etcetera.

has the reconfigurable architecture connected to the system bus. This section describes each model in more detail.

2.1 Reconfigurable Architecture

The reconfigurable architecture is organized as an $W \times H$ array of interconnected resource cells (RCs). Figure 1(b) shows a 4×4 array with two types of RCs, processing cells (PCs) and memory cells (MCs). Any behavioral dataflow component may be encapsulated in a PC, as seen in Figure 2(a), which shows a processing cell reading two ports, calculating average, and output the result. Behavioral models are useful for rapid development and evaluation of new functionality. To allow such behavioral code to be swapped in and out, which is necessary to simulate dynamic reconfiguration, a *reconfigurable module class* is provided. Any module of that type can be repeatedly inserted and removed from a specific PC during simulation. Another mechanism to model reconfigurability is suggested in [95], but it is based on modifications to the SystemC core library.

In addition to *behaviorial* descriptions of processing cells, a *structural* model has been developed. The structural PC allows both *ALU arrays* and *processor arrays* to be constructed and modeled. It contains a computational kernel, built either as processor, or ALU core. The ALU/processor model, seen in Figure 2(a) and (b), has a customizable instruction set and provides an extensive set of exploration parameters in order to evaluate and tune its performance.

Memory cells are used to provide intermediate storage during computations

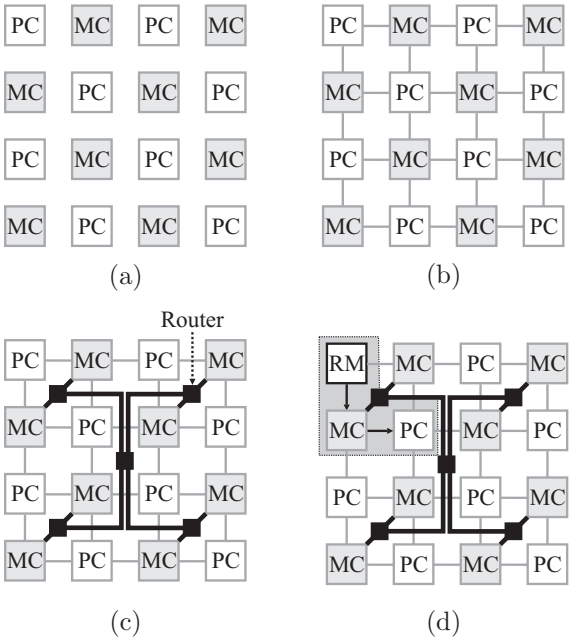


Figure 3: A reconfigurable architecture are constructed from a set of generic modules. First it is decided how the cells should be placed (a). Then the amount of local communication is decided (b). At the top of the local interconnect, a hierarchical routing network is added (c). When the system have been elaborated, configurations can be downloaded to control the operation and *reconfigurable modules* (RMs) can be swapped in and out from a specific PC (d), in which grey area indicate configured resources.

and they may be configured to operate as random access memory (RAM), or first-in first-out (FIFO). They are also used as inter-processor communication buffers as they generate a back-pressure signal to a sending cell if the buffer is full, and a valid signal to a receiving cell if there is available data. Hence, processing is self-synchronizing, so that any computation that uses output from another processing cell is stalled until data becomes available, and result will not be transmitted unless there is available buffer space. Each port on a processing cell is connected to a memory cell, which handles communication with other parts of the system. A memory cell can be configured to forward incoming data to any arbitrary memory cell (global communication), or to simply supply neighboring processing cells with the received data (local communica-

tion). The memory cell can be customized with how many banks it should contain, the storage capacity for each bank, and latency when accessing it.

The interconnect topology is a hybrid approach with dedicated local communication and a global hierarchical network. Resource Cells have a number of I/O ports from all four sides, which define *local communication* bandwidth capacity. In addition, RCs have an optional *uplink* and *downlink* port to connect to the *global network*. *Local communication* uses dedicated links to connect neighboring resource cells. The *global network* connects any two cells with single-path hierarchical routing, in which transfers are dynamically routed to the destination using an identification field in the transfer. Routers are customized with link and routing capacity, and output buffer size.

With these flexible modules for processing, memory, and routing, a reconfigurable architecture is composed. Constructing a reconfigurable architecture from these modules is divided into four phases, as shown in Figure 3. First, the array size, placement, and parameter setup for processing and memory cells are considered Figure 3(a). Secondly, the local interconnect is specified Figure 3(b), were it is used to connect the 4 nearest neighbors. However, higher orders of neighboring connectivity is supported if required. Third, on top of the local interconnect is added a hierarchical routing network Figure 3(c), and the capacity of links and routers are specified. Last, when the system has been constructed, configurations that controls operations in processing cells, memory cells, and the hybrid network are downloaded Figure 3(d). In this phase, *reconfigurable modules* (RM) can be swapped in and out from processing cells, to allow behavioral descriptions to act as being a processing cell, as was described in Figure 2(a).

2.2 Instruction Set Simulators

Instruction set simulators (ISSs) are one of the most required models in a system simulation. They need to be generic so that a wide range of processors can be modeled end explored and they need to support various degrees of temporal accuracy so that simulation speed and accuracy can be selected to current design activity. ArchC [96–98], is an architectural description language and an ISS generator, which has been modified to the SCENIC environment and used to model different types of processors in a reconfigurable computing platform. ArchC addresses modeling of instruction set simulators covering a wide range of instruction set architectures (ISA) and various abstraction levels. A SystemC model of an ISS is generated from a high-level architecture description language (ADL). Several existing processor such as MIPS-I, PowerPC, SPARC-V8, and Intel 8051 are described using the ArchC ADL and these descriptions are available at the ArchC website [96]. The temporal accuracy in ArchC mod-

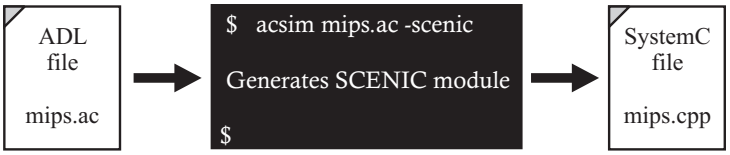


Figure 4: The ArchC source code have been modified, so that any architectural description can be parsed into SystemC file that contains a SCI_MODULE of the instruction set simulator. Objects in the ADL file, such as memories and register, are automatically reflected, and custom commands to control and observe the model are generated.

els are defined in three levels as either *instruction accurate*, *cycle-approximate*, or *cycle-accurate*. The first level lacks time information in order to achieve a high simulation speed, the second level has approximate timing, and the latter has precise timing of the internal behavior but with reduced simulation performance. Cross-compilation to existing models is supported through *gcc* and binary code generation for customized instruction-set architectures supported through the ArchC assembler generator [99].

The ArchC source code has been modified to optionally generate an ISS that utilizes features in the SCENIC core library. Hence, a SCI_MODULE of any ADL file is automatically generated, as shown in Figure 4. It means that internal resources such as memories and registers, declared in the ADL file, are automatically reflected in the SCENIC user interface for introspection. In addition, system parameters used to control different aspects of the simulation can be changed during simulation and customized profiling data observed and collected for visualization and exploration. Customized commands to load applications, start GDB debugging, and read and write to variables are automatically generated and reached in SCENIC environment.

ArchC models allow connection to external components through OSCI TLM ports. Any memory or register declaration in the ADL specification can be exchanged to the corresponding TLM port. This allows cache systems, coprocessors, functional units, buses, and external memories to be connected to allow these components to be explored when profiling an application. ArchC models connects to SCENIC library components, such as RA, SPB, or MPMC, through *transactors* that are used to adapt the ArchC specific TLM *request-response* pairs to the once used in RA, MPMC, or SPB protocol. Transactors are also used to model different coupling mechanisms between the processor and the reconfigurable architecture, as shown in Figure 5. In Figure 5(a), the ISS and the reconfigurable architecture are connected to a bus, and in Figure 5(b) a dedicated link is used so that the RA can be coupled as a coprocessor, or as a

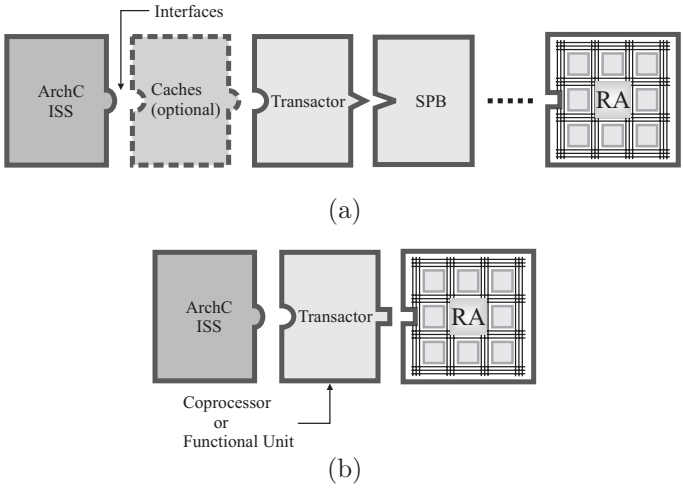


Figure 5: A *transactor* is used to convert the ArchC specific interface to interfaces used in other components, and it allows different coupling mechanisms between processor and reconfigurable architecture to be modeled. The ArchC ISS is connected to the reconfigurable architecture through the bus system (SPB) (a), or through a coprocessor or functional unit connection (b).

functional unit.

2.3 Bus and Memory Models

Generic TLM models of embedded components, such as system processor bus (SPB) and multi-port memory controller (MPMC) have been developed for the SCENIC environment, as shown in Figure 1(a). Models have an extensive set of parameters for tuning the architecture and collect profiling data. The MPMC can be configured to emulate different data rates (SDRAM, DDR, DDR2), operating frequencies, wordlengths, internal parameters for controlling the memory timing, and arbitration policy for the ports. The SPB can be configured to emulate buses with different characteristic such as arbitration cycles, arbitration policy, wordlength, and latency. SCENIC also supports integration of unmodified existing TLM models such as GreenBus [75] or open source protocol (OCP) [100], which supports several abstraction levels of bus based communication with different characteristics of time accuracy and simulation performance.

Processors or any other bus-master may access the reconfigurable archi-

tecture's internal memories, for configuration and data, using the bus *bridge*, shown in Figure 1(a). In addition, a *stream memory controller* (SMC), which is seen Figure 1(a), is used for fast transportation of data between main memory and the reconfigurable array. The SMC have direct memory access through a dedicated port to the multi-port memory controller, and it can connect to any router in the hierarchical routing network. Data that should be read or written using SMC are defined by a *pointer to the first element*, an *access shape*, and an *identification field* which is used to route data to/from a unique resource cell in the array. These parameters are controlled by the processor, which initiate new data streams when the array is reconfigured. The access shape is described with three parameters [46]: *stride*, *span*, and *skip*. Stride defines the distance between addresses. Span defines how many addresses that should be accessed before applying the skip value.

3 SystemC Simulation and Exploration Environment

To address the discussed aspects for efficient design exploration, a SystemC environment with interactive control (SCENIC) has been developed. SCENIC is based on OSCI SystemC 2.2 and extends the functionality with features to define exploration scenarios using a customized scripting environment, construct and configure simulations from extensible markup language (XML), interact with simulation modules during run-time, and the ability to control the simulation kernel using micro-step simulation. SCENIC extends OSCI SystemC *without modifying* the core library and hence integration of 3rd party modules are supported without any modifications. These extensions provide means for more effective simulation and exploration of complex systems, and are briefly described below and presented in more detail in the following sections:

- **Scripting environment** – A customized set of commands to define simulation scenarios, interact with modules, and gather and post-process performance metrics.
- **Module construction** – New user modules are rapidly constructed from existing base classes that contain common building blocks, convenience functions, and macros. Instruction set simulators that utilize features in the SCENIC core library are automatically generated by running a modified version of ArchC [96]. Highly flexible modules of buses and memories have been developed to allow modeling of complete embedded systems.
- **System construction** – User modules are automatically registered during program start-up and stored in a *module library*, from which new

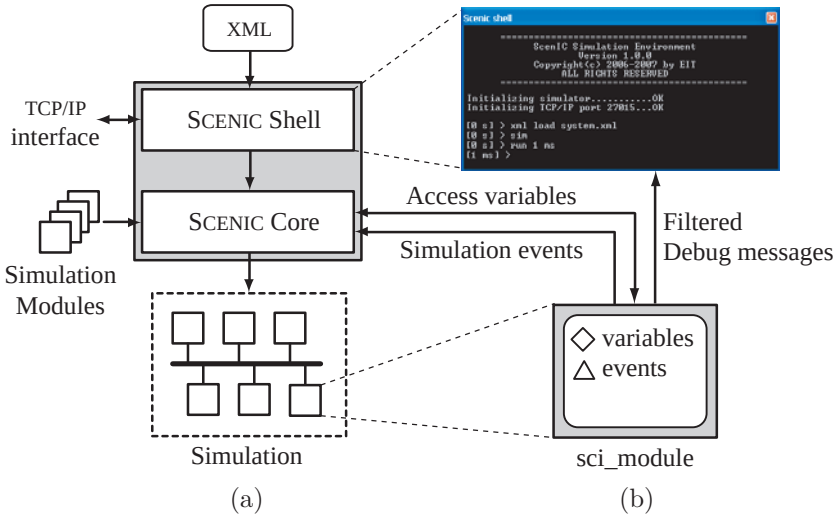


Figure 6: (a) A simulation is created from an XML description format using a library of simulation modules. (b) The modules can communicate with the SCENIC shell to access member variables, notify simulation events, and filter debug messages.

simulation modules are dynamically created and instantiated. Systems are described with XML, which can be parsed from the scripting environment to construct and configure simulations without re-compilation when systems are altered.

- **Interactive simulation** – Users can interactively start and stop the simulation using the non-blocking commands RUN and STOP. Internal data structures and module hierarchy are accessible from the scripting environment, allowing performance metrics to be observed and exploration parameters controlled. Dynamic simulation events allow simulation modules to notify the simulator of specific conditions on which to observe, reconfigure, or halt the simulation.

Figure 6(a) shows the design flow from XML description to an interactive SystemC simulation model, and the environment is divided into two tightly-coupled parts: the SCENIC *core* and the SCENIC *shell*. The SCENIC core handles the SystemC extensions to interact with simulation modules, and the SCENIC shell handles the user interface and a TCP/IP socket to communicate with external programs.

3.1 Scripting environment

SCENIC handles user interaction from a command line interface, script files, or from a connection to a graphical user interface (GUI) over a TCP/IP socket. It can also be controlled from Matlab to support a more powerful scripting environment, and a library of Matlab functions has been developed to easily connect and communicate with the simulator. The command line interface enables efficient scripting and allows users to setup different scenarios in order to evaluate system architectures.

Commands are executed from the command line interface in the SCENIC shell by typing the command name followed by a parameter list. There are two types of commands: *Built-in commands* and *Module commands*. Built-in commands are implemented in the SCENIC shell and include setting environment variables, evaluate basic arithmetic or binary operations, loop-constructs, handle file I/O, etcetera. Module commands are implemented inside User Modules to allow customized module-specific functionality to be called from the command line. If a hierarchical name of an instantiated SystemC module is given instead of a built-in command, the command is passed to and evaluated by that module. For example, memory modules can implement customized commands to inspect memory contents, and processor modules can implement customized commands to load programs, or inspect performance metric. All simulation modules have a common interface to manipulate data structures from the command line. From the command line, internal variables of modules are viewed, modified, and also periodically logged to capture trends over time. Structural reflection is supported by using the built-in *list* command, which list names and types of modules in the command window.

3.2 Module Construction

SCENIC interacts with simulation modules during simulation-time, implemented by extending the OSCI SystemC *module* class with run-time reflection. To take advantage of the SCENIC extensions, the only required change is to use SCI_MODULE, which is an extension of SC_MODULE that encapsulates functionality to access the simulation modules from the SCENIC shell. However, SCENIC can still simulate standard OSCI SystemC modules without any required modifications to the source code. This is an important aspect since it allows integration and co-simulation with existing modules. The class hierarchy and the extended functionality provided by the SCI_MODULE class is shown in Figure 7. A user module that inherits from SCI_MODULE gets access to macros and convenience functions to register variables that should be controlled and observed during simulation, to extract constructor parameters, to output debug messages, etcetera.

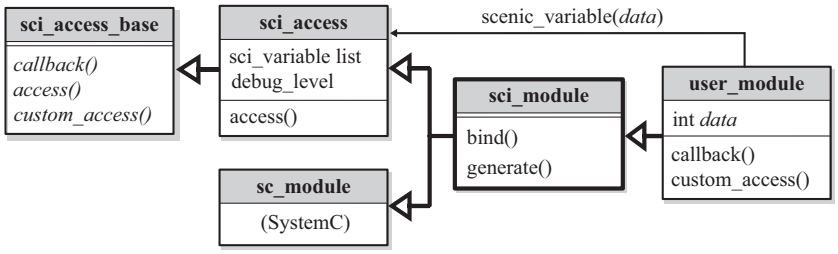


Figure 7: Class hierarchy used to implement a user module. The registered variables are stored in SCI_ACCESS, which is also used for variable registration. The *callback*, *access* and *custom_access* functions are virtual and only implemented where actually needed. Arrows show how derived classes are created from base classes, with arrows pointing to base classes.

3.3 System Construction

Generation of an embedded system simulation in SCENIC is done either using the traditional SystemC procedure for instantiation, configuration and binding of the components using a C++ file, or by using an XML specification. The XML specification contains information about how to configure and connect components and provides information about design hierarchy. When a system is constructed from an XML specification, all components are created dynamically during elaboration. Consequently, source files does not have to be re-compiled when the system is modified.

SCENIC contains a *module library* from which SystemC modules are created. User modules derived from SCI_MODULE are automatically registered during program start-up and stored in the module library. In XML specifications, modules can be instantiated at any location in design hierarchy. Instantiation and configuration are described in XML as

```
<INSTANTIATE>
  <MODULE type="SPARC_V8" name="processor">
    <PARAMETER name="PERIOD" type="TIME" value="2.5 ns"/>
    <PARAMETER name="DCACHE" type="UINT" value="8192"/>
    <PARAMETER name="ICACHE" type="UINT" value="8192"/>
    <PARAMETER name="MULTEN" type="BOOL" value="TRUE"/>
    ...
  </MODULE>
</INSTANTIATE>
```

Internally, instances of a module are created by calling the module library and providing a list of module parameters. All SCI_MODULE uses the same constructor arguments, which are passed to the module using a parameter

vector supporting arbitrary data types. The module extracts parameters of any type during system elaboration to obtain its static configuration. After elaboration, internal configuration variables are controlled from the command line, so that they can be changed during an interactive exploration scenario.

3.4 Interactive simulation

Interactive exploration requires internal variables to be reflected and controlled during run-time, and that simulations may be halted when reaching user-defined observation points. In SCENIC, variable reflection and controllability are supported through *access variables*, and user-defined observation points supported through *dynamic* or *static simulation events*. By introducing micro-step simulation it is possible to interactively pause the simulation to modify or view internal variables, and then continue execution. Simulation is controlled using the non-blocking commands RUN and STOP, or the blocking command RUNB followed by a time. Non-blocking commands are useful for interactive simulations, whereas blocking commands are useful for scripting.

Access Variables

Variables inside a SCI_MODULE are exported to the SCENIC shell as *access variables*. This reflects the variable type, size and value during simulation. Hence, access variables enable dynamic configuration (controllability) and extraction of data during simulation (observability). Figure 8 shows how the user module exports an access variable named *data*, which is created and stored internally and visible from the SCENIC shell. Access variables are implemented using a templated class, SCI_VARIABLE, that encapsulates the member variable and enable operations, for example *get* and *set*, to be evaluated on arbitrary data types using standard C++ `iostreams`. Hence, for user-defined types the stream operators (`<<` and `>>`) must be supported.

Performance data, such as number of bus transactions, can be directly reflected using a SCI_VARIABLE. However, in some cases a mathematical expressions are required when the performance metric is data or time dependent. This requires functional code to evaluate the expression and assign the result to the access variable, and it is supported in SCENIC by using *callbacks*. When the variable associated with a callback is accessed, a callback function is evaluated and assigns a value to the variable. This value is then reflected in the SCENIC shell. The callback functionality may also be used when assigning values to variables that affect functionality or configuration. For example, a variable that represents a memory size, which requires the memory to be resized (reallocated) when the variable changes. The following code shows an example of how to reflect variables and how to implement a custom callback

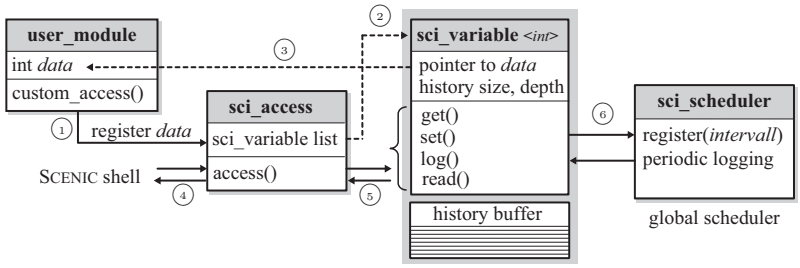


Figure 8: A local variable is registered (1), which creates a SCI_VARIABLE object (2) pointing to the variable (3). It contains features to create a history buffer and to register a periodic logging interval with a global scheduler, to capture variable values at constant intervals (6). The SCENIC shell can access a variable through the SCI_ACCESS object (4), which contains a list of all registered SCI_VARIABLE objects and functions to access those (5).

function:

```
class ScenicModule : public sci_module {                  /* extended module      */
private:
  void*   m_mem;                                          /* pointer to data      */
  short   m_size;                                         /* scalar data type     */
  double  m_average;                                      /* scalar data type     */
  int     m_data[5];                                      /* static vector        */
public:
  ScenicModule(sc_module_name nm, SCI_PARAMS& params) :
    sci_module(nm, params)
  {
    scenic_variable(m_data,    "data");                   /* variable "data"      */
    scenic_callback(m_size,    "size");                   /* callback "size"      */
    scenic_callback(m_average, "average");                /* callback "average"   */
  }

  virtual SCI_RETURN callback(SCI_OP op, string name) {   /* callback function    */
    if (name == "size")
      if (op == WRITE) m_mem = realloc(m_mem, m_size);    /* realloc "size" bytes */
    if (name == "average")
      if (op == READ)  m_average = average(m_data);       /* return average value */
  }
};
```

When the value of the *size* is modified it triggers the callback function to be executed and the memory to be reallocated. The variable *average* is also evaluated on demand, while the *data* vector is constant and reflects the current simulation value. The registered variables are accessed from the SCENIC shell using *set* and *get* commands, as shown in the following command line sequence.

```
[0 ns]> ScenicModule set -var size -value 256
[0 ns]> ScenicModule set -var data -value "4 7 6 3 8"
[0 ns]> ScenicModule get
  data     5x4 [0]     { 4 7 6 3 8 }
  average  1x8 [0]     { 5.6 }
  size     1x2 [0]     { 256 }
```

The value of an access variable can be periodically logged to a history buffer to study trends over time. The periodical time interval and the depth of the history buffer are configured from the SCENIC shell. To save memory, history buffers are created dynamically when variable logging is enabled. Periodical logging requires the variable class to be aware of SystemC simulation time. Normally, this would require each SCI_VARIABLE to inherit from SC_MODULE and contain a SystemC process. This would consume a substantial amount of simulation time due to increased context switching in the SystemC kernel. To reduce context switching, we propose an implementation using a global scheduler that handles logging for all variables in all modules. The global scheduler, shown in Figure 8, improves simulation time since it is only invoked once for each time event, which can log multiple variables in a single context switch.

Simulation Events

Simulation parameters and module configuration affect module functionality and therefore also the required simulation run-time. When running a batch of simulations using different configurations the required simulation run-time for each system is often unknown. A passive solution is to always run the simulation for a longer time than optimally required. This guarantees simulation completeness, but leads to poor simulation performance and problems to correctly evaluate performance data that depends on simulation time.

We propose the use of static and dynamic *simulation events*, which can be configured to execute SCENIC commands when triggered. In this way, the simulation modules can notify the simulator of specific events or conditions on which to observe, reconfigure, or halt the simulation. A simulation event is a SCI_VARIABLE of string type which can be assigned a SCENIC shell command. *Static* simulation events can be inserted in user code to represent special conditions, which trigger execution of a any assigned shell command if satisfied. The following code shows how a static event, named `onComplete`, is created.

```
ScenicModule::ScenicModule() {
  ...
  scenic_event(m_event, "onComplete", "Completed!");
  SC_THREAD(SimThread);
}

void ScenicModule::SimThread() {
  ...
  if (sc_time_stamp() == sc_time(1, SC_US))
    SCI_EVENT_NOTIFY(m_event);            /* notify */
  ...
}
```

The event can be configured from SCENIC to execute a shell command when the event is triggered. The following SCENIC command line sequence shows how the static event, onComplete, is assigned to execute the STOP command when triggered.

```
[0 ns]> ScenicModule set -var onComplete -value "stop"
[0 ns]> run 2 us
  [ScenicModule] Completed!
  [Scenic] Simulation halted @ 1 us.
[1 us]>
```

The module generates a simulation event at time 1 us, which executes the command STOP to halt the simulation.

When a condition that should trigger an event is unknown at compile time, dynamic instead of static simulation events are used. Dynamic simulation events may be registered at run-time from the SCENIC shell to notify the simulator when a boolean condition associated with an *access variable* is satisfied. The following example registers a new dynamic event that executes a SCENIC script called *overflow*:

```
[0 ns]> ScenicModule log -var average -every "1 ns"
[0 ns]> ScenicModule event onOverflow -when "average > 8"
[0 ns]> ScenicModule set -var onOverflow -value "overflow"
[0 ns]> run 10 us
  [4700 ns] [ScenicModule] Executing script "overflow"
[10 us]>
```

When the condition, average > 8, is satisfied, a user-defined script named *overflow* is executed. Since the condition is only evaluated on data in the history buffer, repeated assignments and false values during delta execution is effectively ignored. The internal function calls to create a dynamic simulation event is shown in Figure 9. A *condition* is created from the SCENIC shell, which register itself with the access variable and creates a new *simulation event*. When the access variable is logged, the condition is evaluated and triggers the event if the condition is satisfied.

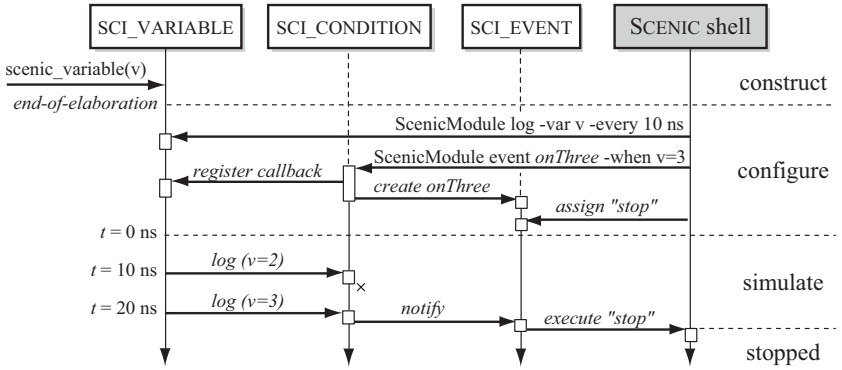


Figure 9: The function call flow to create a dynamic simulation event from the SCENIC shell. An *access variable* is configured to be logged every 10 ns. A condition is created and registered with the *access variable*. During simulation, the condition is evaluated each time logging to history buffer is carried out. When the condition is satisfied, the associated event is triggered, which in this example occurs after a simulation time of 20 ns.

Debug message filtering

The use of debug messages is a natural approach to trace if simulation models execute correctly. However, for large simulations the flow of messages often contains too detailed information and from too many simultaneous sources. To address this problem, we introduce functionality to control the message flow from each simulation module and to trace the source of each message. The SCI_MODULE contains a SCI_VARIABLE to configure the *debug level*, which determines how detailed information each module produce. A group of modules can be configured simultaneously to the same debug level and a child-module initially inherits the debug level from its parent simulation module. The hierarchical SystemC name is used to specify the source of the message. Debug messages are produced by inserting macros in the functional code, and is checked at run-time against the current debug level value.

For inactive messages, the impact on simulation performance is negligible since the debug level is accessed without the overhead of a function call. However, printing debug messages, either to command line or to file, results in a major simulation overhead, and that is why this functionality has been introduced. As an example, experiments with a system showed that the overhead for producing verbose debug messages was 30x the original simulation run-time. Limiting the debug message flow to a specific source of interest and to a certain

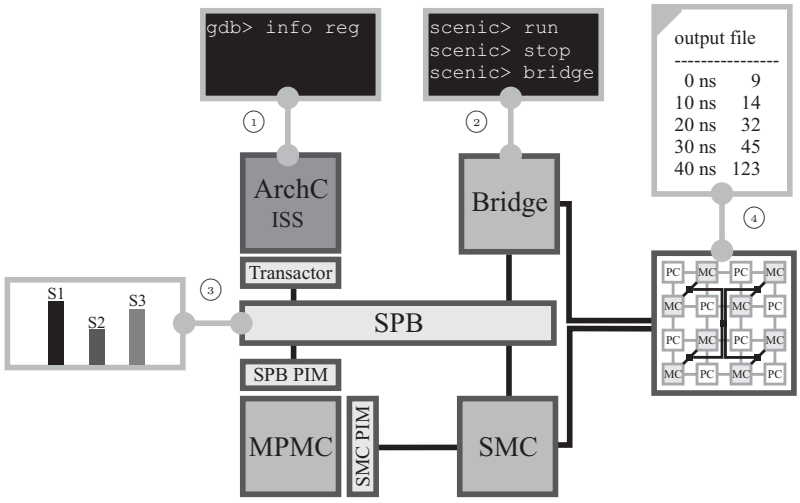


Figure 10: There are several ways to observe the system for debugging or exploration purposes: Using GDB session (1), using the SCENIC shell (2), using an external plot program (3), or using file I/O (4).

type of messages dramatically improve simulation run-time, and is a trade-off between observability and simulation performance.

4 Exploring Reconfigurable Computing Platforms

This section describes how a *virtual platform* with processor, bus, external memory and a reconfigurable architecture is simulated in SCENIC. Different aspects of tool support for exploration, debugging, and simulation performance-accuracy trade-offs are discussed.

4.1 Performance Exploration

When a system is constructed and an application mapped to it, it needs to be evaluated to ensure that its performance is sufficient. Evaluation requires both software and hardware performance metrics to be gathered and analyzed. This section discusses how SCENIC, and our library components support such exploration. It first describes how software profiling is performed, secondly how hardware components can be explored, and last how systematic batch simulations can be generated. Figure 10 shows an overview of different techniques to observe models during simulation.

Software Profiling

A common approach to application profiling is to use a tool, such as *gprof* [101], to analyze data that have been collected during run time. Collected data is used to generate a *flat profile*, that shows the amount of time the program spent executing each function and a *call graph*, that shows how much time was spent in each function and its children. Based on this information, system architects can suggest candidate functionality for acceleration in reconfigurable fabric.

In SCENIC the application profile may be generated by using the built-in functionality to periodically log variables to a history buffer. By configuring logging of the program counter each clock cycle, the flat profile and the call graph is generated in a post-processing step that uses the logged program counter and the application object dump. This approach prevents special compilation of the application code and consequently it is non-intrusive. The overhead of logging the program counter every clock cycle is about 10% for an embedded system with a MIPS-I, SPB, and MPMC with *cycle-approximate* temporal behavior. The same method could be used to generate more fine-grained profiling information such as inner loop profiling.

System Level Profiling

Application profiling provides a good indication of functionality that should be hardware-accelerated or optimized to achieve the required performance. However, to identify and eliminate bottlenecks at system level, customized profiling data in other system components needs to be gathered. This profiling data is annotated into the models at design time together with exploration parameters that are used to tune performance. During exploration, periodic logging of profiling data can be switched on when required in order to analyze trends over time. When amount of gathered data is large and complex scenarios need to be analyzed, visualization is an important aspect to aid designers interpret data and draw conclusions. This is supported in SCENIC through a TCP/IP connection to any external program with plotting capabilities and a library of Matlab functions has been developed to connect and communicate with the simulator.

The SCENIC environment provides the capability to measure elapsed simulation time between different events in the system. The practical use of this is to gather time estimates of algorithm execution time, or any other time information which can not be annotated into the models at design time. This is supported using dynamic simulation events, which are created in run-time from the SCENIC shell. The shell command may be to simply stop the simulation, or to execute a script which take a time stamp, gather annotated profile data, do mathematical operations and writes any result to a file. Consequently, to

measure execution time for an algorithm, events can be created to log elapsed time as the program counter reaches specific values.

Systematic Batch Simulations

When embedded systems are developed there are multiple design parameters that need to be tuned in order to optimize performance. To manually change parameters, and collect performance data for further analysis can be very time consuming. Within the proposed methodology, the ability to explore and tune system parameters in batch mode has proven to be very useful. To allow batch simulations system generation, configuration, scenario setup, and data collection have to be automated. This is supported in SCENIC using the built-in customized script language. Built-in commands include for example setting environment variables, looping constructs, and evaluating basic arithmetic operations. In particular, each modules hierarchical SystemC name is used to communicate with the system components. Hence, loop constructs which systematically generate, configure, run and capture performance metrics may be described in a script file. The following script shows how to evaluate different processor cache sizes, when running a JPEG encoder.

```
% Define data cache size in kB
set cache "8 16 32";

% Run JPEG encoding with different cache sizes
foreach -var $cache -command run(embbeded.xml, cjpeg, $cache)
```

For each cache size, a new system is generated from an XML file, the cache size is configured, the JPEG application is downloaded, and after simulation has finished the execution time is appended to a result file.

4.2 Debugging

This section describes how SCENIC library components are interactively debugged.. There are several mechanisms added in all library modules to allow them to be debugged and an overview of these mechanisms are shown in Figure 10. In addition, external debug programs can be connected to allow efficient software debugging.

Software

All processor resources such as registers, memories and TLM ports are made accessible in SCENIC through callback functions, or access variables. Consequently, basic debugging capabilities are offered when running SCENIC standalone. For example, breakpoints can be dynamically configured by using

dynamic simulation events which execute the STOP command when the program counter reaches a specific value. However, some activities require more powerful debug environments, which is supported by connecting the SystemC module to an external debug program such as GNU debugger (GDB).

Software debugging using GDB remote target is supported for all ArchC generated modules instantiated in the system. Connecting one GDB session to a single ArchC module and running the program does not give any measurable simulation overhead. All ArchC modules have a command which dynamically opens a GDB port and multi-core systems are supported by opening a GDB session for each module. In addition all SCENIC commands to control and observe the system are available. When connection has been established between the simulation models and GDB, the non-blocking command RUN is used to start the simulation in SCENIC. Stepping and setting breakpoints is then applied the traditional way from each GDB session which controls how time advance in the simulation. When GDB access memory over TLM ports it uses a backdoor access to a global memory map, in order to avoid that simulation time advance or that number of bus transactions are accumulated. All memory-mapped storage is automatically registered for backdoor access.

System Components

The use of debug messages is a natural approach to trace if simulation models execute correctly. However, for large simulations the flow of messages often contains too detailed information and from too many simultaneous sources. Dynamic simulation events can be used to activate more detailed debug information when specific observation points in simulation are reached. During simulation, a designer may setup conditions that should halt simulation, write observation data to log file, change debug level or similar. It could be used to stop simulation after a specific amount of bus transactions, or when a transaction from a master with a specific identification occurs on the bus etcetera.

In order to trace communication bugs, models using TLM channels to communicate may optionally be configured to output a timeout message after a specific simulation time when a call has not returned successfully. This is very useful to find communication bugs that occurs in a chain of connected TLM models with master, transactor, bus, and memory and also to isolate communication bugs caused by improper configuration of memory map or port connections.

4.3 Accuracy and Simulation Performance

To make practical use of a virtual platform in software development and exploration scenarios, simulation speed has to be fast. Simulation speed is affected

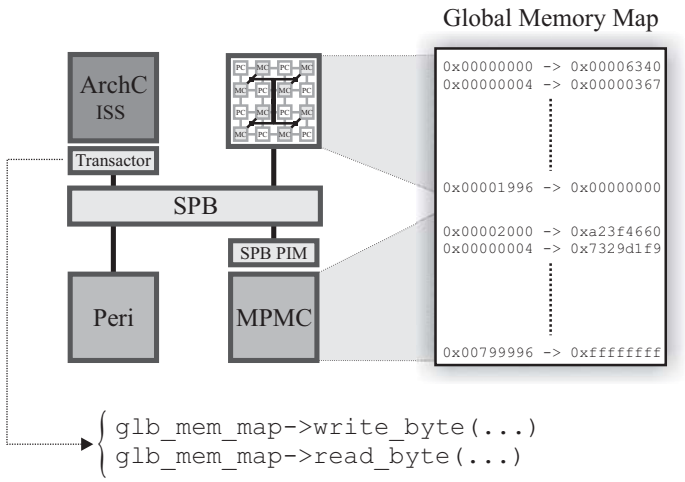


Figure 11: All modules may access memory mapped storage, such as registers and memory, using a backdoor access that is implemented using a global memory map. In this example, the ArchC *transactor* use backdoor access to increase simulation performance when temporal accuracy may be reduced.

by the abstraction level but also by simulator implementation such as event-driven or cycle based [70]. The OSCI simulator is event-driven and the following discussion is limited to the abstraction levels that can be applied to increase simulation speed of virtual platforms using the OSCI simulator. A number of abstraction modeling styles have been proposed, each falling into one of the categories: untimed, estimated timing and cycle accurate. While untimed models are mainly used for software development, estimated timing and cycle accurate models are used for architectural exploration. An untimed model without pin accurate interfaces is expected to provide a simulation speed several orders of magnitude higher than a cycles accurate and pin accurate SystemC model. It is argued that a virtual platform needs to provide support to switch abstraction levels during simulation. With abstraction level interactivity, architects may increase accuracy of the model only during periods when profile data is gathered, or when debugging the system on a detailed level.

The SCENIC models offers two ways to interactively switch abstraction levels: By reducing number of wait calls inside a SystemC thread or by reducing number of SystemC processes that need to be invoked by the scheduler to exchange a transaction. These abstractions can be used to increase simulation speed in parts of the simulation which does not need that level of detail in

Table 1: Simulation time required when executing JPEG encoding on a virtual platform with a MIPS processor, a system bus, and a multi-port memory controller. Temporal accuracy is varied from instruction accurate, to using memory backdoor access, to using bypassed wait statements.

Accuracy	**Simulation Time** (s)	**Relative Simulation Time** (time units)
Cycle-appoximate	1340	15
Backdoor	502	5.5
Untimed	92	1.0

timing. For example, profiling is to be performed after the operating system has been booted and the application starts it's execution. For this purpose, a dynamic simulation event can be configured to run a script which changes to a more accurate abstraction level when the program counter reaches the application code. The first optimization, invocation of wait statements, is controlled by setting *batchsize* which controls number of instructions to execute before wait is called in a thread. It is implemented in processing cells and ArchC generated modules. The second optimization, concerns reducing context switching when chain of processes need to be activated to complete a transaction. For example, a transaction that requires processes in processor, then bus, and then memory to be activated to exchange a single transaction. By using backdoor access to read and write to the system's memory mapped storage, such context switching is reduced. Transactors that connects ArchC modules to the system bus can be configured to use the backdoor to find and access the memory through a global memory map, which prohibit process activation in bus and memory controller. Consequently, reducing the amount of context switching during simulation at the cost of reduced temporal accuracy. The global memory map is shown in Figure 11, and it may be accessed by any module in the system.

Table 1 shows the effect on simulation time as *backdoor access* and *bypassed wait calls* are applied to a virtual platform executing JPEG encoding. It is shown that reducing temporal accuracy from instruction accurate to untimed behavior results in a 15x performance improvement. In this specific simulation the untimed behavior is accomplished by setting *batchsize* to the maximum value, which assures that no *wait* statement is called in the instruction set simulator. It is clear, that if performance exploration is required only in a fraction of the total simulation run-time and if temporal accuracy may be sacrificed otherwise, interactive switching of abstraction levels increases simulation

performance without reducing the accuracy of the observed metric.

5 Conclusions

Development of reconfigurable computing platforms requires methods and tools that aid designers in a variety of activities such as architectural exploration, debugging, and software development. This part has presented a SystemC environment with interactive control (Scenic), which addresses these activities through an amount of features added to the OSCI SystemC library. Furthermore, a set of flexible transaction level models have been presented, so that a complete reconfigurable platform with reconfigurable fabrics, processors, memories, and buses may be constructed. Different aspects of tool support to build, configure, observe, and interactively control simulation models were discussed. With *interactive simulations* a number of features are controlled at run-time, so that simulation performance and accuracy are adapted to current design activity.

Part III

Modeling and Exploration of a Reconfigurable Processor Array

Abstract

This part presents a coarse-grained reconfigurable architecture built as an array of interconnected processing and memory cells. A hybrid interconnect network that consists of local communication with dedicated wires and a global hierarchical routing network, is proposed. Memory cells are distributed and placed close to processing cells to reduce memory bottlenecks. Processing cells are instruction set processors with enhanced performance for communication-intensive inner loops. Inter-processor communication is performed using a self-synchronizing protocol that simplifies algorithm mapping and manages unpredictable time variations. The reconfigurable architecture is described as a scalable and parameterizable SystemC transaction level model, which allows rapid architectural exploration. Our exploration environment SCENIC is used to setup scenarios, control the simulation models and to extract performance data during simulation.

Based on: Henrik Svensson, Thomas Lenart, and Viktor Öwall, "Modelling and Exploration of a Reconfigurable Array using SystemC TLM ," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, Miami, USA, April 2008.
and: Thomas Lenart, Henrik Svensson, and Viktor Öwall, "A Hybrid Interconnect Network-on-Chip and a Transaction Level Modeling approach for Reconfigurable Computing," in *Proceedings of IEEE International Symposium on Electronic Design, Test and Applications*, Hong Kong, China, January 2008, pp 398–404.

1 Introduction

Reconfigurable computing platforms can combine high performance with flexibility [7]. These platforms are typically organized as a run-time reconfigurable architecture (RA) connected to a general purpose processor (GPP), so that computation intensive kernels are mapped to the RA and the remaining application is executed in the GPP [38].

Coarse-grained reconfigurable architectures (CGRAs) are promising in many application areas and several architectures have been proposed over the last two decades [64]. Even FPGA vendors currently include specialized coarse-grained blocks, such as signal processing datapaths, and processor kernels, to improve performance. A CGRA trade mapping flexibility, as offered by a traditional FPGA, to reduce delay, area, power, and configuration time [54]. Proposed CGRAs diverge in aspects such as granularity, interconnect topology, memory system, and programming model. In particular, presented CGRAs have different objectives when balancing resources for processing, memory and communication in a constrained chip area. The design space for these architectures is complex and architects are faced with the challenge of finding systematic approaches to evaluate and explore architectures before a chip is fabricated.

In this part, the coarse-grained reconfigurable architecture ACELRAY is presented. To obtain insights into the design space before a chip is fabricated, ACELRAY is developed as a scalable and parameterizable SystemC architecture model. The architecture is organized as an array of *processing cells* and *memory cells*, which communicate using a local and a global communication network. The processing cells contain programmable processors, communicating with a self-synchronizing mechanism. The memory cells contain memory banks that can be used as FIFO or RAM. The interconnect combines local communication on dedicated links with a flexible global hierarchical routing network. Hierarchical routing reduces the router complexity since there is only a single path between processing nodes. Dedicated local links provide high speed communication between neighbor resources.

Design of a reconfigurable architecture needs to be addressed at a system level, and simulation-based performance exploration is required to analyze impact of design parameters. Simulation-based performance exploration is limited by simulation run-time and the required design effort to develop, change, and refine models. This is addressed using transaction level modelling (TLM), which improves the exploration abilities, since time spent in design, simulation and refinement is reduced, as compared to RTL code. Designer interactive exploration is addressed using our SystemC environment with interactive control (SCENIC). SCENIC is used to automate construction of simulation models, simulation scenario set-up, and collection of performance data. As an

approach to exploration we propose having a set of parameterizable library models and a topology and network generator which can generate a simulation model of a reconfigurable array. Simulations are then interactively controlled with SCENIC user interface from which architects can inspect behavior and performance metric during simulation.

This part is organized as follows: Section 2 describes related work, Section 3 presents the modelling and exploration methodology, and Section 4 gives an overview of the architectural organization. In Section 5 and Section 6 processing and memory cells are proposed. In Section 7 routers used in the hierarchical network are proposed. In Section 8, experimental studies and results are presented. It demonstrates the performance for the communication network, shows how to map and analyze a filter algorithm, and presents synthesis results from a prototype implementation. Finally, Section 9 concludes the part.

2 Related Work

There are several research fields that relate to the work presented in this part: Organization of CGRAs, network-on-chip (NoC) communication, and methodologies to enable efficient design space exploration of these.

Coarse-grained Reconfigurable Architectures – An overview on reconfigurable computing is presented in [7,38]. Different CGRAs are briefly summarized in [64]. There are several well known examples of industrial CGRAs such as PACT XPP [44] and Field Programmable Object Arrays from MathStar [61]. The presented work reminds of MATRIX [56], RAW [41], CHESS [57], PicoArray [58], WPPA [59], MPPAs from Ambric [60], and FPOAs from MathStar [61], because they are built as arrays of small processors.

We propose an array architecture that are build from *processing* and *memory* cells. The processors are RISCs with a customizable instruction set and with enhanced functionality for fast access to the interconnect ports. Each processor has a small program memory and can execute one ore more instructions of a task that has been spatially partitioned. The memory cells are distributed and shared resources that are used to store coefficients or intermediate results during computation. They can operate as first-in first-out (FIFO) buffers, or as random access memory (RAM). To allow *address generation* and *datapath* to execute in different processors, the RAM mode allows data and address ports to be associated with different processing cells.

Topology and routing – Recent work compares many popular NoC architectures, including *2D mesh*, *ring*, *torus*, *folded torus*, and *spidergon* networks [102] [37]. *Hybrid topologies* are discussed in [103] as a way of aggregating bandwidth between adjacent processing cells, but introduce longer latency and require bridges to convert between protocols. The 2D mesh architecture in Fig-

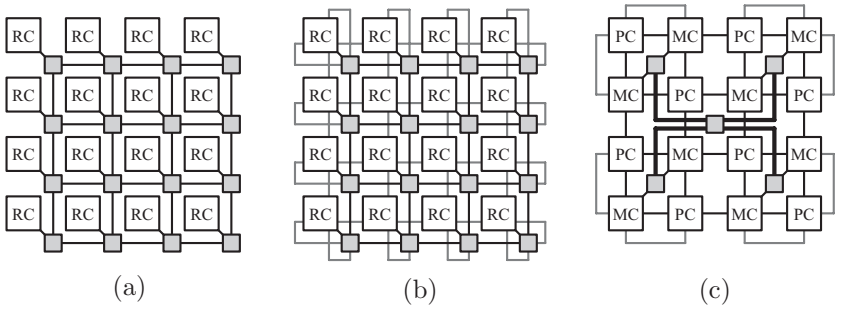


Figure 1: (a) 2D mesh with resource cells, (b) Torus, (c) Proposed architecture with nearest neighbor and hierarchical communication. In this example, four basic cells are connected together using dedicated wires for local communication and five network routers for global communication.

ure 1(a) is a popular and well researched architecture in academia, with several variations such as torus shown in Figure 1(b) and the folded torus. Mesh architectures are both regular and scalable, but comes with the downside of poor global communication and often large router logic overhead. Global communication is routed along the horizontal and vertical wires from source to destination, consuming local bandwidth along the way. Each processing cell connects to the two-dimensional mesh using a network router, which contains buffer queues to store and forward packets from all directions. Since every router can potentially handle communication from any source to any destination, the router becomes unnecessarily complex. Hence, pure mesh architectures face problems concerning both bandwidth and high complexity.

This part proposes a hybrid network that combines fast dedicated local communication with a flexible global network. The global network is build as hierarchical connections of routers and it allows communication between any two cells to be dynamically initiated during task execution. The path between sender and receiver in the global network is deterministic, which simplifies the router implementation. The local network is based on switches controlled by the current configuration and the instructions executed in processors. We propose the use of distributed memories to handle communication between processing cells, avoiding large routing buffers in the network. Our approach is to keep communication simple, using the main part of the area to implement memory and processing cells.

Exploration methodology – Traditionally, computer network simulations have been used to simulate NoC communication [104]. However, these approaches focus only on network communication and omits other system aspects

required for design exploration such as implementation of processing elements and accurate hardware modeling. Development of new CGRAs requires scalable and parameterizable *architectural models*, high simulation performance, and tools to evaluate and explore performance at system level. In [59], a programmable processor array is described and tuning to specific application domains is addressed through a parameterizable architecture description. An event-driven approach to enhance simulation performance for this specific architecture is proposed in [105]. However, this approach focus mainly on simulation speed and omits aspect such as system level simulations, which require interoperability with legacy models, and that models of custom components can be developed and explored. SystemC is one of the candidates that addresses system level design aspects [73] and provides high simulation performance. Experiments presented in [106] show that SystemC simulations can increase simulation performance several orders of magnitude, as compared to traditional RTL simulations. Simulation frameworks and NoC models based on SystemC has been presented in [107] and [108], but focusing mainly on communication.

In order to build, configure, and explore reconfigurable architectures, a simulation and exploration tool based on the open SystemC initiative (OSCI) SystemC library, has been developed. Furthermore, a consistent use of SystemC and TLM to evaluate and explore our architecture, is proposed. As a result of the raised abstraction level, models are fast to develop, become more flexible, and simulate faster. A system level modeling perspective is suggested, and several flexible system components have been developed in order to allow evaluation of our architecture integrated into an embedded system.

3 Modelling and Exploration Methodology

Modules communicate through channels and interfaces specified by the OSCI TLM standard [66], which provides a communication mechanism based on blocking and non-blocking *put* and *get* functions. The major advantages with TLM are high simulation speed and simulation models that are easy to modify compared to register transfer level (RTL) models. In TLM, pin-accurate models are replaced with an abstract communication layer, which uses *transactions* to transfer data. A transaction is a data structure that contains information about the transfer, for example memory address, transfer size and payload. The simulation speed increases compared to cycle activated models because processes are activated or resumed only when transactions are exchanged between modules. Figure 2 shows a point-to-point communication example between a master and a slave device, using separate FIFO channels in each direction. The slave module blocks on a call to the get function until the master puts a *request* after 10 ns. The master blocks while waiting for a *response*, which is provided by the

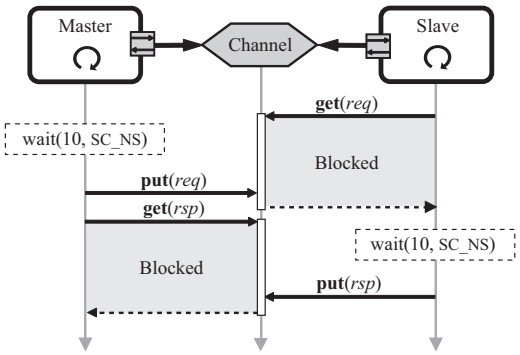


Figure 2: Communication between a master and slave module connected with a master/slave channel (containing two FIFO instances). Function calls are from the module to the channel. The OSCI-TLM library provides channels implementing functions to exchange transactions between modules.

slave after another 10 ns. Note that function calls are always made from the module to the channel, but the flow of information depends on the function.

A transaction exchanged between modules is refereed to as a *transfer* in this part. A transfer contains destination address, source address, payload, and payload type specification, as shown in Figure 3. The type specification is needed as the payload can be either data, memory address, configuration, or flow control exchanged between network routers in the global network. The size of the payload is typically the same as used in computational operations in processing cells.

3.1 Exploration Environment

Our SystemC environment with interactive control (SCENIC) is based on the OSCI SystemC library [73] and provides additional functionality to rapidly develop SystemC simulations for evaluation and exploration of systems. User interaction is enabled through module instrumentation, run-time variable access, real-time simulation control and a scripting environment. An introduction to key aspects in SCENIC is given below, and illustrated in Figure 4.

Variable access

Interactive simulations require access to internal variables during run-time. The standard SystemC module class has been extended with functionality to access local member variables from the user interface. In addition, it is possible

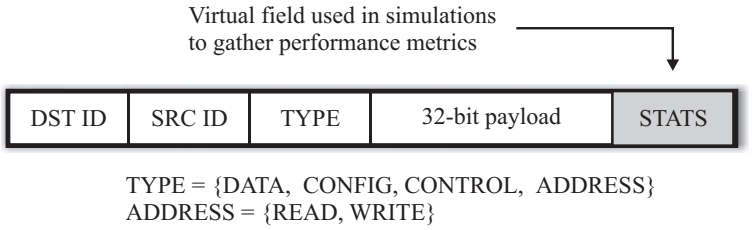


Figure 3: A transfer contains routing information (IDs), a payload type field (TYPE), and the payload. It also contains annotated statistics (STATS), such as identification number, produce time, consumption time, etcetera.

to periodically capture content of member variables to a history buffer, which is required when analyzing trends over time.

User interface

SCENIC has a user interface that supports basic scripting to simplify simulation setup and design space exploration. Registered variables are reached from the user interface to setup parameters before simulation and inspect performance metric and result data during simulation.

Modules and architectures

User modules are automatically registered at program start-up and stored in a module library. The module library is an object creation factory that handles module instantiation during elaboration. To allow design exploration of partially reconfigurable systems, a *reconfigurable module class* is provided to dynamically insert and remove modules from the simulation [95]. It is used to configure synthetic modules, such as traffic generators into a processing cell to create simulation scenarios in the scripting environment. *Architectural templates* are user objects that defines how to construct the system in terms of modules and connections from a set of parameters.

4 Architectural Organization

The ACELRAY architecture is organized as an array of interconnected resource cells (RCs). We propose a topology that splits the interconnect network into local communication and a global network. Local communication use dedicated links to connect neighboring cells. The global network connects any two cells using hierarchical routing. The motivation for this topology is the fact that

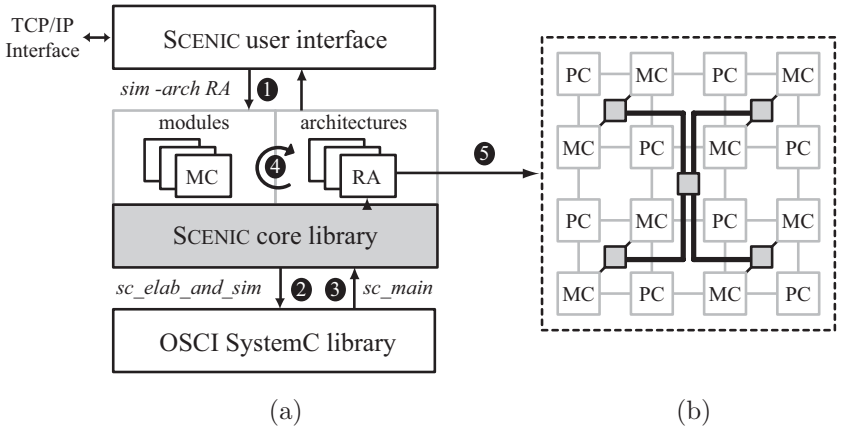


Figure 4: (a) The user interface communicate with the SCENIC core library. A simulation is initiated from the user interface, which triggers the SystemC library to initialize. When ready, SCENIC calls the requested *architecture* which construct a simulation platform from the module library. (b) The SystemC simulation constructed with SCENIC has runtime control and observability from the user interface.

an optimally mapped computational structure often results in a high degree of local communication. In contrast, the mesh architecture has a shared network for both local and global communication, where even neighboring cells communicate over the global network. It also means that global communication consume bandwidth from local communication. The proposed topology can be viewed as two independent networks interacting through network routers, shown in Figure 1(c).

Resource cells have a number of I/O ports from all four sides, defining its local communication bandwidth capacity and an optional uplink and downlink port to connect to the global network. The global network connects any two cells with single-path hierarchical routing, in which transfers are routed to the destination using an identification field. The global network is used for data communication between RCs to enhance routing flexibility offered by local communication. It is also used to connect resources to external memory and to a configuration manager that can configure tasks to execute on the array. Connections over the hierarchical routing network are dynamically changed by altering the *destination identification* field (DST ID) in transfers.

Figure 5 shows modules, ports, bindings, and channels of a 2×2 array with two type of RCs, processing cells (PCs) and memory cells (MCs). Processing

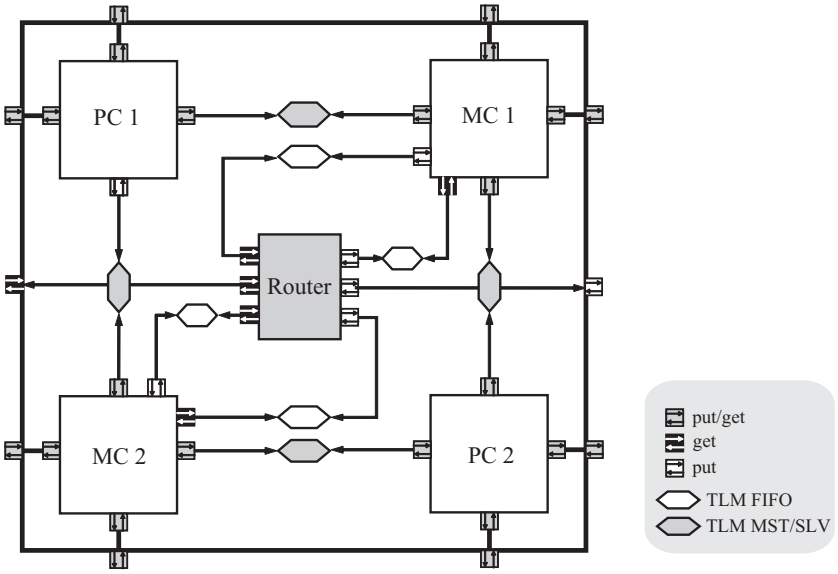


Figure 5: Transaction Level Modeling of ACELRAY using OSCI SystemC TLM libary.

cells contain computational kernels, built either as processor cores, or dataflow components. Memory cells are used to separate processing from communication and to provide intermediate storage during computations. Each port on a processing cell is connected to a neighboring memory cell, which handles the communication with other parts of the system. A memory cell can be configured to forward incoming data to any arbitrary memory cell (global communication), or to simply supply the neighbor processing cell with the received data (local communication). Processing is self-synchronizing, so that any computation that uses output from another processing cell is blocked until data becomes available and result will not be transmitted to the output unless there is available buffer space. Implementation details are presented in Section 5 and 6. The hierarchical global network is implemented with routers, depicted in Figure 5. The coarse-grained granularity is reflected both in the communication and in basic computations, as routing (global network) and switching (local communication) is performed on *transfers*, and computations performed on *payload* in *transfers*. The size of the payload is selected as 32-bit wide for the work presented in this part.

Models of embedded system components have been developed to explore and evaluate ACELRAY integrated in an embedded system. This is important

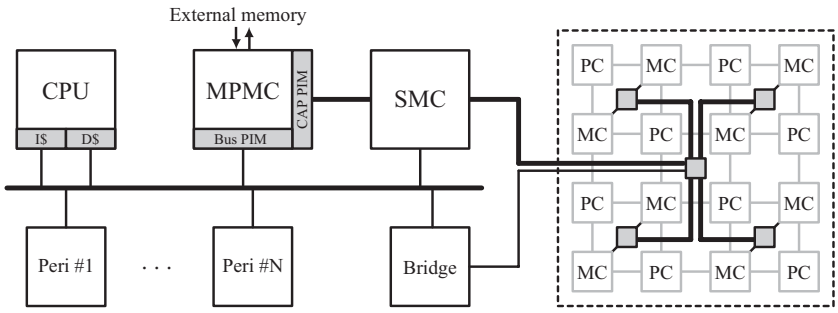


Figure 6: System components such as processor, multi-ported memory controller (MPMC) and bus have been developed and integrated into our environment to address design exploration of ACELRAY integrated into an embedded system. The stream memory controller (SMC) handles data transfers to and from the reconfigurable accelerator.

as performance depends on bandwidth to external memory and the interface between the array and an embedded processor. Figure 6 shows the models developed and illustrates ACELRAY integrated in a system with a shared bus, a stream memory controller (SMC), a multi-port memory controller (MPMC) and a processor (CPU).

5 Processing Cells

Implementations of algorithms are composed of either *computation*, *control*, or *memory* blocks. Processing cells are responsible for implementation of *computation* and *control* functionality. Computational operations are performed in a datapath that consists of connected processing cells. Data enters and leaves the datapath from/to *another datapath*, *memory* or from any *external device*. Control operations are performed in processing cells that implements operations such as *switch*, *fork*, and *address generation* used to access data in memory cells. A key property during processing is the *self-synchronizing communication mechanism* between processing cells. It reduces the complexity involved in programming and mapping algorithms to the array and it copes with time variations that can not be statically predicted. Unpredictable timing is observed when connecting processing cells to a shared resource such as the global network or an external memory.

This section describes the processing cell architecture and propose mechanisms to implement the functionality described above. Insights into machine parameters and balance of communication and processing capacity is discussed.

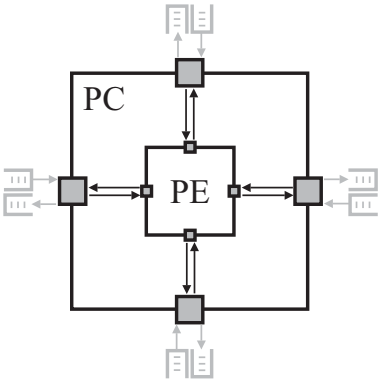


Figure 7: A processing cell contains a processing element, which connects to its port. The ports are connected to FIFO buffers implemented in memory cells.

It is also shown how the model supports exploration of the instruction set, processor resources, and timing parameters. First however, the *processing element* used as building block inside processing cells, is introduced.

5.1 Architecture of processing elements

A processing element (PE) is used as a building block for processing cells. The processing element is organized similarly as a conventional RISC processor core. It contains a small program memory, general purpose register (GPR), and execution pipeline. Larger memories, for data or instructions, are implemented by connecting to a memory cell, as described in Section 6.

Processing elements connect to the I/O ports of the processing cell, which are connected to *FIFO buffers* implemented inside the memory cells, seen in Figure 7. Input and output FIFO buffers are used to synchronize communication between processing elements and their contents can be read and written to/from GPRs with the *move* instruction. However, *move* instructions may become a bottleneck in a short inner loop that implements both the *I/O communication* and the *computation*. Therefore, a new addressing mode is proposed. In the proposed addressing mode, I/O communication is embedded into the instruction. It means that register-register and register-immediate instructions allow operands to be fetched directly from input buffers, and the result to be written directly to any of the output buffers. Any such instruction must stall the pipeline when input operands are not available or when the output buffer is full. As the timing is not statically predictable this must be handled

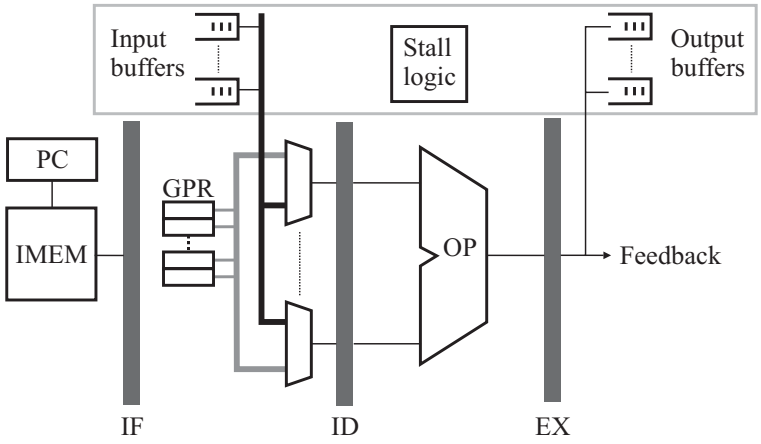


Figure 8: A modified three stage execution pipeline to enhance performance for I/O intensive computations. Instruction source references can be input buffers and instruction destination references can be output buffers.

dynamically in hardware.

Figure 8 shows how the proposed addressing mode affects the execution pipeline. For each source operand the input multiplexers are extended to select any of the input buffers. At the output stage each output buffer is connected to the result register and control logic is inserted to write to referenced destination buffers. Each output transmitted to any buffer is also written back to the general purpose registers to be used in subsequent instructions. The stall logic stalls all pipeline stages if decoding an instruction with a *source reference* to an empty buffer or if there is a *destination reference* to a full buffer in the execution stage.

To implement functionality that allows PEs to operate as external address generators, another addressing mode is suggested. Traditional load and store instructions are complemented with an addressing mode that allows any instruction to select the *transfer type* to be transmitted to the output port. The transfer type transmitted from a PE is either *data*, *read address*, or *write address*, see Figure 3. These types are used by memory cells to determine their operation.

Table 1 summarizes the addressing modes proposed. In the assembly code notation, a general purpose register is denoted R and an external port that connects to memory cells denoted with X. As described in Section 5.2, a cluster of processing elements can be contained in a processing cell, and references to

Table 1: Addressing modes proposed

Addressing mode	Example
Register	**add** RD RA RB
I/O data	**add** (R|I|X)D (R|I|X)A (R|I|X)B
I/O read address	**add** r@XD (R|I|X)A (R|I|X)B
I/O write address	**add** w@XD (R|I|X)A (R|I|X)B

I/O buffers in the local cluster interconnect are denoted with *I*, in the assembly notation. To distinguish the transfer types that can be transmitted to an external port, *w@* and *r@*, are used for *address write transfers* and *address read transfers*, respectively.

5.2 Cluster of processing elements

A processing cell built with several processing elements is referred to as a cluster of processing elements. The cluster is introduced to balance processing capacity against I/O capacity. Some PEs use the full I/O capacity of the processing cell and hence only one instance per processing cell is created. Conversely, there are PEs which will not use the full I/O capacity offered by the processing cell and several such PEs are clustered in a single processing cell.

Figure 9(c) shows one possible configuration, with a cluster containing 4 PEs and a local cluster interconnect. The number of PEs contained in a PC and the type of interconnect inside the cluster are generic settings to the model. Every PE output port has a FIFO buffer that connects to input ports of other PEs in the cluster, and every PE input is connected to the multiplexer at the decode stage, as shown in Figure 8. Consequently, references to buffers in the cluster network have the same effects on synchronization and stalling as references to processing cells' I/O ports. If several PEs in a cluster are connected to the same external PC I/O port, signals are sent through a bitwise *or-gate* network. Hence, mapping of programs needs to prevent that two PEs attempt to receive or transmit to the same port simultaneously. In addition to balancing I/O against processing capacity, the cluster interconnect allows to distinguish the FIFO capacity offered internally from the capacity offered by memory cells. For example, small FIFOs are implemented inside the cluster and algorithms that need a larger FIFO or RAM use a memory cell, as described in Section 6.

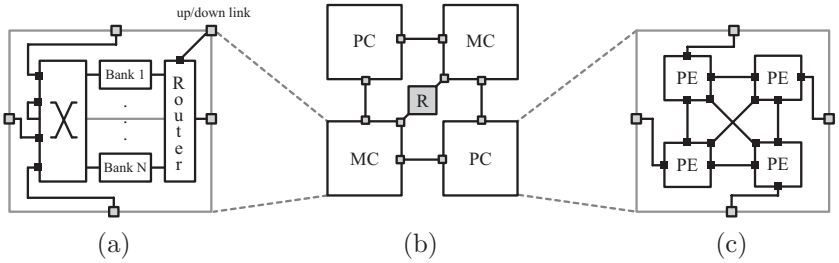


Figure 9: (a) A memory cell contains a generic number of memory banks, a switch and a router. (b) A 2x2 ACELRAY with a router that connects the memory cells to the global network (c) A processing cell that contains a cluster of processing elements and a local interconnect.

5.3 **Instruction extensions**

The PE model has been developed so that architects can evaluate performance of application-specific extensions of the *instruction set*, or the *instruction format*. To develop a custom instruction, architects write a new class that inherits from the instruction base class. The base class contains functionality to model behavior, timing, and side effects. The following code is required to model an *add* instruction.

```
PE_REGISTER_INSTRUCTION(add);
add::add(sc_module_name nm) :
    pe_instruction_module(nm)
{   set_execution_time(1);
    set_nbr_source(2);
    set_nbr_destination(1);
    set_nbr_immediate(0);          }

void add::operation(const vector<CA_WORD>&  source,
                    vector<CA_WORD>&        destination,
                    const vector<CA_WORD>&  immediate,
                    SPECIAL_REGS&           sregs )
{       destination[0] = source[0] + source[1];         }
```

The instruction behavior is implemented in the *operation* function, which is called from the execution stage. In the constructor the architect set parameters that are used to propagate inputs and outputs and simulate timing. An added instruction is automatically registered through the *macro* on the first line, so that any PE instantiated during elaboration finds it. During system construction or simulation, customization of PEs are controlled from the SCENIC user interface.

5.4 Exploration Parameters

In all modules there is a set of parameters that can be explored, such as size of memories, clock period and latency. Some parameters directly connects to a physical entity such as clock period to execution time and memory size to hardware area. Other parameters reflect aspects of the system connected to functionality such as the set of supported instructions in a PE, which in a more abstract way are connected to performance and area. The goal of exploration is to understand how the system parameters affect performance, area and functionality of the architecture. The processing element has machine parameters such as clock period, pipeline latency, execution time for instructions, number of outputs/inputs, size of registers and instruction memory. Furthermore, the instruction format may easily be adapted to model special-purpose formats or multi-issue architectures.

There are two optimizations that can be performed to reduce complexity when having multiple PEs on the chip as compared to a single processor core: Not all PEs need to support all instructions and some PEs may act as *single-instruction* execution units which statically execute one of its operations per configuration. The first optimization that concerns heterogeneity is quite natural and has been proposed and implemented in other RAs. Typically, hardware support for multiplication, division, or other area consuming functionality is not implemented in all PEs. The second optimization that concerns the execution pipeline is proposed because many algorithms will utilize the array as a deep pipeline were each processing element executes a single instruction. Hence, unnecessary control logic and program memory associated with sequential execution can be removed in some PEs to reduce complexity. However, it is also recognized that there are tasks which needs an instruction flow to implement the control behavior. Both approaches, static *single-instruction* and program-flow execution, have been proposed in RAs. In [44] they suggest a single instruction per configuration to build static structures during task execution, in [59] they suggest an array with processors with a short program memory, and in [61] they suggest a control flow implemented in a configurable state-machine in which each state is associated with a specific instruction.

6 Memory Cells

Algorithms require intermediate storage to buffer and reorder data. The use of on-chip memory for intermediate storage may significantly increase performance, as compared to implementing such storage in external memory. Performance may be further improved by distributing the storage capacity to several memories to improve data delivery and by placing memories close to datapaths to reduce latency. These aspects have been considered in design of the memory

system, which is implemented in memory cells. Memory cells are distributed and shared resources that can be allocated by any processing cell. When balancing memory and computing resources in a constrained chip area, this may be advantageous as compared to memories that are dedicated to a specific processing cell. Five major functions for the memory cells are identified:

- Act as a self-synchronizing barrier between communicating processing cells,
- implement configurable number of initial transfers,
- support random access,
- allow inter-task communication by keeping data while tasks are sequentially mapped to ACELRAY, and
- support larger memory structures to be emulated from a set of smaller distributed memories.

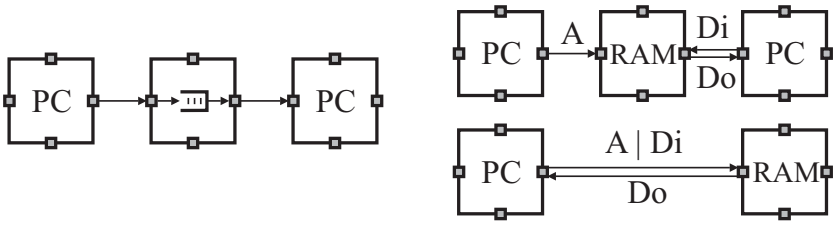
A memory cell is sub-divided in three types of blocks: a generic number of memory banks (MBs), a switch, and a router. Figure 9(a) shows a block diagram of a memory cell. Memory banks contain the actual storage, implemented as dual port RAM, or registers for smaller memories. The configurable switch connects a memory cell's external I/O ports to memory banks without latency. Communication with the global network is handled through a bidirectional connection between each memory bank and the router.

6.1 FIFO emulation

The basic functionality in all memory banks is the FIFO mode. With a memory bank operating as a FIFO, a link which supports self-synchronization can be allocated to connect two processing cells with an initial number of transfers in the buffer, see Figure 10(a). A configurable number of initial transfers is another key property, which allows algorithms described as a data-flow graphs (DFGs), with delays associated with an edge that connects two nodes, to be naturally mapped to the array. This functionality is utilized in the case study presented in Section 8.

6.2 RAM emulation

In addition to the FIFO mode a memory bank may support RAM mode, which provides other necessary primitives to efficiently support algorithm mapping. The RAM mode is used to randomly access local data such as coefficients or instructions and to reorder data communicated between two processing cells.



(a) (b)

Figure 10: (a) A memory bank inside a memory cell configured to operate as FIFO that synchronizes communication between processing cells and has a configurable number of initial transfers. (b) A memory bank inside a memory cell operating as RAM which receives input data (Di) and address (A) and transmits output data (Do).

When a bank supports both FIFO and RAM mode, swapping between these operational modes is done by transferring configuration data to that bank, while the same physical storage is reused. Hence, a memory bank can act as a FIFO in one task and as RAM in a subsequent task, which randomly access result data.

In RAM mode, a *write access* is accomplished by streaming *data transfers* and *address transfers* to a memory bank. *Address* and *data transfers* are synchronized, so that *address transfers* are blocked until the *data transfer* has been received and conversely *data transfers* are blocked until a valid *address transfer* has been received. A read access is done by streaming *address transfers* to one of the ports in a memory bank, which triggers a read operation with resulting data transmitted to one of the output ports.

Algorithms that have data independent address generation can be partitioned into *address generation* and *computation*. These two partitions may be mapped onto separate processing cells to improve performance. Hence, address generation can be pipelined and access latency experienced during repeated read operations hidden. Figure 10(b) shows two possible connections to a memory bank operating in RAM mode. In one of the examples shown in Figure 10(b) address generation is mapped to a processing cell different from the processing cell that consume and produce data. The connections shown in Figure 10(b) can be further expanded to connections where either data or address is communicated over the global network.

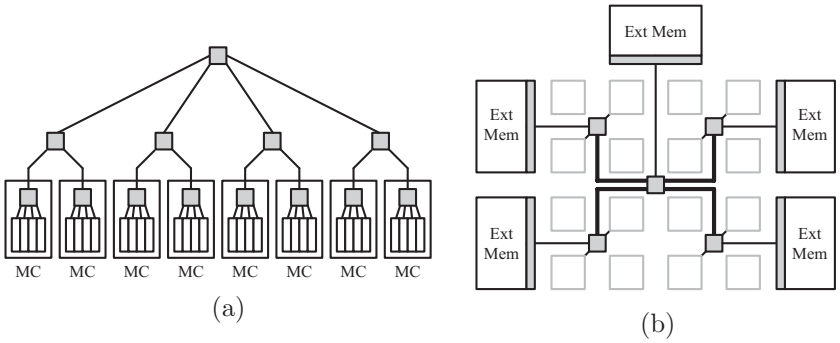


Figure 11: (a) Hierarchical connection of routers to memory cells, (b) On-chip scratch pads or external memory connected to routers. The designer specifies where to connect memory controllers for external memory.

6.3 Emulating larger memories

Memory banks can be concatenated to emulate larger memories. This is important because algorithms might require larger storage capacity than offered by any single bank available. Mapping such storage to external memory might lead to serious drops in throughput, due to traffic contention in global network, bus, or external memory.

The concatenation mechanism suggested is straightforward for banks operating in FIFO mode. Any number of banks can be cascaded with incoming data forwarded to the next memory bank over the global network. The performance penalty without traffic contention in the global network is two clock cycles latency for each bank added to form the cascaded FIFO. The throughput remains the same under the condition of no traffic contention.

In RAM mode there is a similar mechanism. Each memory bank supporting RAM mode can be configured to have a *base address* and *high address*. If *address transfers* received have an address that is out of range, it is forwarded to the next memory in the cascade. *Address data* and *write data* enter a cascaded memory network through the first memory and *read data* leaves through the last memory. Any processing cell connecting to a cascaded memory will experience the same interface and behavior as with a single memory bank.

6.4 Connecting to external memory

Local memory can store data during computations, but the data to be processed is usually located in on-chip scratch pads or in external memory. To fully benefit from the computational power, data has to be streamed in and out

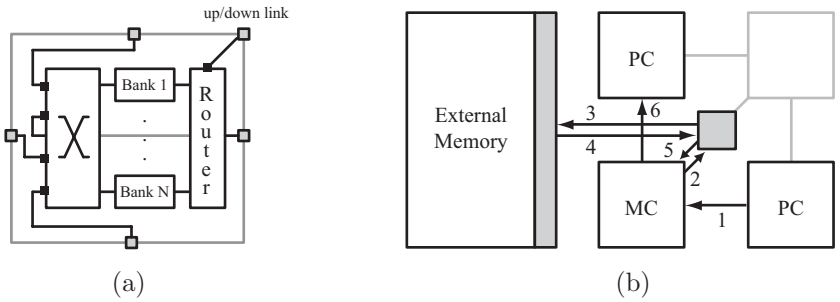


Figure 12: (a) Internal building blocks in the memory cell, consisting of a port switch and four memory banks. Each memory bank has a unique ID for routing data over the network. (b) Two memory cells using the external memory interface to transparently emulate a larger buffer.

at high speed. External memory interfaces can be connected to any router in the network and accessed using a unique identification in the same way as memory banks, which is shown in Figure 11(b). Every memory bank that communicates with the external memory can setup a transfer to or from the memory. This transfer can be either a linear memory access or a more complex stream specified using *stride*, *span*, and *skip* parameters. The external memory controller handles the memory access mode and provides the memory bank with data.

External memory may also operate as an extended buffer when required. Local memory is limited and sometimes larger memory buffers are required between computations. A transfer between two processing cells is usually routed from one memory cell to the other, but can transparently be connected through an external memory controller to act as a large intermediate buffer, shown in Figure 12(b).

6.5 Exploration parameters

The storage capacity in a memory bank is an important system parameter, which needs to be elaborated and balanced to the processing capacity to adapt to specific application domains. It is likely that an optimal balance for a certain application will be a heterogenous distribution of capacity to memory banks. For example, some application might not utilize any RAM and needs only FIFOs with a storage capacity of a few transfers. The architecture model facilitate exploration by setting number of memory banks, size of memories, clock period, latencies and supported operating mode, RAM, FIFO, or both.

7 Routers

In the proposed architecture routing is *deterministic*, which means that there is only one single valid path to route network traffic. To change the routing path, the sender must select a different receiving memory bank to reach its final destination. This means that routing is a simple task of forwarding data between memory cells and network routers, which reduce hardware complexity as compared to *adaptive* routing algorithms. The router is built as a switch that connects input ports to output ports. Each output port is associated with a range of consecutive addresses to that an incoming transfer only requires a table look-up to be routed to the correct output port. If an incoming address is not found in the routing table, the transfer is sent to the default port which is upwards in the hierarchical routing network, as shown in Figure 11(a). Once the path is recognized by a router, the transfers are propagated downwards until reaching its final destination. The data width and the capacity of the switch are parameterizable in order to evaluate different router capacities when exploring Acelray .

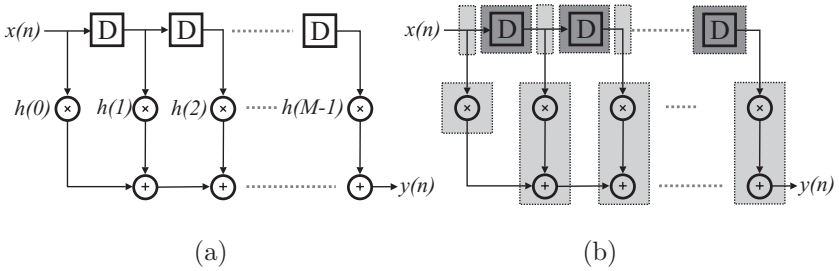
Several proposed router architectures contain output buffer queues to temporarily store packets traveling through the network [103]. When the communication load increases the buffer queues fills up, resulting in dropped packets or poor throughput. The reason is that a packet inside the communication network is only half-delivered, still waiting for the next router to accept outgoing packets. In case the receiving node is unable to accept a packet, it is trapped inside the communication network, resulting in increased latency in other parts of the network. In the proposed architecture memory banks are used as an alternative to large output queues in the network routers. Instead of dividing the total memory resources in two static pools, i.e. local memory and router memory, local memory will be assigned for either of these tasks.

8 Experiments and Results

This section presents a case study in which a FIR filter is implemented on Acelray and performance data gathered to evaluate, explore and tune the implementation. It also presents a exploration study that determines performance characteristic for the proposed hybrid interconnect network under different traffic loads and communication patterns. The sections ends with showing synthesis result from a prototype implementation.

8.1 Filter Implementation

To demonstrate the usefulness of our Acelray architecture and the advantages offered by our methodology, a case study implementing a finite impulse



(a) (b)

Figure 13: (a) A FIR filter in direct form (b) One possible partition of functionality were dark grey is mapped to memory bank or PEs internal output buffer and light grey to a processing element.

response (FIR) filter is presented. The SCENIC scripting environment is used to configure modules to define a scenario, reconfigure modules during simulation (controllability), and to extract performance data during and after simulation (observability) to evaluate the system.

Filter mapping

This case study demonstrates implementation of a fixed-point FIR filter using direct and time-multiplexed form. Figure 13(a) depicts the data-flow graph for the direct form, in which D denotes initial delays between the computation nodes. The time multiplexed implementation is derived by folding the direct form into a single multiply and accumulate operation and additional control that sequence inputs to the computation from memory.

A task mapped to ACELRAY is described with a DFG were each *node* is associated with a program and each *edge* is associated with an initial delay. When mapping the DFG to ACELRAY, several constraints have to be satisfied:

- Map node j to PE_k such that PE_k supports all instructions in the program and that the program fits into the instruction memory, and
- map edge (j, i) to a communication link such that initial delays are less than the total buffer size.

There are also several optimizations that can be considered:

- Reduce communication latency by mapping communicating nodes closely,
- ensure sustained throughput by preferably use dedicated local communication links, and

- reduce traffic contention when mapping nodes which communicate through the global network.

The direct form implementation requires three *single-instruction* programs as depicted in Figure 13(b): fork, multiply and add, and multiply. Partitioning the functionality into these instructions allows having maximum throughput. The time-multiplexed implementation separates address generation and computation to maximize throughput. It requires three different programs which all needs more than a single instruction: coefficient address generation (AG), data input AG, and the multiply and accumulate (MAC) program. The implementation also requires two memories configured to operate in RAM mode both storing M values, where M is the filter length. Addresses from the AG programs are connected to corresponding memory inputs and the memory outputs are both connected to the PE that performs the multiply and accumulate and moves the result to an output FIFO.

The programs take parameters, which specify to which port an external I/O reference should be bound during execution. This allow programs to be mapped with configurable I/O orientation. It is currently accomplished by changing programs before mapping, but may also be handled with hardware support that performs a look-up when referencing I/O ports.

Experiments

An exploration loop requires scenario setup and performance data extraction to be automated. SCENIC provides this capability through a scripting environment, so that experiments can be completely setup from the user interface. In the scripting environment architects can define the architecture to be simulated in terms of machine parameters such as: number of resources, type of interconnect, memory sizes, supported instructions, execution times, and latency. It also operates as configuration manager so that tasks can be programmed from the script. The following script code shows a condensed example of how a scenario is defined:

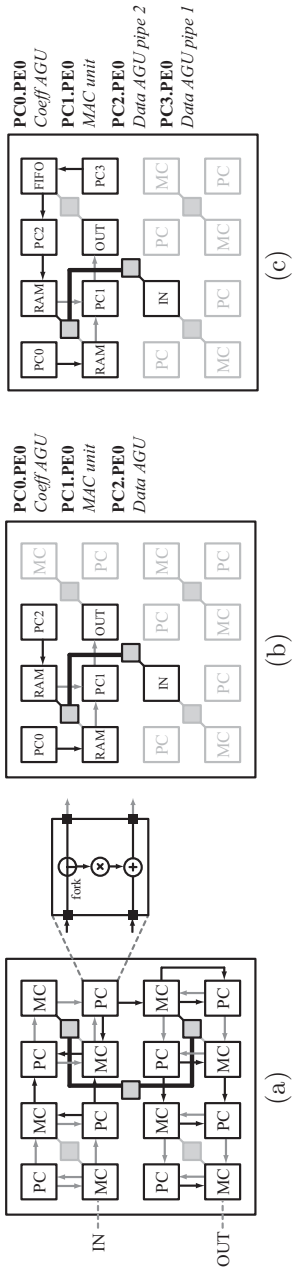


Figure 14: (a) Direct form mapping of a FIR filter with 8 taps (b) Time multiplexed mapping of a FIR filter (c) Time multiplexed implementation were address generation is pipelined to reduce execution time.

```
%  Create architecture
   sim -arch CellArray -width 4 -height 4 -nr_mb 4 -nr_pe 4

%  Load library modules
   pc2 reconfig -mod PECluster

%  Configure progam memory with AG assembly
   config_cell(pc2.pe0, agu_data.asm(South))

%  Configure round buffer memory
   config_link(mc0.Bank0, RAM(East, South))
            .
            .
%  Download test data input
   mc4.Bank0 write -file data.txt

%  Run simulation
   *.* set -var clock_period -value "10 ns"
   runb 1000 ns

%  Inspect output data
   mc3.Bank0 read

%  Read performance data
   pc2.pe0 get -var performance_vector
```

Figure 14 shows one possible mapping of an 8th order FIR filter, in (a) direct form and in (b) time multiplexed form, to ACELRAY. The experimental setup is a ACELRAY with 4×4 resource cells allocated as 8 memory cells and 8 processing cells. Each resource cell has 4 bidirectional I/O ports connecting to the nearest neighbour. Each processing cell contains 4 processing elements and each memory cell contains 4 memory banks. Three of these has 64 bytes and supports only FIFO mode and the other has 1024 bytes and supports both RAM and FIFO. Local and global communication supports transfers with a 32-bit payload field and arithmetics in PEs are 32-bit. The design is further constrained such that only one processing element in each PC can handle a program with more than one instruction and support for multiply and MAC instructions. The other 3 PEs are single instruction execution units without MAC or multiply instructions. The instructions considered in this case study are 32-bit format. Both standard RISC instructions as well as domain specific instructions, such as *fork*, *mac*, and *loop* instruction have been developed.

Results and Discussion

Figure 15 shows execution time of direct mapped (DM) and time multiplexed (TM) implementation, when filter length is varied from 8 to 32. The execution time is measured as filtering a block of 256 32-bit wide input samples stored in a FIFO in the array at execution start. The speedup for DM compared to TM is ranging from a factor of 18 down to 15, because the loop overhead of AGs

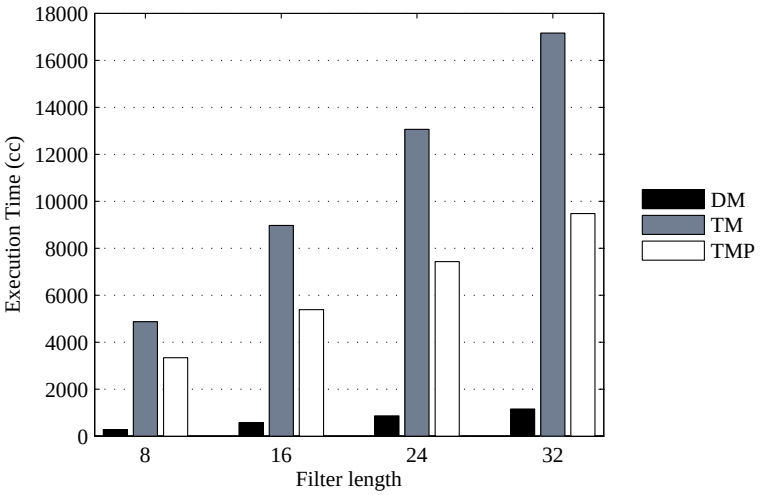


Figure 15: Execution time in clock cycles for direct mapped (DM), time multiplexed (TM), and pipelined time multiplexed (TMP).

are reduced in the TM implementation when the filter length increase.

The DM implementation is executing 8 taps and is then reconfigured with the next 8 taps with intermediate result stored in a memory bank. The throughput of the DM implementation is one output per clock cycle after an initial latency. The latency is 30 clock cycles for an 8-tap filter and over 1000 clock cycles for the 32-taps filter, as final outputs will not be produced until the last configuration out of 4 has been mapped. The latency of the TM implementation is ranging from 26 to 74 clock cycles.

Figure 16 shows utilization of PEs allocated to perform the TM implementation. The bottleneck is the processing element generating the addresses for the round buffer, PC2.PE0 in Figure 14, as it is utilized 100% while the other two PEs have significantly lower utilization. The code executing in PC2.PE0 is:

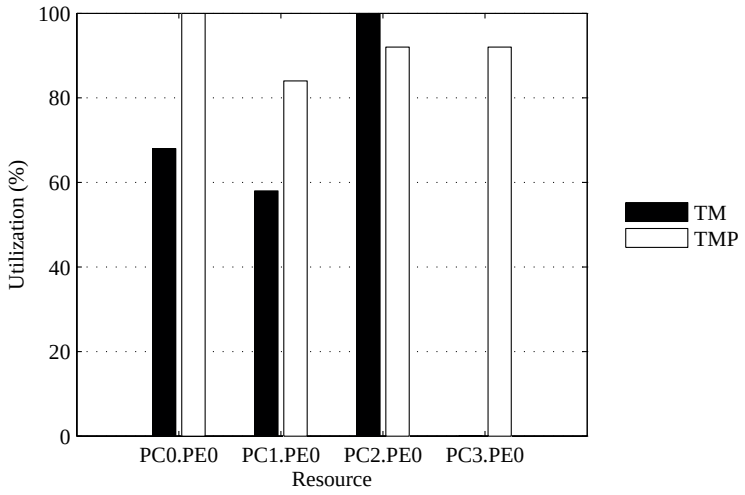


Figure 16: Utilization of processing elements after execution of time multiplexed (TM) and pipelined time multiplexed (TMP), see Figure 14.

```
  % Program in PC2.PE0: Round buffer stores M data
  %                     inputs. Example with M = 8
  movei   R1    256     % Block length counter
  movei   R9    28      % Round buffer start write
loop:                   % R9 mirrors X1
  move    w@X1  R9      % Write last input
  move    r@X1  R9      % Read first input
  loop    start end 7   % Loop next 2 instr.
start:                  % Next is loop start
  addi    R9    R9  4   % Increment address
end:                    % Next is loop end
  modi    r@X1  R9  32  % Modulus and read packet
  subi    R1    R1  1   % Decrement length
  bne     R1    R0  loop % Check length counter
  terminate             % Terminate program
```

The bottleneck is the inner loop (start to end) which implements addition and modulo operation. To increase performance a new processing cell is allocated and the inner loop operation is pipelined, as shown in Figure 14(c). The time multiplexed filter with the pipelined address generation (TMP in figure), significantly reduces execution time and increase utilization of the MAC operation, as shown in Figure 15 and 16, respectively.

Compared to measurements carried out with a standard embedded RISC processor, the speedup offered by ACELRAY is a factor of 10 and 90 for the

TMP and DM, respectively, assuming the same operating frequency. This confirms the assumption that CGRAs are an interesting candidate to increase performance. In addition, the case study illustrated how parallelism in an algorithm can be exploited and how implementations with different characteristics in execution time and resource allocation are supported by the architecture.

8.2 Network Characterization

This section presents simulation results and performance analysis of the proposed hybrid interconnect network. SCENIC is used to configure modules to define a scenario and to reconfigure modules during simulation. Throughput, latency, and other performance metrics are extracted during simulation to evaluate the system.

Experimental Setup

Processing cells are configured to operate as traffic generators, which send random transfers to neighboring cells and to the global network. Transfers are annotated with timestamps to keep track of when transfers were produced and injected into the network. The traffic generators also monitor the incoming traffic, counting the number of received transfers, and calculating the transport latency as the number of clock cycles from successful injection to final consumption. The throughput is measured by counting transfers consumed at the final destination. Since communication is both local and global, a localization factor μ is defined as the ratio between local and global communication, where $\mu = 1$ corresponds to pure local communication and $\mu = 0$ corresponds to fully global communication.

The traffic generators inject transfers into the network according to the Bernoulli process, which is a commonly used injection process to characterize a network. For traffic injected into the global network the traffic pattern is modeled as uniform spatial distribution, which means that every processing cell communicates with every other processing cell with equal probability. *Injection rate*, r, is the number of transfers per clock cycle and per processing cell injected into the local or global network. Throughput is the *accepted traffic*, T, measured in transfers per clock cycle and per processing cell. Ideally, accepted traffic should increase linearly with the injection rate. However, due to traffic contention in the global network the amount of accepted traffic will saturate at a certain level.

Results and Discussion

Figure 17 shows accepted traffic for a network with 2×2 *basic cells* were all routers and links between routers have a capacity of one transfer per clock cy-

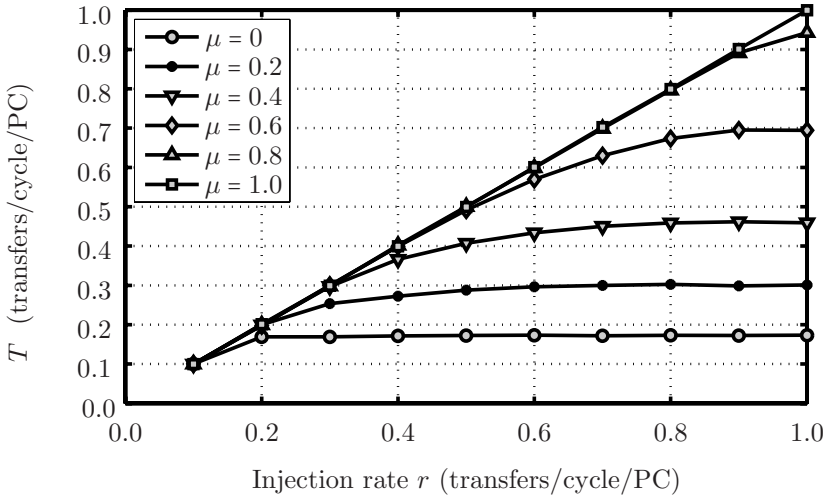


Figure 17: Accepted traffic as a function of injection rate and localization.

cle. Accepted traffic is measured for $0.1 \leq r \leq 1$ and $0 \leq \mu \leq 1$. When $\mu = 1$ there is no traffic contention and the network achieves the optimal linear relationship between injection rate and accepted traffic. When $\mu = 0$, the saturation point for accepted traffic is $T = 0.17$ transfers/cycle/PC. Assuming a realistic localization factor, $\mu = 0.8$, throughput is 94% of the optimal performance.

Figure 18 shows the average transport latency which is measured using the same injection process and traffic pattern as for the throughput measurement. The average transport latency is defined as $L = \sum L_i/N$ where L_i is the transport latency for transfer i and N is the total number of transfers consumed at destination after local or global transport over the interconnect network. Transport latency is measured as the number of clock cycles between successful injection into the network and consumption at the final destination. As shown in Figure 18, the average transport latency saturates at $L \approx 100$ clock cycles for a localization factor $\mu = 0$. The latency is bound by the available buffer capacity inside the network. Since routers use round robin arbitration, a transfer is guaranteed to be delivered within $\sum B_i$ clock cycles, where B_i is the buffer capacity in router i.

In data-driven and streaming applications, latency has small impact on system performance. However, due to random access and feedback loops, the impact of latency can not be omitted in realistic applications. Hence, for these applications latency is an important parameter for the overall performance. With the realistic localization factor $\mu = 0.8$, the average transport latency for

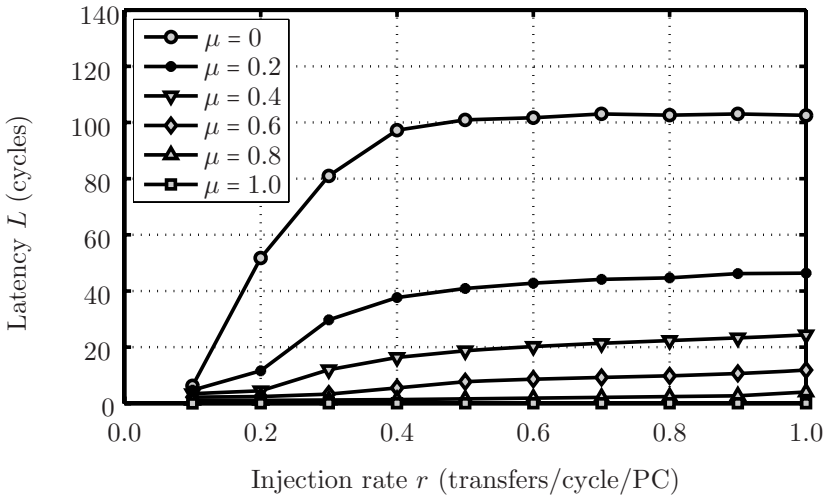


Figure 18: Average transport latency as a function of injection rate and localization.

the proposed network is only 4 clock cycles.

The experiments show the advantages when combining high-performance local communication with a flexible global network for realistic localization factors. This is useful in reconfigurable computing, were the flexibility can be utilize during algorithm mapping to achieve a high localization factor. Functional units with high communication rate are mapped to adjacent processing cells, or to closely located processing cells on the hierarchical network.

8.3 Implementation

The ACELRAY architecture has been translated to VHDL and synthesized with a 0.13 μm technology. Two types of processing elements have been developed: a 32-bit *DSP processing element* with radix-2 butterfly support, and a 16/32-bit *MAC processing element* with multiplication support [34]. The VHDL implementation of these processing elements are based on a customized instruction set, a program memory to hold processor instructions, and configurable number of I/O ports. The configuration for the synthesized design are:

- **Array** – Half of the resource cells are memory cells and the other half are processing cells. The 4×4 array contains 5 routers and the 8×8 array contains 21 routers.

Table 2: Synthesis results for 4×4 and 8×8 array. The results are based on a $0.13\,\mu$m cell library.

Architecture	**Area** (mm^2)	**Frequency** (MHz)	**Storage** (bytes)	**Router overhead**
4x4 Array (5 routers)	2.48	325	10 K	15.8%
8x8 Array (21 routers)	9.18	325	40 K	17.7%

- **Processing Cells** – Half of the processing cells are DSP processors and the other half are MAC processors. Each processing cell contains one of either MAC, or DSP processing element. Each processing element has 8 general purpose registers, 4 I/O ports, and 64 32-bit word program memory.
- **Memory Cells** – Each memory cell contain one bank, each having a storage capacity of 256 32-bit words.
- **Routers** – Each router has an output que of 4 transfers.

Synthesis results for a 4×4 and 8×8 ACELRAY are presented in Table 2. Area estimates and maximum clock frequency are taken from the synthesis tool. The router overhead shows how much of the total area that are spent on routers. It is seen that it is less than 20% for both designs. The estimated area for an 8×8 array is less than 10 mm^2, when synthesized for a maximum clock frequency of 325 MHz.

9 Conclusions

This part has presented the coarse-grained reconfigurable architectures ACELRAY and a SystemC simulation methodology used to explore the design space of coarse-grained reconfigurable platforms. Concepts on hybrid interconnect networks, memory distribution, and data driven communication have been proposed. Experiments show that the hybrid network achieves a high throughput for realistic localization factors. In reconfigurable computing, high localization factors can be achieved by mapping functional units with high communication rate to adjacent processing cells.

Processing cells that contains instruction-set processors with enhanced performance for I/O intensive programs have been proposed. Processors communicate with a data-driven protocol, which uses FIFO buffers to synchronize inter-processor communication. The FIFO buffers are implemented in Memory Cells containing multiple memory banks. Memory banks are global resources that can be allocated by any processing cell to operate as FIFO or RAM. Larger

memories can be emulated by allocating multiple memory banks, or connecting memory banks to external memory. External memory transfers can be either a linear memory access or a more complex stream specified using stride, span, and skip parameters.

A simulation-based exploration methodology based on transaction level models were used to design, evaluate and explore our architecture. The raised abstraction level significantly improve design time and increase simulation speed. The case study, implementing a filter algorithm, illustrated how our design methodology allows an architect to setup scenarios and inspect result and performance data during execution. It also added insight into the flexibility and performance offered by our proposed ACELRAY architecture and how implementations with varying characteristics can be reached.

Part IV

Algorithm and Coprocessor Implementation of a Speech Packet Loss Concealment Method

Abstract

A speech data packet loss concealment algorithm based on pitch period repetition is presented in this part and a novel low complexity method to refine a pitch period estimate is introduced. Objective performance measurements show that this pitch refinement improves the quality of packet loss concealment. Hardware-software co-design techniques have been investigated to implement the algorithm. Using a co-processor approach a processing delay of 0.9 ms and a overall speedup of 3.3 was achieved.

Based on: Henrik Svensson, Viktor Öwall, and Krzysztof Kuchcinski "Implementation Aspects of a Novel Speech Packet Loss Concealment Method," in *Proceedings of IEEE International Symposium on Circuits and Systems*, Kobe, Japan, May 2005, pp 2867–2870.

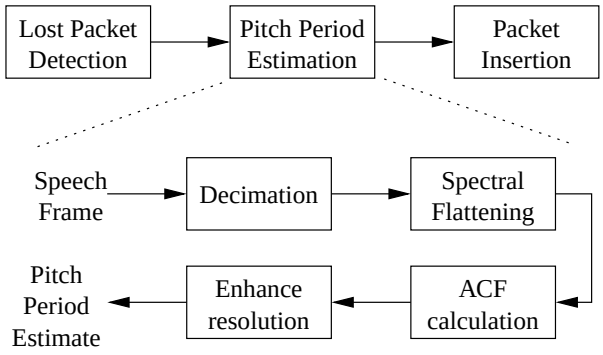


Figure 1: General scheme of a pitch period repetition PLC method.

1 Introduction

Packet based communication is a commonly used method for data transmission in various communication standards. A synchronous connection oriented data transmission such as speech conversation, has an upper bound end-to-end delay. If a packet is delayed beyond this upper bound it is discarded from the network. Hence, the receiver has to replace the missing speech segment, i.e. perform *packet loss concealment* (PLC).

The PLC method presented in this part is based on pitch period repetition. Such a PLC method consists of: detection of lost packet, estimation of pitch period, and packet insertion. The part gives a brief background to pitch estimation and presents a pitch estimation algorithm which includes a novel method to refine the pitch estimate. Hardware-software design space exploration is used to find a co-processor architecture which fulfills a processing delay less than 1 ms for the proposed PLC method.

2 Background

The least complex PLC methods for speech include muting and packet repetition. More advanced methods, which successfully have been used in pulse code modulated (PCM) speech, include pitch period repetition and linear prediction of the missing speech segment [109], [110].

The upper part of Figure 1 shows a general scheme for pitch period repetition PLC methods. Lost packet detection affects both the receiver and transmitter and is an integral part of the standard in which the PLC algorithm should be implemented. Pitch estimation and packet insertion affects only the receiver and do not need to be standardized.

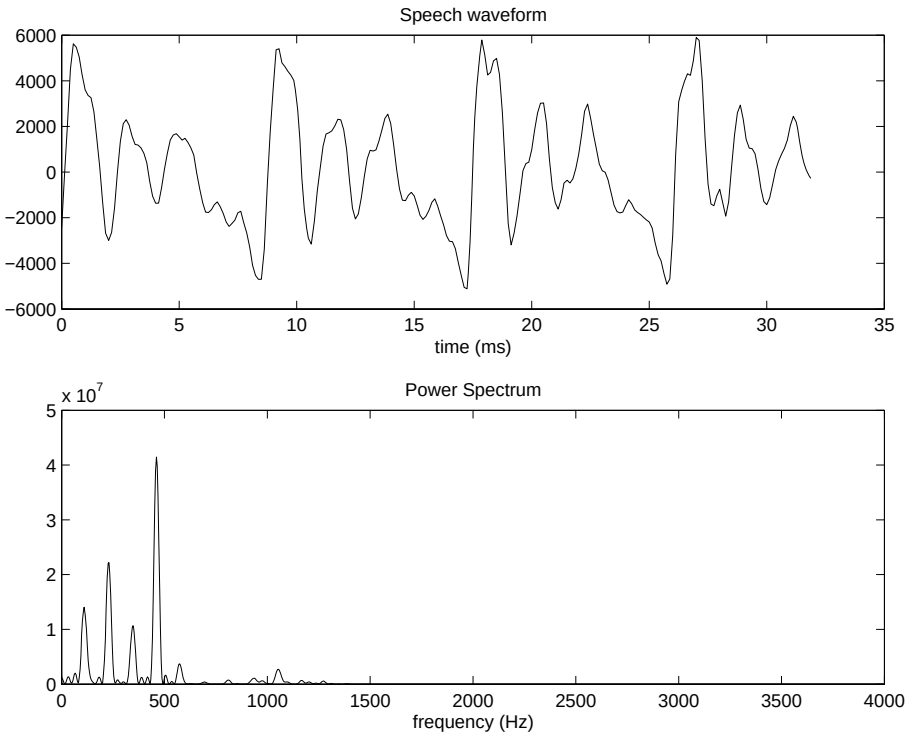


Figure 2: Waveform and Power Spectrum of voiced speech

The human speech production system can be modeled as a excitation signal that is filtered by a Linear Time Invariant filter. The filter in the model corresponds to the cavity with tongue, lips and even the nose in some languages. The excitation signal is either a glottal pulse that resembles a line spectrum or air flow through mouth that resembles a flat spectrum in the frequency domain. Compare this model with a guitar: The string causes the excitation signal and the sound is filtered by the guitar cavity. The type of excitation signal classifies the resulting speech into voiced (line spectrum) and unvoiced (flat spectrum). Compare to utter an "a", which is voiced with an "s" which is unvoiced. When the speech signal is voiced, it has a pitch period (or fundamental frequency) and the signal can be considered piecewise stationary. A PLC algorithm based on pitch repetition uses this to replace lost data with stored data one pitch period back in time.

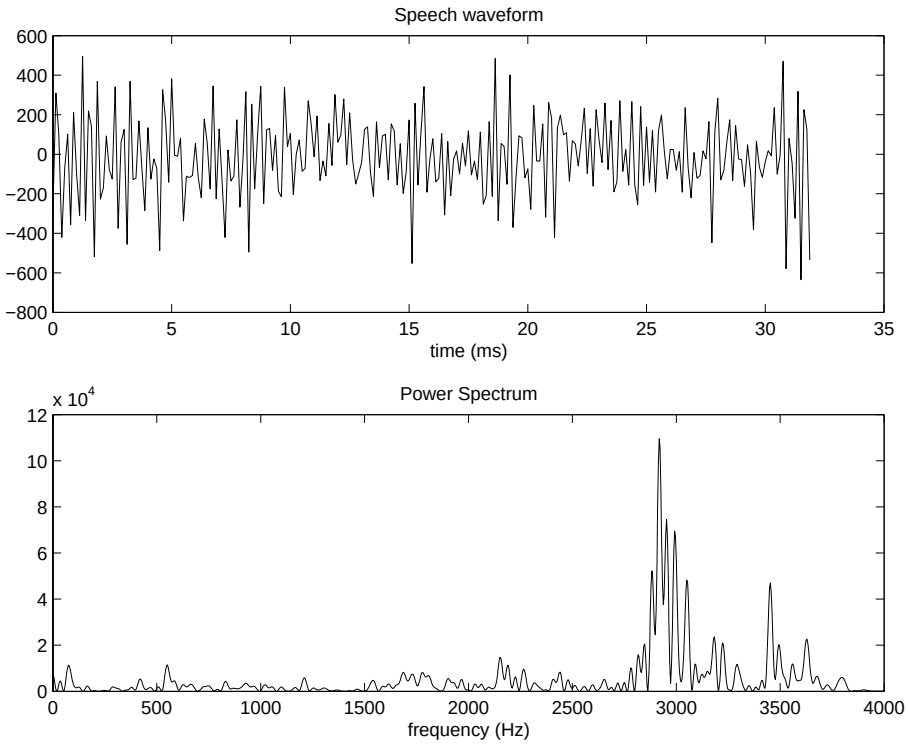


Figure 3: Waveform and Power Spectrum of unvoiced speech

A speech signal containing voiced speech is shown in Figure 2 with the corresponding power spectrum. The periodicity of the signal is clearly seen already in the waveform. The pitch period in this example is about 8 ms. The power spectrum is almost a line spectrum with peaks at even multiples of the fundamental frequency, which is 1/8 ms = 125 Hz.

A speech signal containing unvoiced speech is shown in Figure 3 with the corresponding power spectrum. The waveform has no periodicity and this is reflected in the power spectrum of the signal.

The pitch period is expected to vary between 2 – 20 ms for different speakers. Estimation of the pitch period has been investigated in numerous papers, as it is an important property used in many different applications [111]. A popular method, due to low complexity, straight forward implementation, and good performance is maximizing the autocorrelation function (ACF) [112].

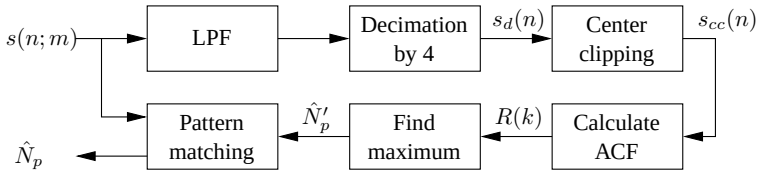


Figure 4: Proposed PPE algorithm.

An overview of commonly used functions in a pitch period estimation (PPE) algorithm based on the ACF is shown in the lower part of Figure 1. There are several steps involved in the algorithm. As the fundamental frequency lies below 400 Hz the signal is low-pass filtered and decimated, in order to reduce the computational complexity of the autocorrelation computation. To compensate for degradation in time resolution of the estimated pitch period due to decimation, the last step is to enhance the pitch period resolution.

After decimation the signal is spectrally flattened in order to reduce spectral peaks at the vocal tract resonance frequencies, because they might cause higher correlation than the fundamental frequency. This is to prevent the first formant frequency, which is often near or even below the fundamental frequency, to cause higher correlation than the fundamental frequency, which would corrupt the detection. Spectral flattening can be done with the inverse filter of the all-pole model as in the SIFT algorithm [113], or by center clipping of the signal, as described in the algorithms [114] and [115].

Last, the autocorrelation function is calculated and the maximum in the interval from minimum and maximum expected pitch period, is taken as the pitch period estimate.

3 Proposed Method

The novelty of the proposed method is that pattern matching is used to enhance the pitch period estimate. A precise pitch estimate is important in a PLC algorithm to diminish discontinuities at the boundaries of the replaced packet. The proposed method for PPE is shown in Figure 4, where pattern matching is the last step.

Samples after decoding at the receiver are denoted as $x(n)$. If a packet is lost at sample m, the content of the analysis frame is denoted as

$$s(n;m) = \begin{cases} x(m-M+n), & 0 \le n \le M-1 \\ 0, & \text{otherwise,} \end{cases} \tag{1}$$

and contains the last M samples of the received speech waveform. Values for

M are in the range of 240–320 samples for a 8 kHz sampling frequency.

As the fundamental frequency lies in the range of 50–400 Hz the signal is first low-pass filtered to remove unwanted information. Moreover, the signal is decimated to reduce the computational complexity in the subsequent processing steps. A low-pass filter with cut-off frequency at 800 Hz and decimation by 4 are proper values for a signal sampled at 8 kHz, as described in [113]. After low-pass filtering and decimation the signal is denoted as $s_d(n)$.

The signal is subjected to spectral flattening using a center-clipping function defined as

$$s_{cc}(n) = \begin{cases} s_d(n) - C_L, & s_d(n) > C_L \\ s_d(n) + C_L, & s_d(n) < -C_L \\ 0, & |s_d(n)| \le C_L, \end{cases} \tag{2}$$

where the center clipping level C_L is calculated according to [111].

The short time autocorrelation function is defined as

$$R(k) = \sum_{n=0}^{M/4-1} s_{cc}(n) s_{cc}(n+k), \quad 0 \le k \le M/4, \tag{3}$$

and the pitch period estimate becomes

$$\hat{N}_{p,D} = \arg \max_{N_{min} \le k \le N_{max}} R(k), \tag{4}$$

where values N_{min} and N_{max}, for a signal decimated to 2 kHz, are 5 and 40, respectively. These values correspond to a pitch period in the range of 2.5–20 ms or the fundamental frequency in the range 50–400 Hz. $\hat{N}_{p,D}$ is the pitch estimate in number of samples in the decimated domain. Hence, multiplication of the estimate with the decimation factor, $D = 4$, yields the pitch period in the 8 kHz domain, denoted as $\hat{N}'_p$.

The time resolution of the estimate is 0.5 ms (2 kHz). This is not sufficient in a PLC algorithm, because it would cause discontinuities at the substitution boundaries. These discontinuities are additionally caused by fine grained pitch errors due to size and position of the analysis frame [112]. Thus, a pitch refinement step based on pattern matching is proposed to cope with these problems. If a packet is lost starting at sample m, the sequence $w(n)$ is defined as

$$w(n) = \begin{cases} s(M - L + n; m), & 0 \le n \le L - 1 \\ 0, & \text{otherwise}, \end{cases} \tag{5}$$

and contains the last L samples of the analysis frame, where $L << M$. The idea is to perform pattern matching between the sequences $w(n)$ and $s(n)$, in order to correct fine grained pitch errors and enhance the time resolution. The

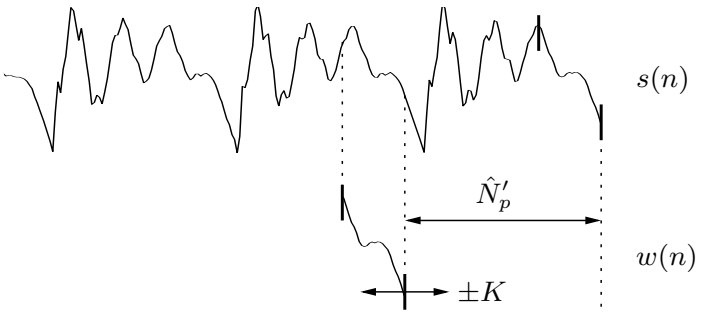


Figure 5: Pattern matching is used to enhance the time resolution and correct fine grained errors in the PPE.

operation is interpreted as sliding the sequence $w(n)$ one pitch period back in sequence $s(n)$, as shown in Figure 5. Mathematically the operation of pattern matching is computed using the mean squared error (mse) denoted as $e^2(k)$.

$$e^2(k) = \sum_{n=0}^{L-1} \{s(M - L - \hat{N}'_p + k + n) - w(n)\}^2. \tag{6}$$

and the refined pitch estimate is written as

$$\hat{N}_p = \hat{N}'_p - \arg \min_{-K \le k \le K} e^2(k). \tag{7}$$

The performance of pitch refinement is highly dependent on L and K. The value L determines the summation period. A small value makes the refinement more sensitive to noise whereas a large value degrades the advantages of fine grained pattern matching. K determines the range around the pitch estimate in which to perform pattern matching. As only fine grained pitch errors should be corrected, the value K is expected to be a few samples.

Evaluation of the pattern matching scheme was performed by comparing the PPE systems in Figure 6 with an objective measurement method. As it was important to demonstrate that the pattern matching scheme, in addition to enhancing time resolution, also was able to correct fine grained errors in the pitch estimate, decimation was excluded in the comparison.

As an objective measurement the average signal-to-noise ratio per missed packet in voiced regions (SNRL) was computed. The signal-to-noise ratio for packet i, lost in a voiced region, is referred to as $\mathrm{SNR_i}$. The $\mathrm{SNR_i}$ is calculated

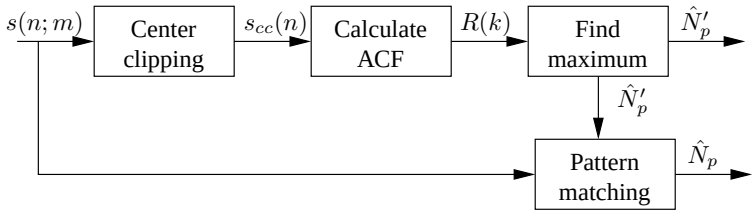


Figure 6: Systems compared to evaluate the performance of pattern matching.

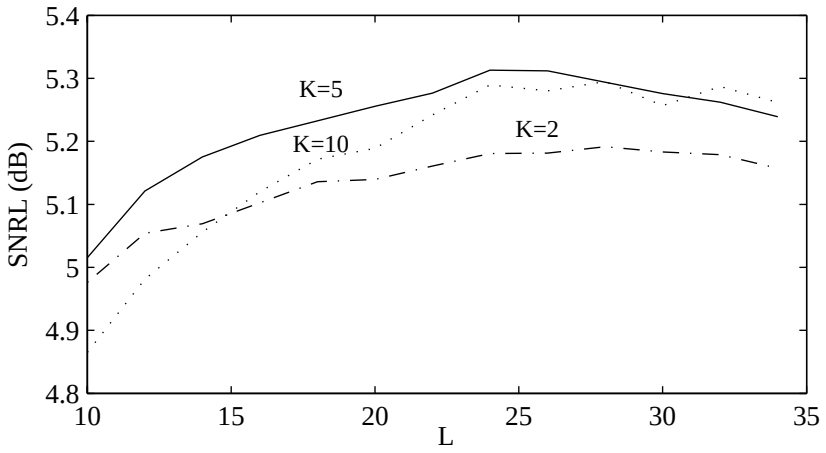


Figure 7: SNRL values plotted for M=240, FER=0.2 during 13 min of speech.

between the replaced packet and the correct packet at the receiver side. SNRL is calculated as the average of the sequence $\mathrm{SNR_i}$.

In the simulations performed, the sampling frequency of the test vectors is 8 kHz, the packet length 30 samples, and approximately 20% of the packets are lost, i.e., the frame erasure rate (FER) is 0.2. The test vectors described in [116] are used as input.

SNRL for 13 minutes of speech with different speakers and background noise was computed for different values of L and K in order to find optimal values for the parameters. Figure 7 shows the result and the optimal values for L and K, using this objective measurement, are 25 and 5, respectively.

Table 1 presents SNRL values for both systems illustrated in Figure 6. Column A presents computed SNRL without pitch refinement and column B presents SNRL with the pitch refinement included. In all test vectors, which contain different speakers and background noise, the SNRL is improved using

Table 1: SNRL for test sequences with (B) and without pitch refinement (A). FER=0.2, M=240, K=5 and L=25

Test	Description	A (dB)	B (dB)
T04	Female, -19.4 dBov	5.9	11
T05	Male, -18.7 dBov	3.4	5.5
T06	Female, -35.0 dBov, ambient noise	4.2	9.8
T09	Female, -35.5 dBov, car noise	6.1	8.0
T12	Male, -34.9 dBov, ambient noise	3.1	5.4
T15	Male, -34.1 dBov, babble noise	4.7	7.0

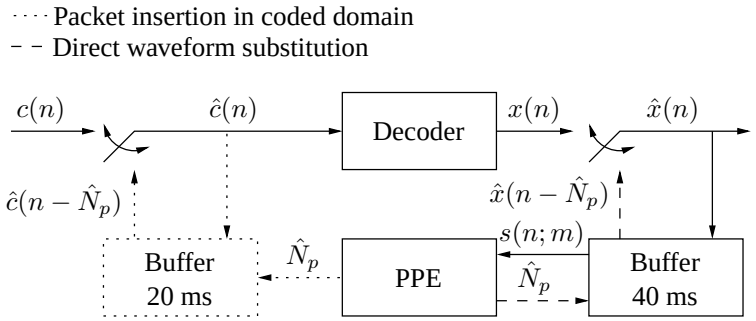


Figure 8: Packet insertion can be done in the coded domain or as direct waveform substitution.

pattern matching to enhance the pitch estimate. Hence, the pattern matching scheme is able to correct fine grained errors in the pitch period estimate and additionally enhance the time resolution.

The strategy for packet insertion depends on the specific codec in the system where the PLC algorithm is implemented. Two strategies are distinguished: direct waveform substitution and substitution in the coded domain, as shown in Figure 8. The waveform substitution technique is straightforward. If the segment is classified as unvoiced, the samples from the last packet are repeated, otherwise pitch period repetition in the decoded domain is carried out.

The second strategy is to perform replacement in the coded domain. Whether this is a feasible solution depends on the specific codec in use. Some decoders have an internal state. Hence, if the decoder is bypassed, as in direct waveform substitution, the state will not be updated and the decoder will behave

Table 2: Profiling results

Function	Clock cycles	Execution time @ 24 MHz
LPF and Decimation	29600	1.234 ms
Center clipping	6900	0.288 ms
ACF and Maximize	32400	1.350 ms
Pattern matching	4800	0.200 ms
All	73700	3.070 ms

erroneously when the first correctly received packet arrives. The concept of replacement in the coded domain was verified with the continuous variable slope delta modulation decoder. Subjective listening tests indicated that replacement in the coded domain improved quality compared to direct waveform substitution.

4 Implementation

The processing delay of the algorithm must be kept to a minimum since it is added to the delay budget even though no packets are lost. As the PPE is only performed once, for a burst of lost packets, the worst case scenario is that every second packet is lost. This result in a processing deadline less than the duration of two packets. A packet that contains 30 samples at 8 kHz has a processing deadline less than 7.5 ms. In order to make the solution attractive, the implementation targets a processing delay less than 1 ms.

The parameters described in Section III are selected as: $M = 320$, $N_{min} = 5$, $N_{max} = 40$, $L = 25$, $K = 5$ and the packet length 30 samples for a 8 kHz signal. The low-pass filter is a linear phase FIR filter of length 31, with 10 dB attenuation at 1 kHz.

The considered target architecture contains an embedded RISC processor with load-store architecture, running at a 24 MHz clock frequency. The algorithm was first developed using floating-point operations, for simulation purposes. Next, the algorithm was transformed to ANSI C 16-bit fixed-point operations with 32-bit accumulate. Table 2 shows the profiling results for the algorithm mapped to the embedded RISC processor. It is assumed that code and data reside in the on-chip memory. Penalty due to cache misses is not considered. It is concluded that the proposed pattern matching scheme only uses 6.5 % of the total execution time.

A computation time of 3.1 ms, as indicated in Table 2, does not meet the

Table 3: Speedup using accelerator

Function	Speedup Arch. A	Speedup Arch. B
LPF and Decimation	1.4	5.5
ACF and Maximize	2.3	6.1
Overall	1.6	3.3

desired 1 ms in processing delay. Acceleration of the algorithm with a factor of 3 is not possible to achieve with software techniques. Two different combined hardware-software techniques were evaluated to accelerate the algorithm. As the decimation and autocorrelation accounts for almost 85 % of the execution time, these functions were investigated for acceleration. The common factor for these functions are that they are both based on multiply-accumulate (MAC).

The considered processor offered an interface, where up to 8 point-to-point links, with FIFO based communication, could be added. The links can send and receive data, from and to, the processors general purpose registers using blocking or non-blocking communication. These links were used as interface to a co-processor.

The architecture investigated first was to stream data from the processor to a 16-bit MAC co-processor, as shown in Figure 9 as Architecture A. After performing loop unrolling, the inner loop code that describes the MAC operations are removed and replaced by instructions to stream data to the co-processor. As both the ACF and filter computation have linear memory access in the direct form, two 16-bit words can be fetched from the memory and sent to the co-processor in two instructions. The switches in Figure 9 alternate to provide the datapath with data in correct order. The speedups achieved using this solution are presented in Table 3 as Architecture A. The overall achieved speedup was not found sufficient for the application. If the processor had supported data to be moved directly from processor memory to the co-processor links, the speedups would have been 3.3 and 2.1 for the ACF maximization and filtering and decimation, respectively.

The second investigated architecture was to add cache memory to the co-processor, as shown in Figure 9 as Architecture B. One memory with 80 16-bit words is sufficient to hold data used for the ACF calculation. 80 16-bit words is also enough during the filtering and decimation to hold the filter coefficients and the 31 16-bit words needed to calculate one filter output. This solution adds extra hardware as the control logic in Architecture A must be extended. For the ACF calculation both operands are fed to the MAC from the internal

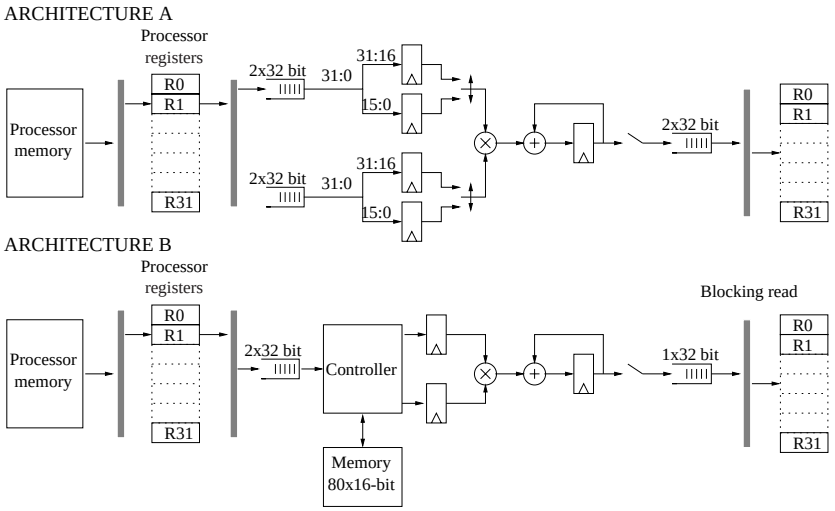


Figure 9: Hardware acceleration.

memory and the processor fetches the output from the last FIFO by a blocking read operation. During the filtering function the processor must download 4 16-bit data and fetch one filter output through a blocking read operation to calculate one filter output. The speedups achieved using this solution are presented in Table 3, as Architecture B. The overall algorithm speedup using this solution is 3.3. This gives a computation time of 0.93 ms, which meets the 1 ms processing delay constraint.

5 Conclusions

A speech packet loss concealment method based on pitch period repetition has been presented in this part. A novel technique with low complexity has been proposed to refine a pitch estimate. Experiments using an objective performance measure show that the proposed method is able to correct fine grained errors in the pitch estimate and enhance time resolution.

Hardware-software co-design techniques have been used to explore the design space for the implementation. Two different co-processor architectures have been investigated to accelerate the algorithm. An algorithm speedup of 3.3 and a processing delay of 0.9 ms was achieved.

Conclusion

Coarse-grained reconfigurable architectures are an emerging technique for programmable acceleration of computation intensive processing kernels with word-level arithmetic. In this thesis, it has been shown that a coarse-grained reconfigurable architecture that consists of 12 processing elements, 3 memory banks, and a coprocessor communication mechanism is able to reduce the number of clock cycles for a speech codec by 83% as compared to instruction set processor execution. This was accomplished by accelerating less than 25 functions that were responsible for almost 90% of the execution time on the targeted instruction set processor. As automatic mapping tools mature, bottlenecks encountered with manual mapping are reduced and larger fractions of applications may be accelerated.

System level evaluation of reconfigurable platforms is necessary to predict performance, and to find and cure bottlenecks prior to fabrication. A SystemC simulation methodology with high abstraction level and automated approaches to build, simulate, evaluate, and tune models, was presented. The high abstraction level improved design time and increased simulation speed. Furthermore, automated approaches enabled efficient architectural exploration. A set of flexible transaction level models were developed to allow complete embedded systems with instruction set processors, buses, and memories to be evaluated together with the reconfigurable architecture.

Parameterizable models of reconfigurable architectures are required in order to understand the design parameters and their relationship to performance and area. This thesis presented a reconfigurable processor array and demonstrated its design parameters. Concepts on processing cells, hybrid interconnect networks, memory distribution, and data driven communication, were proposed. Processing cells built as processors with enhanced performance for I/O intensive programs were suggested. Memory cells were proposed as global resources that could be allocated by any processing cell to operate as FIFOs or RAMs. The hybrid interconnect network consists of dedicated local links and a global hierarchical network. Experiments showed that the hybrid network achieved a high throughput for realistic localization factors.

Bibliography

[1] G. Estrin, "Organization of computer systems: The fixed plus variable structure computer," in *Proceedings of Western Joint Computer Conference*, 1960, pp. 33–40.

[2] ——, "Reconfigurable computer origins: The UCLA fixed-plus-variable (F+V) structure computer," *IEEE Annals of the History of Computing*, vol. 24, no. 4, pp. 3–9, 2002.

[3] T. Glökler and H. Meyr, *Design of Energy-Efficient Application-Specific Instruction Set Processors (ASIPs)*. Kluwer Academic Publisher, 2004.

[4] A. Abnous and J. Rabaey, "Ultra-low-power domain-specific multimedia processors," in *Proceedings of Workshop on VLSI Signal Processing*, 1996, pp. 461–470.

[5] A. Chandrakasan, S. Sheng, and R. Brodersen, "Low power techniques for portable real-time DSP applications," in *Proceedings of International Conference on VLSI Design*, 1992, pp. 203–208.

[6] A. Chandrakasan and R. Brodersen, "Minimizing power consumption in digital CMOS circuits," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 498–523, 1995.

[7] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–211, 2002.

[8] (2008) ARM embedded cores. [Online]. Available: http://www.arm.com/

[9] (2008) MIPS. [Online]. Available: http://www.mips.com/

[10] (2008) PowerPC embedded cores. [Online]. Available: http://www.ibm.com/

[11] (2008) ARM DSP Extensions. [Online]. Available: http://www.arm.com/

[12] (2008) MIPS DSP ASE. [Online]. Available: http://www.mips.com/

[13] H. Shi, "RISC+SIMD=DSP?" in *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, 2000, pp. 3211–3214.

[14] A. Iranpour and K. Kuchcinski, "Evaluation of SIMD architecture enhancement in embedded processors for MPEG-4," in *Proceedings of Symposium on Digital System Design*, 2004, pp. 262–269.

[15] (2008) AMD processors. [Online]. Available: http://www.amd.com/

[16] O. Silven and K. Jyrkk, "Observations on power-efficiency trends in mobile communication devices," *EURASIP Journal on Embedded Systems*, vol. 2007, 2007.

[17] F. Gruian, "Energy-Centric Scheduling for Real-Time Systems," Ph.D. dissertation, Lund University, Department of Computer Science, 2002.

[18] (2008) Pocket Guide to Processing Engines for DSP. [Online]. Available: http://www.bdti.com/

[19] (2008) DSP Selection Guide. [Online]. Available: http://www.ti.com/

[20] (2008) C6000 High Performance DSPs. [Online]. Available: http://www.ti.com/

[21] J. van Eijndhoven, F. Sijstermans, K. Vissers, E. Pol, M. Tromp, P. Struik, R. Bloks, P. van der Wolf, A. Pimentel, and H. Vranken, "TriMedia CPU64 architecture," in *Proceedings of International Conference on Computer Design*, 1999, pp. 586–592.

[22] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors, Scheduling and Synchronization.* Signal Processing and Communications Series, 2000.

[23] N. Clark, H. Zhong, and S. Mahlke, "Automated custom instruction generation for domain-specific processor acceleration." *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1258–1271, 2005.

[24] K. Kuchcinski and C. Wolinski, "Global approach to assignment and scheduling of complex behaviors based on HCDG and constraint programming," *Journal of Systems Architecture*, vol. 49, no. 12-15, pp. 489–503, 2003.

[25] C. Wolinksi and K. Kuchcinski, "Computation patterns identification for instruction set extensions implemented as reconfigurable hardware," in *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2007, pp. 175–181.

[26] C. Wolinski and K. Kuchcinski, "Identification of application specific instructions based on sub-graph isomorphism constraints," in *Proceedings of IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2007, pp. 328–333.

[27] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2003.

[28] C. Kozyrakis, "Scalable Vector Media-processors for Embedded Systems," Ph.D. dissertation, University of California Berkeley, Computer Science Division, 2002.

[29] C. Kozyrakis and D. Patterson, "Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, 2002, pp. 283–293.

[30] ——, "Overcoming the limitations of conventional vector processors," in *Proceedings of International Symposium on Computer Architecture*, 2003, pp. 399–409.

[31] A. Berkeman, "ASIC Implementation of a Delayless Acoustic Echo Canceller," Ph.D. dissertation, 2002.

[32] (2008) CatapultC. [Online]. Available: http://www.mentor.com/

[33] K. Kuchcinski, "Constraints-driven design space exploration for distributed embedded systems," *Journal of Systems Architecture*, vol. 47, no. 3-4, pp. 241–261, 2001.

[34] T. Lenart, "Design of Reconfigurable Hardware Architectures," Ph.D. dissertation, Lund University, Department of Electrical and Information Technology, 2008.

[35] T. Lenart, M. Gustafsson, and V. Öwall, "A hardware acceleration platform for digital holographic imaging," *Journal of VLSI Signal Processing Systems*, vol. 52, no. 3, pp. 297–311, 2008.

[36] M. Kamuf, "Trellis Decoding From Algorithm to Flexible Architectures," Ph.D. dissertation, Lund University, Department of Electroscience, 2007.

[37] P. P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh, "Performance Evaluation and Design Trade-Offs for Network-on-Chip Interconnect Architectures," *IEEE Transactions on Computers*, vol. 54, no. 8, pp. 1025–1040, Aug. 2005.

[38] T. Todman, G. Constantinides, S. Wilton, O. Mencer, W. Luk, and P. Cheung, "Reconfigurable computing: Architectures and design methods," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 2, pp. 193–207, 2005.

[39] O. Albaharna, P. Cheung, and T. Clarke, "Virtual hardware and the limits of computational speed-up," in *Proceedings of Circuits and Systems*, vol. 4, 1994, pp. 159–162.

[40] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A quantitative analysis of the speedup factors of FPGAs," in *Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2004, pp. 162–170.

[41] 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*, vol. 30, no. 9, pp. 86–93, 1997.

[42] P. Lysaght and P. Subrahmanyam, "Guest editors' introduction: Advances in configurable computing," *IEEE Design & Test of Computers*, vol. 22, no. 2, pp. 85–89, 2005.

[43] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proceedings of International Conference on Field-programmable Logic and Applications*, 2003, pp. 61–70.

[44] V. Baumgarte, G. Ehlers, F. May, A. Nckel, M. Vorbach, and M. Weinhardt, "PACT XPP: A self-reconfigurable data processing architecture," *The Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, 2003.

[45] S. Khawam, I. Nousias, M. Milward, Y. Ying, M. Muir, and T. Arslan, "The reconfigurable instruction cell array," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 16, no. 1, pp. 75–85, 2008.

[46] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The reconfigurable streaming vector processor (RSVP)," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, 2003, pp. 141–150.

[47] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "PipeRench: A reconfigurable architecture and compiler." *IEEE Transactions on Computers*, vol. 33, no. 4, pp. 70–78, 2000.

[48] J. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, 1997, pp. 12–21.

[49] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "PipeRench: A coprocessor for streaming multimedia acceleration," in *Proceedings of International Symposium on Computer Architecture*, 1999, pp. 28–39.

[50] P. Chow, S. O. Seo, J. Rose, K. Chung, G. Paez-Monzon, and I. Rahardja, "The design of an SRAM-based field-programmable gate array. i. architecture," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 2, pp. 191–197, 1999.

[51] A. Alsolaim, J. Becker, M. Glesner, and J. Starzyk, "Architecture and application of a dynamically reconfigurable hardware array for future mobile communication systems," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000, pp. 205–214.

[52] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD - reconfigurable pipelined datapath," in *Proceedings of International Workshop on Field-Programmable Logic and Applications*, 1996, pp. 126–135.

[53] M. Zabel, S. Kohler, M. Zimmerling, T. Preusser, and R. Spallek, "Design space exploration of coarse-grain reconfigurable DSPs," in *Proceedings of International Conference on Reconfigurable Computing and FPGAs*, 2005, pp. 15–22.

[54] T. Miyamori and K. Olukotun, "REMARC: Reconfigurable multimedia array coprocessor," in *Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 1998, pp. 261–272.

[55] (2008) XPP-III Processor Core. [Online]. Available: http://www.pactxpp.com/

[56] E. Mirsky and A. DeHon, "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, 1996, pp. 157–166.

[57] A. Marshall, T. Stansfield, J. Kostarnov, V. J., and B. Hutchings, "A reconfigurable arithmetic array for multimedia applications," in *Proceedings of ACM/SIGDA Symposium on Field programmable gate arrays*, 1999, pp. 135–143.

[58] G. Panesar, D. Towner, A. Duller, A. Gray, and W. Robbins, "Deterministic parallel processing," *International Journal of Parallel Programming*, vol. 34, no. 4, pp. 323–341, 2006.

[59] D. Kissler, A. Kupriyanov, F. Hannig, and J. Teich, "A highly parameterizable parallel processor array architecture," in *Proceedings of IEEE International Conference on Field Programmable Technology*, 2006, pp. 105–112.

[60] (2008) Am2000 Family Massively Parallel Processor Array. [Online]. Available: http://www.ambric.com/

[61] Mathstar, "Field Programmable Object Array," http://www.mathstar.com.

[62] J. Rabaey, "Reconfigurable processing: The solution to low-power programmable DSP," in *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 1, 1997, pp. 275–278.

[63] H. Singh, L. Ming-Hau, L. Guangming, F. Kurdahi, N. Bagherzadeh, and F. E. Chaves, "MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[64] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Proceedings of Design, Automation and Test in Europe*, 2001, pp. 642–649.

[65] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*. Kluwer Academic Publisher, 2000.

[66] A. Rose, S. Swan, J. Pierce, and J.-M. Fernandez, *Transaction Level Modeling in SystemC*, OSCI-TLM WG, 2006.

[67] L. Cai and D. Gajski, "Transaction level modeling: An overview," in *Proceedings of Hardware/Software Codesign and System Synthesis*, 2003, pp. 19–24.

[68] S. Swan, "SystemC transaction level models and RTL verification," in *Proceedings of Design Automation Conference*, 2006, pp. 90–92.

[69] W. Klingauf, "Systematic transaction level modeling of embedded systems with SystemC," in *Proceedings of Design, Automation and Test in Europe*, 2005, pp. 566–567.

[70] C. Shin, P. Grun, N. Romdhane, C. Lennard, G. Madl, S. Pasricha, N. Dutt, and M. Noll, "Enabling heterogeneous cycle-based and event-driven simulation in a design flow integrated using the SPIRIT consortium specifications," *Journal on Design Automation for Embedded Systems*, vol. 11, no. 2, pp. 119–140, 2007.

[71] D. C. Black and J. Donovan, *SystemC: From the Ground Up*. Kluwer Academic Publisher, 2004.

[72] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer Academic Publisher, 2002.

[73] Open SystemC Initiative (OSCI), "OSCI SystemC 2.2 Open-source Library," http://www.systemc.org.

[74] A. Donlin, "Transaction level modeling: Flows and use models," 2004, pp. 75–80.

[75] W. Klingauf, R. Gunzel, O. Bringmann, P. Parfuntseu, and M. Burton, "GreenBus: A generic interconnect fabric for transaction level modelling," in *Proceedings of Design Automation Conference*, 2006, pp. 905–910.

[76] (2008) SystemC OCP Models. [Online]. Available: http://www.ocpip.org/

[77] IBM Systems and Technology Group, *128-Bit Processor Local Bus Architecture Specifications Version 4.7*, IBM, 2007.

[78] W. Wolf, "Hardware-software co-design of embedded systems," *Proceedings of the IEEE*, vol. 82, no. 7, pp. 967–989, 1994.

[79] S. Carta, D. Pani, and L. Raffo, "Reconfigurable coprocessor for multimedia application domain," *The Journal of VLSI Signal Processing*, vol. 44, no. 1, pp. 135–152, 2006.

[80] R. Enzler, C. Plessl, and M. Platzner, "System-level performance evaluation of reconfigurable processors." *Microprocessors and Microsystems*, vol. 29, no. 2-3, pp. 63–74, 2005.

[81] *Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s*, Std. Version 03/96, March 19 1996.

[82] H. Safizadeh, H. Noori, M. Sedighi, A. Jahanian, and N. Zolfaghari, "Efficient host-independent coprocessor architecture for speech coding algorithms," in *Proceedings of Euromicro Conference on Digital System Design*, 2005, pp. 227–230.

[83] V. Chouliaras and J. Nunez, "Scalar coprocessors for accelerating the G723.1 and G729A speech coders," *IEEE Transactions on Consumer Electronics*, vol. 49, no. 3, pp. 703–711, 2003.

[84] *ITU-T Software Tool Library 2000 User's Manual*, Std. Version 1.0, February 24 2001.

[85] S. Mishra and A. Balaram, "Efficient hardware-software co-design for the G.723.1 algorithm targeted at VoIP applications," in *Proceedings of IEEE International Conference on Multimedia and Expo*, vol. 3, 2000, pp. 1379–1382.

[86] A. Langi, "Rapid development of a real-time speech coder on a TMS320C54x DSP," in *Proceedings of Canadian Conference on Electrical and Computer Engineering*, 2002, pp. 1045–1048.

[87] Thomas J. Dillon, *G.723.1 Dual-Rate Speech Coder: Multichannel TMS320C62x Implementation*, Texas Instruments, 2000.

[88] S. Carta and L. Raffo, "Processing time saving in low power voice coding applications using synchronous reconfigurable co-processing architecture," in *Proceedings of IEEE ICECS*, vol. 2, 2002, pp. 529–532.

[89] B. Mathew and A. Davis, "A loop accelerator for low power embedded VLIW processors," in *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis*, 2004, pp. 6–11.

[90] E. Ramo, J. Resano, D. Mozos, and F. Catthoor, "A configuration memory hierarchy for fast reconfiguration with reduced energy consumption overhead," in *Proceedings of International Parallel and Distributed Processing Symposium*, 2006, pp. 1–8.

[91] (2008) PowerPC 405 Processor. [Online]. Available: http://www.xilinx.com/

[92] (2008) Embedded Development Kit. [Online]. Available: http://www.xilinx.com/

[93] *IEEE Standard SystemC Language Reference Manual*, Std. IEEE Std 1666-2005, March 31 2006.

[94] Open SystemC Initiative (OSCI), "OSCI SystemC 2.2 Open-source Library," http://www.systemc.org.

[95] A. V. Brito, M. Kühnle, M. Hübner, J. Becker, and E. U. K. Melcher, "Modelling and simulation of dynamic and partially reconfigurable systems using SystemC," in *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, 2007, pp. 35–40.

[96] Computer Systems Laboratory, University of Campinas , "The ArchC Architecture Description Language," http://archc.sourceforge.net/.

[97] The ArchC Team, *The ArchC Architecture Description Language v2.0*, Computer Systems Laboratory, University of Campinas , 2007.

[98] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros, "The ArchC architecture description language and tools," *International Journal of Parallel Programming*, vol. 33, no. 5, pp. 453–484, 2005.

[99] A. Baldassin, P. Centoducatte, and S. Rigo, "Extending the ArchC language for automatic generation of assemblers," in *Proceedings of International Symposium on Computer Architecture and High Performance Computing*, 2005, pp. 60–68.

[100] Open Core Protocol International Partnership, "An Initiative towards Open Network-on-Chip Benchmarks," http://www.ocpip.org.

[101] (2008) The GNU Profiler. [Online]. Available: http://www.gnu.org/

[102] L. Bononi and N. Concer, "Simulation and Analysis of Network on Chip Architectures: Ring, Spidergon and 2D Mesh," in *Proceeding of IEEE Conference on Design, Automation and Test in Europe*, 2006, pp. 154–159.

[103] J. Chan and S. Parameswaran, "NoCGen: A template based reuse methodology for networks on chip architectures," in *Proceeding of IEEE Conference on VLSI Design*, 2004, pp. 717–720.

[104] A. Hegedus, G. M. Maggio, and L. Kocarev, "A ns-2 simulator utilizing chaotic maps for network-on-chip traffic analysis," in *Proceedings of IEEE International Symposium on Circuits and Systems*, 2005, pp. 3375–3378.

[105] A. Kupriyanov, D. Kissler, F. Hannig, and J. Teich, "Efficient event-driven simulation of parallel processor architectures," in *Proceedings of International Workshop on Software and Compilers for Embedded Systems*, 2007, pp. 71–80.

[106] T. Rissa, A. Donlin, and W. Luk, "Evaluation of SystemC modelling of reconfigurable embedded systems," in *Proceedings of Design, Automation and Test in Europe*, 2005, pp. 253–258.

[107] X. Ningyi, L. Xianglun, L. Renfei, and Z. Zucheng, "A SystemC-based NoC Simulation Framework supporting Heterogeneous Communicators," in *Proceedings of IEEE International Conference on ASIC*, 2005, pp. 1032–1035.

[108] A. Portero, R. Pla, and J. Carrabina, "SystemC Implementation of a NoC," in *Proceedings of IEEE International Conference on Industrial Technology*, 2005, pp. 1132–1135.

[109] D. Goodman, O. Jaffe, G. Lockhart, and W. Wong, "Waveform substitution techniques for recovering missing speech segments in packet voice communications," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 34, no. 6, pp. 1440–1448, Dec. 1986.

[110] E. Gündüzhan and K. Momtahan, "A linear prediction based packet loss concealment algorithm for PCM coded speech," *IEEE Transactions on Speech and Audio Processing*, vol. 9, no. 8, pp. 778–785, Nov. 2001.

[111] L. Rabiner, M. Cheng, A. Rosenberg, and C. McGonegal, "A comparative performance study of several pitch detection algorithms," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 24, no. 5, pp. 399–418, Oct. 1976.

[112] L. R. Rabiner, "On the use of autocorrelation analysis for pitch detection," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 25, no. 1, pp. 24–33, Feb. 1977.

[113] J. D. Markel, "The SIFT algorithm for fundamental frequency estimation," *IEEE Transactions on Audio and Electroacoustics*, vol. 20, no. 5, pp. 367–377, Dec. 1972.

[114] J. Dubnowski, R. Schafer, and L. Rabiner, "Real-time digital hardware pitch detector," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 24, no. 1, pp. 2–8, Feb. 1976.

[115] M. M. Sondhi, "New methods of pitch extraction," *IEEE Transactions on Audio and Electroacoustics*, vol. 16, no. 2, pp. 262–266, Aug. 1968.

[116] *Adaptive Multi-Rate (AMR), speech codec test sequences*, Std. Release 5, 2002-06.