

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

**MODELING AND MAPPING FOR DYNAMICALLY RECONFIGURABLE
HYBRID ARCHITECTURES**

by

Kiran Kumar Bondalapati

A Dissertation Presented to the
**FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA**

In Partial Fulfillment of the
Requirements for the Degree
**DOCTOR OF PHILOSOPHY
(COMPUTER ENGINEERING)**

August 2001

Copyright 2001

Kiran Kumar Bondalapati

UMI Number: 3054713

**Copyright 2001 by
Bondalapati, Kiran Kumar**

All rights reserved.

UMI[®]

UMI Microform 3054713

**Copyright 2002 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.**

**ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346**


UNIVERSITY OF SOUTHERN CALIFORNIA
THE GRADUATE SCHOOL
UNIVERSITY PARK
LOS ANGELES, CALIFORNIA 90007

This dissertation, written by

KIRAN K. BONDALAPATI

.....
*under the direction of his..... Dissertation
Committee, and approved by all its members,
has been presented to and accepted by The
Graduate School, in partial fulfillment of re-
quirements for the degree of*

DOCTOR OF PHILOSOPHY



.....
Dean of Graduate Studies

Date May 11, 2001

DISSERTATION COMMITTEE

V. N. Prasad
.....
Chairperson

[Signature]
.....

[Signature]
.....

Dedication

To My Grandfather M.V. Ramaiah

Acknowledgments

I would like to thank Dr. Viktor Prasanna, my advisor at USC, for his guidance, encouragement, support, and vision throughout my PhD program. In addition to academic guidance, he has been a motivating force in the development of my professional skills and perspective of research. In the long road to my dissertation he provided me excellent direction and moral support when I faltered in my approach. I understood the process of abstracting the underlying principles and applying problem solving techniques to multiple domains wholly due to his teaching. I really appreciate the flexibility and understanding Dr. Prasanna has shown towards helping me balance my professional and personal responsibilities. His contribution towards my professional and career growth goes far beyond technical and academic advisement. I have been extremely fortunate to have him as my advisor and to have known him as a person over the last few years.

I also thank the members of my qualifying examination and defense committees, Dr. Peter Beerel, Dr. Michel Dubois, Dr. Mary Hall, and Dr. Ulrich Neuman for their suggestions.

It has been a wonderful experience to participate in collaborative research efforts with several of my fellow graduate students during my Ph.D. program. I have shared

thoughts on research and other things over daily lunches with Andreas Dandalis, who has been an excellent sounding board for all my ideas. I thank the other founding members of the MAARC research group Seonil Choi and Reetinder Sidhu for making the joint effort a success. Through the years, other students of Pgroup have made my Ph.D. study an enriching experience - Ammar Alhusaini, Amol Bakshi, Prashant Bhat, Myungho Lee, Young Won Lim, Wenheng Liu, Vaibhav Mathur, Sumit Mohanty, Neungsoo Park, Michael Penner, Cauligi Raghavendra, Mark Redekopp, Mitali Singh, Jinwoo Suh, among others. Henryk Chrostek and Christine Contreras have been instrumental in making my non-academic affairs run smoothly throughout my stay.

I thank the Department of Defense and National Science Foundation for supporting our research and providing opportunities to frequently visit and interact with active researchers all over the world.

I thank my parents Prasad and Bharathi, my sister Dr. Lalitha, brother-in-law Dr. Narendra and nephew Monish for being extremely encouraging and patient with my seemingly never ending academic pursuits for the last twenty years, culminating in my Ph.D. Throughout my Ph.D. study, several of my friends provided the needed encouragement for working and diversion from academics as appropriate. There are one too many to name, but it would have been tough to get through the long haul without them. Last, but not the least, I would like to thank my undergraduate teachers and notably Prof. P.C.P. Bhatt, my undergraduate advisor, for instigating in me the scientific curiosity to pursue a doctoral degree.

Contents

Dedication	ii
Acknowledgments	iii
List of Figures	viii
List of Tables	x
Abstract	1
1 Introduction	3
1.1 Thesis Contributions	9
1.1.1 Hybrid System Architecture Model (HySAM)	9
1.1.2 Mapping Techniques	10
1.1.3 Simulation Framework	12
1.2 Thesis Outline	12
2 Reconfigurable Computing	14
2.1 Classification	15
2.2 Field Programmable Gate Array Systems	19
2.2.1 SPLASH	22
2.2.2 Xilinx 6200	23
2.2.3 NSC CLAY	24
2.2.4 Xilinx Virtex	25
2.2.5 Dynamically Programmable Gate Array (DPGA)	26
2.3 Hybrid Architectures	27
2.3.1 Convergence in Hybrid Architectures	28
2.3.2 OneChip	29
2.3.3 Berkeley Garp	30
2.3.4 National Adaptive Processing Architecture (NAPA)	30
2.3.5 Xilinx Platform FPGA	31
2.3.6 Triscend	33
2.3.7 Chameleon Systems RCP	35

3	Approach	41
3.1	Challenges	42
3.1.1	Static vs. Dynamic Reconfiguration	42
3.1.2	Design Methodologies	44
3.1.3	Multi-dimensional Optimization	46
3.1.4	Design Tools	48
3.2	Approach	51
3.2.1	Model based Reconfigurable Computing	53
3.2.2	Loop Computations	55
3.2.3	Definitions	57
4	Related Work	59
5	Hybrid System Architecture Model (HySAM)	65
5.1	Hybrid System Architecture Model (HySAM)	67
5.2	Functions and Configurations	69
5.3	Attributes	71
5.4	Memory Access	72
5.5	Reconfiguration and Configuration Cache	73
5.6	Generative Aspect	75
5.6.1	Generators	76
5.6.2	Reconfiguration Cost	78
5.7	Execution Model	79
6	Mapping Techniques	81
6.1	Generic Mapping Problem (GMP)	82
6.1.1	NP-Completeness	83
6.2	Loop Synthesis	84
6.2.1	Linear Loop Synthesis	85
6.2.2	Linear Loop Mapping Problem	86
6.2.3	Optimal Solution	86
6.2.4	Illustrative Example	91
6.2.5	Mapping Configurations onto Multiple Contexts	93
6.2.6	Multicontext Loop Mapping Problem	95
6.3	Dynamic Precision Management	98
6.3.1	Overview of Dynamic Precision Variation	101
6.3.2	Precision Requirement Analysis	103
6.3.3	Precision Variation Curve	105
6.3.4	Theoretical Analysis of Loops	105
6.3.5	Run-time Analysis	108
6.3.6	Dynamic Precision Management	110
6.3.7	Precision Management Problem (PMP)	111

6.3.8	Precision Management Algorithms	115
6.3.9	An Illustrative Example	117
6.3.10	Application	120
7	Mapping onto Reconfigurable Pipelines	122
7.1	Mapping Nested Loops	123
7.1.1	Parallelizing Nested DSP Loops	125
7.1.2	Pipelining	127
7.1.3	Limitations on the Throughput	129
7.1.4	Data Context Switching (DCS)	131
7.1.5	DSP/Microprocessor Implementations	136
7.1.6	Performance Summary	137
7.1.7	Performance Results	137
7.2	Reconfiguring Pipelines	140
7.2.1	Definitions	141
7.2.2	Pre-processing and Mapping	143
7.2.3	Partitioning	144
7.2.4	Routing Considerations	145
7.2.5	Pipeline Segmentation	147
7.2.6	Performance Results	148
8	DRIVE Simulation Framework	152
8.1	Motivation	155
8.2	DRIVE Overview	157
8.3	Other Simulation Tools	159
8.4	DRIVE Framework Implementation	160
8.5	Visualization	165
8.6	DRIVE Summary	167
9	Conclusions and Future Directions	169
9.1	Future Directions	174
	References	178

List of Figures

1.1	International Technology Roadmap for Semiconductors	7
1.2	Makimoto's wave	8
2.1	Typical FPGA Board, Device and Logic block architecture	20
2.2	DPGA architecture and composition	26
2.3	Triscend A7 CSoC architecture block diagram	34
2.4	Chameleon Reconfigurable Communications Processor (RCP) Architecture	35
2.5	Reconfigurable Processing Fabric (RPF)	37
2.6	Chameleon software tool flow	40
3.1	Static configurable computing	43
3.2	Dynamic configurable computing	43
3.3	Traditional Design Synthesis Approach	45
3.4	Constraint space of optimization in reconfigurable architectures	47
3.5	Multi-dimensional configurable architecture characteristics	48
3.6	Chameleon design tools flow	50
3.7	Model-based Approach	54
5.1	Hybrid System Architecture Model and example architecture	68
5.2	Example Reconfiguration	74
5.3	Two different compositions	79
5.4	HySAM execution model	80
6.1	Example reconfiguration costs and optimal configuration sequence	88
6.2	Overview of our approach for dynamic precision management in loops(shaded and rounded regions indicate our contributions)	102
6.3	Example code for simulations	106
6.4	Precision Variation Curves for <i>RSQ</i> using theoretical and run-time analysis	108
6.5	Multiplication operation from sample code	117
7.1	Mapping of loop body to one stage	128
7.2	Pipelined datapath of all ten stages	129

7.3	Dataflow in original design	132
7.4	Dataflow in dynamic context switching	132
7.5	Optimized datapath for one stage	135
7.6	(a) N-body simulation task DAG and (b) FFT task DAG with partition numbers	146
7.7	Algorithm to generate the pipeline segments	149
8.1	DRIVE framework	157
8.2	Major components in the DRIVE framework and the information flow	161
8.3	Sample DRIVE visualization	166

List of Tables

5.1	HySAM model parameters and definitions	69
6.1	Representative Model Parameters for Garp Architecture	92
6.2	Theoretical and simulated iteration numbers for $N = 1024$	117
6.3	HySAM model parameters for XC6200 multiplier configurations	118
6.4	Execution times using different approaches	120
7.1	Analytical performance summary	137
7.2	Platform characteristics	138
7.3	Performance Results and Speedups	138
7.4	Schedules for N-body simulation (a) S_0 : Greedy Scheduling (b) S_I : Schedule after segmentation	148
7.5	Schedules for FFT (a) S_0 : Greedy Scheduling (b) S_I : Schedule after segmentation	148
7.6	Virtex module characteristics	150
7.7	Reconfiguration costs in number of Virtex slices	150

Abstract

Reconfigurable computing is a new paradigm based on dynamically adapting the hardware to reconfigure the computation and communication structures on the chip. Reconfigurable circuits and systems have evolved from application specific accelerators to a general purpose computing paradigm. Various reconfigurable devices have been developed by researchers and the industry. These devices promise a high degree of flexibility and superior performance. But, the algorithmic techniques and software tools are also heavily based on the hardware paradigm from which they have evolved.

This thesis addresses the fundamental challenges in achieving high performance using reconfigurable architectures. The diverse range of issues in mapping applications onto reconfigurable architectures are identified. A formal framework for mapping application tasks onto reconfigurable architectures is proposed in this thesis. The proposed framework includes a parameterized system level model, algorithmic mapping techniques and system level interpretive simulation environment.

A parameterized model of hybrid reconfigurable architectures, Hybrid System Architecture Model (HySAM), is developed to facilitate application mapping. Hybrid reconfigurable architectures include traditional processing units and memory on the same

die as reconfigurable logic. The parameterized abstract model, HySAM, is general enough to capture a wide range of configurable systems. Loop statements in traditional programs consist of regular, repetitive computations which are the most likely candidates for performance enhancement using configurable hardware. This thesis develops a formal methodology for mapping loops onto reconfigurable architectures. The abstract model is used to define and solve the problem of mapping loop statements onto reconfigurable architectures. The complexity of the problems and our proposed solutions is also addressed. Performance improvements are achieved on various architectures using our algorithmic techniques for mapping. In addition, existing design and simulation tools do not include the reconfiguration aspect in their methodology. A simulation methodology for reconfigurable architectures is proposed and validated by implementing a proof of concept tool. The Dynamically Reconfigurable systems Interpretive simulation and Visualization Environment (DRIVE) facilitates high level performance evaluation framework for design space exploration.

Chapter 1

Introduction

It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in 5 years.

– John Von Neumann (ca. 1949)

Microprocessors are at the heart of most current high performance computing platforms. They provide a flexible computing platform and are capable of executing large class of applications. Software for microprocessors is developed by implementing higher level functions using the instruction set of the architecture. As a result, the same fixed hardware can be used for many general purpose applications. Unfortunately, this generality is achieved at the expense of performance. The software program stored in memory has to be fetched, decoded and executed. In addition, data is fetched from and stored back into memory. These conditions force explicit sequentialization in the execution of

the program. Casting all complex functions into simpler instructions to be executed sequentially on the processor results in degraded performance.

Application Specific Integrated Circuits (ASICs) provide an alternate solution which addresses the performance issues of general purpose microprocessors. ASICs are designed for a specific application and hence, each ASIC has fixed functionality and superior performance for a highly restricted set of applications. However, ASICs restrict the flexibility of the architecture and exclude any post-design optimizations and upgrades in features and algorithms.

A new computing paradigm using *reconfigurable computing* promises an intermediate trade-off between flexibility and performance. Reconfigurable computing utilizes hardware that can be adapted at run-time to facilitate greater flexibility without compromising performance. Reconfigurable architectures can exploit fine grain and coarse grain parallelism available in the application because of the adaptability. Exploiting this parallelism provides significant performance advantages compared to conventional microprocessors. The reconfigurability of the hardware permits adaptation of the hardware for specific computations in each application to achieve higher performance compared to software. Complex functions can be mapped onto the architecture achieving higher silicon utilization and reducing the instruction fetch and execute bottleneck.

Reconfigurable logic¹ can be defined to consist of matrix of programmable computational units with a programmable interconnection network superimposed on the computational matrix. The basic differences of reconfigurable logic compared with traditional processing include:

- *Spatial Computation*: The data is processed by spatially distributing the computations rather than temporally sequencing through a shared computational unit.
- *Configurable Datapath*: The functionality of the computational units and the interconnection network can be adapted at run-time by using a configuration mechanism.
- *Distributed Control*: The computational units process data based on local configuration rather than an instruction broadcast to all the functional units.
- *Distributed Resources*: The required resources for computation, such as computational units and memory are distributed throughout the device instead of being localized in a single location.

The spatial distribution of the computations and the distributed control and resources result in higher computational power efficiency for reconfigurable computing compared to microprocessors, DSPs and ASICs. Computational power efficiency is defined as

¹The distinction between configurable and reconfigurable logic is fuzzy at best. Some definitions of *configurable* restrict it to logic that can be programmed one time and used for computation. Such devices do not have the non-recurring engineering (NRE) costs associated with ASICs. *Reconfigurable* logic is defined to be a device that can be reprogrammed at run-time, in between computations, in the field. In this thesis, we use *adaptive*, *configurable* and *reconfigurable* to mean *reconfigurable* at run-time.

the number of the number of gates actively working in a clock cycle to solve a problem to the total number of gates in a device. In traditional architectures like microprocessors and DSPs large portion of the chip is utilized to support active computation in a much smaller portion of the chip. Reconfigurable computing can have significantly higher computational power efficiency compared with conventional microprocessors and ASICs [54].

Reconfigurable architectures have evolved from Field Programmable Gate Arrays (FPGAs) [65]. FPGAs consist of a matrix of logic blocks and interconnection network. The functionality of the logic blocks and the connections in the interconnection network can be modified by downloading bits of configuration data onto the hardware. Currently, hybrid architectures which integrate programmable logic and interconnect together with a microprocessor on the same chip are being developed. On-chip integration of reconfigurable logic reduces the memory access costs and the reconfiguration costs. The availability of increasingly larger number of transistors [2] (see Figure 1.1) facilitates the integration of reconfigurable logic with other components on system on a chip (SoC) architectures.

Applications are mapped onto reconfigurable architectures by analyzing the computations performed. Computations that can be speeded up by using reconfigurable hardware are identified and mapped onto the reconfigurable hardware. In the presence of a microprocessor, the computations which have complex control and data structures are

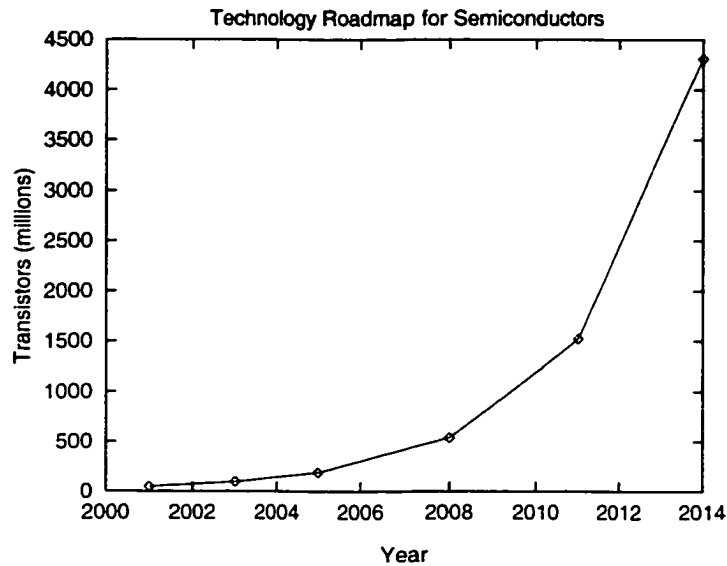


Figure 1.1: International Technology Roadmap for Semiconductors

executed on the microprocessor. The partitioning of the computations of an application between the microprocessor and the reconfigurable hardware is performed manually or by using automatic/semi-automatic tools. The partitioned computations have to be compiled into executable code on the microprocessor and hardware configurations on the reconfigurable hardware. The reconfigurable hardware needs to be configured using the configuration information before the actual execution can be performed. This configuration can be updated at run-time to execute a different set of computations from the application.

Development of systematic scheduling and mapping techniques for computing architectures require high level abstractions. Computing models, which are high level abstractions of the architectures, can be utilized to develop algorithmic techniques for mapping applications onto the architectures. Reconfigurable computing is different from the von-Neumann paradigm of computing and requires computational models different

from conventional models. This gives rise to a design crisis in the tools for mapping applications onto reconfigurable architectures. Makimoto predicts this changing paradigm in the wave illustrating technology trends in semiconductors (see Figure 1.2).

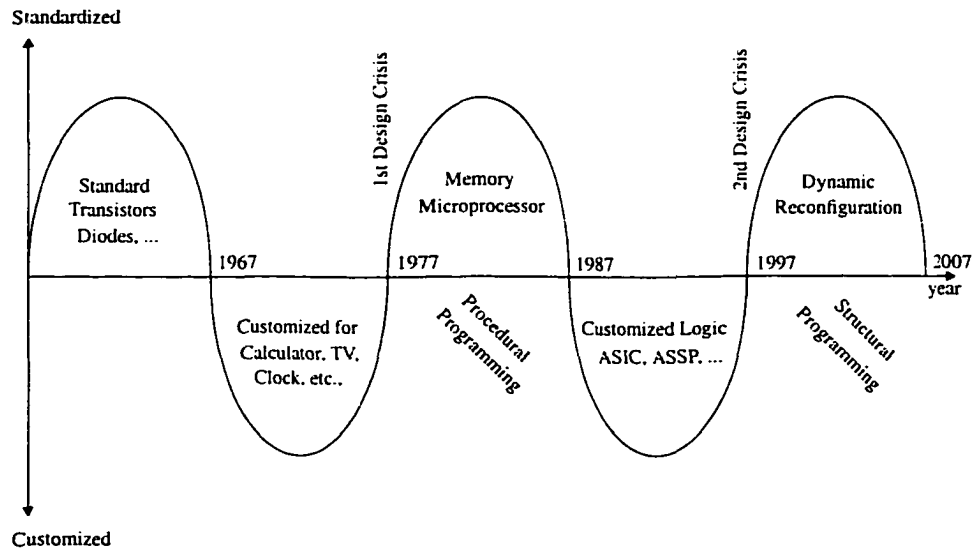


Figure 1.2: Makimoto's wave

There are several application areas where reconfigurable computing has been shown to achieve significant performance. These include long multiplication [76], cryptography [30, 76], genetic algorithms [39, 70], image processing [3, 25], genomic database search [46], signal processing [29, 58, 69]. The nature and diversity of the reconfigurable architectures results in a wide variety of implementation issues with respect to applications.

This thesis addresses the basic issues involved in the design process for facilitating reconfigurable computing. We identify fundamental challenges in developing a framework for effective design and mapping techniques. A formal framework based on a

parameterized model of hybrid reconfigurable architectures is the focus of this thesis. A formal approach is a critical requirement as accepted by many researchers, but addressed by very few. This thesis defines generic problems and presents solutions for mapping application tasks onto reconfigurable architectures. A simulation methodology for high level performance evaluation is also proposed and a prototype tool is illustrated. We list the basic contributions of the thesis and the thesis outline in the following sections.

1.1 Thesis Contributions

This dissertation addresses several important issues in developing a formal framework for mapping applications onto reconfigurable architectures. Our focus is on developing algorithmic techniques that can enable automatic mapping tools including compilation frameworks for current and future reconfigurable architectures. This thesis is one of the earliest efforts to develop a parameterized model of reconfigurable architectures and mapping techniques. The contributions of this thesis include:

1.1.1 Hybrid System Architecture Model (HySAM)

A high level model of reconfigurable hardware is needed to abstract the low level details. Existing models supplied by the CAD tools have either multiple abstraction layers or are

very device specific. We developed Hybrid System Architecture Model (HySAM), a parameterized model of a configurable computing system, which consists of configurable logic attached to a traditional microprocessor. This model can be utilized for developing the actual mapping and scheduling of these tasks onto the configurable system. Our model cleanly partitions the *capabilities* of the hardware from the *implementations* and presents a very clean interface to the user.

1.1.2 Mapping Techniques

The main focus of this thesis is in developing mapping techniques that use the HySAM model to map application tasks onto reconfigurable architectures. The mapping problems that we address focus on loops in application tasks which constitute a large fraction of the execution time of most applications. The contributions of the thesis in this area include:

- Defining a generic mapping problem based on the HySAM model that identifies the architectural and application constraints and the optimization criteria. We address the complexity of the problem by showing that this generic problem is NP-complete.
- Developing polynomial time algorithms to address a subset of the application loops which have linear dependencies between tasks. Our algorithmic techniques are based on dynamic programming and result in optimal total execution times for the loops.

- Developing algorithms for dynamic precision variation for loops on reconfigurable architectures. Our work is the first in this research area to develop algorithmic techniques to exploit dynamic reconfiguration to improve the performance of applications on reconfigurable architectures.
- Developing techniques to map computations onto high performance reconfigurable pipelines to exploit architectural features of reconfigurable architectures. Heuristic algorithms are developed to reduce the reconfiguration overheads in the presence of resource constraints.
- Developing techniques for parallelizing applications by using data context switching. Data context switching treats each execution of a repetitive computation as a context. Instead of switching the configuration, the data on which the datapath is operating on is changed every cycle dynamically. This technique can parallelize loop computations that cannot be parallelized using existing techniques on conventional architectures.
- Optimization techniques to simultaneously exploit multiple dimensions of reconfigurable architectures. The thesis addresses the issues in developing mapping techniques which exploit multiple aspects of the reconfigurable logic to deliver superior performance compared to traditional techniques. Our techniques address various application and architectural characteristics and resource limitations in developing mapping techniques to optimize application performance.

1.1.3 Simulation Framework

As part of this thesis we proposed a novel interpretive simulation and visualization environment based on modeling and module level mapping approach. The **Dynamically Reconfigurable systems Interpretive simulation and Visualization Environment (DRIVE)** can be utilized as a vehicle to study the system and application design space and performance analysis. *Interpretive* simulation measures the performance of the abstract application tasks on the parameterized abstract system model. This is in contrast to simulating the exact behavior of the hardware by using HDL models of the hardware devices. A prototype of the framework was developed in Java exploiting novel features of the language.

1.2 Thesis Outline

Chapter 2 presents the background for the thesis. The evolution and the state of reconfigurable architectures are discussed in detail. The features of several architectural implementations are discussed to motivate the issues that need to be addressed in developing mapping techniques.

Chapter 3 outlines the challenges in reconfigurable computing and our approach to address some of the challenges. The paradigm shift from a fixed hardware to a reconfigurable hardware gives rise to challenges in abstraction, algorithms and design tools. We also introduce the basic definitions that are used throughout the thesis.

Chapter 4 describes some work by researchers and industry in addressing issues related to those in this thesis.

Chapter 5 describes in detail our Hybrid System Architecture Model (HySAM). The various components of the architecture and the abstraction of the architecture and the application by the model parameters are explained in detail. This model is used inherently in all the mapping techniques presented in the later chapters of this thesis.

Chapter 6 defines mapping problems and develops solutions for mapping applications onto reconfigurable architectures. Several variations of the mapping problems are defined and their complexity is addressed in this chapter. The developed techniques are used to illustrate how dynamic precision variation can be used to exploit reconfigurable architectures.

Chapter 7 illustrates algorithmic techniques for mapping applications onto reconfigurable pipelines by addressing several resource and application constraints that exist in reconfigurable computing. This chapter focuses on developing high performance reconfigurable pipelined datapath to map applications onto reconfigurable architectures.

Chapter 8 describes in detail the motivations and our philosophy in developing the interpretive simulation framework: **Dynamically Reconfigurable systems Interpretive simulation and Visualization Environment (DRIVE)**.

Chapter 9 discusses the conclusions from this dissertation work and addresses directions for future research based on the evolution of reconfigurable architectures.

Chapter 2

Reconfigurable Computing

They always say that time changes things, but you actually have to change them yourself.

- Andy Warhol

Reconfigurable architectures have evolved from Field Programmable Gate Arrays (FPGAs). Currently, there are a large class of FPGAs available commercially. Various computing systems have been constructed by integrating multiple FPGAs and dedicated memory. Some systems also couple a general purpose microprocessor or an ASIC such as a Digital Signal Processor (DSP) to the FPGAs. To alleviate the communication and memory access bottleneck for configuration and data, future systems are integrating configurable logic onto the same chip as a microprocessor. Such hybrid architectures can distribute the computations between different components of the system. We give a brief overview of the different classes of architectures in this chapter.

2.1 Classification

A large number of reconfigurable architectures have been developed over the years by researchers and the industry. Reconfigurable architectures can be classified based on several different parameters. In this section, we list some of the most distinguishing architectural parameters which can be used to classify reconfigurable architectures. Examples of each type of architectures are discussed in detail in the following sections.

- **Granularity**

The granularity of the reconfigurable logic is the size of the smallest functional unit that is addressed by the mapping tools. Typically FPGAs have smaller granularity such as 2-input and 4-input functional units. Several reconfigurable architectures such as Chameleon implement coarse grain arithmetic units of larger size such as 32 bits.

Lower granularity provides more flexibility in adapting the hardware to the computation structure. But, lower granularity has a performance penalty due to larger delays when constructing computation modules of larger size using smaller functional units. Some architectures implement features that are specifically targeted towards reducing these overheads. For example, some FPGAs implement fast carry chains to permit construction of larger arithmetic modules from small functional units.

- **Host Coupling**

A large fraction of reconfigurable logic is utilized as a processing fabric attached to a host processor. The host processor performs the control functions to configure the logic, schedule data input and output, external interfacing, among other things. The type of coupling to such a host system dictates the overheads in utilizing reconfigurable logic to speed-up computations. The degree of coupling affects the reconfiguration and the data access costs.

The degree of coupling can be roughly partitioned into three classes:

- *Loose System-level Coupling*: This includes architectures which have reconfigurable logic communicating to the host through an I/O interface similar to a disk drive and other peripherals. A large number of initial FPGA based boards were architected with this degree of coupling. Such architectures include

SPLASH [19].
- *Loose Chip-level Coupling*: These systems reduce the overheads in communicating to the host by using direct communication between the host and the reconfigurable logic. An example of such an architecture is the PRISM [4].

A large number of existing embedded architectures with reconfigurable logic are architected using this technique.

– *Tight On-chip Coupling*: The availability of large number of transistors has resulted in the integration of reconfigurable logic on the same chip as a host processor, significantly reducing the communication overheads between different components of the architecture. Such architectures include Garp [41], Chameleon [72], among others.

- **Reconfiguration Methodology**

Typically, a reconfigurable device is configured by downloading a sequence of bits known as a *bitstream* onto the device. The speed and methodology of download depend on the interface supported by the device. Two possible interfaces are bit-serial and bit-parallel interface. The time for configuration is directly proportional to the size of the bit-stream. Fine-grain and coarse-grain devices have difference in the configuration time because coarse grain devices typically need smaller configuration bitstreams.

The flexibility of reconfiguration is achieved at the cost of reconfiguration cost. Reconfigurable logic has to stop computation for initiating a new configuration. This reconfiguration time can be significant, especially for fine-grain multi-million gate FPGAs. Some architectures support *partial* and *dynamic* reconfiguration [82]. Partial reconfiguration permits reconfiguration of the functionality of a portion the device while the remaining portion retains its functionality. Dynamic reconfiguration permits reconfiguration of a portion of the device while other portions of the device are performing computations.

Some other architectures address this problem by utilizing multiple contexts [67] or a reconfiguration cache [41, 72]. Both are similar in principle. Some configurations of the device can be stored in buffers (possibly on-chip memory). At runtime, it is less expensive to switch to one of the configurations in these memory buffers compared with loading a new configuration from external memory. The organization of the cache varies among the architectures. Some architectures implement the architecture as an external memory, whereas some architectures have distributed context memories. For example, Chameleon [72] RCP has a cache holding one configuration on-chip which permits single cycle reconfiguration.

- **Memory Organization**

The computation performed on the reconfigurable logic needs to access data from memory. Intermediate results from computations also need to be stored before the logic can be reconfigured to perform the next computation. The organization of the memory affects the data access cost and is a significant fraction of the actual execution time. Currently, most reconfigurable architectures include large memory on the reconfigurable logic device. This memory can be implemented as large blocks of memory (Virtex BlockRAMs [83]) or as distributed memory blocks (Chameleon LSMs [72]).

2.2 Field Programmable Gate Array Systems

A Field Programmable Gate Array consists of an array of combinational logic blocks overlaid with an interconnection network of wires (See Figure 2.1). Both the logic blocks and the interconnection network are configurable. The configurability is achieved by using either anti-fuse elements or SRAM memory bits to control the configurations of transistors. Anti-fuse technology utilizes strong electric currents to create a connection between two terminals and is typically less reprogrammable. SRAM based configuration can be reprogrammed on the fly by downloading different configuration bits into the SRAM memory cells.

Typical logic block architectures contain a look-up table, a flip-flop, additional combinational logic and SRAM memory cells to control the configuration of the logic block (See Figure 2.1). The logic blocks at the periphery of the device also perform the I/O operations. The interconnection network can be reconfigured by changing the connections between the logic blocks and the wires and by configuring the switch boxes which connect different wires. The switch boxes for the interconnection network are also controlled by SRAM memory cells. The functions computed in the logic block, the interconnection network and the I/O blocks can be configured using external data. FPGAs typically permit unlimited reconfiguration. These versatile devices have been used to build processors and coprocessors whose internal architecture as well as interconnections can be configured to match the needs of a given application. For a detailed architectural survey of FPGAs and related systems, see [18, 40, 65].

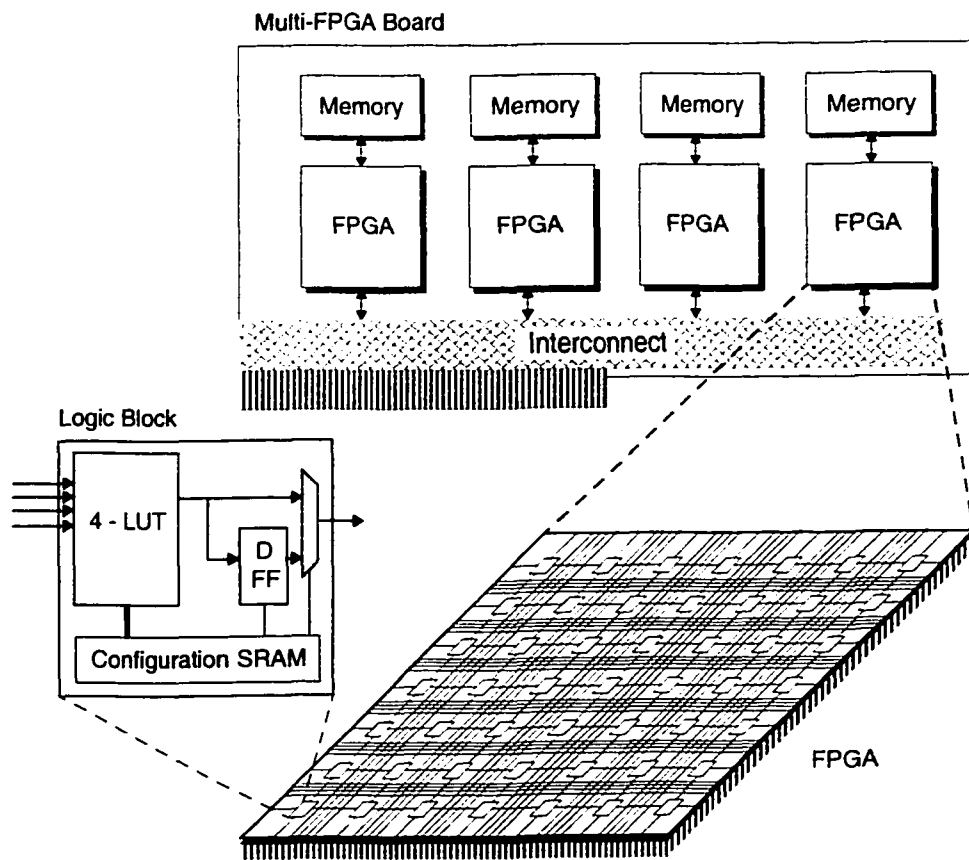


Figure 2.1: Typical FPGA Board, Device and Logic block architecture

Current and future generation reconfigurable devices ameliorate the reconfiguration cost by providing *partial* and *dynamic* reconfigurability [32, 41, 67, 70, 83]. In *partial* reconfiguration, it is possible to modify the configuration of a part of the device while the configuration of the remaining part is retained. In *dynamic* reconfiguration devices permit this partial reconfiguration even while other logic blocks are performing computations. Devices in which multiple contexts of the configuration of a logic block can be stored in the logic block and the context switched dynamically have also been proposed [32, 67].

Typically, the application requirements increase at a rate faster than the increase in the size of logic resources on most FPGA devices. FPGA architectures also have limits on the I/O capability due to the limitation on the number of I/O pins on the device. To map large applications onto configurable logic, various systems have been designed which have several FPGAs on a board. These architectures also provide local memory and dedicated or programmable interconnect between the FPGAs. These board level architectures are usually designed to function under an external controller or use one of the on-board FPGAs as a controller. Examples of such systems include the experimental DECPeRLe board [76], SPLASH-2 [19] and Teramac [1] and the commercial WILD series from Annapolis Microsystems [56]. Some software tools exist which can automatically partition the design between multiple FPGAs on a board using higher level abstractions [42]. For a detailed overview of FPGA devices and multi-FPGA architectures see [40].

2.2.1 SPLASH

SPLASH and SPLASH 2 [19], its successor, can be conceptually viewed as a system of linear array of processing elements. This architecture makes SPLASH 2 a good candidate for systolic applications with limited neighbor-to-neighbor interconnect. Applications which are not linear systolic are possible by utilizing a crossbar interconnecting the FPGAs. The system consists of a board with 16 Xilinx 4010 chips (plus one for control) arranged in a linear systolic array and a crossbar switch which allows establishment of a limited number of non-linear communication channels. Each chip has a 36-bit connection to its two nearest neighbors. Each Xilinx 4010 is connected to a 512 KB memory(16-bit word size). Up to 16 boards can be daisy chained together to provide a large linear-systolic array of 256 elements. The whole system is connected to a Sparc processor using the Sbus system bus interface. The Sparc processor acts as the host and controls the initialization and configuration of the system.

SPLASH has been one of the extensively utilized system for developing applications which have shown enormous speed-up over traditional microprocessors and even supercomputers. Though the application domain is usually restricted to systolic array style of computation, there are several critical applications which satisfy this criteria. SPLASH has been used for Image Processing, Motion Detection, DNA sequence matching among others [19]. On DNA sequence matching SPLASH achieved over 300x the performance of a Cray-II and over 200x performance of a 16K processor CM-2.

2.2.2 Xilinx 6200

The XC6200 FPGA [82] architecture from Xilinx is the first SRAM based FPGA architecture designed for implementing reconfigurable coprocessors. The XC6200 architecture features a fine-grained cell structure, abundant routing, built-in processor interface and supports fast partial reconfiguration.

The programmable logic of an XC6200 consists of large array of reconfigurable logic cells each of which contains both programmable logic and routing resources. Each cell contains a flip-flop and combinatorial logic capable of implementing any two-input function or any type of 2-to-1 multiplexer. Cells are arranged in 4-by-4 blocks and 16-by-16 tiles. The interconnection network consists of a hierarchy of programmable routing wires. Each cell can be used for logic or memory functions. When cells are configured as memory, each cell provides two bytes of ROM or RAM memory which can be accessed externally or internally.

The most important feature in the new XC6200 device is the FastMap interface, designed to connect directly to an external processor's system bus. The FastMap interface places the whole FPGA into the processors address space. The processor can read and write the logic and the configuration memory by using normal load and store. This parallel interface allows the entire configuration memory to be programmed in under 100 micro-seconds.

A random access feature allows arbitrary areas of the FPGA memory to be changed. This provides a fast partial reconfiguration capability. This partial reconfiguration can

be performed without disturbing circuits running in other parts of the device. This facilitates sharing of hardware space by swapping in and out designs at runtime. A reconfigurable hardware platform based on the XC6200 architecture has been designed and is being offered as a commercial product by Virtual Computer Corporation [27].

2.2.3 NSC CLAy

The National Semiconductor CLAy [68] architecture is an SRAM based Configurable Logic Array. CLAy was designed to support real-time algorithm and logic sharing by using dynamic partial reconfiguration.

The logic cell layout is similar to existing FPGA devices, with a flip-flop and 5-input lookup tables. The interconnection network is made up of nearest neighbor connections, local and express bus wires. The full device can be configured in 640 micro-seconds. Larger designs are supported by an integrated Field Configurable Multi-Chip Module (FCMCM) which consists of a 2×2 array of CLAy devices.

CLAy supports partial reconfiguration by which a single cell's functionality can be changed. This is much faster than programming the complete device and reconfiguration time is the order of 1 micro-second. This partial reconfiguration can be done without functional interruption of the remaining parts of the device. These features of the CLAy devices have been exploited in designing novel applications [80].

2.2.4 Xilinx Virtex

Virtex is one of the latest in a series of high-performance FPGAs from Xilinx [83]. It has different versions which have capacities ranging from 50 thousand to 1 million system gates. Virtex architecture comprises of an array of Configurable Logic Blocks (CLBs), encircled by programmable I/O blocks, and dedicated block memories of 4096 bits each. The is a hierarchical routing matrix with local routing and varying number of global routes of different lengths. There are 24 single length routes, 96 routes of length six and 12 long lines spanning the chip. There are additional I/O routing resources around the periphery of the logic blocks.

The CLBs contain four logic cells each. Each logic cell has a 4-input function generator (Look Up Table - LUT), a flip-flop and some carry logic. The LUTs can operate as function generators or they can be used as distributed RAMs. Additional multiplexors and wires in a CLB provide flexible combination of different logic cell outputs and routing of input signals to CLB output. High speed arithmetic is facilitated by providing additional carry logic in each of the logic cells. A dedicated AND gate in each logic cell improves multiplier implementations.

On-chip local memories can be realized on the Virtex architecture in two different ways. The logic cells can be combined and configured as memory cells to obtain multi-ported RAM of required sized. Each Virtex also has large Block SelectRAM memories. These are organized along the two vertical edges of the FPGA. Each memory block is four CLBs high and the number of such blocks is as much as 32 for large size Virtex

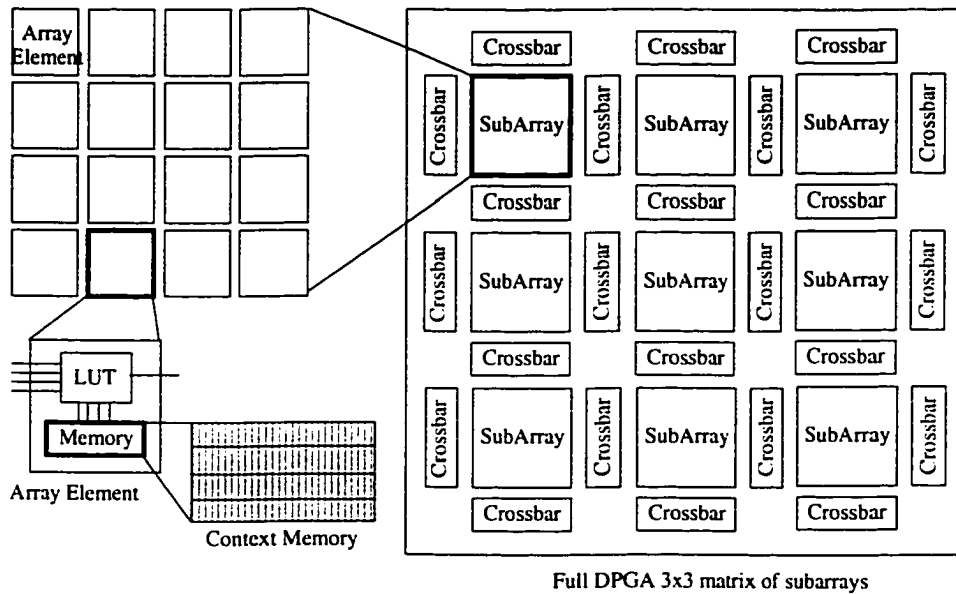


Figure 2.2: DPGA architecture and composition

chips with 64 CLBs height. Each such memory cell is a fully synchronous dual-ported 4096-bit RAM with independent control signals for each port and independent data-widths.

2.2.5 Dynamically Programmable Gate Array (DPGA)

Dynamically Programmable Gate Array (DPGA) is a multi-context, fine-grained computing device [31]. DPGA consists of a matrix of computational cells with hierarchical interconnect. The basic computational unit is a 4-LUT (with optional flip-flop output) similar to commercial FPGA architectures. Clusters of such units of size 4x4 are grouped into sub-arrays. These sub-arrays are tiled to compose the arrays. Crossbars between subarrays serve to route inter-subarray connections (see Figure 2.2).

The important feature of the architecture is the multiple contexts for the configuration of each cell. Small context memories local to each functional unit contain multiple descriptions of the functionality of the unit. A 4x32 bit DRAM memory cell provides four context configurations for both the LUT and the interconnection network. The local interconnect switch is also controlled by such a context memory. A single, 2-bit, global context identifier is distributed to all the array elements. A broadcast context identifier selects the context to be executed (similar to an instruction) in each system clock cycle. The control is centralized but the decentralized distribution of the context memories permits various algorithm implementations.

2.3 Hybrid Architectures

Configurable platforms which have shown impressive results typically have configurable logic attached to a host system through some interface such as the system bus or the I/O channels. These systems have shown significant speedups for specific applications. The limiting factor in this case in achieving higher performance on all applications is the delay in communicating with the configurable logic. This delay results in higher data transfer and reconfiguration overheads. Currently, systems try to alleviate this problem by moving configurable logic to the processor die. We term these architectures as hybrid architectures. There are various terms used for these architectures, including Systems-on-Chip (SoC), Configurable System-on-Chip (CSoC), Reconfigurable Systems-on-Chip (RSoC), Systems on Programmable Chip (SoPC), among others.

2.3.1 Convergence in Hybrid Architectures

We discuss the features of some of the hybrid architectures in the following sections. These vary from conceptual research architectures to industrial implementations by start-up companies. The list of architectures described below is not supposed to be exhaustive, but just a sampling of the large space of hybrid architectures.

The methodology of implementing hybrid architectures has experienced a convergence. There are several commercial devices which integrate programmable logic on the same die as the processor [26, 72]. FPGA vendors are also aggressively approaching the same design space by providing customized processor and other IP cores on their devices. These include the Platform FPGA initiative by Xilinx [84] with PowerPC cores and the Altera Excalibur offering [61].

In this thesis, we consider hybrid architectures as the concept of a chip containing multiple components as defined above and not the actual chip. Hence, an FPGA which integrates multiple architectural components¹ is considered a hybrid architecture if the chip has reconfigurable logic that is programmable by the end-user.

In the following sections, we outline several hybrid architectures and illustrate their key features. These architectures illustrate large variation in the implementation approach. For example, Xilinx Platform FPGA integrates an embedded PowerPC processor as a hard core on a fine-grain Virtex-II FPGA. On the other end of the spectrum,

¹conventional processor or DSP cores, embedded memory, and peripheral interfaces

Chameleon Systems Reconfigurable Communications Processor (RCP) is a hybrid architecture with an ARC 32-bit processor integrated with proprietary coarse-grain Reconfigurable Processing Fabric (RPF) on the same chip.

2.3.2 OneChip

OneChip illustrates an architecture with a very tight integration of reconfigurable logic with a traditional microprocessor core. Programmable logic called Programmable Functional Units (PFUs) were co-located with Basic Functional Units (BFUs) in a MIPS microprocessor core. The execute stage of the microprocessor was thus enhanced with reconfigurable logic.

The basic advantage of OneChip is the high data bandwidth that is shared by the PFU and the BFU. A similar approach was earlier suggested by the PRISC project [64]. The PFUs are associated with dedicated configuration memories located close to the PFUs to facilitate fast operation. The interconnection network is similar to hierarchical interconnect found in Xilinx FPGAs. The chip can execute typical MIPS applications with binary compatibility by not exploiting the configurability. Pre-compiled configurations of the PFUs can be stored in the configurations and can be dynamically switched to perform specific application tasks.

2.3.3 Berkeley Garp

The Berkeley Garp architecture combines configurable logic with a standard MIPS processor on the same chip [41]. The configurable array is composed of a matrix of logic blocks which are organized into 32 rows of 24 blocks. One block in each row is a control block and the remaining are logic blocks which can implement a 2-bit operation. Four memory buses run vertically through the rows for moving information into and out of the array. They can be used for data transfer and memory accesses. A separate wire network provides interconnection between the logic blocks. The loading and execution of the configurations is under the control of the main processor. A transparent integrated configuration cache holds the equivalent of 128 total rows of configurations (as 4 cached configurations for each row). Reconfiguration from this cache takes 4 cycles irrespective of the number of rows. The operation of the reconfigurable array is carried out by using some extended instructions to the MIPS instruction set. The reconfigurable array, however, can perform data cache or memory accesses independent of the MIPS core.

2.3.4 National Adaptive Processing Architecture (NAPA)

National Adaptive Processor Architecture (NAPA) is part of the Adaptive System on a Chip (ASC) series which is aimed at providing an integrated hardwired standard and application specific “softwired” programmable functional blocks [66]. It is targeted towards DSP applications, Imaging, Feature Extraction problems, Encryption/Decryption etc.

NAPA 1000 is a high performance, low power Adaptive Processing Architecture from National Semiconductor. The device consists of a 32 bit RISC processor core and a 50K gate Adaptive Logic Processor(ALP). It also consists of a multiple processor bus interface and 16K bytes of data memory and 256 byte scratch-pad memory for the ALP.

The microprocessor is a custom Compact RISC processor. The ALP consists of a homogeneous array of 96x64 core cells. Each cell can implement up to a three input logic function with up to two outputs. The interconnection network is hierarchical with nearest neighbor connections and chip wide switched bus network. The ALP is provided with a pair of memory banks of 8K each and 8 scratch-pad memories of 256 bytes for distributed local storage.

The application development environment on NSC NAPA is targeted towards providing a C/C++ based transparent interface. The adaptive logic is integrated as functional building blocks which will be integrated as a layer between the C compiler and the low level object code. ALP generators for common functions are intended to provide high performance and utility. A Reconfigurable Pipeline Instruction Set(RPIS) coordinates the interaction between the RISC and the ALP components.

2.3.5 Xilinx Platform FPGA

Xilinx Platform FPGA is an evolution from high capacity Virtex series FPGAs. Future generations of these FPGAs will integrate embedded processors, DSP functions and high speed communication interfaces.

The integration of hard and soft intellectual property (IP) cores permits hardware upgrades on the fly. The enabling technology for Platform FPGAs is the Virtex-II architecture. It is expected to scale up to 10 million system gates with internal system clocks up to 200 MHz. Virtex-II is also expected to have higher capacity configurable logic blocks and larger size block RAM and distributed RAM. The memory block sizes are increased to provide a higher memory-to-logic ratio and facilitate memory rich and data intensive application.

As a hybrid reconfigurable architecture, Platform FPGA is expected to integrate several hard cores in addition to the available soft cores. Soft cores are intellectual property provided by Xilinx and other developers as modules that can be integrated and compiled onto the FPGA by using EDA tools. Virtex-II has *IP ImmersionTM* and *Active InterconnectTM* technologies that facilitate easy integration of the soft cores into designs. Platform FPGA also supports high bandwidth serial and parallel interfaces supporting several industry standards.

Xilinx has announced that an embedded PowerPC 405 microprocessor core from IBM will be available as a hard core on the Platform FPGA. The PowerPC core is expected to operate at 300 MHz and communicate with the reconfigurable logic at more than 6 GB/sec. Platform FPGA also integrates multiple 18-bit multipliers onto the Virtex-II platform to enable theoretical performance of up to 600 billion MAC (multiply accumulates per second).

The ability of the Platform FPGA to integrate multiple aspects of different architecture paradigms onto the Virtex-II FPGA provides a flexible platform for design development. Platform FPGA design tools combine aspects of embedded processor compilation, electronic design automation (EDA), real time operating systems (RTOS), digital signal processing (DSP), among others.

2.3.6 Triscend

Triscend corporation's first configurable system-on-a-chip E5 architecture integrates a 8032 8-bit microcontroller, on-chip programmable logic, RAM and I/Os on a chip [26]. The E5 is targeted towards embedded systems and promises fast development and high level of customization. The 8032 microcontroller is a "turbo" version that runs at 40MHz. The programmable logic consists of 3,200 Configurable System Logic (CSL) cells. There is up to 64KByte of on-chip, dedicated system RAM.

The A7 Configurable System-on-Chip family consists of four devices that share the same architecture. All members of the A7 family feature the ARM7TDMI processor core rated at up to 60MHz, 8Kbytes of mixed instruction/data cache, a high-performance dedicated internal bus, and an external memory interface unit that supports Flash, EEPROM, SRAM and SDRAM. Each device in the family also includes 16Kbytes of Scratch-Pad SRAM that can be used as regular internal RAM or as a trace buffer during debug. The A7 also includes four independent DMA channels for memory-to-memory transfers, linked-list DMA, and frame transfer support.

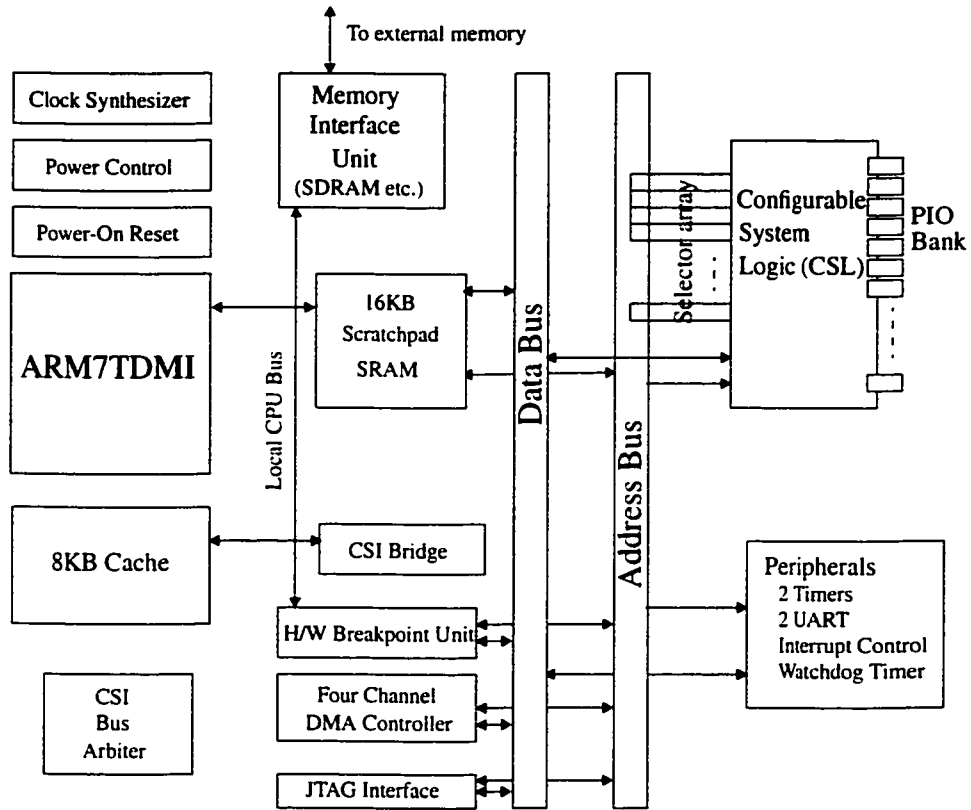


Figure 2.3: Triscend A7 CSoC architecture block diagram

The devices in the A7 CSoC family vary by the amount of Configurable System Logic (CSL) cells, which range from 512 to 3,200 cells or approximately 40,000 logic gates. The devices also offer between 123 and 315 programmable I/Os depending on the size. A block diagram of the Triscend chip with some of the major components is shown in Figure 2.3.

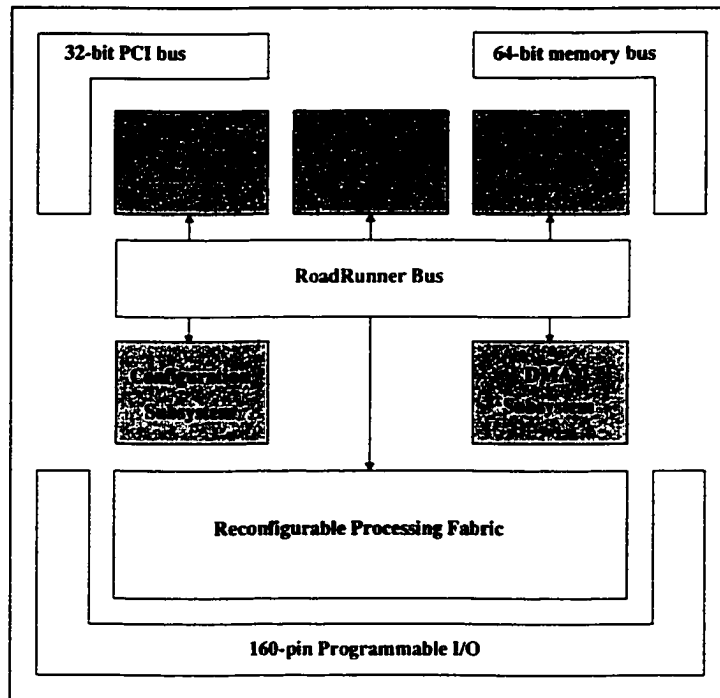


Figure 2.4: Chameleon Reconfigurable Communications Processor (RCP) Architecture

2.3.7 Chameleon Systems RCP

Chameleon Systems has designed the industry's first Reconfigurable Communications Processor (RCP). It is a high performance reconfigurable system on a chip optimized for compute intensive signal processing tasks found in communications, VoIP, software defined radio (SDR), protocol processing, among others. The architecture and the software mapping tools are briefly described in the following sections.

Chameleon Architecture

The Chameleon Reconfigurable Communications Processor (RCP) architecture consists of a 32-bit embedded processor core, 32-bit reconfigurable processing fabric, a high-speed system bus, and a programmable I/O system. The processor can interface to memory and other processing systems through a PCI-controller and memory/DMA controllers in addition to the programmable I/O. The different components are linked by a 128-bit, split transaction, high performance RoadRunner bus, which provides 2GByte/sec on-chip communication bandwidth. An overview of the architecture is given in Figure 2.4.

The different components in the Chameleon architecture are briefly described below:

Embedded Processor: The RCP has a 32-bit ARC processor that delivers 120 MIPS at 125 MHz. The processor has a four-stage pipeline, 64 general purpose 32-bit registers and large set of instructions. The memory interface in the processor has a 4Kbyte instruction cache and a 4 KByte data cache.

Reconfigurable Processing Fabric: The reconfigurable processing fabric is divided into Slices with various versions of the family supporting different number of slices. Each slice consists of three tiles and can be independently reconfigured. A tile consists of 32-bit datapath units (DPUs), 16x24-bit multipliers, local store memories (LSMs) and control logic unit (CLU). A block diagram of the tile is shown in Figure 2.5. A 32-bit DPU implements most standard arithmetic operations and several specialized operations. There are several registers in the datapath to support efficient pipelining of

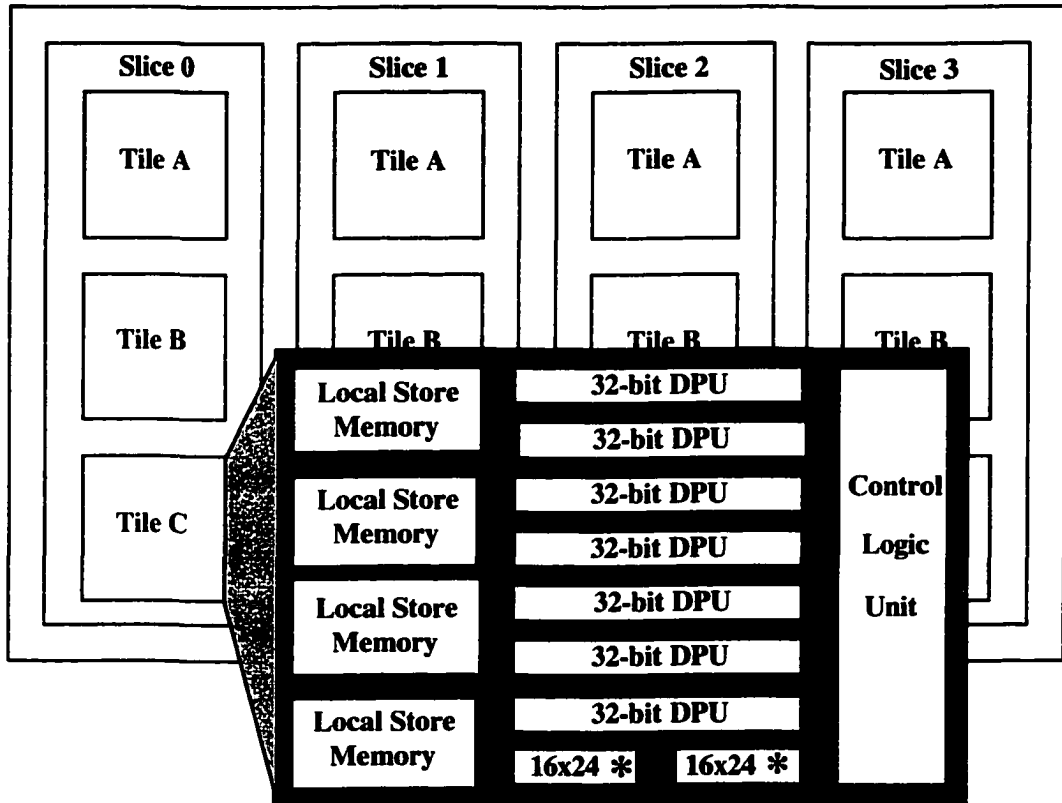


Figure 2.5: Reconfigurable Processing Fabric (RPF)

computations. Each tile has four 32-bit wide by 128 word deep Local Store Memories (LSMs). The LSMs are multi-ported allowing simultaneous read and write operations. The Control Logic Unit (CLU) can store eight user specified instructions for each of the seven DPUs in the Tile where each instruction represents a complete DPU configuration.

Dynamic Interconnect: The datapath is overlaid with a rich interconnect architecture which is deterministic and provides 100% routability. Dynamic interconnect includes three types of routes: local routes, intra-slice routes and inter-slice routes. Local routes

connect 8 DPUs on either side of any DPU with a delay of one clock cycle. Intra-slice routes connect all DPUs within a slice with a delay of one clock cycle. Inter-slice routes connect DPUs in different slices with a delay of two clock cycles.

Embedded Computing Subsystem: In addition to the main processing units, the Chameleon architecture has several components to interface to external systems. RCP has a programmable I/O bank of 160 pins providing an aggregate bandwidth of 2 GByte/sec. The PIO bank can be interfaced to external components such as A/D, D/A, FPGAs, SRAM memory and other sensors. A 32-bit PCI controller supporting Master/Slave operation provides an interface solution to a PCI bus. A 64-bit memory controller can interface to external SDRAM and can support 1 GByte/sec transfer rate. A 16 channel DMA subsystem can transfer data between the various modules on the chip and the LSMs.

Chameleon Reconfiguration Potential

Each DPU is associated with a control logic unit that can store 8 different instruction words. The instruction words dictate the configuration of the DPU including the control signals associated with various DPU components such as registers and multiplexers. The configuration to be executed can be determined based on control state machine and datapath intermediate results. This facilitates on-the-fly execution of a different configuration every cycle from among the 8 possible configurations.

The Configuration subsystem consists of a controller and a background configuration buffer. Configuration information can be transferred from off-chip memory into the

background configuration plane while the active configuration is performing computations at peak performance. RCP can switch from active to background configuration in just one clock cycle (few nanoseconds) facilitating a single cycle context switch. This is in contrast to reconfigurations times of the order of few milliseconds for most FPGAs.

Chameleon Software Environment

The Chameleon Systems Integrated Development Environment (C SIDE) is a complete toolkit for designing, debugging and verifying RCP designs. Figure 2.6 gives an overview of the development flow using the RCP. C code is compiled using an optimized GNU C compiler for the ARC processor and Verilog code is synthesized for the Reconfigurable Processing Fabric. The two streams are linked together for execution on the RCP. Chip-Sim, a complete cycle-accurate simulator for the complete chip, has a standard GDB interface for debugging. The Chameleon PCI development board provides a PCI bus or JTAG interface to debug and verify designs on the RCP hardware.

The Chameleon eConfigurable Basic I/O Services (eBIOS) provides a seamless interface between the embedded processor system and the fabric. eBIOS provides resource allocation, configuration management and DMA services. The software tools can generate eBIOS calls automatically or they can be specified by the user for precise orchestration of the execution.

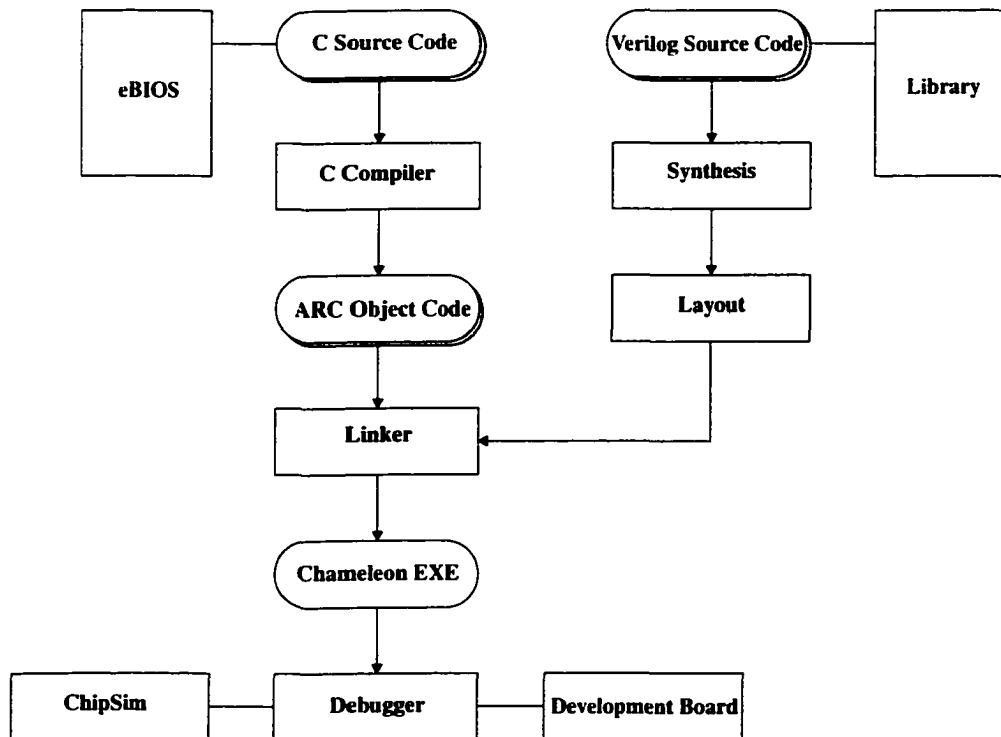


Figure 2.6: Chameleon software tool flow

Chapter 3

Approach

Challenges can be stepping stones or stumbling blocks.

It is just a matter of how you view them.

- Unknown

Several research and commercial efforts in configurable computing concentrated on designing new configurable architectures and developing design tools and applications for these architectures. These efforts are specific to the respective architectures and there is a lack of developmental effort in designing a uniform framework for mapping applications onto configurable architectures. Our approach is to develop a framework which will facilitate mapping and provide an algorithmic foundation for automatic mapping.

The problem of mapping of tasks to configurable architectures is different from mapping to traditional fixed architectures. Configurable architectures have a variable and

adaptive target which can be chosen at runtime to suit the computation unlike fixed architectures. Traditional mapping techniques assume a given fixed target and try to optimally map an application(set of tasks) to minimize execution time and other overheads such as communication and data reorganization. In configurable computing the reconfiguration costs are not costs associated with the tasks but rather with the target architecture.

3.1 Challenges

Mapping applications onto configurable architectures requires utilization of the dynamic reconfiguration potential available. Tools and techniques which have been designed for static architectures cannot realize this potential. Traditional CAD tools concentrate on reducing the timing delays and other overheads in mapping a task onto a configurable architecture. They do not consider the issues in reconfiguring the architecture for each task and the new overheads incurred in doing the reconfiguration. We outline below the challenges in developing the tools and techniques for utilizing reconfigurable architectures.

3.1.1 Static vs. Dynamic Reconfiguration

Utilizing FPGAs for speeding-up applications has been mostly limited to developing configurations which optimize the computation time for a given task. The optimized

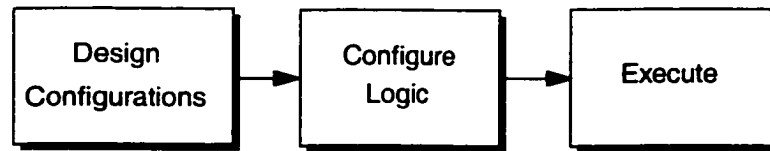


Figure 3.1: Static configurable computing

configuration is then used to execute the task. This process is illustrated in Figure 3.1. A given computational task is analyzed and an optimized configuration is developed for that computational task. The configurable logic device is then configured, usually under the control of the host, with this optimized configuration. Finally the configuration is executed by initiating the computation and communicating the data to the device. The programmability of the device is not exploited and the logic resources are not reused during a computation.

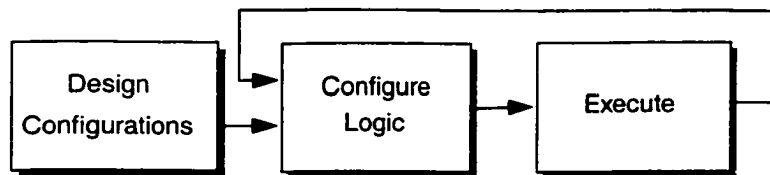


Figure 3.2: Dynamic configurable computing

The conventional approach is *static*, because the hardware is configured just once, followed by execution. The concept of *dynamic* configurable computing is illustrated in Figure 3.2. The configurable resources are reused by reconfiguring the hardware after a computation is completed. The configuration of the logic and the interconnection network are adapted on the fly during the execution. The run-time reconfiguration

can be based on intermediate results generated by the computations. This approach has enormous opportunities to achieve higher performance than conventional approach by closely adapting the hardware to the nature of the computation.

3.1.2 Design Methodologies

To achieve high performance using configurable architectures, effective configuration design techniques need to be developed. Existing design methodologies are based on ASIC design tools and fail to realize the full potential of configurable logic. These logic synthesis tools are geared towards compiling a hardware oblivious algorithm. The behavioral description of the algorithm is mapped to logic using synthesis tools in several phases. In this process, the structure of the algorithm is not utilized resulting in sub-optimal designs with respect to area and delay performance. Also, this design methodology does not incorporate the input data knowledge into the configuration. Algorithm specific and instance-aware configurations are the key to achieving large speed-ups on configurable architectures.

Current design compilation times are too long and preclude any run-time, dynamic modification of the configurations. Existing designs also lack modularity and scalability and have low performance (e.g. clock rates) unless optimized by hand. One major problem in using reconfigurable logic to speed-up a computation is the design process. The “standard CAD approach” used for digital design is typically employed (see Figure 3.3). The required functionality is specified at a high level of abstraction via an HDL

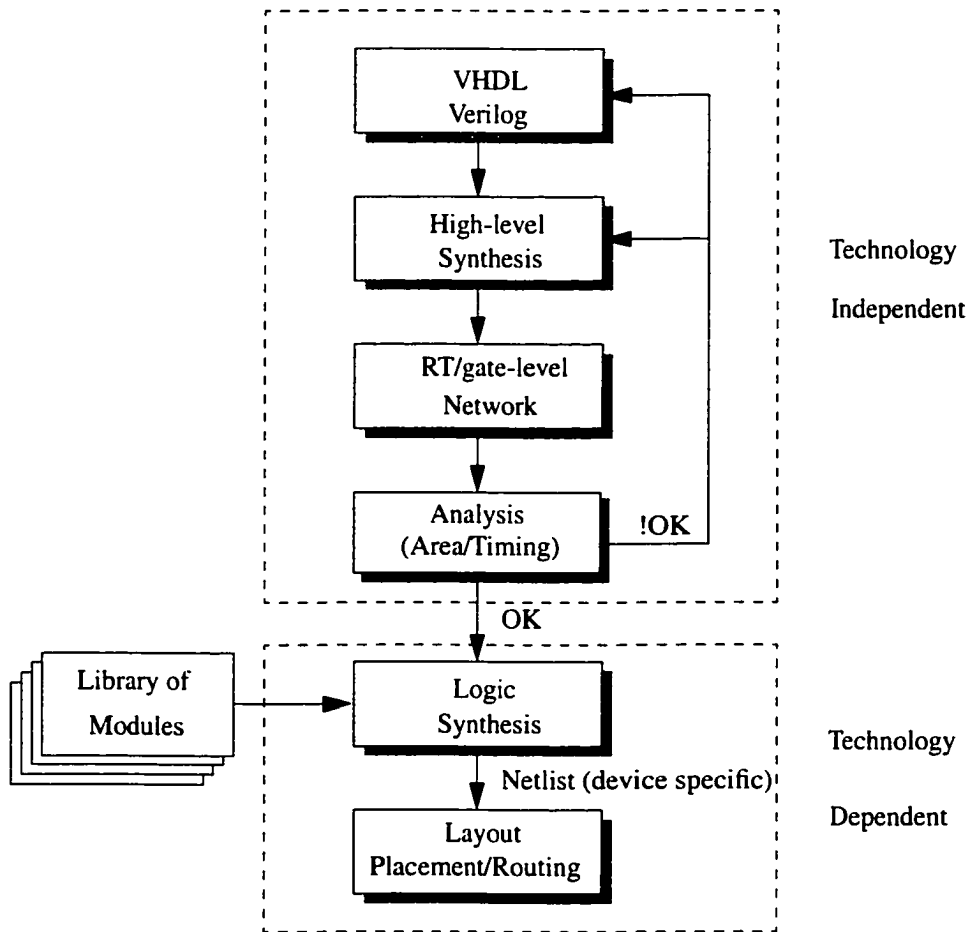


Figure 3.3: Traditional Design Synthesis Approach

or a schematic. Logic libraries specific to a given device (e.g. Xilinx FPGA, Altera FPGA, Chameleon Component Library etc.) and time consuming placement and routing steps are required to perform the logic mapping. This approach of *logic synthesis* as opposed to *algorithm synthesis* allows the user to specify the design using a behavioral model. But this abstraction is achieved at the expense of performance. The semantics and nature of the algorithm are lost in the mapping phases.

3.1.3 Multi-dimensional Optimization

Configurable architectures possess multi-dimensional characteristics which are more diverse than systems based on microprocessors, digital signal processors and other integrated multi-component architectures. The constraint space that dictates the optimization process is illustrated in Figure 3.4. In addition to the traditional two-dimensional space of architectural and application constraints, reconfigurable architectures also have dynamic adaptation constraints.

Application constraints deal with the type of applications tasks and the dependency among the tasks. As we describe in later chapters, for loop computations these constraints include the structure of loops, dependencies inside the loop and loop carried dependencies. Architectural constraints include the granularity of the functional units, the structure of the functional units, the performance characteristics, among others. In reconfigurable architectures, the additional factor of adaptation constraints affects the

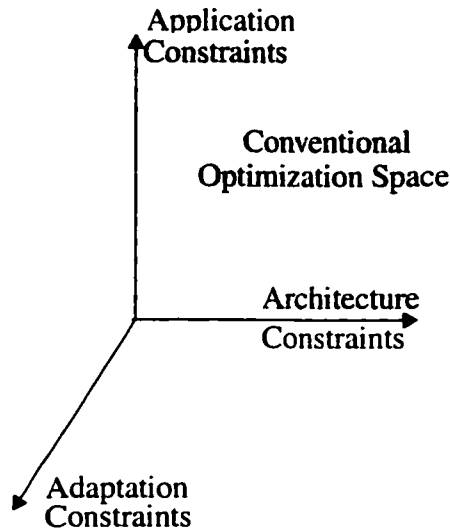


Figure 3.4: Constraint space of optimization in reconfigurable architectures

optimization process. The method of reconfiguration, partial reconfiguration, reconfiguration caches, reconfiguration overheads, etc. need to be incorporated into the optimization process. The algorithmic mapping techniques that are developed have to include the variables and costs for these multiple dimensions.

These characteristics of configurable architectures need to be exploited to achieve high performance. The multi-dimensional characteristics of reconfigurable architectures is illustrated in Figure 3.5. When reconfigurable logic is integrated with other computing architectures in hybrid systems, it also gives rise to further complexities due to such integration. Multiple dimensions of the problem and architecture space need to be considered for optimizing the performance of the applications on hybrid reconfigurable systems. These include:

- Fine-grain and coarse-grain parallelization

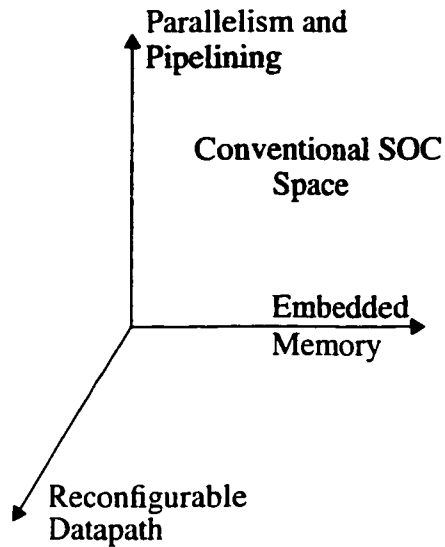


Figure 3.5: Multi-dimensional configurable architecture characteristics

- Customized application specific configurations
- Reconfiguration of the hardware
- Larger space of cost variables - reconfiguration cost, data communication cost
- Non-traditional memory organization and access

3.1.4 Design Tools

Complex systems on a chip have evolved due to various technological, application, and market forces. The design methodologies are being developed as a *reaction* to the evolution of the hybrid architectures. Design tools are being developed after the evolution

of the hybrid architectures and have not co-evolved with the architectures. Current design processes¹ are based on independent design flow for each architectural component. The programming models and the design tools for each of the individual components are utilized to map an application. The integration is performed at a much later stage. A standard interface between different components of the hybrid architecture is the only integration that exists during the design phase. An example design tool flow is illustrated in Figure 3.6. The independent development of C and HDL code is compiled, synthesized and linked only late in the compilation phase. The development activities for each aspect are separate and are based on a fixed application programmer interface (API).

The different characteristics of reconfigurable logic are outlined in Figure 3.5. These characteristics are utilized independently either in different stages or by using semi-automatic design tools. For example, *register balancing* is a technique used to pipeline designs. A design can be implemented as a combinatorial circuit without regard to pipelining. Synthesis tools can be used to automatically generate a balanced pipelined design by inserting registers in all the datapaths leading to the output.

The development of designs using such ad-hoc techniques results in sub-optimal designs. Also, the integration of multiple design methodologies and mapping techniques involves significant effort by the application designer. An integrated methodology is required that simultaneously exploits multiple aspects of hybrid architecture. Design tools

¹by design processes, methodologies and tools we refer to the aspects of developing applications *using* a hybrid architectures and not designing the chip.

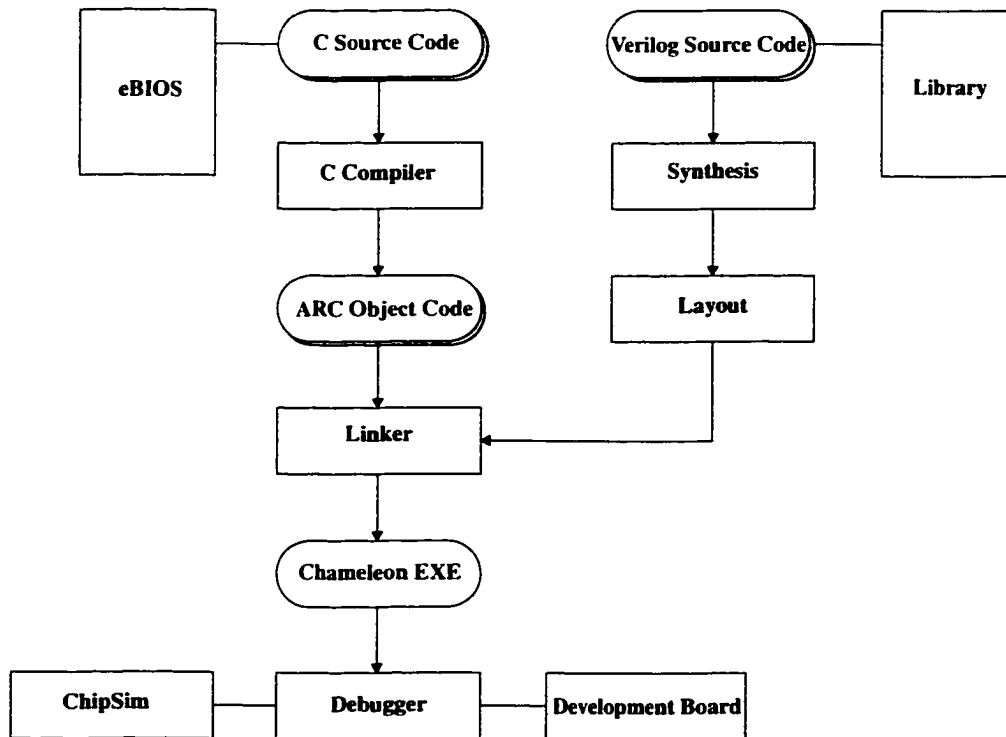


Figure 3.6: Chameleon design tools flow

that facilitate such a process can be developed only by developing mapping techniques that focus simultaneously on multiple aspects of the hybrid architecture characteristics.

3.2 Approach

Traditional high level tools such as compilers for microprocessors or parallel processors are also not suitable for mapping applications. Compilers map an application(program) by targeting to a fixed machine architecture. They do not take into account the dynamic behavior of reconfigurable architectures. Static analysis of the task computation is not enough to perform optimal mapping to reconfigurable architectures. We will develop algorithmic techniques which will enhance such a compiler framework by analyzing the issues in dynamic behavior of the architecture.

In this thesis we address the reconfigurable computing issues and challenges discussed earlier by developing a model and algorithmic framework for mapping applications onto reconfigurable architectures. The techniques developed are evaluated on example architectures and simulated using a simulator, DRIVE (see Chapter 8), developed as part of the thesis.

The Hybrid System Architecture Model (HySAM) that we developed is discussed in detail in Chapter 5. HySAM is a parameterized model of a reconfigurable computing system, which consists of reconfigurable logic attached to a traditional microprocessor. This model is utilized for analyzing application tasks and developing the mapping and scheduling of these tasks onto the reconfigurable system.

The absence of mature design tools impacts the simulation environments that exist for studying reconfigurable systems and the benefits that they offer. Simulation tools are a very important component of the design cycle. Simulations provide users with practical feedback when developing applications and designing systems. This allows the designer to determine the correctness and performance of a design before the system is actually constructed. The user can explore the merits of alternative designs without actually building the systems. Simulation tools provide a means to explore the architecture and the design space in real time at a very low resource and time cost.

As part of this thesis, we have developed a simulation framework - **Dynamically Reconfigurable systems Interpretive simulation and Visualization Environment (DRIVE)**. DRIVE can be utilized as a vehicle to study the system and application design space and performance analysis. Reconfigurable hardware is characterized by using a high level parameterized model. Applications are analyzed to develop an abstract application task model. *Interpretive* simulation measures the performance of the abstract application tasks on the parameterized abstract system model. This is in contrast to simulating the exact behavior of the hardware by using HDL models of the hardware devices.

The application tasks that we consider in this thesis are LOOP computations. It is a well known rule of thumb that 90% of the execution time of a program is spent in 10% of the code. This code usually consists of repeated executions of the same set of instructions. The typical LOOP constructs used for specifying iterative computations in various programming languages are DO, FOR and WHILE, among others.

Computations which operate on a large set of data using the same set of operations are most likely to benefit from configurable computing. Hence, loop structures will be the most likely candidates for performance improvement using configurable logic. Configurations which execute each task can be generated for the operations in a loop. Since each operation is executed on a dedicated hardware configuration, the execution time for the task is expected to be lower than that in software.

The algorithmic techniques developed in the remainder of this thesis address the different aspects of the mapping challenges outlined in Chapter 3.1. The structure of the applications tasks as dictated by the dependencies in loop computations and the characteristics of the hardware as captured by our HySAM model are addressed in the mapping problems. We address the space of these dependencies and characteristics by defining several mapping problems and developing algorithms to solve the mapping problems.

3.2.1 Model based Reconfigurable Computing

A model of the hardware that abstracts the hardware without sacrificing the core features of the hardware can significantly aid in the mapping process (see Figure 3.7). An example of a model that permits application mapping and analysis is the Reconfigurable Mesh model [10, 16]. The model based mapping environment takes into account the capabilities and limitations of current as well as projected hardware technologies. Parameterized models for algorithm design and analysis will possess the following characteristics:

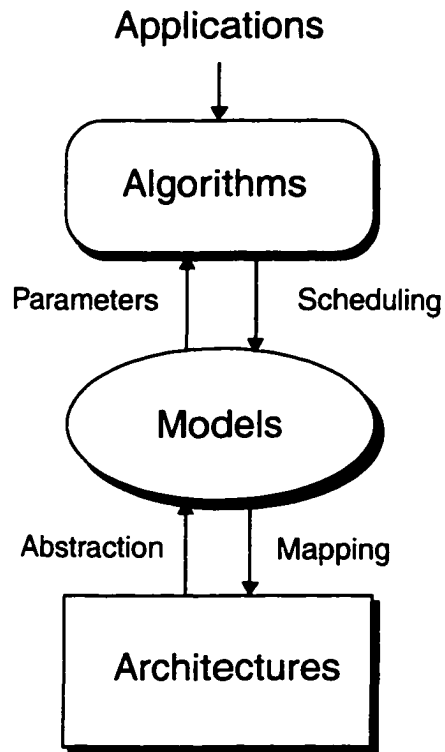


Figure 3.7: Model-based Approach

- Cost models for analysis of reconfigurable architectures.
- Techniques for partitioning and placement of designs exploiting algorithm and input structure.
- Cost analysis incorporating the cost of reconfiguration and *partial* and *dynamic* reconfigurability.
- Impact of off-chip communication in designing reconfigurable computing solutions.
- Tradeoffs between reconfigurability and redundancy of hardware.

The HySAM model developed as part of this thesis is described in detail in Chapter 5.

3.2.2 Loop Computations

There are various methods of exploiting the locality in execution present in loops. Instruction caches utilize the spatial locality to improve memory performance by keeping the most frequently accessed instructions in a faster memory. Loop parallelization techniques rely on utilizing the parallelism in independent iterations by executing different iterations of the same loop on multiple functional units at the same time [81]. Software pipelining techniques are utilized to exploit pipelining at the architectural level by pipelining the successive iterations of a loop construct.

The loop statements which can be executed on configurable logic should be *well behaved*. The characteristics of *well behaved* loops are described below:

- Constant step size for the index
- No function calls in loop body
- No pointer operations or arithmetic
- Statically computable memory accesses

Though not all loop constructs satisfy these constraints, most commonly occurring loops, including those in benchmark programs, satisfy these constraints. Loop transformation techniques [81] can be utilized to convert arbitrary loops to conform to the above constraints. The operations in a loop can be executed by setting up a specific configuration for each of the operations. Each of the operations in the loop statement might be a simple operation such as an addition of two integers or can be a more complex operation such as a square root of a floating point number. The problems and solutions that we present are independent of the complexity of the operation.

Loop computations provide opportunity for parallelizing the computations on reconfigurable architectures. Reconfigurable architectures have a large number of functional units which can be utilized for concurrent computations (parallel or pipelined). Pipelined functional units support high clock frequency and hence high performance. The rich dynamic interconnection network provides enough data bandwidth for parallel and pipelined computations. We first introduce some definitions used in the remainder of the thesis.

3.2.3 Definitions

Loop: A set of statements $\{S_1, S_2, \dots, S_p\}$ executed repeatedly for N times. The number of statements in the loop is p and the set of statements is referred to as the *loop body*.

Iteration: One specific execution of the loop body is called an *iteration*. The statements are superscripted with the iteration number to indicate a specific iteration. For example, execution of a statement S_i in the j th iteration is indicated as S_i^j ($1 \leq i \leq p$ and $1 \leq j \leq N$).

Nested Loop: Nested loop is a general case in which a loop can contain other loops in the loop body. For a k -level nested loop, a specific execution of a statement S_i is indicated by superscripting S_i with a vector of size k as $S_i^{j_1, j_2, \dots, j_k}$ ($1 \leq i \leq p$ and $1 \leq j_r \leq N_r$). N_r denotes the number of iterations of the r th loop. The vector indicates the specific iteration of each of the nested loops from the outermost to the innermost loop. A statement executed independent of a specific loop is indicated by a '*' in that position in the vector.

Data Dependency: If a value computed by a statement S_j is used by another statement S_i , then S_i is said to be data dependent on S_j and is indicated as $S_i \prec S_j$. All dependencies discussed in the remainder of the thesis are data dependencies unless explicitly mentioned otherwise. Dependencies can be illustrated by a dependency graph which has statements as nodes and dependencies as directed edges.

Transitive Dependency: The dependency relation defined above satisfies the transitive property:

$$S_i \prec S_j \text{ and } S_j \prec S_k \Rightarrow S_i \prec S_k.$$

Loop-carried Dependency: If a statement executed in one iteration is data dependent on the statement executed in another iteration then the dependency is classified as a loop-carried dependency. Loop-carried dependency indicates a cycle in the dependency graph: $S_i^a \prec S_i^b$ and $a \neq b$.

Chapter 4

Related Work

If I have seen farther, it is by standing on the shoulders of giants.

- Isaac Newton

The following topics outline the different aspects of reconfigurable computing that research has been addressing in the past several years:

- *Architectures* [8, 20, 26, 34, 41, 44, 72, 77]: Device and system architectures are being developed which propose various ways of organizing and interfacing configurable logic. Some architectures are also based on coarse grain functional units that are configured on the fly to execute an operation from a given set of operations. Commercial architectures are exploring integration of reconfigurable logic and microprocessors on the same chip.
- *Applications* [3, 23, 29, 60, 63, 69, 85]: Specialized configurable architectures which are utilized for speeding up specific applications are replacing some ASICs.

Some applications also exploit optimization based on a specific input instance of the computation.

- *Algorithmic Synthesis* [5, 11, 15, 19, 21, 22, 28, 44, 45, 50, 53, 59, 62, 71, 76, 78, 80]: Dynamically reconfigurable architectures give rise to new classes of problems in mapping computations onto the architectures. New algorithmic techniques are needed to schedule the computations. Existing algorithmic mapping techniques focus primarily on loops in general purpose programs. Loop structures provide repetitive computations, scope for pipelining and parallelization and are candidates for mapping to reconfigurable hardware. We describe briefly some of the algorithm synthesis research in this chapter.
- *Software Tools* [7, 9, 12, 33, 37, 44, 47, 51]: Current software tools still rely on CAD based mapping techniques. But, there are several tools being developed to address run-time reconfiguration, compilation from high level languages such as C, simulation of dynamically reconfigurable logic in software and complete operating system for dynamically reconfigurable platforms.

There is a significant lack of research in development of models of reconfigurable architectures that can be utilized for developing a formal framework for mapping applications. The Reconfigurable Mesh model was the earliest theoretical model that addressed dynamic reconfiguration in computation and communication structure [10]. However, Reconfigurable Mesh model is more theoretical and hardware implementations have

only been able to approximate the delay and speed assumptions in the model. Our HySAM model is a more realistic computation and compilation model that facilitates development of algorithmic mapping techniques.

There have been several research efforts that focused on developing architectures and the associated software tools for mapping onto their specific architecture. Some of these projects have addressed generic mapping techniques that can be extended to a class of reconfigurable architectures. Such projects include the Berkeley Garp [75, 21, 41], National Semiconductor NAPA [36, 66], Xputer [44], Northwestern MATCH [6], MIT RAW [77], CMU PipeRench [38], DEC PeRLe [76], SPLASH [19].

The Garp and NAPA projects address some of the issues in mapping loops onto reconfigurable architectures. However, they are heavily based on loop analysis and do not develop a model-based mapping framework. Our algorithmic techniques also exploit loop analysis performed in conventional compilation and parallel compilation areas and use the HySAM model for developing optimal mappings. Hartenstein et. al. developed the Xputer paradigm and an operating system for machines based on the paradigm [44]. CoDe-X takes as input a C like application program and compiles and executes on Xputer hardware. The framework does the partitioning, compiling and library mapping of the application tasks. Data scheduling to improve performance is also addressed. Time

Multiplexed FPGA [74] stores multiple FPGA contexts in SRAM cells. This architecture can potentially switch between configurations in about 30 ns. The mapping techniques that we develop can exploit such an architecture by minimizing the reconfiguration overhead.

Customizing configurable hardware to suit the computations has been acknowledged as the most significant advantage of such architectures. Some researchers have adapted the hardware to perform computations with exactly the required precision for the computations [69, 73]. Such *static* approaches do not exploit the ability of configurable hardware to be adapted to the exact required precision as the computations progress. The maximum possible precision of variables which is determined in the *static* approach can still involve execution with superfluous precision and unnecessary overheads. Several efforts have also focused on developing parameterized libraries and components, precision being one of the parameters. Most FPGA device vendors provide such highly optimized parameterized libraries for their architectures. Efforts have also been made to generate such modules using high level descriptions [24, 55].

Pipelined designs have been studied by several researchers in the configurable computing domain. The concept of virtual pipelines and their mapping onto physical pipelines has also been analyzed. Cadambi et. al. address some of the issues in mapping virtual pipelines onto a physical pipeline by using incremental reconfiguration in the context of PipeRench [20]. Their work addresses the issues in controlling the movement of configuration data and computation data.

Luk et. al. describe pipeline morphing and virtual pipelines as an idea to reduce the reconfiguration costs [50]. A pipeline configuration is morphed into another configuration by incrementally reconfiguring stage by stage while computations are being performed in the remaining stages. Virtual pipelines are mapped onto physical pipelines by morphing between pipeline stages. But, morphing is limited to architectures which support fast reconfiguration of the order of a single pipeline stage execution. We consider the case when the pipeline reconfiguration cost is significant. The problem that we are addressing in this thesis is to generate distinct pipeline phases between which the reconfiguration cost is reduced.

Weinhardt describes the generation of pipelined circuits from parallel-FOR loops in high level programming language [78]. Weinhardt et. al. also developed pipeline vectorization techniques [79]. The loop candidates which are ideal for mapping onto hardware pipelines and loop transformations which can be performed to increase the parallelism are described.

Several simulation tools have been developed for reprogrammable FPGAs. Most tools are device based simulators and are not system level simulators. Some of the efforts in this area are briefly described here. The most significant effort in this area has been the Dynamic Circuit Switching (DCS) based simulation tools by Lysaght et.al. [51]. These tools study the dynamically reconfigurable behavior of FPGAs and are integrated into the CAD framework. Though the simulation tools can analyze the dynamic circuit behavior of FPGAs, the tools are still low level. The simulation is based on CAD tools

and requires the input design of the application to be specified in VHDL. The parameters for the design are obtained only after processing by the device specific tools.

Luk et.al. describe a visualization tool for reconfigurable libraries [48]. They developed tools to simulate behavior and illustrate design structure. Their emphasis is on visualization of library modules and not system level simulation or application performance analysis. CHASTE [17] was a toolkit designed to experiment with the XC6200 at a low level. The toolkit allows circuit specification and performs timing analysis and simulation. But, the target of the CHASTE system is low level design exploration and not system level analysis. There are other software environments such as JHDL [7], HOTWorks [27], Riley-2 [52], etc. But, they are software systems for low level hardware design and evaluation and are not system level interpretive simulation frameworks. Our proposed DRIVE framework permits system level simulation and analysis at an abstraction level much higher than existing simulations.

Chapter 5

Hybrid System Architecture Model

(HySAM)

The heartfelt cry of many scientists is that computers force them to make square holes for their round and fuzzy edged pegs.

- Anon

A model is a mathematical abstraction that captures to some degree of accuracy the form and function of an entity, in a way that makes the model useful for specific purpose.

The complexity of the low level hardware features and application make it infeasible to explore the direct mapping of the application onto the actual hardware. There is a semantic gap in the application description and the hardware architecture abilities. An abstraction of the application and the hardware architecture is needed to bridge this

semantic gap. In this chapter we develop Hybrid System Architecture Model, a mathematical abstraction of the applications tasks and the reconfigurable hardware features.

A high level model of reconfigurable hardware is needed to abstract the low level details. Existing models supplied by the CAD tools have either multiple abstraction layers or are very device specific. We present a parameterized model of a configurable computing system, which consists of configurable logic attached to a traditional microprocessor. This model can be utilized for developing the actual mapping and scheduling of these tasks onto the configurable system. Our model cleanly partitions the *capabilities* of the hardware from the *implementations* and presents a very clean interface to the user.

The model consists of two complementary aspects. A *declarative* aspect and a *generative* aspect. Similar to automata, declarative aspect of a model specifies the parameterized model of the hybrid architecture. This forms the foundation of the algorithmic analysis. The generative aspect specifies the ways in which the declarative aspect can evolve based on a set of generative functions. The evolution of the model defines the rule based transformation of the parameters of the model for each generative function.

We first describe our model of configurable architectures and then discuss the components of the model and how they abstract the actual features of configurable architectures. Then the generative aspect of the model is further elaborated.

5.1 Hybrid System Architecture Model (HySAM)

The *Hybrid System Architecture Model* is a general model consisting of a von-neumann style processor (CPU such as a microprocessor or an embedded processor) with additional Configurable Logic Unit (CLU). Figure 5.1 shows the architecture of the HySAM model. The architecture consists of a traditional processor, standard memory, configurable logic, and configuration memory communicating through an interconnection network.

The model consists of two complementary aspects. A *declarative* aspect and a *generative* aspect. The declarative aspect of a model specifies the parameterized model of the hybrid architecture. This forms the foundation of the algorithmic analysis. The generative aspect specifies the ways in which the declarative aspect can evolve based on a set of generative functions. The evolution of the model is a rule based transformation of the parameters of the model, for each generative function.

Our model partitions the *capabilities* of the hardware from the *implementations* and presents a formal interface for algorithmic optimization. The model abstracts the actual implementation choices for each of the components of the model. The interconnection network can be implemented using a wide variety of choices. It can be a bus based architecture or dedicated connection between various modules. An example realization of the HySAM model is shown in Figure 5.1.

The configurable logic consists of a matrix of size $\mathcal{W} \times \mathcal{D}$ logic units. The granularity of the logic is ω which is the granularity of an individual functional unit. For example,

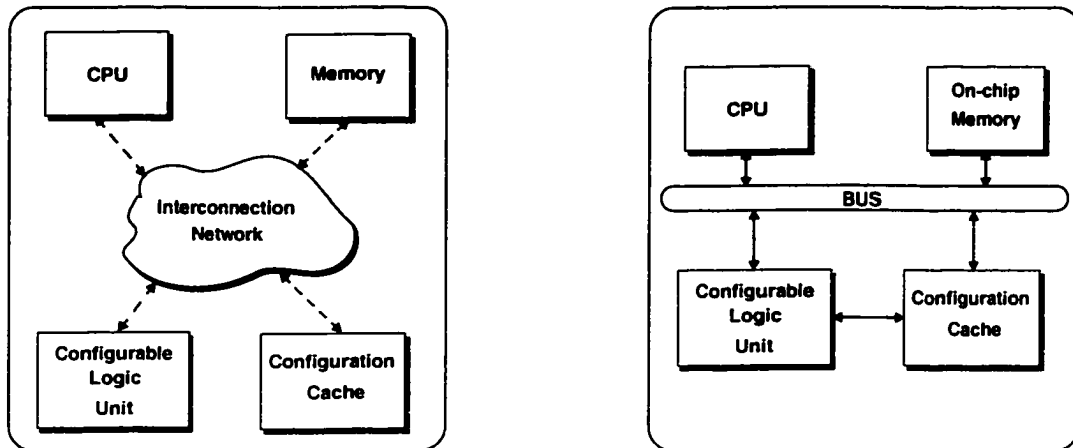


Figure 5.1: Hybrid System Architecture Model and example architecture

ω is 2 for many FPGAs, and is 32 for Chameleon RCP datapath. Configurations for executing a function are composed from the configurations of the individual functional units which are active in that configuration. Each functional unit can be configured to perform an operation from a set of basic operations.

The parameterized HySAM which is outlined above can model a wide range of systems from board level architectures to systems on a chip. Such systems include SPLASH [19], Triscend [26], Berkeley Garp [41], Chameleon RCP [72], DEC PerLE [76] among others. The values for each of the parameters establish the architecture and also dictate the class of applications which can be effectively mapped onto the architecture. For example, a system on a chip would have smaller size configurable logic (lower \mathcal{W} and \mathcal{D}) than an board level architecture but would have potentially faster reconfiguration times (lower τ_c and Γ_c).

Table 5.1: HySAM model parameters and definitions

	Parameter Definition
\mathcal{F}	Set of functions $\mathcal{F}_1 \dots \mathcal{F}_n$ (<i>capabilities</i>)
\mathcal{C}	Set of possible configurations $\mathcal{C}_1 \dots \mathcal{C}_m$ of the CLU (<i>implementations</i>)
\mathcal{A}_{ij}	Set of attributes for function F_i using configuration C_j
\mathcal{R}_{ij}	Reconfiguration cost from C_i to C_j
\mathcal{G}	Set of generators that transform configurations
\mathcal{B}	Bandwidth of the interconnection network (bytes/cycle)
χ	Size of the configuration cache
τ_c, Γ_c	Access cost for configuration data from the cache and memory respectively
τ_d, Γ_d	Access cost for data from the on-chip and external memory respectively
\mathcal{M}	Amount of on-chip data memory
ω	Granularity of the configurable logic functional unit
\mathcal{W}, \mathcal{D}	Width and Depth in units of ω of CLU

5.2 Functions and Configurations

The input to the system is an application which is to be executed on the hybrid system architecture. This input application is partitioned into tasks which are to be executed on the CPU and the Configurable Logic Unit. The applications tasks to be executed are then decomposed into a sequence of CLU functions(\mathcal{F}).

A configuration denotes a specific structure and arrangement of the configurable logic which has a given functionality. A configuration describes the state of basic functional units of the architecture. The uninitialized state of the CLU is denoted by \mathcal{C}_0 . Execution of a function on the CPU is represented as execution in a special configuration \mathcal{C}_{cpu} .

The functions \mathcal{F} and configurations \mathcal{C} have a *many-to-many* relationship. Each configuration \mathcal{C}_j , can potentially contain more than one function \mathcal{F}_i . For example, a configuration can contain both addition and logical OR. The execution cost of a function \mathcal{F}_i in configuration \mathcal{C}_j is specified as one of the attributes in the set \mathcal{A}_{ij} . Each function \mathcal{F}_i can be executed by using any one configuration from a subset of the configurations. The different configurations can potentially be generated by different tools, libraries or algorithms. These configurations might have different area, time, reconfiguration, precision, power, etc. characteristics. For example, it is possible to design multipliers of various area/time characteristics by choosing various degrees of pipelining and carry look ahead techniques. The multiplier implementation can have different values for the area, pipeline stages, cycle time and number of cycles for finishing the computation. Similarly, fixed point operation configurations can be designed with various degrees of precision.

The configuration required for executing a specific function has to be generated either at compile time or on-the-fly at runtime. Compile time configurations can be generated by using schematic techniques or CAD mapping tools or dynamic pipeline generation by using component libraries. Runtime configurations can be generated by using run-time parameterized circuits or by dynamic modification of the configuration information before loading the configuration [47].

5.3 Attributes

The HySAM model associates parameters with each function-configuration pair which are called *attributes*. The *attributes* define the relationships between a function \mathcal{F}_i and a configuration \mathcal{C}_j . They contain parameters such as computation costs and the data access costs in terms of the amount of data accessed. These costs can be translated to actual timing costs based on the register/memory/bus access costs. The *attributes* can be extended to include additional variables that reflect other optimization criteria such as power and area. We define below some attributes which are utilized in the algorithms in this thesis.

Attributes Matrix (for \mathcal{F}_i in \mathcal{C}_j) -

- t - execution latency(execution time) of function F_i in configuration C_j .
- ρ - execution throughput or the data rate of function F_i in configuration C_j .
- π - precision of the operands. Depending on the context the precision can be the sum of the bit-widths of the operands or a vector representing the individual bit-widths of all input and output operands.
- β_{in} and β_{out} - input and output data bandwidth required. The overhead required for data communication is absorbed by this parameter. The precision (π) and the number of samples required determines β_{in} and β_{out} . These parameters in conjunction with the data access costs τ_d and Γ_d define the total input and output data access cost for executing a function \mathcal{F}_i in a configuration \mathcal{C}_j .

The input and output memory bandwidth requirements are specific not only to the function being executed but also the configuration that is utilized. For example, a 32-bit adder executed by using a bit-parallel adder has 32 bit input in one cycle. A bit-serial adder needs 32 bits at a rate of 1 bit/cycle.

The bandwidth required also depends on the mode of data access and execution. If the configuration is executing in a pipeline configuration or in a streaming mode, then the bandwidth requirement reflects the instantaneous bandwidth for accessing one set of input data samples. But if the data is to be first accessed from external memory and stored into on-chip memory and then utilized then the bandwidth requirements requirements reflect the cost of accessing all the data samples from external memory. The cost parameters also include any possible overlap of input, output and execution.

The actual total data access costs are dependent on the bandwidth requirement parameters (β_{in} and β_{out}) and the cost of data access from memory (τ_d and Γ_d). For example, when the execution is performed by reading data from on-chip memory and writing back to on-chip memory then the total data access overhead is given by $\tau_d * (\beta_{in} + \beta_{out})$.

5.4 Memory Access

The total system memory access bandwidth is modeled by the bandwidth of the interconnection network, \mathcal{B} . At any specific instant in the execution, the peak instantaneous bandwidth required can be computed based on the attributes of the functions executing using given set of configurations on the hardware. The configuration and data access

costs in accessing data elements from various memory modules are specified by the access cost parameters given below:

- τ_c and Γ_c - These constants define the costs in accessing configuration data from configuration cache and memory respectively. The configuration cache might be a distributed multi-context cache memory or an off-chip cache. The costs specified as cycles/byte denote the cost in accessing the configuration bit-stream information.
- τ_d and Γ_d - The data elements required for computation in a given configuration can be streamed from memory or loaded from on-chip memory words. The access costs τ_d and Γ_d denote the costs in accessing the data words from on-chip memory and off-chip memory, respectively.

5.5 Reconfiguration and Configuration Cache

To execute a different task the logic needs to be reconfigured to execute a different configuration. Changing the configuration of the logic has some associated overheads. In current reconfigurable devices and system architectures, the reconfiguration time is significant compared to the execution time for the operation.

We assume C_0 denotes the null configuration when configurable logic is not initialized. Hence, \mathcal{R}_{0i} denotes the time to configure the logic to a configuration C_i from the initial unconfigured state. \mathcal{R}_{ij} denotes the cost of changing the configuration from C_i

to C_j . This cost is a measure of the amount of logic reconfigured and the time spent in reconfiguring. It is possible to reduce the reconfiguration overhead by exploiting partial and dynamic reconfiguration cost. For a given CLU design and a set of configurations, partial and dynamic reconfiguration is included in the definition of the \mathcal{R}_{ij} cost.

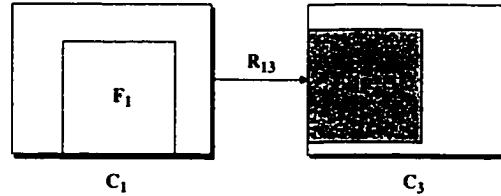


Figure 5.2: Example Reconfiguration

Figure 5.2 graphically portrays the reconfiguration of the logic from configuration C_1 to C_3 where C_1 executes the function \mathcal{F}_1 and C_3 executes the function \mathcal{F}_4 . \mathcal{R}_{13} is the reconfiguration cost in changing from configuration C_1 to C_3 .

The configuration cache acts as a *function cache* holding various configurations which can execute different functions. It can be utilized to achieve lower reconfiguration times between tasks which are executed large number of times. The configuration cache has the capacity to hold χ configurations. The cost of accessing the cached configuration data is τ_c cycles/byte and the cost of accessing configuration data not in the cache is Γ_c cycles/byte. The configuration cache might be realized as distributed configuration store or even as off-chip configuration memory. The physical realization is not relevant, but it does effect the concurrent activities that can occur in the system.

5.6 Generative Aspect

Generator \mathcal{G} is a composition function, $\mathcal{G} : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}'$. It abstracts the process of composition of configurations to generate other configurations. The physical process of constructing reconfigurable computing solutions is abstracted using the generators. Generators are a class of functions which define the various forms of composition. The range of the generator function, \mathcal{C}' , is a superset of \mathcal{C} and potentially includes new configurations generated by the transformation. Each generator defines not only the composition function but also describes how the attributes are transformed based on the composition.

$$\mathcal{A}(\mathcal{G}(\mathcal{C}_1, \mathcal{C}_2)) = f_{\mathcal{G}}(\mathcal{A}(\mathcal{C}_1), \mathcal{A}(\mathcal{C}_2))$$

The definition of function $f_{\mathcal{G}}$ depends on the specific generator \mathcal{G} that is utilized in the composition. The composition functions can be evaluated for feasibility based on the constraints imposed by the parameters of the available resources such as logic area, time and memory. The generators can be utilized recursively to compose larger configurations. We do not exhaustively describe the possible set of generator functions here but give examples of some fundamental compositions to illustrate the concept. The transformation functions for attributes of the generated configurations can be defined as given below.

5.6.1 Generators

- **Parallel:** \mathcal{G}_{par}

This generator abstracts the execution of more than one configuration in parallel on the CLU. The resulting configuration can execute multiple functions concurrently. The generator can compose multiple versions of the same configurations to achieve parallelization or compose different configurations. The mapping function for the attributes can be specified as:

$$t(\mathcal{G}_{par}(\mathcal{C}_1, \mathcal{C}_2)) = \max(t_{\mathcal{C}_1}, t_{\mathcal{C}_2})$$

$$\rho = \rho(\mathcal{C}_1) + \rho(\mathcal{C}_2)$$

$$\beta_{in} = \beta_{in}(\mathcal{C}_1) + \beta_{in}(\mathcal{C}_2)$$

$$\beta_{out} = \beta_{out}(\mathcal{C}_1) + \beta_{out}(\mathcal{C}_2)$$

- **Serial:** \mathcal{G}_{ser}

This generator simply composes larger configurations from different configurations by connecting them together. After computation in a configuration, the data is communicated to the subsequent configurations. The delay of the configurations is accumulated.

$$t(\mathcal{G}_{ser}(\mathcal{C}_1, \mathcal{C}_2)) = t(\mathcal{C}_1) + t(\mathcal{C}_2)$$

$$\rho = 1 / \left(\frac{1}{\rho(C_1)} + \frac{1}{\rho(C_2)} \right)$$

$$\beta_{in} = \beta_{in}(C_1)$$

$$\beta_{out} = \beta_{out}(C_2)$$

- **Pipelining:** \mathcal{G}_{pipe}

The process of constructing a design by pipelining data through a set of configurations can be abstracted using this generator. The configurations are not only executing concurrently, but are also operating on data items from other configurations in the same set. Only one configuration on the CLU receives input from the external source (memory, processor etc.) and one configuration communicates data to an external source. (data is stored between configurations using clocked memory elements such as registers).

$$t(\mathcal{G}_{pipe}(C_1, C_2)) = t(C_1) + t(C_2)$$

$$\rho = \min(\rho(C_1), \rho(C_2))$$

$$\beta_{in} = \beta_{in}(C_1)$$

$$\beta_{out} = \beta_{out}(C_1)$$

These basic generators define some fundamental compositions. Recursive application of these generators can be utilized to generate a large class of configurations for applications from the basic configurations.

5.6.2 Reconfiguration Cost

The Reconfiguration cost matrix for the additional configurations is also specified by the generator functions. The composition functions define the arithmetic of the reconfiguration cost. The general expression for the approximate reconfiguration cost computation for the above generators is given below:

$$\begin{aligned}\mathcal{R}(\mathcal{G}(\mathcal{C}_i, \mathcal{C}_j), \mathcal{C}_k) &= \min \{ \mathcal{R}(\mathcal{C}_i, \mathcal{C}_0) + \mathcal{R}(\mathcal{C}_j, \mathcal{C}_k), \\ &\quad \mathcal{R}(\mathcal{C}_j, \mathcal{C}_0) + \mathcal{R}(\mathcal{C}_i, \mathcal{C}_k), \\ &\quad \mathcal{R}(\mathcal{C}_i, \mathcal{C}_k) + \mathcal{R}(\mathcal{C}_j, \mathcal{C}_k) - \mathcal{R}(\mathcal{C}_0, \mathcal{C}_k) \}\end{aligned}$$

The three different terms in the *min* pertain to the cases when the reconfiguration costs of different pairs are based on some partial reconfiguration. The additional $\mathcal{R}(\mathcal{C}_0, \mathcal{C}_k)$ factor in the third term considers the fact that the $\mathcal{R}(\mathcal{C}_i, \mathcal{C}_k)$ and $\mathcal{R}(\mathcal{C}_j, \mathcal{C}_k)$ terms of the expression contribute twice to the configuration cost specific to \mathcal{C}_k .

The reconfiguration cost between two different configurations generated using the recursive application of the generators can be computed by using the equation defined above for the Reconfiguration cost. As an illustrative example, the reconfiguration cost between the two pipelined configurations shown in Figure 5.3 can be computed to be equal to $\mathcal{R}_{\mathcal{C}_2, \mathcal{C}_4}$. This reconfiguration cost computation is used implicitly throughout the thesis when determining the reconfiguration cost between complex combinations of configurations.

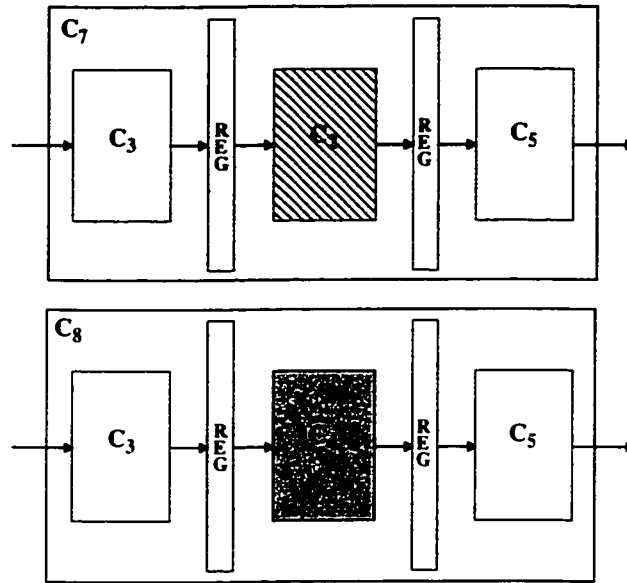


Figure 5.3: Two different compositions

5.7 Execution Model

The input to the system is an application which is to be executed on the reconfigurable hybrid system architecture. This input application is partitioned into tasks which are to be executed on the CPU and the Configurable Logic Unit. The applications tasks to be executed are then decomposed into a sequence of CLU functions(\mathcal{F}). The representation of the complete application is a task graph with functions as nodes and the edges representing the precedence constraints. If a task T_j is dependent on T_i (control or data dependency) then it is indicated as $T_j \propto T_i$. An edge is labeled with the input and the output bandwidth (β_{in} and β_{out}) of the tasks related by that edge.

This sequence of tasks is then mapped onto a sequence of configurations, σ ($\sigma_1 \sigma_2 \dots \sigma_p$, where $\sigma_i \in \mathcal{C}$). Note that it is possible to have a configuration repeat in the sequence, possibly for sequential tasks ($\sigma_i = \sigma_j$ and $i \neq j$). Algorithmic optimization techniques

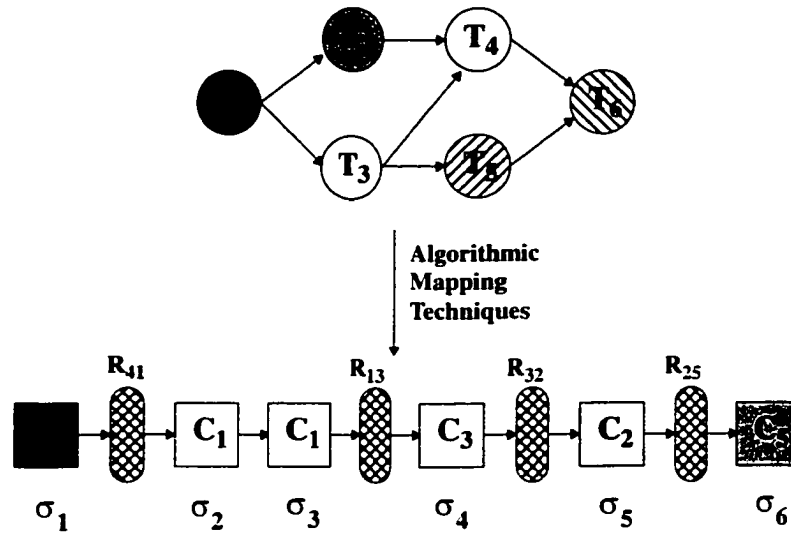


Figure 5.4: HySAM execution model

are utilized to arrive at an optimal sequence of configurations. The tasks are executed on the CPU or on the CLU. Execution of a function on the CLU involves loading the configuration onto the CLU and communicating the required data to the CLU. After executing in a configuration C_i , to execute in a different configuration C_j , the CLU has to be reconfigured which takes R_{ij} time. The total execution time is the time spent executing in each configuration and the time spent in reconfiguration between the executions.

Chapter 6

Mapping Techniques

Computer science is no more about computers than astronomy is about telescopes.

– E.W. Dijkstra

The problem of mapping operations(tasks) of a loop to a configurable system involves not only generating the configurations for each of the operations, but also reducing the overheads incurred. The sequence of tasks to be executed have to be mapped onto a sequence of configurations that are used to execute these tasks. The objective is to reduce the total execution time.

Scheduling a general sequence of tasks with a set of dependencies to minimize the total execution time is known to be an NP-complete problem. We consider the problem of generating this sequence of configurations for loop constructs which have a sequence of statements to be executed in linear order. There is a linear data or control dependency

between every pair of adjacent tasks. Most loop constructs, including those which are mapped onto high performance pipelined configurations, fall into such a class.

The total execution time includes the time taken to execute the tasks in the chosen configurations and the time spent in reconfiguring the logic between successive configurations. We have to not only choose configurations which execute the given tasks fast, but also have to reduce the reconfiguration time. As we have seen in the previous sections, it is possible to choose one of many possible configurations for each task execution. Also, the reconfiguration time depends on the choice of configurations that we make. Since reconfiguration times are significant compared to the task execution times, our goal should be to minimize this overhead.

6.1 Generic Mapping Problem (GMP)

GMP Problem: Given a set of tasks, T_1 through T_p ($T_i \in F$) and a partial order α on the tasks, find an optimal sequence of configurations $\sigma(\sigma_1 \sigma_2 \dots \sigma_p)$ ($\sigma_i \in C$). The goal is to minimize the execution time cost E given by

$$E = \sum_{i=1}^p (t_{i,i} + \beta_{in}^i + \beta_{out}^i + \mathcal{R}_{i,i+1})$$

where $t_{i,i}$ is execution time for task T_i in configuration σ_i , β_{in}^i and β_{out}^i denote the input and output data access cost and $\mathcal{R}_{i,i+1}$ is the reconfiguration cost from σ_i to σ_{i+1} .

6.1.1 NP-Completeness

Theorem 1 *The general scheduling problem, GMP, defined above is NP-complete.*

Proof: We prove the complexity of the GMP problem by transforming a known NP-complete problem to a simpler variation of the mapping problem defined above. The mapping problem GMP with constant cost for each reconfiguration and functions with same execution time in all configurations is considered. We do not formally establish the order of complexity of the simple problem. But, it is evident that the variable reconfiguration and execution costs only increase the complexity of the GMP problem.

In the first step, the decision problem for the GMP problem can be verified to be in NP. Given a constant K and a schedule σ , it is possible to verify whether the cost of the schedule is less than K in polynomial time. In the second step, the problem of *Sequencing with deadlines and set-up times (SS6)* from [35] can be transformed to the GMP problem. The SS6 problem is defined as:

SS6 Problem: Given a set C of compilers, set T of tasks, for each $t \in T$ a length $l(t) \in \mathbb{Z}^+$, a deadline $d(t) \in \mathbb{Z}^+$, and a compiler $k(t) \in C$, and for each $c \in C$ a set-up time $l(c) \in \mathbb{Z}_0^+$, is there a one-processor schedule σ that meets all task deadlines and satisfies that whenever two tasks t and t' are scheduled consecutively and have different compilers, then $\sigma(t') \geq \sigma(t) + l(t) + l(k(t'))$?

The transformation is carried out by defining the set of configurations \mathcal{C} in GMP based on the set of compilers C and the set of tasks \mathcal{T} in GMP based on the tasks T in SS6. The latencies of the tasks in SS6 are transformed to the execution times of the

tasks in GMP. The compiler set-up times in SS6 are transformed to the reconfiguration costs for each configuration in GMP. The deadlines $d(t)$ in the SS6 problem define the partial order α in the GMP. If a task t has an earlier deadline than t' in the SS6 problem then $t \alpha t'$ in the GMP problem. The time taken for the transformation is polynomial in the sizes of the set of tasks ($\| T \|$) and the set of compilers ($\| C \|$). The transformation proves that the simplified version of the GMP problem is itself NP-complete. Therefore, the GMP problem is NP-complete \odot .

In later sections and chapters we develop algorithms which are polynomial in time complexity for variants of the problem when the partial order α is more constrained. We examine such problems in the context of loop computations that occur in most compute intensive applications.

6.2 Loop Synthesis

Computations which operate on a large set of data using the same set of operations are most likely to benefit from configurable computing. Hence, loop structures will be the most likely candidates for performance improvement using configurable logic. Configurations which execute each task can be generated for the operations in a loop. Since each operation is executed on a dedicated hardware configuration, the execution time for the task is expected to be lower than that in software.

The loops that are mapped onto configurable logic do not have complex control or address calculation. The characteristics of such loops are outlined below:

- Constant step size for the index
- No function calls in loop body
- No pointer operations or pointer arithmetic
- No loop carried dependencies

Though not all loop constructs satisfy these constraints, many commonly occurring loops, including those in benchmark programs, satisfy these constraints. Loop transformation techniques [81] can be utilized to convert arbitrary loops to conform to the above constraints.

6.2.1 Linear Loop Synthesis

We consider the problem of generating this sequence of configurations for loop constructs which have a sequence of statements to be executed in linear order. There is a linear data or control dependency between every pair of adjacent tasks. The total execution time includes the time taken to execute the tasks in the chosen configurations and the time spent in reconfiguring the logic between successive configurations. We have to not only choose configurations which execute the given tasks fast, but also have to reduce the reconfiguration time.

In the following sections we define the problem based on our model and then develop an optimal solution for the problem of mapping a linear loop onto an architecture that does not employ a configuration cache.

6.2.2 Linear Loop Mapping Problem

Find an optimal sequence of configurations to execute a linear sequence of statements in a loop.

LMP: Given a set of tasks, T_1 through T_p ($T_i \in F$) to be executed in linear order ($T_{i+1} \propto T_i$, $1 \leq i < p$), find an optimal sequence of configurations $\sigma(\sigma_1 \sigma_2 \dots \sigma_p)$ ($\sigma_i \in \mathcal{C}$). The goal is to minimize the execution time cost E given by

$$E = \sum_{i=1}^p (t_{i,i} + \beta_{in}^i + \beta_{out}^i + \mathcal{R}_{i,i+1})$$

where $t_{i,i}$ is execution time for task T_i in configuration σ_i , β_{in}^i and β_{out}^i denote the input and output data access cost and $\mathcal{R}_{i,i+1}$ is the reconfiguration cost from σ_i to σ_{i+1} .

6.2.3 Optimal Solution

The input consists of a sequence of statements $T_1 \dots T_p$ ($T_i \in F$) and the number of iterations N . We can compute the execution times t_{ij} for executing each of the tasks T_i in configuration C_j . The reconfiguration costs R_{ij} can be pre-computed since the configurations are known beforehand. In addition there is a loop setup cost which is the cost for the system to initiate computation by the Configurable Logic Unit.

A simple greedy approach of choosing the best configuration for each task is not optimal since the reconfiguration costs for later tasks are affected by the choice of configuration for the current task. The complete solution space needs to be searched by considering all possible configurations in which each task can be executed.

We use dynamic programming to search the solution space. In the solution space the same sequence of configurations for a given sequence of tasks occurs as part of multiple larger problems. Dynamic programming can be utilized to compute the optimal solution for the complete sequence of tasks by using solutions for smaller subsequences . Once an optimal solution for executing up to task T_i is determined, the cost for executing up to task T_{i+1} can be determined. This approach is used recursively to compute the optimal solution.

Theorem 2 *Given a sequence of tasks $T'_1 T'_2 \dots T'_r$, an optimal sequence of configurations, σ , for executing these tasks **once** can be computed in $O(rm^2)$ time.*

Proof: We use the dynamic programming approach to compute the optimal sequence, σ . We define the optimal cost of executing up to task T'_i ending in a configuration C_j as E_{ij} . We initialize the E values as $E_{0j} = 0, \forall j : 1 \leq j \leq m$.

We assume that optimal solutions for executing up to task T'_i ending in all possible configurations, σ_j , are computed. Now for each of the possible configurations ($\sigma_i \in \mathcal{C}$) in which we can execute T'_{i+1} we have to compute an optimal sequence of configurations ending in that configuration, σ_j . We compute this by using the recursive equation:

$$E_{i+1j} = t_{i+1j} + \beta_{in}^{i+1} + \beta_{out}^{i+1} + \min_k (E_{ik} + \mathcal{R}_{kj}) \quad \forall j : 1 \leq j \leq m$$

We have examined all possible ways to execute the task T'_{i+1} once we have finished executing T'_i . If each of the values E_{ik} is optimal then the value E_{i+1j} is optimal. Hence

we can compute an optimal sequence of configurations by computing the E_{ij} values. The minimum cost for the complete task sequence $(T'_1 T'_2 \dots T'_r)$ is given by $\min_j [E_{rj}]$. The corresponding optimal configuration sequence can be computed by using the E matrix.

Computation of each value takes $O(m)$ time as there are m configurations. Since there are $O(rm)$ values to be computed, the total time complexity is $O(rm^2)$. \odot

Theorem 2 provides a solution for an optimal sequence of configurations to compute one iteration of the loop statement. But repeating this sequence of configurations is not guaranteed to give an optimal execution for N iterations. Figure 6.1 shows the configuration space for two tasks T_1 and T_2 and four possible configurations C_1, C_2, C_3, C_4 . T_1 can be executed in C_1 or C_3 and task T_2 can be executed in C_2 or C_4 . The edges are labeled with the reconfiguration costs and cost for the edges and configurations not shown is very high. We can see that an optimal sequence of execution for more than two iterations will be the sequence $C_1 C_4 C_3 C_2$ repeated $N/2$ times. The repeated sequence of $C_1 C_4$ which is an optimal solution for one iteration does not give an optimal solution for N iterations.

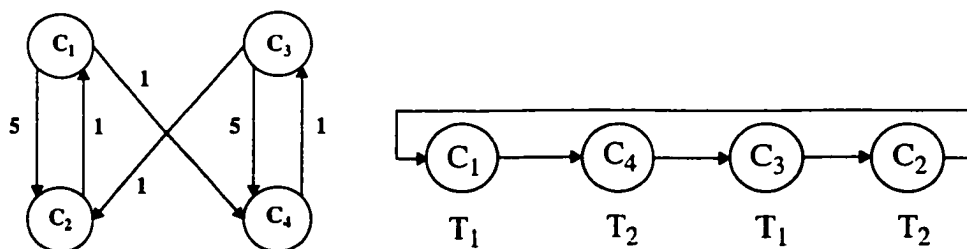


Figure 6.1: Example reconfiguration costs and optimal configuration sequence

One simple solution is to fully unroll the loop and compute an optimal sequence of configurations for all the tasks. But the complexity of algorithm will be $O(Npm^2)$, where N is the number of iterations. Typically the value of N is very large (which is desirable since higher value of N gives higher speedup compared to software execution). We assume $N \gg m$ and $N \gg p$. We also assume that $N \bmod p * m \equiv 0$ ($p * m$ divides N exactly). We show that an optimal configuration sequence can be computed in $O(pm^3)$ time.

Lemma 1 *An optimal configuration sequence can be computed by unrolling the loop only m times.*

Proof: Let us denote the optimal sequence of configurations for the fully unrolled loop by $\sigma_1 \sigma_2 \dots \sigma_{N/p}$. After executing one iteration, configuration σ_{p+1} executes task σ_1 . Therefore, task T_1 is executed in each of configurations $\sigma_1, \sigma_{p+1}, \sigma_{2p+1}, \dots, \sigma_{x-p+1}$. There are at most m configurations for each task. If the number of configurations in $\sigma_1, \sigma_{p+1}, \sigma_{2p+1}, \dots, \sigma_{N/p-p+1}$ is more than m then some configuration will repeat. Since we assumed $N \gg m$, some configuration will repeat. Therefore, $\exists y_1$ and y_2 s.t. $\sigma_{y_1+p+1} = \sigma_{y_2+p+1}$. Also, $y_2 - y_1 \leq m$.

The execution cost is a monotonically non-decreasing function since there are no negative values for any of the execution, reconfiguration or input/output costs. Let $y_3 = y_2 + y_2 - y_1$. The subsequences of tasks from $y_1 * p + 1$ to $y_2 * p$ and $y_2 * p + 1$ to $y_3 * p$ are identical. These subsequences are the same number of multiple iterations of the loop. The subsequences are identical and the first configuration for the first task

in each subsequence is same. Therefore, dynamic programming will select the same subsequence of configurations for both the subsequences of tasks. Given that

$$T_{y_1=p+1}T_{y_1=p+2} \dots T_{y_2=p} \equiv T_{y_2=p+1}T_{y_2=p+2} \dots T_{y_3=p}$$

the optimal sequence will have

$$\sigma_{y_1=p+1}\sigma_{y_1=p+2} \dots \sigma_{y_2=p} \equiv \sigma_{y_2=p+1}\sigma_{y_2=p+2} \dots \sigma_{y_3=p}$$

Applying the same argument to the complete sequence, it can be proved that all subsequences are identical. The identical subsequence is the subsequence between the first repetitions of matching task and configuration pair. Intuitively, the proof states that there exists a cycle in the sequence of configurations and the maximum length of the cycle is bounded.

There are p tasks with m possible configurations for each task. Therefore, the longest possible length of such a subsequence is $p*m$. This subsequence of $p*m$ configurations is repeated to give the optimal configuration sequence for $N * p$ tasks. Hence, we need to unroll the loop only m times. ⊙

Theorem 3 *The optimal sequence of configurations for N iterations of a loop statement with p tasks, when each task can be executed in one of m possible configurations, can be computed in $O(pm^3)$ time.*

Proof: From Lemma 1 we know that we need to unroll the loop only m times to compute the required sequence of configurations. The solution for the unrolled sequence of $p * m$ tasks can be computed in $O(pm^3)$ by using Theorem 2. This sequence can then be repeated to give the required sequence of configurations for all the iterations. Hence, the total complexity is $O(pm^3)$. \odot

The complexity of the algorithm is $O(pm^3)$ which is better than complexity by fully unrolling, $O(Npm^2)$, by a factor of $O(N/m)$. This solution can also be used when the number of iterations N is not known at compile time and is determined at runtime. The decision to use this sequence of configurations to execute the loop can be taken at runtime from the statically known loop setup and single iteration execution costs and the runtime determined N .

6.2.4 Illustrative Example

The Discrete Fourier Transform(DFT) is a very important component of many signal processing systems. Typical implementations use the Fast Fourier Transform(FFT) to compute the DFT in $O(N \log N)$ time. The basic computation unit is the butterfly unit which has 2 inputs and 2 outputs. It involves one complex multiplication, one complex addition and one complex subtraction.

There have been several implementations of FFT in FPGAs [58, 60]. The computation can be optimized in various ways to suit the technology and achieve high performance. We describe here an analysis of the implementation to highlight the key features

of our mapping technique and model. The aim is to highlight the technique of mapping a sequence of operations onto a sequence of configurations. This technique can be utilized to map onto any configurable architecture. We use the timing and area information from Garp [41] architecture as representative values.

For the given architecture we first determine the model parameters. We calculated the model parameters from published values and have tabulated them in Table 6.1 below. The set of functions(\mathcal{F}) and the configurations(\mathcal{C}) are outlined in Table 1 below. The values of n and m are 4 and 6 respectively. The Configuration Time column gives the reconfiguration values \mathcal{R} . We assume the reconfiguration values are same for same target configuration irrespective of the initial configuration. The Execution Time column gives the t_{ij} values for our model. The configuration \mathcal{C}_3 for the multiplier uses a sequential shift-add configuration to do the multiply and has a high execution cost.

Table 6.1: Representative Model Parameters for Garp Architecture

Function	Operation	Configuration	Configuration Time (μs)	Execution Time (ns)
\mathcal{F}_1	Multiply(Fast)	\mathcal{C}_1	14.4	37.5
	Multiply(Slow)	\mathcal{C}_2	6.4	52.5
	Multiply(Adder)	\mathcal{C}_3	4.8	480.0
\mathcal{F}_2	Addition	\mathcal{C}_4	1.6	7.5
\mathcal{F}_3	Subtraction	\mathcal{C}_5	1.6	7.5
\mathcal{F}_4	Shift	\mathcal{C}_6	3.2	7.5

The input application which is the FFT innermost loop is analyzed and decomposed. First, the loop statements have to be decomposed into functions which can be executed

on the CLU, given the list of functions in Table 6.1. One complex multiplication consists of four multiplications, one addition and one subtraction. Each complex addition and subtraction consist of two additions and subtractions respectively. The statements in the loop are mapped to multiplications, additions and subtractions which will result in the task sequence $T_m, T_m, T_m, T_m, T_a, T_s, T_a, T_a, T_s, T_s$. Here, T_m is the multiplication task mapped to function \mathcal{F}_1 , T_a is the addition task mapped to function \mathcal{F}_2 and T_s is the subtraction task mapped to function \mathcal{F}_3 .

When we find the optimal sequence of configurations for this task sequence using our algorithm, the solution is the configuration sequence $\mathcal{C}_1, \mathcal{C}_4, \mathcal{C}_5, \mathcal{C}_4, \mathcal{C}_5$ repeated for all the iterations. The most important aspect of the solution is that the multiplier configuration in the solution is not the fastest implementation. The reconfiguration overhead is lower for \mathcal{C}_2 and hence the higher execution cost is amortized over all the iterations of the loop. But, the configuration with the least configuration cost has a very high execution cost which does not ameliorate the lower configuration cost. The total execution time is using the optimal sequence of configurations is $N * 13.055\mu s$ where N is the number of iterations.

6.2.5 Mapping Configurations onto Multiple Contexts

Each of the operations in the loop statement might be a simple operation such as an addition of two integers or can be a more complex operation such as a square root of a floating point number. The problems and solutions that we present are independent of

the complexity of the operation. As we described in Section 5.1, a single operation can be implemented using various optimizations to provide several implementations. These different configurations can have different performance characteristics.

The mapping problem is to select the configuration to be utilized for each function and the configurations which are stored in the contexts. To select the configuration for executing a given function we can employ the greedy strategy. The greedy algorithm chooses the best possible configuration for executing a given function, i.e., the configuration with the lowest execution cost. But this configuration might have a large reconfiguration cost which increases the total execution time and gives a sub-optimal solution. For selecting the configurations to be pre-loaded the greedy strategy is still sub-optimal. Pre-loading the configuration with the highest reconfiguration cost gives a sub-optimal solution. Selecting a different configuration to be pre-loaded and using a configuration with lower execution cost can give a better solution [14]. We assume the following regarding the model as explained in Section 5.1:

1. The χ configurations are loaded on to the device at the start of the computation.
2. The active context can be configured from any of the χ configurations with a cost χ_c .
3. The pre-loaded configurations can not be modified during the execution of the complete application. Only the active context can be reconfigured externally.

6.2.6 Multicontext Loop Mapping Problem

MMP: Given a set of tasks, T_1 through T_p ($T_i \in F$) to be executed in linear order ($T_{i+1} \propto T_i, 1 \leq i < p$) and a multi-context buffer of size χ , find an optimal sequence of configurations $\sigma(\sigma_1 \sigma_2 \dots \sigma_p)$ ($\sigma_i \in \mathcal{C}$). The goal is to minimize the execution time cost E given by

$$E = \sum_{i=1}^p (t_{i,i} + \beta_{in}^i + \beta_{out}^i + \mathcal{R}'_{i,i+1})$$

where $t_{i,i}$ is execution time for task T_i in configuration σ_i , β_{in}^i and β_{out}^i denote the input and output data access cost and $\mathcal{R}'_{i,i+1}$ is the reconfiguration cost from σ_i to σ_{i+1} given by:

$$\begin{aligned} \mathcal{R}'_{ij} &= \chi_c \text{ if } \mathcal{C}_j \in \Lambda \\ &= \mathcal{R}_{ij} \text{ otherwise} \end{aligned}$$

Solution: We compute the optimal schedule σ and the set of contexts χ by using a dynamic programming approach. We first discuss how the optimal solution can be computed for a fully unrolled loop. All the iterations of the loop are unrolled to give a linear task sequence. We define the following variables:

- $E_{ij}, 1 \leq j \leq m$: the cost of executing tasks T_1 to T_i with T_i being executed using configuration \mathcal{C}_j and the configuration \mathcal{C}_j is added to the contexts in Λ if not already in Λ .

- $E_{ij}, m+1 \leq j \leq 2*m$: the cost of executing tasks \mathcal{T}_1 to \mathcal{T}_i with \mathcal{T}_i being executed using configuration \mathcal{C}_j and the configuration $cal\mathcal{C}_j$ is **not** added to the contexts in Λ if not already in Λ .
- $\Lambda_{ij}, 1 \leq j \leq 2 * m$: the set of contexts which are added to Λ for executing tasks \mathcal{T}_1 to \mathcal{T}_i with \mathcal{T}_i being executed using configuration \mathcal{C}_j .
- $|\Lambda_{ij}|$: the number of contexts in set Λ_{ij} .

The E_{ij} and the Λ_{ij} values are computed using dynamic programming. The recursive equations for computing them are given below:

$$\hat{k} = k \text{ such that } 1 \leq k \leq 2 * m \text{ and } \min[E_{ik} + \delta_{kj}]$$

δ_{kj} denotes the reconfiguration cost and can be evaluated based on the various possible scenarios:

- Configuration \mathcal{C}_j is already in cache. The reconfiguration cost is the cost of performing a context switch, χ_c .
- Configuration \mathcal{C}_j has not been cached. The reconfiguration cost is based on the set $|\Lambda_{ik}|$. If there is space in this set of configurations to be pre-loaded, then the configuration \mathcal{C}_j is added to the set and reconfiguration cost is χ_c . If the cache is already full then the full reconfiguration cost \mathcal{R}_{ij} is incurred.

The value of δ_{kj} is computed at each step as

if ($C_j \in \Lambda_{ik}$)

$$\delta_{kj} = \chi_c$$

else if ($|\Lambda_{kj}| < \chi$ and $1 \leq j \leq m$)

$$\delta_{kj} = \chi_c$$

else

$$\delta_{kj} = \mathcal{R}_{ij}$$

Given the value of \hat{k} , the E_{i+1j} and the Λ_{i+1j} values are computed as follows:

$$E_{i+1j} = t_{i+1j} + E_{i \hat{k}} + \delta_{\hat{k} j}$$

$$\Lambda_{i+1j} = \Lambda_{i \hat{k}} \cup C_j$$

if $|\Lambda_{i \hat{k}}| < \chi$ and $1 \leq j \leq m$)

$$= \Lambda_{i \text{ mink}} \text{ otherwise}$$

The minimum execution cost E and the corresponding set of contexts Λ for executing tasks \mathcal{T}_1 to \mathcal{T}_z for any z are given by:

$$\hat{j} = j \text{ such that } 1 \leq j \leq 2 * m \text{ and } \min[E_{zj}]$$

$$E = E_{z \hat{j}}$$

$$X = X_{z \hat{j}}$$

The required optimal schedule and the set of contexts can be computed by fully unrolling the loop and computing E and Λ for $z = N * p$ where N is the number of the iterations and p is the number of tasks in the loop. ⊙

6.3 Dynamic Precision Management

There are several methods of generating custom hardware configurations suited to the computations to be performed. The ability to perform variable precision arithmetic is one of the significant advantages of reconfigurable hardware.

Reconfigurable hardware such as FPGAs and various custom computing machines contain fine-grained configurable resources. Such fine-grained configurable logic can be utilized to build computing modules of various sizes. The modules can be built to

perform computations on various bit-widths. For example, it is possible to build a standard 16-bit \times 16-bit multiplier or a 8-bit \times 12-bit multiplier using reconfigurable hardware. The 8-bit \times 12-bit multiplier would consume less area and execute faster than the standard 16-bit \times 16-bit multiplier.

In configurable hardware, using higher precision usually results in wastage of resources such as logic area, time and power. For example, performing 32-bit multiplications when the operands have only 8 significant bits will typically require 16 times more area and 4 times more execution time. Redundant computations also expend more clock cycles and increase the power consumption. The ability to construct modules of required precision is one of the key advantages of reconfigurable hardware. Variable precision computations can be implemented by using a *static* approach. In the *static* approach, the precision of the operands and operation is fixed at compile time and can be different from the standard precision(e.g. 8-bit, 16-bit, 32-bit, etc.) used on microprocessors. Reconfigurable architectures also support *dynamic precision*, which is the ability of the hardware to change its precision at run-time in response to variant precision demands of the algorithm.

Applications are typically developed to perform operations on standard 32-bit variables. The precision of the operands and the operations is sufficient to guarantee the correctness of the operations in the worst case. But in most applications, the actual precision required for computations is usually much lower than the precision implemented. This is typically the case in computations which accumulate values as the computations

progress, as in iterative computations such as loops. The precision of the operands increases as the iterations of the loops progress. Loop computations offer the most potential for pipelining and parallelizing in most applications. Configurable hardware is an excellent match for computations with fine-grain pipelining and parallelism. In addition to the performance benefits obtained by mapping of computations in a loop onto configurable hardware, loops can also take advantage of variable precision.

Applications are currently mapped to reconfigurable hardware either by high level behavioral compilers or exhaustive hand-tooled designs. To extract the performance advantages of configurable hardware for variable precision, the trade-offs in performing computations using a very high precision versus changing the precision of computations as the execution progresses need to be evaluated. Performing this analysis by hand and tuning the implementation to the requirements of the application entails significant effort on the part of the designer. Dynamic precision management can result in implementations with lower execution times, logic area and power consumption compared to previous approaches.

For managing dynamic precision in loop computations, intelligent choices on the use of appropriate modules from the available set of modules with different precision need to be made. These configurations then have to be scheduled to achieve optimal execution schedule. We consider a schedule to be optimal if the schedule has minimum *total execution time*, which includes both the execution time in various configurations

and the reconfiguration time between configurations. Currently, a framework for managing dynamic precision computations for any class of computations does not exist. We develop such a framework for loop computations in this thesis [13].

In Section 6.3.1 we give an overview of our approach to the *dynamic precision* management problem. Each of the steps in our approach are then described in detail in the later sections. Analysis of the required precision for loop computations is discussed in Section 6.3.2. The variable precision loop mapping problem is defined and our Dynamic Precision Management Algorithm (DPMA) for computing the optimal schedule is presented in Section 6.3.6. We illustrate the utility of our approach by showing an example mapping in Section 6.3.9.

6.3.1 Overview of Dynamic Precision Variation

This section details an approach to managing the task of adapting the precision of the implementation to that of the application. An overview of our approach is shown in Figure 6.2. We focus our efforts on dynamic precision management for loop computations since they are the most compute intensive tasks in typical applications. For the loop computations in applications, we describe an approach to determine the required precision using theoretical analysis and run-time instrumentation. The required precision for the computations in a loop can be expressed as the variation in precision as the iterations of the loop progress. We introduce the concept of the *precision variation curve* to represent this variation. The *precision variation curve* for the operations and operands in the

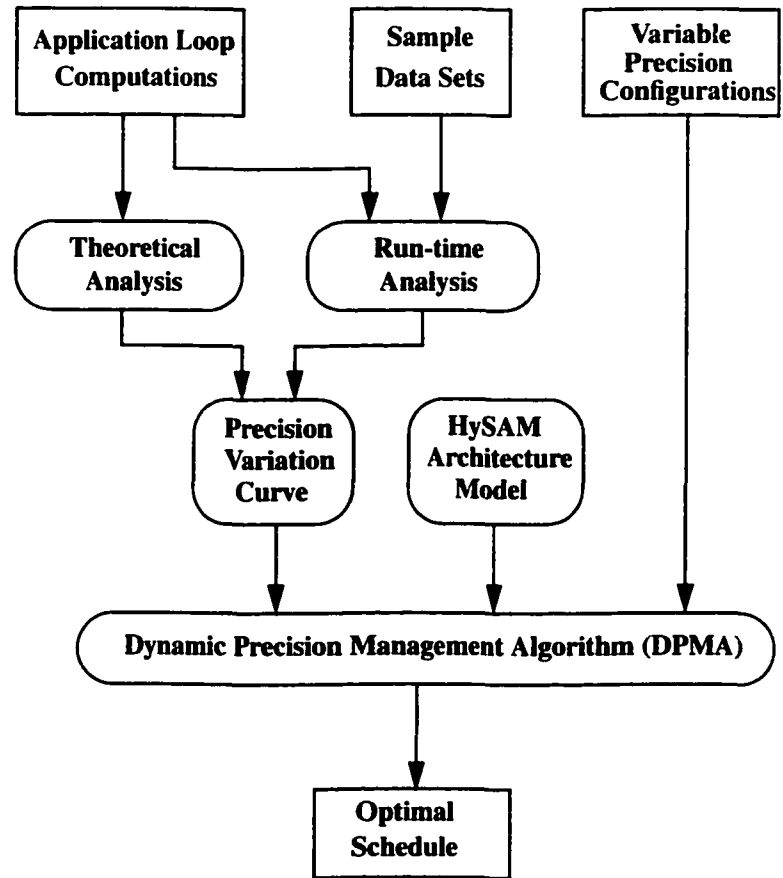


Figure 6.2: Overview of our approach for dynamic precision management in loops (shaded and rounded regions indicate our contributions)

loop can be identified either by theoretical analysis or by run-time analysis as described in Section 6.3.2.

Given the required precision for the iterations of the loop, we need to determine the mapping of the iterations to a set of configurations which are used to execute the operations in the loop. For each iteration the precision of the configuration which executes the iteration should be equal to or greater than the required precision for that iteration. The configurations are chosen from the set of library components or parameterized modules that are provided for the architecture.

Given the requirements for the precision of the computations and the available module configurations, we compute the set of configurations and the schedule of reconfigurations. We compute these by developing algorithmic techniques for precision management. First, we develop an abstract model of reconfigurable architectures, the Hybrid System Architecture Model(HySAM). This parameterized abstract model is general enough to capture a wide range of configurable systems. We define the precision management problem in loop computations using our model. A dynamic precision management algorithm is then developed to compute the optimal sequence of configurations for minimizing the total execution time including the reconfiguration time.

6.3.2 Precision Requirement Analysis

Applications typically use more precision than is necessary for computations. The precision used for operations is based on the maximum possible precision the operands

can attain. Usually, the actual precision required for computations is different from the standard precision (e.g. 8-bit, 16-bit, 32-bit etc.) supplied by microprocessors or ASIC hardware. By determining the exact precision required for operations we can reduce the resources required for the operation. This reduction can be in logic area, execution time or power consumption. Determining the required precision is not a trivial task as the actual data input to the application is not known until run-time. Evaluation of the required precision at run-time without any framework for analyzing the nature of the variation and the algorithmic techniques for utilizing this variation cannot provide any performance benefits.

The precision required for the computations in an application might not only vary with the specific operation but also change as the execution progresses. For iterative computations in which values are accumulated over the execution time of the application, the precision varies as the iterations progress. Loop computations are the most typical iterative computations which show such behavior. In addition to the varying precision, loops are the most compute intensive tasks in a program. In this thesis we focus on the varying precision of operations in loop computations. This variation can be measured by analyzing the variation of the precision of the operands and the operations as the iterations progress. We represent this variation in terms of the loop iterations by using the *precision variation curve*.

6.3.3 Precision Variation Curve

The *precision variation curve* facilitates the representation of the notion of the variation in the precision of the operands and the operation as the execution of the loop progresses. A simple method to represent such a variation is to indicate the precision of the operand for each iteration so that the precision is defined for the whole iteration space. But as we shall show in the subsequent sections, the precision usually varies very slowly as the iterations progress. Thus the *precision variation curve* can be represented by specifying the points where the precision of the operands or the operation changes.

Definition: The *precision variation curve* for a given operation or operand in a loop computation can be represented by the sequence (η_i, ϕ_i) , $1 \leq i \leq \ell$. η_i denotes the iteration number at which a change in precision takes place due to the computation. $\eta_i \leq N$ where N is the total number of iterations. ϕ_i denotes the precision required for performing iterations η_i to $\eta_{i+1} - 1$ for $1 \leq i < \ell$ and ϕ_ℓ denotes the precision required for performing iterations η_ℓ to N .

Examples of *precision variation curves* are shown in Figure 6.4. We develop theoretical and run-time instrumentation methods for determining the *precision variation curve* in the next two sections.

6.3.4 Theoretical Analysis of Loops

We can theoretically determine the *precision variation curve* for the operations in a given computation. The precision of computed variables in a loop is determined by the

```

-----
DO 10 I=1,N
  DO 20 J=1,N
    RSQ(J) = RSQ(J)+XDIFF(I,J)*YDIFF(I,J)
20   IF (MAXQ.LT.RSQ(J)) THEN
      MAXQ = RSQ(J)
      POVERR = POVERR / MAXQ
10   VIRTXY = VIRTXY + MAXQ * SCALE(I)
-----

```

Figure 6.3: Example code for simulations

precision of the variables before the iteration, the number of iterations and the operations performed on the variable. For each type of arithmetic operation, the maximum possible precision of the result can be expressed using the above values. For example, the precision of a variable X (initially 0) after N iterations of a loop which contains the statement $X = X + \Delta$ (where Δ is a constant) is bounded by

$$\pi_X \leq \pi_\Delta + \log(N + 1)$$

where π_X denotes the precision (bit size) of the variable X . The analysis is not limited to simple expressions, but extends to complex arithmetic expressions in loops. In recursive expressions in loops where the value of the variable X in iteration i is given by X_i , if

$$X_i = \delta_1 * X_{j_1} + \delta_2 * X_{j_2} + \dots + \delta_k * X_{j_k} = \sum_{l=1}^{l=k} \delta_l * X_{j_l}$$

The upper bound on the precision of X_i is given by

$$\pi_{X_i} \leq (i - 1) * \log \Delta + (i - 1) * \log k + \pi_{X_1}$$

where $\Delta = \max[\delta_1, \delta_2, \dots, \delta_k]$, the maximum of the constant coefficients. Similarly, for the expression $X = X \times \Delta$, the upper bound of precision for X with an initial value 1 and after N iterations is given by

$$\pi_X \leq N * \pi_\Delta$$

The *precision variation curve* can be computed theoretically for all expressions in loops which are polynomials of variables and constants. Since many scientific applications consist of such computations, theoretical analysis can be performed for all such applications. It is to be noted however, that such an analysis is not entirely feasible for floating point computations. But the analysis can be performed for integer and fixed point data and computations. This does not limit the applicability of the analysis or the algorithms we present later as many signal and image processing computations and several benchmark problems operate on integer and fixed point data.

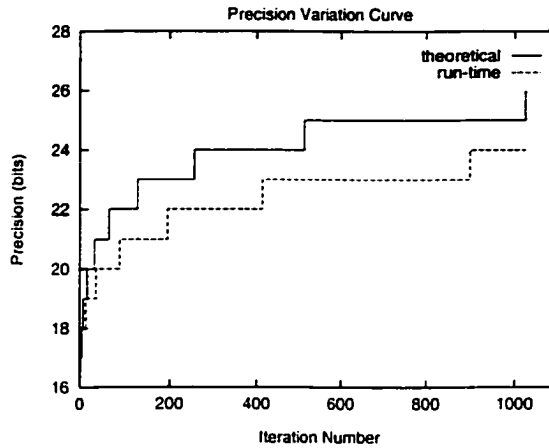


Figure 6.4: Precision Variation Curves for *RSQ* using theoretical and run-time analysis

6.3.5 Run-time Analysis

Theoretical analysis of expressions in loops computes the upper bounds on the precision of the variables and computations. This determines the minimum precision required to represent these variables. The estimates using theoretical analysis are conservative and can usually be much higher than the actual precision of the operands. For example, using the above analysis for the Fibonacci series $X_i = X_{i-1} + X_{i-2}$, we obtain $\pi_{X_i} = i - 1$ and hence, $\pi_{X_{15}} = 14$. But, $X_{15} = 610$ which needs only 10 bits. Even in the case when the bound is actually tight for expressions, the actual precision might be lower than theoretical estimate. This can occur when the data inputs are assumed to have maximum precision, but are actually randomly distributed over the complete input range.

For example, consider the code segment shown in Figure 6.3. We performed simulations with uniformly distributed random values for the 8-bit non-negative data inputs

XDIFF and *YDIFF*. The precision of the *RSQ* variable was measured by tracing the earliest iteration in which a new higher significant bit was set. Since the maximum bits in the result of $XDIFF(I, J) * YDIFF(I, J)$ are 16, the iteration in which the k th most significant bit of the result is set is given by 2^{k-16} . The *precision variation curves* obtained using the theoretical and run-time analysis are plotted in Figure 6.4. The actual precision required for the computations is significantly lower than the theoretical estimate as evident from the graph.

This run-time measurements illustrate a very important advantage in exploiting variable precision computations. The actual *XDIFF* and *YDIFF* values have significantly lower precision than the maximum possible precision of 8 bits. The assumption of maximum precision for all the input *XDIFF* and *YDIFF* values has a rolling effect on precision of other operands and operations. The repeated accumulation of the product of these numbers results in a precision difference in the final values which is much larger than the precision difference for one value. It is clearly revealed in simulations where the actual required precision is much lower than the theoretical precision.

For computations which do not have a tight bound on the precision and for computations with complex control flow, computing the required precision by using run-time statistics is a viable alternative. The application can be instrumented to measure the precision of the different variables and the knowledge can be utilized by the mapping tool or the compiler to identify the required precision at various program points. Though

we do not address the run-time mapping issues in this thesis, it is also possible to determine the precision of the operands and the operations by examining the values at run-time and modifying the precision of the operations on the fly. In this thesis, we focus on run-time precision management based on the knowledge of the required precision at compile(mapping) time. The required precision can either be analyzed automatically or can be user specified.

6.3.6 Dynamic Precision Management

Given the *precision variation curve* for the loop, we need to determine the mapping of the iterations to a set of configurations which are used to execute the operations in the loop. For each iteration, the precision of the corresponding configuration which executes the iteration should be equal to or greater than the required precision for that iteration. But, reconfiguring the hardware whenever the required precision changes can result in significant reconfiguration overheads. For architectures in which the reconfiguration times are much higher than the execution times, the reconfiguration overhead might be prohibitive. Thus, it is necessary to identify the optimal set of configurations which result in minimization of the overall execution cost, including the reconfiguration cost. Also, the set of configurations which are available for executing an operation might not encompass all the possible precision values that are required. Some of the operations will have to be executed with more precision than is necessary in the absence of configurations with the exact precision.

We present the Precision Management Problem and the Dynamic Precision Management Algorithm based on the following assumptions:

- Higher precision computations require more resources such as power, logic area and computation time(t_c).
- The required precision for the computations varies monotonically. This is true for most computations which accumulate values as the loop iterations progress. The algorithms we describe can be applied to monotonic subsequences with optimal schedules for each subsequence individually.
- The algorithm determines the optimal schedule for a *given* precision variation curve. When the actual variation is different from the precision variation curve, the schedule might not be optimal.

6.3.7 Precision Management Problem (PMP)

Input: An operation in a loop with N iterations of the loop body and the *precision variation curve* for the operation. The *precision variation curve* for a given operation or operand in a loop computation is the sequence (η_i, ϕ_i) , $1 \leq i \leq \ell$. η_i denotes the iteration number at which a change in precision takes place due to the computation. $\eta_i \leq N$ where N is the total number of iterations. ϕ_i denotes the precision required for performing iterations η_i to $\eta_{i+1} - 1$ for $1 \leq i < \ell$ and ϕ_ℓ denotes the precision required for performing iterations η_ℓ to N .

Output: An optimal schedule of configurations $\sigma = \langle \theta_j, \sigma_j \rangle$, where $1 \leq j \leq \ell$. For $1 \leq j < \ell$, σ_j is the configuration used for iterations $\theta_j \dots \theta_{j+1} - 1$ and σ_ℓ is the configuration used for iterations θ_{j+1} to N .

A schedule σ is said to be valid if it satisfies the precision requirement for all the iterations of the loop, i.e.,

$\forall k$ s.t. $1 \leq k \leq N$, if

$$\pi_i = \phi_i, \text{ for some } i \text{ s.t. } \eta_i \leq k < \eta_{i+1}$$

$$\pi_o = \pi_{\sigma_j}, \text{ for some } j \text{ s.t. } \theta_j \leq k < \theta_{j+1}$$

then $\pi_i \leq \pi_o$.

An optimal schedule has the minimum total execution cost E which includes the reconfiguration cost among all valid schedules. The cost of a schedule is given by

$$E = \sum_{j=1}^{\ell} [(\theta_{j+1} - \theta_j) \times t_{\sigma_j} + \mathcal{R}_{j-1j}]$$

where t_{σ_j} is time for executing one iteration of the loop in configuration σ_j and \mathcal{R}_{j-1j} is the reconfiguration cost between configurations σ_{j-1} and σ_j . ⊙

To minimize the total execution cost, both the execution cost and the reconfiguration cost have to be examined. The set of configurations and the schedule of reconfigurations

need to be determined. We first show that the points of reconfiguration are the subset of the points where the required precision changes, i.e., $\theta \subseteq \eta$, where $\theta = \{\theta_1, \dots, \theta_\ell\}$ and $\eta = \{\eta_1, \dots, \eta_\ell\}$.

Lemma 2 *Given the definitions in the PMP problem, the schedule σ of configurations satisfies the property $\theta \subseteq \eta$.*

Proof. Assume that $\theta \not\subseteq \eta$ in the optimal schedule σ . Then there exists at least one point of reconfiguration which is not a point of change of required precision.

$$\exists i : \theta_i \notin \eta$$

Without loss of generality,

$$\exists j : \theta_{i-1} \leq \eta_{j-1} < \theta_i < \eta_j \leq \theta_{i+1}$$

Consider the schedule σ' where the configurations are the same as σ but the reconfiguration points are different:

$$\sigma = [\theta_1 \dots \theta_{i-1} \theta_i \theta_{i+1} \theta_{i+2} \dots \theta_\ell]$$

$$\sigma' = [\theta_1 \dots \theta_{i-1} \eta_j \theta_{i+1} \theta_{i+2} \dots \theta_\ell]$$

The cost of executing one iteration in configuration σ_i is t_{σ_i} . Since we assume precision variation to be monotonic, $\pi_{\sigma_{i+1}} > \pi_{\sigma_i}$ and $t_{\sigma_{i+1}} > t_{\sigma_i}$. The difference in execution cost of the two schedules is

$$\begin{aligned}
\sigma' - \sigma &= (t_{\sigma_i}(\eta_j - \theta_{i-1}) + t_{\sigma_{i+1}}(\theta_{i+1} - \eta_j)) \\
&\quad - (t_{\sigma_i}(\theta_i - \theta_{i-1}) + t_{\sigma_{i+1}}(\theta_{i+1} - \theta_i)) \\
&= t_{\sigma_i}(\eta_j - \theta_{i-1} - \theta_i + \theta_{i-1}) \\
&\quad + t_{\sigma_{i+1}}(\theta_{i+1} - \eta_j - \theta_{i+1} + \theta_i) \\
&= (t_{\sigma_i} - t_{\sigma_{i+1}})(\eta_j - \theta_i) \\
&< 0
\end{aligned}$$

Since $\eta_j > \theta_i$ and $t_{\sigma_i} < t_{\sigma_{i+1}}$, $\sigma' - \sigma < 0$. The new schedule has lower cost and hence a schedule with reconfiguration points which is the subset of the precision change points has lower execution cost. Since σ is the optimal schedule our assumption must be incorrect. Hence, $\theta \subseteq \eta$. \odot

To determine the choice of configuration at each η_i , we can use a greedy approach where the best configuration with the required precision is chosen at each η_i . The best configuration σ_j ($\sigma_j \in \mathcal{C}_1, \dots, \mathcal{C}_m$) is given by the configuration which has the lowest execution cost t_{σ_j} . But the greedy algorithm will not provide the optimal solution due to two reasons:

- The greedy approach does not consider the reconfiguration costs which are incurred at future reconfiguration points. A configuration with higher execution cost might have a lower reconfiguration cost at the next step, making it a better choice for executing the given iterations.
- With significant reconfiguration costs, it is possible that we use a higher precision configuration than required (even if exact precision configuration is available in C), to avoid a reconfiguration step in future. The greedy approach does not consider this case and thus can result in non-optimal schedule.

6.3.8 Precision Management Algorithms

To determine the choice of configuration at each L_i , we can use a greedy approach where the best configuration with the required precision is chosen at each L_i . The best configuration C_j ($C_j \in C_1, \dots, C_m$) is given by the configuration which has the lowest execution cost t_{C_j} . But the greedy algorithm will not provide the optimal solution due to two reasons:

- The greedy approach does not consider the reconfiguration costs which are incurred at future reconfiguration points. A configuration with higher execution cost might have a lower reconfiguration cost at the next step, making it a better choice for executing the given iterations.

- With significant reconfiguration costs, it is possible that we use a higher precision configuration than required (even if exact precision configuration is available in \mathcal{C}), to avoid a reconfiguration step in future. The greedy approach does not consider this case and thus can result in non-optimal schedule.

In the following, we present an algorithm based on dynamic programming which computes an optimal schedule having the minimum execution cost including the reconfiguration cost.

Dynamic Precision Management Algorithm (DPMA)

Given the Precision Management Problem we can use Theorem 2 from previous section and the above Lemma 2 to solve the problem. The configuration in the optimal schedule changes only when the required precision changes according to Lemma 2. We define T_i as the task executing iterations η_i to $\eta_{i+1} - 1$. Given this task definitions and the set of configurations \mathcal{C} with different precisions, we can use Theorem 2 to compute the optimal schedule.

Theorem 4 *The optimal schedule σ for the PMP problem can be computed in $O(\ell m^2)$ time where ℓ is the number of points in the precision variation curve and m is the number of configurations in the set \mathcal{C} .*

Proof: We can use dynamic programming to compute the E_{ij} values for the schedule σ as described in Theorem 2. Computing one E_{ij} value takes $O(m)$ time since there are m configurations. The total number of values to be computed is $O(\ell m)$, therefore the total time complexity of the algorithm is $O(\ell m^2)$. ◻

6.3.9 An Illustrative Example

We illustrate our approach by mapping the multiplication operation from the example code segment presented in Figure 6.3.

```

-----
      DO 10 I=1,N
          ...
10     VIRTXY = VIRTXY + MAXQ * SCALE(I)
-----

```

Figure 6.5: Multiplication operation from sample code

The input data $SCALE(I)$ is an 8-bit integer. The precision of $MAXQ$ has been analyzed in Section 6.3.5. We present the same result in the form of a table in Table 6.2.

Table 6.2: Theoretical and simulated iteration numbers for $N = 1024$

P_i	L_i	L'_i		P_i	L_i	L'_i
P_r	Theore- tical	Simu- lated		P_r	Theore- tical	Simu- lated
16	1	1		22	64	195
17	2	2		23	128	412
18	4	5		24	256	897
19	8	14		25	512	-
20	16	35		26	1024	-
21	32	87				

We have abstracted the Xilinx XC6200 series device by using our model. The parameters specified are for the HySAM model and have been evaluated from XC6200 documentation [82, 57]. The footprint of each precision is given by the equation $4 \times$

Table 6.3: HySAM model parameters for XC6200 multiplier configurations

Configuration C_i	Precision $Pr(C_i)$	Time $t_{C_i} (ns)$	Reconfig. $R_{0i} (ns)$
C_1	8×8	140	5120
C_2	8×16	250	10240
C_3	8×20	300	12800
C_4	8×24	400	15360
C_5	8×28	520	17920
C_6	$8 \times 32^*$	640	20480

$row \times col$, where row and col are the precisions of the two inputs. For the configurations relevant to mapping the given operation, row is 8. Reconfiguration times are based on a 32-bit data bus running at 50MHz. It is possible to design modular configurations which can be reconfigured in lesser time using partial reconfiguration. For this mapping, we assumed that complete reconfiguration is needed for each configuration. The parameters for various multiplier configurations with different precisions are listed in Table 6.3.

We measured the total execution time for the loop computations using five different approaches. The first two approaches do not exploit the *dynamic precision* by varying the precision of the operation at run-time. The different approaches and the schedule of configurations($\langle Q_j, C_j \rangle$) in each approach are described below.

- **Raw:** The first approach uses a static configuration of $8bit \times 32bit$ precision for all the iterations of the loop.

Schedule: $\langle 1, C_6 \rangle$

- **Static:** We utilize the theoretical analysis where we determine that the highest precision required for 1024 iterations is only $8bit \times 28bit$. But the configuration is still static and is used for all the iterations.

Schedule: $\langle 1, C_5 \rangle$

- **Greedy:** We used the greedy algorithm (see Section 6.3) to compute the schedule of configurations to be utilized for the computations. The precision of the operation is varied dynamically but the greedy choice is based on the lowest execution time for each configuration.

Schedule: $\langle 1, C_2 \rangle, \langle 2, C_3 \rangle, \langle 32, C_4 \rangle, \langle 512, C_5 \rangle$

- **DPMA:** Our dynamic precision management algorithm was utilized to compute the optimal schedule using the *precision variation curve*. This approach uses higher execution cost configurations for some of the computations but reduces the overall execution cost by performing lesser number of reconfigurations.

Schedule: $\langle 1, C_4 \rangle, \langle 512, C_5 \rangle$

- **DPMA-run:** In this approach we performed run-time analysis of the loop and utilized the *precision variation curve* from the run-time analysis as the input to the algorithm. This approach can be implemented easily by adding a run-time check of the precision, which needs very small amount of additional logic and no extra clock-cycles if the precision remains within the run-time statistics. *Schedule:*

$\langle 1, C_4 \rangle$

Table 6.4: Execution times using different approaches

Algorithm	Execution Time (<i>ns</i>)	Reconfiguration Time (<i>ns</i>)	Total (<i>ns</i>)
Raw	655360	20480	675840
Static	532480	17920	550400
Greedy	468010	56320	524330
DPMA	471160	33280	504440
DPMA-run	409600	15360	424960

The execution times including the reconfiguration times are summarized in Table 6.4. The approaches using *dynamic precision* achieve significantly lower execution times compared to the Raw and Static approaches. We noticed that our DPMA algorithm executed all the iterations of the loop in the minimum time for the theoretical and run-time *precision variation curves*. The DPMA-run achieves significant speed-up by exploiting the fact that 28-bit precision is never required.

6.3.10 Application

The dynamic precision management framework gives rise to a wealth of issues which can potentially provide enormous benefits to mapping computations onto configurable hardware. Bit-serial and digit-serial computations are one class of computations which can exploit dynamic precision without large overheads. The control component of the design needs to execute the configurations for a variable number of steps based on the required precision. Run-time precision management where the control modifies the precision of the computations are being explored. Configurable logic can be utilized to

execute multiple iterations of loops in parallel in the absence of dependencies. Reduction of the logic resources due to dynamic precision management can be exploited to execute more number of iterations in parallel. Multi-context devices and configuration caches can be utilized to reduce the reconfiguration overheads by storing variable precision configurations.

Chapter 7

Mapping onto Reconfigurable

Pipelines

One does not learn computing by using a calculator, but one can forget arithmetic.

– Alan Perlis

Pipelining and parallelizing are the two main techniques employed to exploit reconfigurable hardware. Pipelined designs are well structured and map well onto configurable devices. The repetitive computations in loops can be pipelined by generating a configurable pipelined datapath for the loop body or the inner loop and mapping onto the multiple functional units in reconfigurable architectures. In this chapter we address two different variations of the problem of mapping the application onto pipelined designs.

The first problem that we address considers loop computations with loop carried dependencies that have low throughput using existing design techniques. We develop integrated mapping technique that exploits the distributed memory of the reconfigurable logic to dynamically switch data contexts. The data context switching results in interleaved execution of multiple iterations of the loops providing high throughput in spite of the presence of loop carried dependencies. For this problem we assume that the reconfigurable logic has enough resources to map the complete inner loop body onto the reconfigurable logic.

The second problem relaxes some of the assumptions made above and considers reconfigurable logic and memory resource constraints and also considers graph structured dependencies between the tasks in the loop body. The reconfigurable logic resource constraint imposes reconfiguration steps between computation stages. We develop the approach of pipeline segmentation and pipeline construction to minimize the reconfiguration overheads to address this problem. Heuristic techniques are utilized to construct multiple pipeline segments that are executed one after another with reconfiguration steps between every two computation segments.

7.1 Mapping Nested Loops

Loops with simple control and no data dependencies between different iterations of the loop are easy to compile / synthesize for high performance on a variety of architectures.

In this section, we focus on mapping nested loops that have dependencies. These dependencies do not permit existing parallelization techniques to be applied. The dependencies in such computations limit the throughput of the pipelined computations. Some examples of such loops are Infinite Impulse Response (IIR) Filters and adaptive filters. We will use the IIR as a motivating example in the remainder of the section.

We developed an approach to map nested loops by using a combination of pipelining, parallelization and our proposed optimization - *data context switching*. Each iteration of the outermost loop in the computations defines a *data context*. Data context switching interleaves the execution of the iterations of the loops. The inner loops of the nested loop are pipelined to map onto the multiple functional units of the reconfigurable architecture. To operate at a high frequency, each pipeline stage has inherent delays due to pipeline registers. The data dependency in the inner loop reduces the throughput that can be achieved due to the inherent pipeline delays. We use embedded memory blocks in the architecture to parallelize the outer loop of the computation to increase the throughput. The resulting designs have the optimal throughput of one output per cycle, utilizing reduced hardware. The mapping scales with the number of loop iterations and the amount of hardware resources. We compare the performance benefits of experimental mappings using our approach with performance that can be achieved on state-of-the-art microprocessors and DSPs.

We apply the *data context switching* technique to two diverse reconfigurable architectures. The Virtex FPGA [83] from Xilinx represents a state-of-the-art fine-grain reconfigurable architecture. The main functional units are composed using fine-grain 4-input Look Up Tables (LUTs). Chameleon System's Reconfigurable Communications Processor (RCP) represents a high-performance system on a chip with coarse-grain reconfigurable architecture. The CS2000 series architecture of the RCP family combines a 32-bit RISC CPU, proprietary Reconfigurable Processing Fabric (RPF), embedded peripheral cores, high performance system bus, and a programmable bank of I/O pins. The Reconfigurable Processing Fabric is a matrix of coarse grained 32-bit functional units that communicate using a rich hierarchical interconnection network.

7.1.1 Parallelizing Nested DSP Loops

In this section, we focus on the parallelization and pipelining of nested loop computations that have loop carried dependencies. We relax one of the conditions that we discussed in the previous chapter. In this section we permit the loop to have loop carried dependencies. The class of loops that we consider is large but is limited to regular loops which have the following characteristics:

- Loops with predetermined iteration count.
- Loops without pointer operations and pointer arithmetic.
- Loops with loop-carried dependencies in the inner loops and no loop-carried dependencies in the outermost loop.

There are a large number of application loops which satisfy these criteria. Many loops which do not satisfy these criteria can be converted to this form by rewriting without pointers and by using various loop transformations such as normalization, permutation, among others [81]. Such loops occur in several classes of applications including signal processing. An example of such a computation is an Infinite Impulse Response (IIR) filter:

$$y(n) = a_0 * x(n) - \sum_{k=1}^{10} a_k * y(n - k)$$

The code for computing one sample of this IIR filter is given below:

```
for k=1 to 10 do
    sum = sum + a[k]*y[-k];
endfor
y[0] = a[0] * x[0] - sum;
```

The input data required for the computation is $a[0], \dots, a[10]$ (constants for the filter), $y[-10], \dots, y[-1]$ (history or memory of the filter) and $x[0]$ (sampled waveform).

In typical signal processing applications, this filter is executed for a frame (a specified set of samples such as 80) and for a large number of channels (e.g. 100). The filter coefficients are different for each channel and the sampled waveform is different for each channel. The complete code where the i loop denotes different channels is given below:

```

for i=1 to 100 do
  for j=1 to 80 do
    sum = 0;
    for k=1 to 10 do
S1:      sum = sum + a[i,k]*y[i,j-k-1];
    endfor
S2:  y[i,j] = a[i,0] * x[i,j] - sum;
    endfor
endfor

```

In the following sections, we show how such loops can be mapped onto reconfigurable architectures by using pipelining and *data context switching*. The source code for the example will be used as a running example to illustrate our techniques.

7.1.2 Pipelining

In the first phase, the inner loops are transformed into a pipelined datapath. The datapath is constructed for the body of the innermost loop. In the absence of loop-carried dependencies, the loop will have a data flow graph with no cycles. The single stage datapath for the example computation is shown in Figure 7.1. The single stage datapath itself can have internal pipeline registers and internal pipeline delay. This delay is referred to as the single stage pipeline delay, δ . The example shows a datapath with a delay of two.

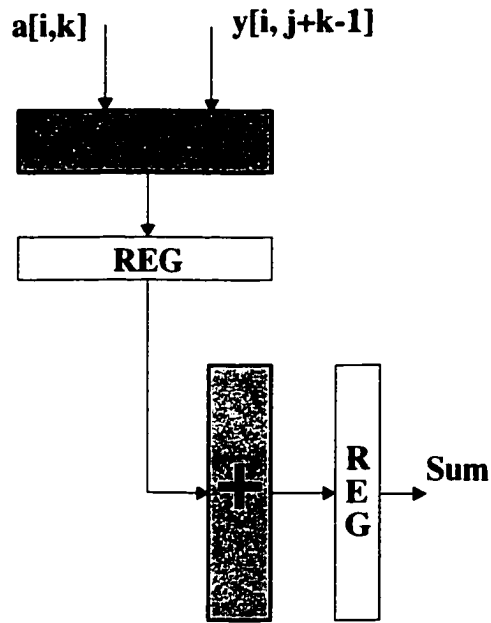


Figure 7.1: Mapping of loop body to one stage

The innermost loop is unrolled to generate the pipelined datapath for loops with feedback. The depth of the pipeline is the number of iterations of the innermost loop. As shown in Figure 7.2, unrolling the example results in a depth of ten stages. If there was no loop-carried dependency in the j loop then this pipeline can be mapped and executed on the hardware to generate results at a throughput of one output/cycle. But, the loop-carried dependency results in a feedback path from the last stage of the pipeline to the first stage. This feedback path and the multiplexer to accommodate the selection of the feedback and the input are shown in Figure 7.2.

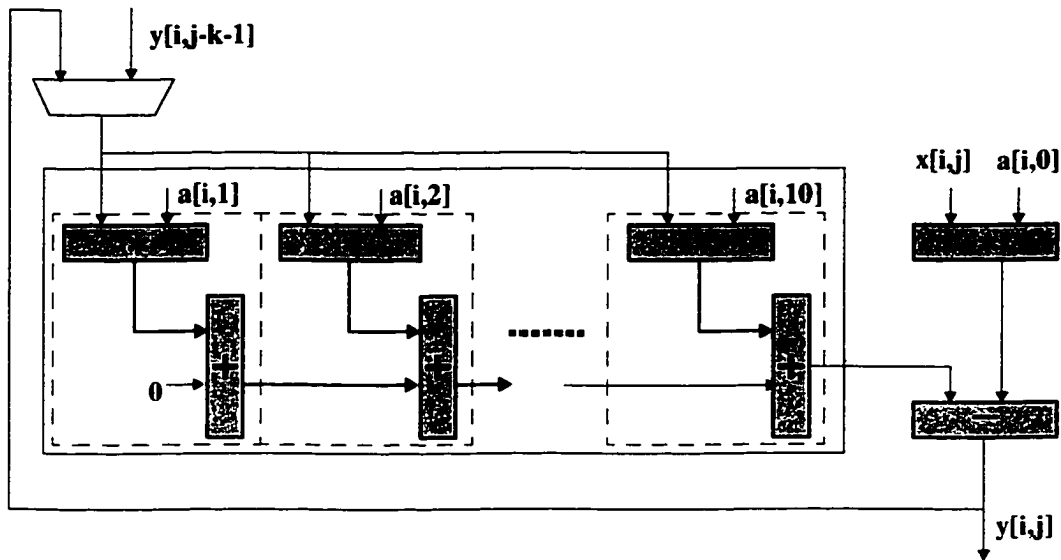


Figure 7.2: Pipelined datapath of all ten stages

7.1.3 Limitations on the Throughput

In the sample code, statement S_1 has a loop-carried dependency due to the feedback. This can be observed based on the transitive dependency property. Consider the execution of the $S_1^{1,12,1}$.

$$S_1^{1,12,1} \prec S_2^{1,11,*} \text{ and } S_2^{1,11,*} \prec S_1^{1,11,10} \Rightarrow S_1^{1,12,1} \prec S_1^{1,11,10}$$

There is a loop-carried dependency in the j loop. Each pipeline stage has inherent delay buffers (registers) to satisfy timing constraints of the functional units and the routing. The design operates at a much lower frequency without these delay buffers. A new sample cannot be fed into the pipeline every cycle due to this dependency. Hence, the

throughput of the pipeline reduces to the delay of each pipeline stage. Also, the outermost loop (i loop) will have to be executed sequentially after finishing the complete j loop. The computation cannot be interleaved due to the loop-carried dependency.

Let δ denote the pipeline stage delay and N_r denote the number of iterations of the iterations of the r loop ($r \in \{i, j, k\}$). The throughput of the pipeline is $\frac{1}{\delta}$ and total time to execute the loop in number of cycles is given by

$$T_{pipe} = \delta * (N_k + N_j) * N_i$$

On typical DSP engines and microprocessors, loop transformation does not provide any performance benefits. Loop unrolling of the k loop can provide additional instructions in the basic block for more Instruction Level Parallelism (ILP). But, the memory bandwidth required for executing each computation, S_1 , limits the performance even if there are multiple functional units. Simple mapping onto reconfigurable architectures will also face similar limitations in spite of multiple functional units.

Loop interchange of the i and j loop only increases the memory bandwidth requirements. The intermediate values computed for each frame will have to be stored and fetched later from memory. In the following section, we address the throughput improvement of the pipeline by using a loop interchange transformation and interleaving. But, unlike DSPs and microprocessors, reconfigurable architectures possess embedded memory blocks. This embedded memory is utilized to perform *data context switching* to eliminate the memory bandwidth problem.

7.1.4 Data Context Switching (DCS)

The outermost loop (i loop) in the example does not have any loop-carried dependencies. Using the pipelined datapath, the outermost loop can be executed sequentially. The loop can be parallelized by replicating the hardware mapping and executing a subset of the iterations on each replicated pipeline. But, there is a limit on the hardware resources that are available on most reconfigurable architectures, including Virtex and Chameleon RPF. We developed an alternate technique to improve the throughput of the pipelined datapath - *data context switching*.

Each iteration of the outermost loop defines a different *data context*. Each data context differs in the data inputs and constants that are used in the computation. In the example computation, each context differs in the x and y input data and the filter coefficients a . By using data context memories, we simulate multiple versions of the pipeline computing on distinct data sets. Figure 7.3 and Figure 7.4 show the differences in the data flow in the two approaches. In the *data context switching approach*, the computational units are switching between a different data context each cycle.

Data context switching uses the embedded and distributed local memory to store the context information and retrieve it at appropriate cycles in the computation. *Data context switching* is achieved by using four steps:

- Rescheduling the input data flow.
- Storing the constant data in distributed local memories.
- Inserting data context memories in each pipeline stage.

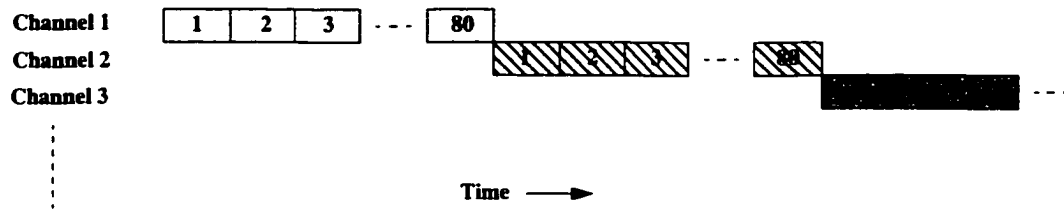


Figure 7.3: Dataflow in original design

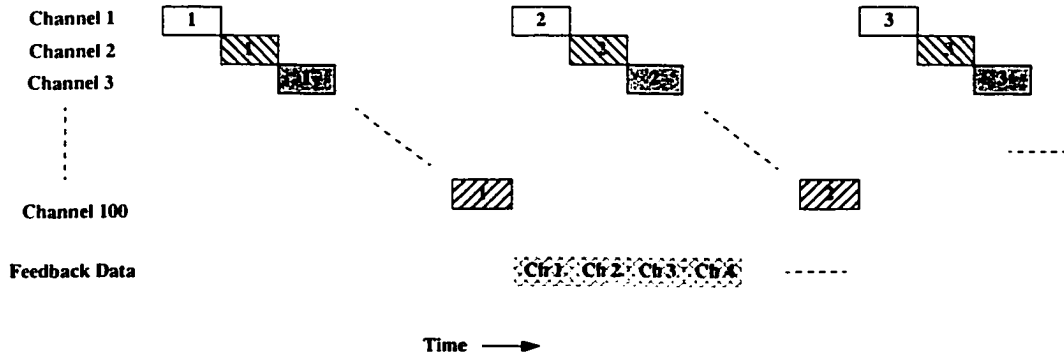


Figure 7.4: Dataflow in dynamic context switching

- Balancing the feedback path delay.

Rescheduling input data: The pipelined design in the previous section computes the full frame (j loop) for one channel (i loop) at a time. In our *data context switching* approach we compute one sample of each channel at a time. This interleaves the execution of the iterations of the outermost loop (i loop) with the execution of the j loop. In conventional architectures this increases the memory bandwidth requirements. In reconfigurable architectures, the inherent pipeline registers in each stage and the data context memories are utilized to store intermediate results. The inherent pipeline delays in each stage are exploited to switch the data context on which the pipeline is operating in each cycle.

Figure 7.3 shows the input data arriving at the first stage in the pipeline in existing pipelined design. Figure 7.4 shows the input data arriving at the first stage in our dynamic context switching approach.

On the Chameleon architecture, the data can be manipulated in the external memory by the ARC processor. The data can be streamed through the programmable I/O (PIO) pins in Chameleon. The data can also be fetched into the on-chip local memory (RAMs in Virtex and LSMs in Chameleon) and rescheduled by addressing the data in a different order.

Memories for constants: The constants used in computations of all the outer loop iterations (such as filter coefficients) are stored in local on-chip memories. This necessitates additional logic for addressing the local memories and accessing the correct constants for each operation in each stage. Since these are in local distributed memories, they can be updated by using distributed computational units. This can be exploited for computations in which the constants change, such as adaptive filters.

This memory for constants is shown in Figure 7.5. A context index counter is utilized to extract the correct constants for each step of the computation as shown in the figure.

Data context memories: At any cycle in the computation, the corresponding functional unit in each pipeline stage operates on the same data context (iteration of outermost loop). The intermediate results flow through the pipeline stage and arrive at the next

stage in the pipeline after a delay δ . To compute on the same data context in the corresponding functional unit in each stage, this pipeline stage delay should equal the number of contexts that are being computed. To match these, we introduce distributed memories as FIFO buffers which store the context information in each pipeline stage. The re-timed data flow ensures that the correct constants and input operands for each context appear at the inputs of each functional unit in the pipeline.

The resulting datapath of one pipeline stage after applying the data context switching optimization is shown in Figure 7.5. The single pipeline stage shows how the data context information is distributed throughout the single stage.

Balancing delay paths: The re-timing of each pipeline stage results in the feedback data arriving at a wrong data context. Delay buffers need to be inserted in the feedback path to balance the two datapaths. The size of the delay buffer in the feedback is the difference in the pipeline stage delay (δ) and the delay in the computations in the j loop (multiply and subtract in the example). When *data context switching* is used, the pipeline stage delay, δ , is equal to the number of contexts.

Figure 7.4 shows how the appropriate feedback data for a specific channel has to arrive at a specific time in the data flow. When the multiplier in the first stage needs to compute using sample 2 from channel 1, the intermediate feedback result for that specific channel should be available on the feedback path shown in Figure 7.2. This is illustrated in the dataflow diagram in Figure 7.4.

The context memories or the delay buffers can be implemented at a low cost on the Chameleon RPF by using the LSMs. The LSMs are used to store the intermediate results. By using a difference of $d + 1$ in the read and write addresses into the LSMs, a FIFO of size d can be implemented. The extra cycle is the inherent turnaround delay for reading and writing into the same location in the LSM.

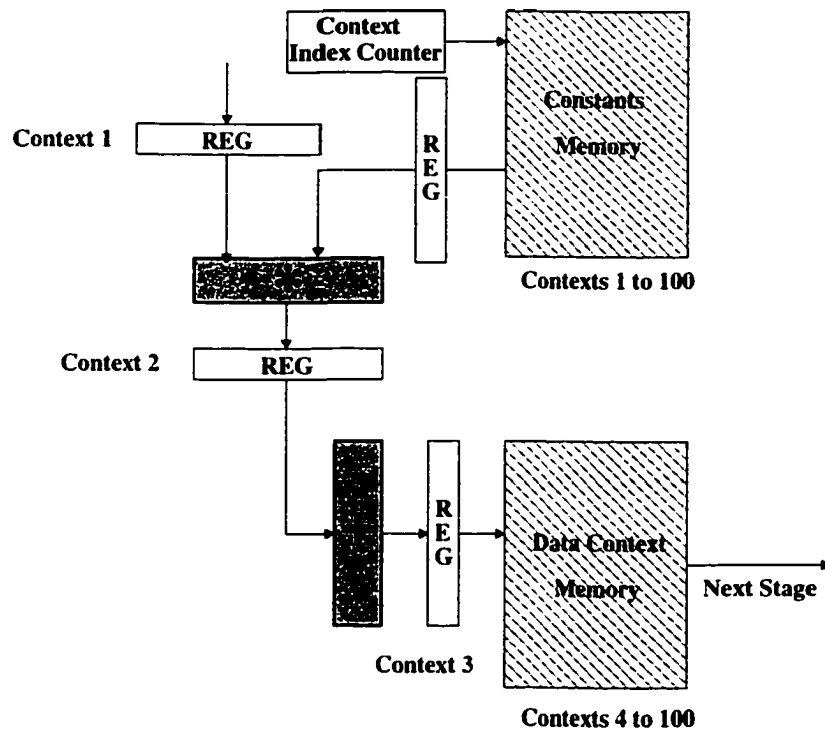


Figure 7.5: Optimized datapath for one stage

The latency of the pipeline increases to $N_k * N_i$ but the total number of computation cycles becomes:

$$T_{dcs} = (N_k + N_j) * N_i$$

The speedup achieved for executing all the iterations of the loops using *data context switching* is δ . On the Chameleon RPF, the most aggressive pipelined design has a δ of 6. In typical reconfigurable hardware (such as Virtex FPGAs) implementations δ is usually much higher due to pipelined functional units with several stages (such as 5-stage pipelined multiplier).

7.1.5 DSP/Microprocessor Implementations

The instruction schedule that can be obtained on a RISC or DSP processor is limited by the bandwidth that can be achieved from memory/registers. The number of data values operated on in the nested loop computation make it difficult to store all values in registers. The filter coefficients in the example computation are the values typically expected to be available in registers. If the innermost loop body is mapped onto a schedule of delay T_{body} then the computation on a DSP or microprocessor will run in the following number of cycles:

$$T_{dsp} = T_{body} * N_k * N_j * N_i$$

where N_i, N_j , and N_k denote the number of iterations of the i, j and k loop respectively. On a microprocessor the example loop body can take up to 20 cycles to execute. Reconfigurable architectures attain considerable speedup over DSPs and microprocessors as evident from the number of cycles.

7.1.6 Performance Summary

The analytical performance measures for the different approaches are summarized in Table 7.1.

Table 7.1: Analytical performance summary

Approach	Value	Cycles
Pipelined	T_{pipe}	$\delta * (N_k + N_j) * N_i$
DCS	T_{dcs}	$(N_k + N_j) * N_i$
DSP/Microprocessor	T_{dsp}	$T_{body} * N_k * N_j * N_i$

7.1.7 Performance Results

We performed experiments on various platforms to validate the performance benefits of *data context switching*. The example DSP nested loop is an Infinite Impulse Response (IIR) filter that occurs in several signal processing applications including the Voice over IP (VoIP) standards. A 10-stage IIR filter is the speech synthesis filter in the G.729 vocoder standard for voice compression. This synthesis filter is a significant component of the computations in both compression and decompression in the G.729 standard.

We mapped the nested loop onto different architectures to obtain performance results. The example DSP nested loop is generic enough in computational structure to reflect the performance benefits that can be achieved for a large class of computations.

We compare the performance of our approach by mapping using the standard pipeline approach and our *data context switching* approach. The basic features of the architectures that we utilized for mapping are outlined in Table 7.2. The Virtex timings are dependent on the design.

Table 7.2: Platform characteristics

Platform	Frequency (MHz)	Cycle Time (ns)
UltraSPARC-II	400	2.5
DSP C62x	300	3.33
Chameleon	125	8
Virtex	up to 200	up to 5ns

Table 7.3: Performance Results and Speedups

Platform	Approach	Cycles	Speedup (in cycles)	Time (μsec)	Speedup (in time)
UltraSPARC-II		800000	1.0	2000	1.0
DSP		200000	4.0	660	3.0
Virtex	Standard	81000	9.8	1426	1.4
Virtex	DCS	9000	88.9	158	12.7
Chameleon	Standard	54000	14.8	432	4.6
Chameleon	DCS	9000	88.9	72	27.8
Chameleon	DCS+Double	4500	177.8	36	55.6

Table 7.3 shows the performance of various architectures and the different mapping techniques on the reconfigurable architectures. The results marked as DCS indicate the results obtained using *data context switching*. The base case for comparing the speedup is the UltraSPARC-II implementation.

The exact cycle counts of the applications on the microprocessor and DSP are based on estimates. The code was compiled using the standard tools to obtain the estimates. The basic loop body timing, T_{body} , for the microprocessor and DSP is 10 and 2 cycles, respectively. This was obtained based on the most aggressive scheduling of the instructions. Memory bandwidth and cache effects on microprocessors were not considered and can only decrease the performance of the microprocessor and DSP.

The mapped design on the Virtex had a pipeline single stage delay of 9 cycles. The design runs at 56.7MHz on a -6 speed grade part which is the fastest Xilinx device. The local memory blocks were implemented as a combination of distributed logic cells and the BlockRAM available on the Virtex. This was necessary to balance the usage of the BlockRAM and the distributed memory. A design using only BlockRAMs would fit only on the largest Virtex device, V1000. On a V600 device, 91% of the BlockRAMs and 43% of the logic cells are utilized by combining distributed RAMs and BlockRAMs.

The Chameleon implementation was developed using the C~SIDE software tools [72]. Two stages of the pipeline (with one multiply-accumulate each) are mapped onto one tile achieving maximum tile usage. The complete design was mapped onto two slices of the Chameleon chip. The design uses 50% of the DPUs and 31% of the LSM memories. The control FSM is simple and is the same for each pipeline stage. Two versions of the design can be mapped onto the Chameleon chip with the available reconfigurable resources. These can operate in parallel to achieve twice the speedup. This speedup is reflected in the last row (DCS+Double) in Table 7.3.

Using standard pipelining approach on reconfigurable architectures, we obtain speedup of 4.6 over UltraSPARC-II. Using our *dynamic context switching* (DCS) approach, we obtain speedup of up to 27.8 over UltraSPARC-II implementation in actual execution timings (in spite of lower clock speed). The optimized Chameleon mapping achieves a speedup of 9.2 over state-of-the-art DSP architecture which is extensively optimized for such nested loops. By fully utilizing the resources and using two duplicate versions

of the mapping, it is possible to further improve the performance by a factor of 2. This is illustrated in the last row (DCS+Double) of Table 7.3. Chameleon chip can achieve a speedup of 55.6 over UltraSPARC-II implementation.

The results indicate that reconfigurable architectures can achieve impressive speedups over microprocessors by exploiting the reconfigurable logic resources. Our novel *data context switching* approach can significantly enhance the speedup that can be obtained on reconfigurable architectures by exploiting the distributed memory resources. The class of signal processing computations we considered are word-oriented. Chameleon architecture has coarse-grain functional units and performs better than Virtex architecture for such signal processing applications.

7.2 Reconfiguring Pipelines

The speed-up that can be obtained by using configurable logic increases as the computations in a loop increase. But, the configurable resources that are available can be lower than the required resources to pipeline all the computations in the loop. In this case, the pipeline has to be segmented to run some of the pipeline stages and reconfigured to execute the remaining computations.

In this section, we consider loops which do not have loop carried dependencies. Such loops do not have any dependencies between different iterations of the loop. Loop transformations can be applied to remove some existing loop carried dependencies. We also assume that the number of iterations to be executed is significantly larger than the

number of pipeline stages. Hence, the cycles involved in filling and emptying the pipeline are insignificant compared to the actual execution cycles of the pipeline stages.

The execution of the complete loop can be decomposed into multiple segments, where a fixed number of iterations of each segment are executed in sequence starting from the first segment. Each segment consists of multiple pipeline stages. The logic is reconfigured after each segment to execute the next segment. The intermediate results from each segment execution are stored in memory. The execution of the sequence of segments is repeated until the required number of iterations of the loop are completed. We assume that the reconfiguration of the different segments is controlled by an external controller (e.g. a host processor).

7.2.1 Definitions

Reconfigurable Logic Memory HySAM model defines the memory available in the system as \mathcal{M} . In this section, we assume that the intermediate memory denotes the memory available in the reconfigurable logic portion of the architecture.

Input Task Specification A dependency graph $G(V, E)$ of the application tasks of the loop to be executed for N iterations. Each task node v_i denotes the operation to be performed on the inputs specified by the incoming edges to the node. The directed edge e_{ij} from v_i to v_j denotes the data dependency between the two nodes. The weight w_{ij} on each edge denotes the number of bits of data communicated between the nodes.

Output Pipeline Configuration A sequence σ of pipeline segments $\sigma_1, \sigma_2, \dots, \sigma_p$ where each segment $\sigma_i (1 \leq i \leq p)$ consists of q number of stages $s_{i1}, s_{i2}, \dots, s_{iq}$. The pipeline stages are the mapping of the computational task nodes V to configurations of the device. Each of the stages s_{ij} is the configuration which executes a specific task in the input task graph. The size of a pipeline stage is given by the length l_{ij} and the width and w_{ij} .

Segment Clock Speed Each pipeline segment σ_i can be executed at a different clock speed f_σ , depending on the maximum clock speed at which the stages in that segment can operate.

Segment Data Output A pipeline stage s_{ij} has global outputs if any of the outgoing edges from a task node are to a node that is not mapped to the same pipeline segment. The size of the segment data output $DO_i (1 \leq i \leq p)$ of all the pipeline stages in a segment $\sigma_i (1 \leq i \leq p)$ is given by the sum of all the global outputs of the stages in the segment.

Segment Iteration Count The number of iterations N_σ for which each pipeline segment is executed before reconfiguring to execute the next segment. N_σ depends on the size of the available memory to store the intermediate results. We assume that the initial and final results are communicated from/to external memory.

$$N_\sigma = \min_i \left\{ \frac{M}{DO_i + DO_{i+1}} \right\} \quad 1 \leq i \leq p - 1$$

Reconfiguration Cost The reconfiguration cost R_{loop} is the total cost involved in reconfiguring between all the segments of the pipeline configuration. This includes the cost of configuring between the last segment and the first segment if $N > N_\sigma$. The reconfiguration cost between any two segments is given by the difference in the two pipeline configurations. Partial reconfiguration of the device in columns is assumed in our computation. We use the number of logic columns in which the configurations are different as the measure of the reconfiguration cost. When the corresponding stages in different segments are dissimilar, the reconfiguration cost accounts for the multiple adjacent stages that need to be reconfigured.

Total Execution Time The total execution time E is given by the sum of the execution times for each segment and the total reconfiguration time.

$$E = N * \left(\sum_{i=1}^p \frac{1}{f_{\sigma_i}} \right) + \frac{N}{N_\sigma} * R_{loop}$$

7.2.2 Pre-processing and Mapping

In this phase the computation tasks in the input DAG are mapped onto components of the given logic device. The components are chosen from the set of library components available for executing the given application tasks in the task graph. Different components can have different logic-area/execution-time tradeoffs and could potentially have

different degrees of pipelining and footprint on the device after layout. The library component of the highest degree of pipelining which satisfies other constraints specified by the task graph (such as precision of inputs) is chosen for a task.

Our proposed approach is illustrated using the mapping and scheduling of the N-body simulation application and the FFT butterfly computation. The resulting task graphs after this phase with the dependency edges are shown in Figure 7.6. In the graph the operations are represented as A - Addition, M - Multiplication, S - Subtraction and Sh - Shift right by 4 bits (Divide by 16). The operations in the graph are all 16 bits so the weights on the edges are not indicated.

7.2.3 Partitioning

The partitioning phase generates multiple partitions where size of each partition is smaller than the size of the device. This phase attempts to optimize two criterion - (1) maximize the size of the partition (2) minimize the weight of the edges between partitions. The first criterion improves the logic utilization and the second criterion reduces the memory required to buffer intermediate results generated by each partition (pipeline segment). A sketch of the partitioning algorithm is given below without the intricate details.

A heuristic based multi-way partitioning is used to incrementally generate each of the partitions. The largest size node is chosen from among the list of *Ready* nodes (whose inputs have been computed) to be added to the current partition. When no more nodes can be added to the current partition, a new partition is initiated. For adding a

Ready node v_i to a partition π_j , the heuristic uses the following sums of weights of edges:

- ω_1 : weight of *in* edges to v_i from nodes in π_j
- ω_2 : weight of *in* edges to v_i from nodes not in π_j
- ω_3 : weight of *out* edges from v_i to nodes in π_j
- ω_4 : weight of *out* edges from v_i to neighbors of π_j

v_k is a neighbor of π_j if there is an edge from a node in π_j to v_k and $v_k \notin \pi_j$

- ω_5 : weight of *out* edges from v_i to nodes not in π_j and not neighbors of π_j

The node chosen is the node with maximum value of $\omega_1 + \omega_3 + \omega_4 - \omega_2 - \omega_5$. The primary inputs and outputs are not considered in computing the weights. The largest node which fits in the current partition satisfying the above condition is added to the current partition. Ties are broken by using the height of the node and the different weights of edges listed above. The resulting partitions are illustrated by the partition number on the nodes of the graph in Figure 7.6.

7.2.4 Routing Considerations

The algorithm for the partitioning of the task graph assumes that there are enough routing resources to communicate between the different pipeline stages and from pipeline stages to the memory. Some of the pipeline stages might have global inputs and outputs. These are data inputs and outputs which are not to adjacent pipeline stages, but

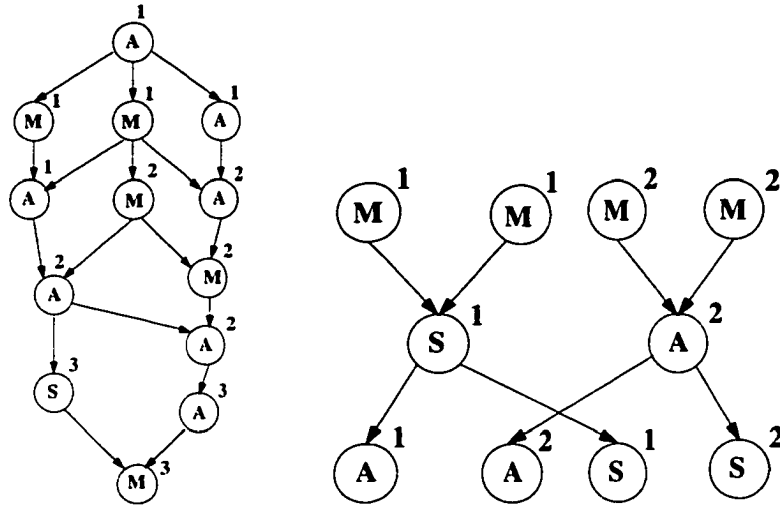


Figure 7.6: (a) N-body simulation task DAG and (b) FFT task DAG with partition numbers

from/to either non-adjacent stages or from/to memory. Some of the data outputs from the pipeline stages might have to be buffered (using registers) before they are consumed in the later stages.

Routing resources are an important consideration when mapping communication between non-adjacent pipeline stages. In our experiments we have discovered that FPGAs such as Virtex [83] are routing and register rich and can support most pipeline-able designs. The number of bits of data computed in each stage is typically less than or equal to the number of logic cells utilized. Hence, the stage to stage communication has enough routing resources by using nearest neighbor interconnect. Extra routing and logic resources (for buffering and multiplexing) have to be utilized for data values communicated across non-adjacent pipeline stages. In the partitioning algorithm, the remaining area in a partition is reduced to reflect the buffering requirements.

A limitation of our approach is that partitions might have bad memory performance when the computation is highly irregular or there are a large number of data dependencies in the DAG. The approximation of routing resources results in infeasible designs in some cases. But, for most applications, the circuits were finally mapped within the available logic and routing resources.

7.2.5 Pipeline Segmentation

The configuration of the pipeline is generated from the partitions that are computed in the previous phase by the algorithm in Figure 7.7. Each partition is utilized to generate one segment of the pipeline. The goal in the segmentation phase is to generate permutations of the pipeline stages in each segment to reduce the reconfiguration costs across segments. We use the heuristic of matching the corresponding stages of the different pipeline segments. In each partition, the nodes of the same height have the flexibility of being mapped in any order onto the pipeline. In addition, once a node has been mapped onto the pipeline, its successors from the same partition can also be mapped.

The algorithm proceeds by first identifying the list of tasks from each partition that are *Ready* to be scheduled. A task node is *Ready* if all of its predecessors have already been scheduled onto the segment. At the next step, a maximal matching set of task nodes are identified from the set of all *Ready* lists from all *Partitions*. A maximal matching set corresponds to the task node which occurs in most partitions. This step schedules similar nodes from different partitions onto the different segments. This

Table 7.4: Schedules for N-body simulation (a) S_0 : Greedy Scheduling (b) S_I : Schedule after segmentation

Segment 1	A	M	M	A	A
Segment 2	M	A	M	A	A
Segment 3	S	A	M	*	*

Segment 1	A	M	M	A	A
Segment 2	A	M	M	A	A
Segment 3	A	S	M	*	*

Table 7.5: Schedules for FFT (a) S_0 : Greedy Scheduling (b) S_I : Schedule after segmentation

Segment 1	M	M	M	*	*
Segment 2	M	S	A	A	A
Segment 3	S	S	*	*	*

Segment 1	M	M	S	A	S
Segment 2	M	M	A	A	S

enables the reduction in the reconfigurations costs at runtime. The *Ready* lists are updated before scheduling the next set of nodes. The resulting pipeline schedules with the different segments are shown in Table 7.4(b) and Table 7.5(b).

7.2.6 Performance Results

We evaluate the performance of our techniques by comparing them with a greedy heuristic based on list scheduling. The greedy schedule chooses the largest available *Ready* node as the next stage of the pipeline. A new pipeline segment is initiated when no more nodes can be added to the current segment. The resulting schedule is shown in Table 7.4(a) and Table 7.5(a). We utilize the modules and the parameters from the Virtex component libraries [83]. Some of the modules utilized are tabulated below in Table 7.6. The number of pipelined stages, precision of the inputs and the size of the module when mapped onto the device are listed in the table.

```

1: Function Segmentation(G, Partition)
2:  $\forall v_i : \text{Mapped}(v_i) \leftarrow \text{FALSE}$ 
3:  $\text{Num\_Partitions} \leftarrow |\text{Partition}|$ 
4: repeat
5:   for  $i = 1$  to  $i = \text{Num\_Partitions}$  do
6:      $\text{Ready}[i] \leftarrow \{v_j | v_j \in \text{Partition}[i] \text{ and}$ 
7:        $\forall v_k : v_k = \text{Predecessor}(v_j) \text{ and } \text{Mapped}(v_k)\}$ 
8:   endfor
9:   for  $i = 1$  to  $i = \text{Num\_Partitions}$  do
10:    for all  $v_j \in \text{Ready}[i]$  do
11:       $\text{Count}(v_j) \leftarrow \sum_{l=1}^{\text{Num\_Partitions}} |\{v_k | \text{Type}(v_k) = \text{Type}(v_j) \text{ and}$ 
12:         $v_k \in \text{Ready}[l]\}|$ 
13:    end for
14:  end for
15:   $V_{\text{curr}} \leftarrow \text{null}$ 
16:  for  $i = 1$  to  $i = \text{Num\_Partitions}$  do
17:     $v_{\text{sel}} = v_j | v_j \in \text{Ready}[i] \text{ and } \forall v_j : \max\{\text{Count}(v_j)\} \text{ and } v_j \in V_{\text{curr}}$ 
18:    if  $v_{\text{sel}} = \text{null}$  then
19:       $v_{\text{sel}} = v_j | v_j \in \text{Ready}[i] \text{ and } \forall v_j : \max\{\text{Count}(v_j)\}$ 
20:    end if
21:     $\text{Segment}[i] \leftarrow \text{Segment}[i] \oplus v_{\text{sel}}$ 
22:    if  $v_{\text{sel}} \neq \text{null}$  then
23:       $V_{\text{curr}} \leftarrow V_{\text{curr}} \cup v_{\text{sel}}$ 
24:       $\text{Mapped}(v_{\text{sel}}) \leftarrow \text{TRUE}$ 
25:    end if
26:  end for
27: until ( $\forall i : \text{empty}(\text{Partition}[i])$ )

```

Figure 7.7: Algorithm to generate the pipeline segments

Table 7.6: Virtex module characteristics

Module	Stages	Input	Slices	Speed
Add	1	16x16	10	173 MHz
Add	1	32x32	20	157 MHz
Subtract	1	16x16	11	141 MHz
Shift	1	16x16	10	180 MHz
Multiply	1	8x8	39	65 MHz
Multiply	4	8x8	48	131 MHz
Multiply	5	12x12	107	117 MHz
Multiply	5	16x16	168	115 MHz

Table 7.7: Reconfiguration costs in number of Virtex slices

	Greedy	Our Approach	Speedup
N-body	624	228	2.74
FFT	702	110	6.38

For the N-body simulation and FFT examples, the number of slices to be reconfigured for each schedule is shown in Table 7.7. This is the reconfiguration cost R_{loop} . The heuristic based algorithms have a significant saving in the reconfiguration cost. This translates to a direct reduction in the total execution time of the configuration. In the worst case, our heuristic algorithms generate a schedule which is at least as good as the greedy heuristic.

The total execution cost was computed for both the applications for a data set size of 4096 data points with the an on-chip memory size of 2KB (M). For the two example applications, reconfiguration cost is the dominant cost in the execution of the application and constitutes more than 95% of the total execution time. The application

speedups are of the same order as the speedups in the reconfiguration costs illustrated in Table 7.7. This shows that our heuristic based approach performs significantly better than the greedy heuristic.

Chapter 8

DRIVE Simulation Framework

The word simulation comes from simia, an ape.

– Dictionary

The general purpose computing area is the most promising to achieve significant performance improvement for a wide spectrum of applications using reconfigurable hardware. But, research in this area is hindered by the absence of appropriate techniques and tools. Current design tools are based on ASIC CAD software and have multiple layers of design abstractions which hinder high level optimizations based on reconfigurable system characteristics. It is also difficult to incorporate dynamic reconfiguration into the current CAD tools framework. The performance of the hardware is limited by the software tools which interact with the hardware. Hence, current CAD software is one of the main bottlenecks in the acceptance of reconfigurable hardware as a general purpose computing platform.

The absence of mature design tools also impacts the simulation environments that exist for studying reconfigurable systems and the benefits that they offer. Simulation tools are a very important component of the design cycle. Simulations provide users with practical feedback when developing applications and designing systems. This allows the designer to determine the correctness and performance of a design before the system is actually constructed. The user can explore the merits of alternative designs without actually building the systems. Simulation tools provide a means to explore the architecture and the design space in real time at a very low resource and time cost.

Current simulation tools for reconfigurable architectures are based on existing CAD design flow and perform mapping of designs to low level hardware for simulation. Furthermore, there are very few tools which provide any ability to study the dynamic behavior of reconfigurable hardware. Most of the existing simulation environments are based on simulation of High-level Description Language(HDL) or schematic designs that implement an application. Existing frameworks are either based on simulation of HDL based designs [7, 48, 51] or they are tightly coupled to specific architectures [17, 44, 52]. System level tools which analyze the interactions between various components of the system such as memory and configurable logic are limited and are mostly tightly coupled to specific system architectures. These tools do not permit evaluation of dynamic architectures and designs. Even for static designs, the user needs to be both a software

designer and a hardware designer to perform the simulations using current tools. Bridging this semantic gap between the user and the hardware needs expertise in multiple domains. This hinders the acceptance of reconfigurable hardware into the general purpose computing area.

In this chapter we present a novel interpretive simulation and visualization environment based on modeling and module level mapping approach. The **Dynamically Reconfigurable systems Interpretive simulation and Visualization Environment (DRIVE)** can be utilized as a vehicle to study the system and application design space and performance analysis. Reconfigurable hardware is characterized by using a high level parameterized model. Applications are analyzed to develop an abstract application task model. *Interpretive* simulation measures the performance of the abstract application tasks on the parameterized abstract system model. This is in contrast to simulating the exact behavior of the hardware by using HDL models of the hardware devices.

The **DRIVE** framework can be used to perform interactive analysis of the architecture and design parameter space. Performance characteristics such as total execution time, data access bandwidth characteristics and resource utilization can be studied using the **DRIVE** framework. The simulation effort and time are reduced and systems and designs can be explored without time consuming low level implementations. Our approach reduces the semantic gap between the application and the hardware and facilitates the performance analysis of reconfigurable hardware. Our approach also captures the simulation and visualization of dynamically reconfigurable architectures. We

have developed the Hybrid System Architecture Model(HySAM) of reconfigurable architectures. This model is currently utilized by the framework to map applications to a system model. We believe such an approach can be utilized to study reconfigurable architectures and application performance and facilitate adoption of such architectures by a larger spectrum of users.

8.1 Motivation

The software design tools for configurable computing are not keeping pace with the hardware device technology. Increasing device capacity, complexity and features are not translated into increased performance and productivity. This is because exploiting the reconfiguration technology for applications entails expertise in multiple domains. The user needs to understand low level device intricacies in addition to the application characteristics. This limits the application of such expertise to a few highly specific architectures and applications. Generalization of such expertise to other architecture domains is currently infeasible. The space of such architecture and application variations is too large for expansion of the expertise. The expertise that is developed is also restricted to the specific user and tools do not exist to translate the expertise to a form which can be utilized by others.

The expert knowledge acquired can be in terms of design methodologies, reusable components, algorithmic techniques etc. Tools which permit transfer of this expert knowledge do not exist. Though tools exist to permit utilization of libraries of optimized, pre-developed components, they are usually very tightly coupled to a single architecture. There are very few efforts which abstract the various available modules to provide a system transparent view and permit cross-platform utilization. There are currently some efforts in developing such abstractions and interface tools(e.g. FLAME [43]). But, there is no framework which exploits such abstract libraries.

Expertise in different domains can be encapsulated by using design abstractions such as system abstractions, module characterization and application analysis. This encapsulated knowledge can be utilized to provide users a modular framework. A developer, architect or a user has to understand only his domain and the appropriate abstractions in the framework. Utilization of high level abstraction reduces the extensive development effort in porting applications to different architectures. The performance of the modules can be characterized by utilizing the information provided by the vendor or the module generator. For low level modules the performance can be obtained by initial simulations and implementations. Utilizing this information, characteristics of high level modules can be simulated or computed without the intervention of CAD tools.

Algorithmic techniques can be utilized to map the abstract specifications to actual designs. A software tool which synthesizes an application specification into an optimal design on a heterogeneous reconfigurable architecture does not exist. Current system

complexities and application diversity does not seem to promise any such tool in the near future. Tools which permit performance estimation and analysis and design space exploration are a more tangible goal. Important characteristics of such a framework would be parametric variation, modularity, extensibility, efficiency, and ease of use. Interpretive simulation of the application model on the system model can provide the required performance analysis.

8.2 DRIVE Overview

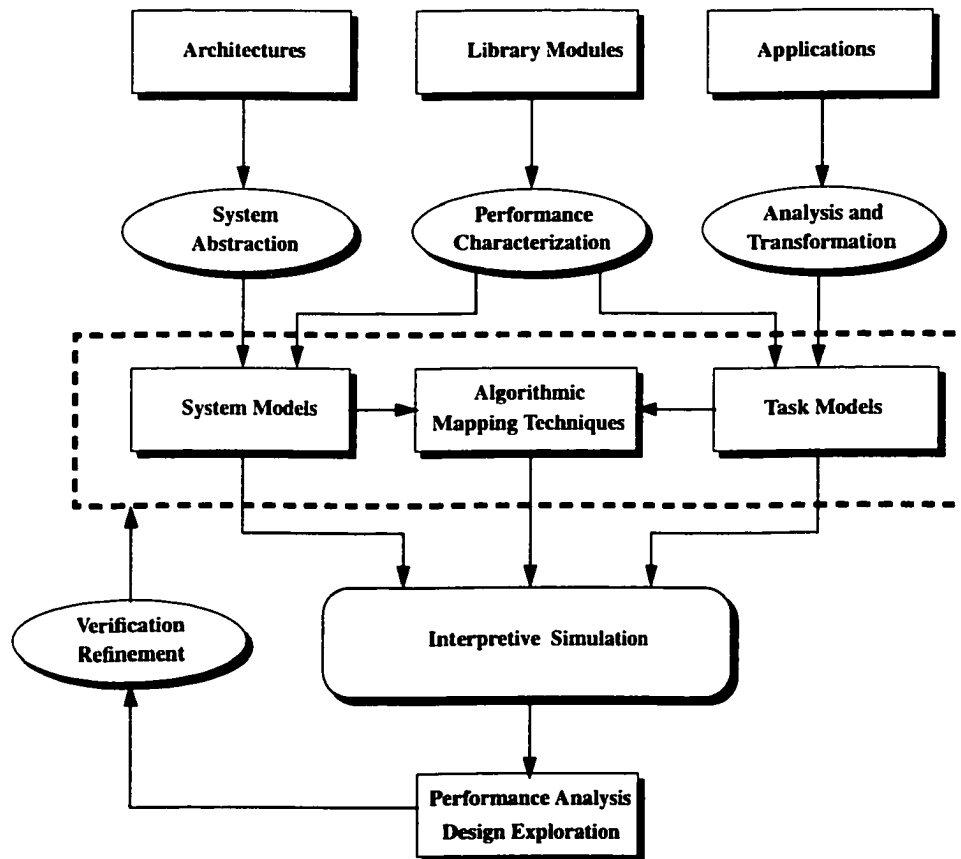


Figure 8.1: DRIVE framework

Figure 8.1 shows an overview of our framework. The system architecture can be characterized to capture the parameter space which affects the performance. The implementations of various optimized modules can be encapsulated by characterizing the performance of the module with respect to the architecture. This characterization is partitioned into the *capabilities* of the system and the actual *implementations* of these *capabilities*. The application is not mapped onto a low level design but is analyzed to develop an application task model. The application model can exploit the knowledge available in the form of the system *capabilities* provided by the module characterization. Algorithmic techniques are utilized to map the application task model to the system models, to perform interpretive simulation and obtain performance results for a given set of parameter values.

Interpretive simulation is performed on the system model which permits a higher level abstract simulation. The application does not need to be actually executed by using device level simulators like HDL models of the architectures. The performance measures can be obtained in terms of the application and model parameters and system characteristics. An interpretive simulation framework will permit design exploration in terms of the architectural choices, application algorithm options, various mapping techniques and possible problem decomposition onto the system components. Development of all the full blown designs which exercise these options is a non-realizable engineering task. Simulation, estimation and visualization tools can be designed to automate this exploration and obtain tangible results in reasonable time.

The abstractions and the techniques that are developed are enclosed in the dashed box in Figure 8.1. Verification of the models, mapping techniques and simulation framework can be performed by mapping some designs onto actual architectures. This verification process can be utilized to expand on the abstraction knowledge and refine the various models and techniques that are developed. The verification and refinement process completes the feedback loop of the design cycle to result in final accurate models and efficient techniques for optimal designs.

8.3 Other Simulation Tools

Several simulation tools have been developed for reprogrammable FPGAs. Most tools are device based simulators and are not system level simulators. Some of the efforts in this area are briefly described here. The most significant effort in this area has been the Dynamic Circuit Switching(DCS) based simulation tools by Lysaght et.al. [51]. These tools study the dynamically reconfigurable behavior of FPGAs and are integrated into the CAD framework. Though the simulation tools can analyze the dynamic circuit behavior of FPGAs, the tools are still low level. The simulation is based on CAD tools and requires the input design of the application to be specified in VHDL. The parameters for the design are obtained only after processing by the device specific tools. Luk et.al. describe a visualization tool for reconfigurable libraries [48]. They developed tools to simulate behavior and illustrate design structure. Their emphasis is on visualization of library modules and not system level simulation or application performance analysis.

Hartenstein et. al. present a hardware/software co-design framework CoDe-X which partitions an application into software executing on a host and hardware configurations onto reconfigurable ALU [44]. But the system does not support a simulation framework which can be utilized in the absence of the hardware and permit performance analysis and architectural alternatives study. CHASTE [17] was a toolkit designed to experiment with the XC6200 at a low level. The toolkit allows circuit specification and performs timing analysis and simulation. But, the target of the CHASTE system is low level design exploration and not system level analysis. There are other software environments such as JHDL [7], HOTWorks [27], Riley-2 [52], etc. But, they are software systems for low level hardware design and evaluation and are not system level interpretive simulation frameworks.

8.4 DRIVE Framework Implementation

An overview of the major components in the **DRIVE** framework and their interactions is given in Figure 8.2. The framework utilizes high level models of reconfigurable hardware. The current prototype uses the HySAM model described in Chapter 5.

The main input requirements to the **DRIVE** framework are the model parameters and the application tasks. The model parameters supply information about the Functions, Configurations, Attributes and the Reconfiguration costs. The user can visualize and update any of the instantiated parameters to explore the design space. For a given model parameters, performance results can be obtained for any set of application tasks

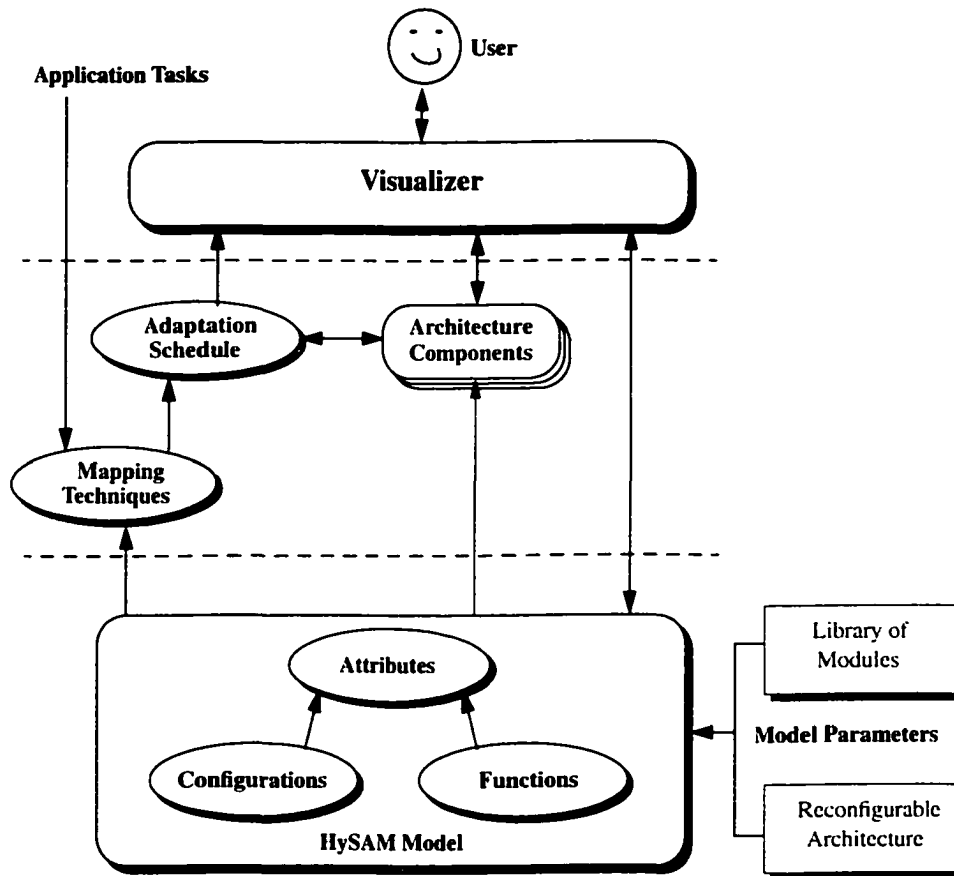


Figure 8.2: Major components in the **DRIVE** framework and the information flow

with various algorithmic mapping techniques.

The high level model partitions the description of the hardware into two components: the Functions (*capabilities*) of the hardware and the Configurations (*implementations*). For example, ability of the hardware to perform multiplication is a capability. The implementations are the different multiplier designs available with varying characteristics such as area, time, precision, structure, etc. The user only needs to have a knowledge of the *capabilities*. This input application is partitioned into tasks which are to be executed on the CPU and the Configurable Logic Unit. The applications tasks to

be executed are then decomposed into a sequence of CLU functions(F). Execution of a function on the CPU is represented as execution in a special configuration C_{cpu} .

The application task model consists of specification of the application in terms of the Functions(*capabilities*). The input to the framework consists of a directed acyclic graph of the application tasks specified with the Functions as the nodes of the graph. The edges denote the dependencies between the tasks. This technique reduces the effort and expertise needed on the part of the user. The application need not be implemented as an HDL design by the user to study the performance on various reconfigurable architectures. Automatic compilation efforts [9, 66] can be leveraged to generate the tasks from high level application programs.

Algorithmic mapping techniques are then utilized to map the application specification to actual implementations. These techniques map the *capabilities* to the *implementations* and generate a sequence of configuration, execution, and reconfiguration steps. This is the *adaptation schedule* which specifies how the hardware is adapted during the execution of the application. The schedule contains a sequence of configurations($C_1 \dots C_q$) where each configuration $C_i \in C$. This *adaptation schedule* can be computed statically for some applications by using algorithmic techniques. Also, the simulation framework can interact with the model and the mapping algorithms to determine the *adaptation schedule* at run-time.

The interpretive simulation framework is based on module level parameterization of the hardware. The framework is independent of a specific hardware platform. The details of the specific hardware are supplied by instantiating the parameters of the model. This information can be obtained from the architecture design and the library components(module generators) for that architecture. These library components or modules form the *implementations* in the model and can be determined for different architectures. Vendors and researchers have developed parameterized libraries and modules optimized for a specific architectures. The proposed framework can exploit the various efforts in design of efficient and portable modules [24, 49, 55]. The framework can incorporate such knowledge as the parameters for the HySAM model.

The user can analyze the performance of the architecture for a given application by supplying the parameters of the model and the application task. Typically the architectural parameters for the model are supplied by the architecture designer and the library designer. But, the user can modify the model parameters and explore the architecture design space. This provides the ability to study design alternatives without the need for actual hardware. The simulation and the performance analysis are presented to the user through a Graphical User Interface. The framework supports incorporation of additional information in the configurations(C) which can be utilized for actual execution or simulation. Using this information, it is possible to link the abstract definitions to actual implementations to verify and refine the abstract models.

The interactions between the components that are shown in Figure 8.2 depict dynamic information flow. The mapping algorithm can make run-time decisions to compute the *adaptation schedule* dynamically. The *adaptation schedule* is utilized by the visualizer to display the results. The visualizer does not possess any knowledge of the current schedule step, system state or model parameters. The appropriate components of the system are queried to obtain the required data and display it at run-time. Note that the framework is based on algorithmic techniques for determining the sequence of operations. Since the simulation is module based, the schedule is based on events. The schedule consists of events such as beginning or completion of execution or reconfiguration. Various modes where the events can either be overlapped or non-overlapped are also supported.

The parameters and attributes of the model can also be evaluated and adapted at run-time to compute the required information for scheduling and visualization. For example, reconfiguration costs can be determined by computing the difference in the configuration information and configurations can even be generated dynamically by future integration of tools like JBits [47]. It is assumed currently that the attributes for configurations are available a priori. It is easy to integrate simulation tools which evaluate the attributes such as execution time by performing simulations as in various module generators [7, 24, 55]. These simulations are based on module generators which do not require mapping using time consuming CAD tools. Once the attribute information

for low level modules are obtained by initial simulations and implementations, the attributes for higher level modules can be simulated or computed without the intervention of CAD tools.

The **DRIVE** framework has been designed using object-oriented methodology to support the modification and addition to the existing components. The framework facilitates the addition of new architectural models, algorithmic mapping techniques, performance analysis tools, etc. in a seamless manner. The framework can also be interfaced to existing tools such as parameterized libraries (Xilinx XBLOX, Luk et. al. [49]), module generators (PAM-Blox [55], Berkeley Object Oriented Modules [24], JHDL [7]), configuration generators (JBits [47]), module interfaces (FLexible API for Module-based Environments [43]), etc. The components of the framework will be made available to the community to facilitate application mapping and modular extensions.

8.5 Visualization

The visualizer for the framework has been developed using the *Java* language AWT toolkit. A previous version of the visualizer was developed using Tcl/Tk. The C programming language was utilized for implementing the simulation engine. The current prototype has been developed in *Java* to utilize the object oriented framework and make the framework modular and easily extensible. Implementing the visualizer and the interpretive simulation in the same language provides for a clearer interface between the

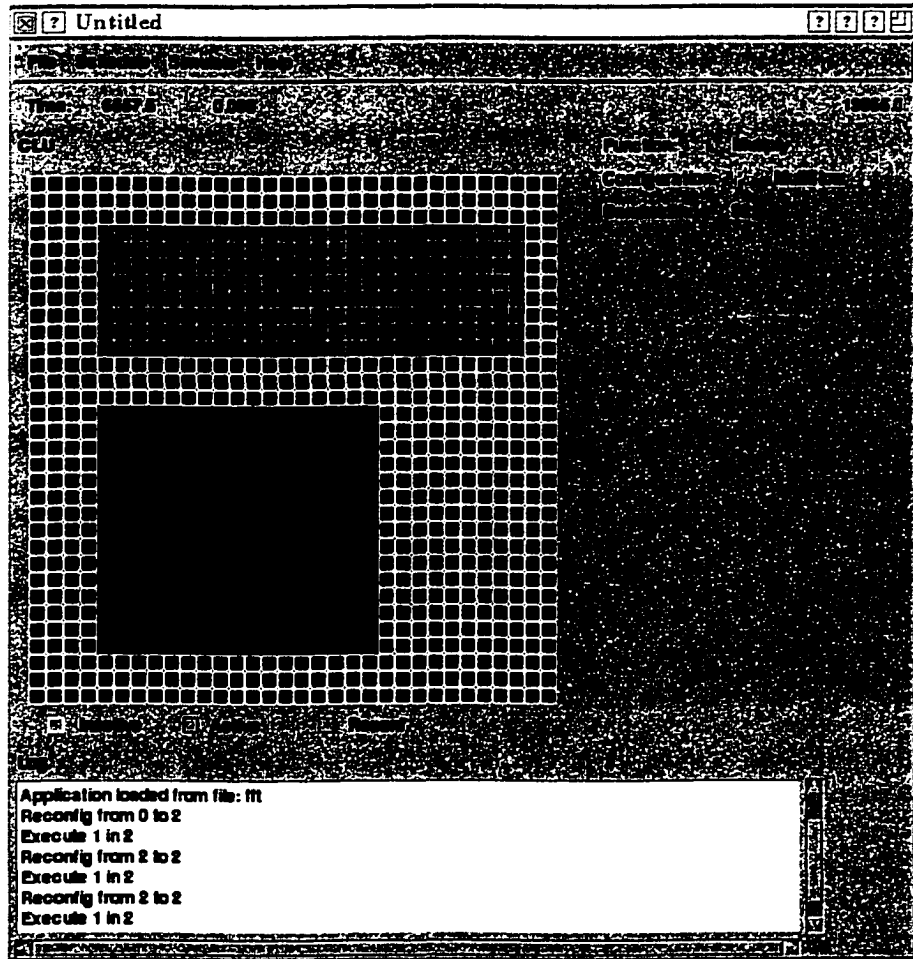


Figure 8.3: Sample DRIVE visualization

components. *Java* is becoming the language of choice for several research and implementation efforts in hardware design and development [7, 24, 47]. Incorporating the results and abstractions from other research efforts is simplified using the current version.

The visualizer acts as a graphical user interface to support the full functionality of the framework. It is implemented as a separate *Java* class communicating with the remaining classes. Any component of the simulation or visualizer framework can be completely replaced with a different component supporting the same interface. The visualizer is oblivious of the algorithmic techniques and implementation details. It accesses information from the different components in the simulation framework on an event by event basis and displays the state of the various architecture components and the performance characteristics. A sample view of the visualizer is shown in Figure 8.3.

8.6 DRIVE Summary

Software tools are an important component of reconfigurable hardware development platforms. Simulation tools which permit performance analysis and design space exploration are needed. The utility of current tools for reconfigurable hardware design is limited by the required user expertise in multiple domains. We have proposed a novel *interpretive* simulation and visualization environment which supports system level analysis. The **DRIVE** framework supports a parameterized system architecture model. Algorithmic mapping techniques have been incorporated into the framework and can be

extended easily. The framework can be utilized for performance analysis, design space exploration and visualization. It is implemented in the *Java* language and supports flexible extensions and modifications.

Chapter 9

Conclusions and Future Directions

Two stone cutters were asked what they were doing.

The first said, "I am cutting this stone into blocks."

The second replied, "I am on a team that is building a cathedral."

– Old Story

Reconfigurable computing is emerging as the platform of choice to design future high performance systems. Reconfigurable architectures meet the performance and flexibility requirements of next generation applications. Future devices which provide dynamic reconfigurability of both combinational logic and interconnection network based on intermediate results promise enormous computational power.

To realize this potential we need tools which exploit these features in a non-trivial manner. Features which future devices need to provide also need to be explored. Logic

synthesis is the current approach to using configurable devices in which a HDL description is statically compiled onto hardware. Using such automated synthesis approach is not amenable to designing tools which analyze the run-time behavior of applications and utilize dynamic reconfiguration.

Collapsing the numerous levels of abstraction in the automated synthesis approach will provide a new paradigm for designing configurable computing solutions. We do this by using a computational model of configurable computing devices which facilitates the algorithm synthesis approach as opposed to the logic synthesis approach. We developed the Hybrid System Architecture Model (HySAM) to facilitate the algorithm synthesis approach. HySAM is a parameterized abstract model which captures a wide range of configurable systems. In our model-based approach user is exposed to the underlying device characteristics which will allow him to make use of the dynamic reconfiguration features. The computational model not only allows the user to implement algorithms in a more natural manner but also permits analysis of runtime behavior.

Automatic mapping and scheduling of applications is necessary for achieving performance improvement for general purpose computing applications on reconfigurable hardware. Mapping of applications in an architecture independent fashion can provide a framework for automatic compilation of applications. Loop structures with regular repetitive computations can be speeded-up by using configurable hardware. Loops form the main portion of the computational load in most applications. In this thesis, we have

developed automatic techniques to map loops from application programs onto configurable hardware. Our algorithms are based on a general Hybrid System Architecture Model(HySAM). We define important problems in mapping of traditional loop structures onto configurable hardware and demonstrate a polynomial time solution for one important variant of the problem.

Our algorithmic techniques address the overheads involved in reconfiguring the hardware. In current architectures the reconfiguration overheads are still significant compared to the execution cost. In this thesis, we have proposed algorithmic techniques for mapping and scheduling loops in applications onto reconfigurable hardware. The heuristics we have developed attempt to minimize the reconfiguration overheads by exploiting designs with partial and runtime reconfiguration. The mapping of example loops from applications illustrates that the proposed algorithms can generate high performance configurations with reduced reconfiguration cost.

Precision variation is one of the customizations that can be provided by configurable hardware for loop computations. In this thesis, we develop a framework for dynamic precision management for loop computations. We have shown how the variable precision in computations can be captured by using the *precision variation curve*. The thesis described our approach to computing the *precision variation curve* using theoretical analysis.

The loop mapping algorithm and the theoretical techniques we developed in the earlier part of the thesis are extended to the dynamic precision management problem. The

DPMA algorithm that we have developed can compute the required optimal schedule for a given operation in a loop using the *precision variation curve* and the set of variable precision configurations. Our Hybrid System Model(HySAM) of reconfigurable architectures facilitates the development of these algorithms using a high level abstract model. The thesis illustrated the performance benefits achievable for an example loop computation using our approach. We expect that the proposed approach can lead to significant improvement in performance and automatic mapping of variable precision computations on reconfigurable architectures.

In this thesis, we developed techniques to map computations onto high performance reconfigurable pipelines to exploit architectural features of reconfigurable architectures. Heuristic algorithms are developed to reduce the reconfiguration overheads in the presence of resource constraints. Applications are parallelized and pipelined by using *data context switching*. Data context switching treats each execution of a repetitive computation as a context. Instead of switching the configuration, the data on which the datapath is operating on is changed every cycle dynamically. This technique can parallelize loop computations that cannot be parallelized using existing techniques on conventional architectures.

Our optimization techniques simultaneously exploit multiple dimensions of reconfigurable architectures. The thesis addresses the issues in developing mapping techniques which exploit multiple aspects of the reconfigurable logic to deliver superior

performance compared to traditional techniques. Our techniques address various application and architectural characteristics and resource limitations in developing mapping techniques to optimize application performance.

Software tools are an important component of reconfigurable hardware development platforms. Simulation tools which permit performance analysis and design space exploration are needed. The utility of current tools for reconfigurable hardware design is limited by the required user expertise in multiple domains. We have proposed a novel *interpretive* simulation and visualization environment which supports system level analysis. The **DRIVE** framework supports a parameterized system architecture model. Algorithmic mapping techniques have been incorporated into the framework and can be extended easily. The framework can be utilized for performance analysis, design space exploration and visualization. It is implemented in the *Java* language and supports flexible extensions and modifications. A prototype version has been implemented and is currently available. The USC Models, Algorithms and Architectures project is developing algorithmic techniques for realizing scalable and portable applications using configurable computing devices and architectures.

The models, algorithmic techniques and simulation methodology proposed in this thesis can be extended to address similar issues in emerging architectures and applications. In the following section we describe briefly future challenges that need to be addressed in reconfigurable computing.

9.1 Future Directions

In this thesis we have presented a model-based framework to develop algorithmic techniques for mapping applications onto reconfigurable architectures. To our knowledge, this is one of the first efforts to address the development of a model-based mapping framework. There are several research areas that have to be addressed in the future to extend this research for emerging architectures and applications. We outline briefly some of these areas:

- *Enhancing the HySAM Model*: Emerging architectures are integrating several different architecture paradigms onto the same chip. Conventional microprocessor cores, DSP cores, reconfigurable logic, embedded memory and peripheral controllers are integrated onto the same chip. Developing a model of computation and communication that can be used to map applications and analyze performance is a challenge. In future applications areas, power dissipation is increasingly becoming as important as performance. Modeling the power dissipation in current and emerging architectures is a critical challenge. HySAM model can be extended to add power characteristics as part of the attribute set \mathcal{A} . Algorithmic techniques similar to those in this thesis can be developed to analyze and optimize the power dissipation.
- *Mapping loops*: Loops continue to remain the focus for mapping onto various architectures. Development of mapping techniques for loop computations can be

utilized to extend the domain of applications that can be mapped onto a given architecture. Further research is needed to address loop computations that have characteristics not satisfied by the assumptions in this thesis. These characteristics include task structure, data and control dependencies, nested loops, and run-time dependencies. Extensive research in the parallel computing on mapping loops can be exploited to address mapping issues. However, existing techniques need to be extended to address the reconfigurable characteristics of the hardware.

- *Dynamic precision variation:* The dynamic precision management framework gives rise to a wealth of issues which can potentially provide enormous benefits to mapping computations onto configurable hardware. Bit-serial and digit-serial computations are one class of computations which can exploit dynamic precision without large overheads. The control component of the design needs to execute the configurations for a variable number of steps based on the required precision. Run-time precision management where the control modifies the precision of the computations are being explored. Configurable logic can be utilized to execute multiple iterations of loops in parallel in the absence of dependencies. Reduction of the logic resources due to dynamic precision management can be exploited to execute more number of iterations in parallel. Multi-context devices and configuration caches can be utilized to reduce the reconfiguration overheads by storing variable precision configurations.

- *Integrated mapping techniques:* Hybrid architectures are experiencing a convergence with future FPGAs providing multiple architectural cores and other System-on-Chip architectures adding reconfigurable logic on-chip. There is a mixture of computational models and programming paradigms in the different components of such architectures. Hybrid architectures and other emerging architectures can facilitate innovative mapping techniques that can exploit multiple aspects of the architectures. Data context switching, developed in this thesis, is an example of such techniques.
- *Design tools:* Design tools are being developed after the evolution of emerging architectures and have not co-evolved with the architectures. Current design processes¹ are based on independent design flow for each architectural component. The programming models and the design tools for each of the individual components are utilized to map an application. The integration is performed at a much later stage. A standard interface between different components of the chip is the only integration that exists during the design phase. Design tools which facilitate a tighter integration in the programming and mapping phase need to be developed.
- *System level simulation and verification tools:* The general purpose computing area is the most promising to achieve significant performance improvement for a wide spectrum of applications using reconfigurable hardware. Current design

¹by design processes, methodologies and tools we refer to the aspects of developing applications *using* a chip and not designing the chip.

tools are based on ASIC CAD software and have multiple layers of design abstractions. Simulation tools provide a means to explore the architecture and the design space in real time at a very low resource and time cost. The absence of mature design tools impacts the simulation environments that exist for studying reconfigurable systems and the benefits that they offer. System level tools which analyze and simulate the interactions between various components of the system such as memory and reconfigurable logic are limited and are mostly tightly coupled to specific system architectures. Future research in reconfigurable computing needs to explore the development of such system level design, simulation and verification tools.

References

- [1] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, and G. Snider. Teramac - Configurable Custom Computing. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 32–38, April 1995.
- [2] S. I. Association. <http://www.semichips.org>.
- [3] P. Athanas and A. Abbott. Real-Time Image Processing on a Custom Computing Platform. *IEEE Computer*, pages 16–24, February 1995.
- [4] P. Athanas and H. F. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.
- [5] J. Babb, M. Frank, and A. Agarwal. Solving graph problems with dynamic computation structures. *SPIE Photonics East: Reconfigurable Technology for Rapid Product Development and Computing*, November 1996.
- [6] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, , and D. Zaretsky. A MATLAB Compiler for Distributed Heterogeneous Reconfigurable Computing Systems. *IEEE Symposium on FPGAs for Custom Computing Machines*, April 2000.
- [7] P. Bellows and B. Hutchings. JHDL - An HDL for Reconfigurable Systems. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.
- [8] R. Bittner and P. Athanas. Wormhole Run-time Reconfiguration. In *ACM International Symposium on Field Programmable Gate Arrays*, pages 79–85, February 1997.
- [9] K. Bondalapati, P. Diniz, P. Duncan, J. Granacki, M. Hall, R. Jain, and H. Ziegler. DEFACTO: A Design Environment for Adaptive Computing Technology. In *Reconfigurable Architectures Workshop, RAW'99*, April 1999.
- [10] K. Bondalapati and V. Prasanna. Reconfigurable Meshes: Theory and Practice. In *Reconfigurable Architectures Workshop, RAW'97*, April 1997.

- [11] K. Bondalapati and V. Prasanna. Mapping Loops onto Reconfigurable Architectures. In *8th International Workshop on Field-Programmable Logic and Applications*, September 1998.
- [12] K. Bondalapati and V. Prasanna. DRIVE: An Interpretive Simulation and Visualization Environment for Dynamically Reconfigurable Systems. In *International Workshop on Field-Programmable Logic and Applications*, September 1999.
- [13] K. Bondalapati and V. Prasanna. Dynamic Precision Management for Loop Computations on Reconfigurable Architectures. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1999.
- [14] K. Bondalapati and V. Prasanna. Hardware Object Selection for Mapping Loops onto Reconfigurable Architectures. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, June 1999.
- [15] K. Bondalapati and V. Prasanna. Loop Pipelining and Optimization for Reconfigurable Architectures. In *Reconfigurable Architectures Workshop (RAW '2000)*, May 2000.
- [16] K. Bondalapati and V. Prasanna. Reconfigurable Computing: Architectures, Models and Algorithms. *Current Science*, 78(7):828–837, April 2000.
- [17] G. Brebner. CHASTE: a Hardware/Software Co-design Testbed for the Xilinx XC6200. In *Reconfigurable Architectures Workshop, RAW'97*, April 1997.
- [18] S. Brown and J. Rose. FPGA and CPLD Architectures: A Tutorial. *IEEE Design & Test of Computers*, Summer 1996.
- [19] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder. *Splash 2: FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, 1996.
- [20] S. Cadambi, J. Weener, S. Goldstein, H. Schmit, and D. E. Thomas. Managing Pipeline-Reconfigurable FPGAs. In *Proceedings ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, February 1998.
- [21] T. J. Callahan and J. Wawrzynek. Instruction-Level Parallelism for Reconfigurable Computing. *International Workshop on Field Programmable Logic*, September 1998.
- [22] D. Chang and M. Marek-Sadowska. Partitioning Sequential Circuits on Dynamically Reconfigurable FPGAs. In *IEEE Transactions on Computers*, June 1999.
- [23] S. Choi and V. Prasanna. Configurable Hardware for Symbolic Search Operations. In *International Conference on Parallel and Distributed Systems*, Dec 1997.

- [24] M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek. Object Oriented Circuit-Generators in Java. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.
- [25] Y. Chung and V. Prasanna. Parallel Object Recognition on an FPGA-based Configurable Computing Platform. In *International Workshop on Computer Architectures for Machine Perception*, Oct 1997.
- [26] T. Corporation. <http://www.triscend.com/>.
- [27] V. C. Corporation. Reconfigurable Computing Products, <http://www.vcc.com/>.
- [28] A. Dandalis, A. Mei, and V. K. Prasanna. Domain Specific Mapping for Solving Graph Problems on Reconfigurable Devices. In *Reconfigurable Architectures Workshop*, April 1999.
- [29] A. Dandalis and V. Prasanna. Fast Parallel Implementation of DFT using Configurable Devices. In *7th International Workshop on Field-Programmable Logic and Applications*, Sept 1997.
- [30] A. Dandalis, V. Prasanna, and J. Rolim. An Adaptive Cryptographic Engine for IPsec Architectures. *IEEE Symposium on FPGAs for Custom Computing Machines (Submitted)*, April 2000.
- [31] A. DeHon. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1994.
- [32] A. DeHon. *Reconfigurable Architectures for General Purpose Computing*. PhD thesis, MIT AI Lab, September 1996.
- [33] M. Donlin. Programmable logic and synthesis strive to get in sync. *Computer Design*, August 1996.
- [34] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - Reconfigurable Pipelined Datapath. In *6th International Workshop on Field-Programmable Logic and Applications*, 1996.
- [35] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
- [36] M. Gokhale and J. Stone. Automatic Allocation of Arrays to Memories in FPGA Processors with Multiple Memory Banks. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1999.

- [37] M. B. Gokhale and A. Marks. Automatic Synthesis of Parallel Programs Targeted to Dynamically Reconfigurable Logic Arrays. In *Proceedings of the 1995 International Workshop on Field-Programmable Logic and Applications, Oxford, England, September 1995*.
- [38] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, , and R. Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *IEEE Computer*, April 2000.
- [39] P. Graham and B. Nelson. Genetic Algorithms In Hardware and In Software - A Performance Analysis of Workstation and Custom Computing Machine Implementations. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1996.
- [40] S. Hauck. The Roles of FPGAs in Programmable Systems. *Proceedings of the IEEE*, 86, April 1998.
- [41] J. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, April 1997.
- [42] R. T. J. Babb and A. Agarwal. Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators. *IEEE Workshop on FPGAs for Custom Computing Machines*, April 1993.
- [43] A. Koch. Unified access to heterogeneous module generators. In *ACM International Symposium on Field Programmable Gate Arrays*, February 1999.
- [44] R. Kress, R. Hartenstein, and U. Nageldinger. An Operating System for Custom Computing Machines based on the Xputer Paradigm. In *7th International Workshop on Field-Programmable Logic and Applications*, pages 304–313, Sept 1997.
- [45] A. Lawrence, A. Kay, W. Luk, T. Nomura, and I. Page. Using reconfigurable hardware to speed up product development and performance. In *5th International Workshop on Field-Programmable Logic and Applications*, 1995.
- [46] E. Lemoine and D. Merceron. Run Time Reconfiguration of FPGA for Scanning Genomic Databases. In *IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.
- [47] D. Levi and S. Guccione. Run-Time Parameterizable Cores. In *ACM International Symposium on Field Programmable Gate Arrays*, February 1999.
- [48] W. Luk and S. Guo. Visualising reconfigurable libraries for FPGAs. In *Asilomar Conference on Signals, Systems, and Computers*, 1998.

- [49] W. Luk, S. Guo, N. Shirazi, and N. Zhuang. A Framework for Developing Parametrised FPGA Libraries. In *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, 1996.
- [50] W. Luk, N. Shirazi, S. Guo, and P. Cheung. Pipeline Morphing and Virtual Pipelines. In *7th International Workshop on Field-Programmable Logic and Applications*, Sept 1997.
- [51] P. Lysaght and J. Stockwood. A Simulation Tool for Dynamically Reconfigurable FPGAs. *IEEE Transactions on VLSI Systems*, Sept 1996.
- [52] P. Mackinlay, P. Cheung, W. Luk, and R. Sandiford. Riley-2: A Flexible Platform for Codesign and Dynamic Reconfigurable Computing Research. In *7th International Workshop on Field-Programmable Logic and Applications*, September 1997.
- [53] T. Maruyama and T. Hoshino. A C to HDL Compiler for Pipeline Processing on FPGAs. *IEEE Symposium on FPGAs for Custom Computing Machines*, April 2000.
- [54] P. Master and K. Lane. Powering up 3G Handsets for MPEG-4 Video. *Communication Systems Design*, January 2001.
- [55] O. Mencer, M. Morf, and M. Flynn. PAM-Blox: High Performance FPGA Design for Adaptive Computing. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.
- [56] A. Microsystems. <http://www.annapmicro.com/>.
- [57] X. A. Notes. A Fast Constant Coefficient Multiplier for the XC6200.
- [58] X. D. A. Notes. The Fastest FFT in the West, <http://www.xilinx.com/apps/displit.htm>.
- [59] R. Payne. Run-time Parameterised Circuits for the Xilinx XC6200. In *7th International Workshop on Field-Programmable Logic and Applications*, pages 161–172, Sept 1997.
- [60] R. Petersen and B. Hutchings. An Assessment of the Suitability of FPGA-Based Systems for use in Digital Signal Processing. In *5th International Workshop on Field-Programmable Logic and Applications*, 1995.
- [61] A. N. S. C. E. Processor. <http://www.altera.com/html/products/nios.html>.
- [62] K. M. G. Purna and D. Bhatia. Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers. In *IEEE Transactions on Computers*, June 1999.

- [63] A. Rashid, J. Leonard, and W. H. Mangione-Smith. Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability. *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.
- [64] R. Razdan. *PRISC: Programmable Reduced Instruction Set Computers*. PhD thesis, Harvard University, May 1994. <ftp.eecs.harvard.edu:users/smith/theses/razdan-thesis.tar.gz>.
- [65] J. Rose, A. E. Gamal, and A. Sangiovanni-Vincentelli. Architecture of Field Programmable Gate Arrays. *Proceedings of the IEEE*, July 1993.
- [66] C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. Arnold, and M. Gokhale. The NAPA Adaptive Processing Architecture. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.
- [67] S. Scalera and J. Vázquez. The design and implementation of a context switching FPGA. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.
- [68] N. Semiconductor. Configurable Logic Array (CLAy) Data Sheet, Dec 1993.
- [69] N. Shirazi, P. Athanas, and A. Abbott. Implementation of a 2-D Fast Fourier Transform on an FPGA-Based Custom Computing Machine. In *International Workshop on Field-Programmable Logic and Applications*, 1995.
- [70] R. P. Sidhu, A. Mei, and V. K. Prasanna. Genetic Programming using Self-Reconfigurable FPGAs. In *International Workshop on Field Programmable Logic and Applications*, September 1999.
- [71] R. Subramanian, N. Ramasubramanian, and S. Pande. Automatic Analysis of Loops to Exploit Operator Parallelism on Reconfigurable Systems. In *Languages and Compilers for Parallel Computing*, August 1998.
- [72] C. Systems. <http://www.chameleonsystems.com/>.
- [73] A. Tenca and M. Ercegovic. A Variable Long-Precision Arithmetic Unit Design for Reconfigurable Coprocessor Architectures. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.
- [74] S. Trimmerger, D. Carberry, A. Johnson, and J. Wong. A Time-Multiplexed FPGA. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 22–28, April 1997.
- [75] B. P. URL. <http://HTTP.CS.Berkeley.EDU/Research/Projects/brass/>.

- [76] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, March 1996.
- [77] 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*, pages 86–93, September 1997.
- [78] M. Weinhardt. Compilation and Pipeline Synthesis for Reconfigurable Architectures. In *Reconfigurable Architectures Workshop(RAW' 97)*. ITpress Verlag, April 1997.
- [79] M. Weinhardt and W. Luk. Pipeline Vectorization for Reconfigurable Systems. In *IEEE Symposium on Field-Programmable Custom Computing Machines(FCCM '99)*, April 1999.
- [80] M. J. Wirthlin and B. L. Hutchings. Improving Functional Density Through Run-Time Constant Propagation. In *ACM International Symposium on Field Programmable Gate Arrays*, pages 86–92, February 1997.
- [81] M. E. Wolf and M. S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.
- [82] Xilinx. XC6200 Field Programmable Gate Arrays, 1996.
- [83] Xilinx Inc.(www.xilinx.com). *Virtex Series FPGAs*.
- [84] Xilinx Inc.(www.xilinx.com). *Xilinx Platform FPGAs*.
- [85] P. Zhong, M. Martonosi, P. Ashar, and S. Malik. Solving Boolean Satisfiability with Dynamic Hardware Configurations. *International Workshop on Field Programmable Logic*, September 1998.