Configurable Computing for Mainstream Software Applications

by

William D. Bishop

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2003

Your file Votre référence
ISBN: 0-612-82973-1
Our file Notre référence
ISBN: 0-612-82973-1

Abstract

A configurable computer is a computing device that may be modified at run-time to provide application-specific computer hardware to support the computation of a task. A coupled configurable computer uses a configurable coprocessor to provide specialized hardware to perform a specific task while the remainder of the computer system works on other tasks. The fact that the hardware functionality of a configurable coprocessor may be specialized at run-time allows this coprocessor to effectively behave like a larger system of coprocessors. Previous research has shown that configurable coprocessors can significantly enhance the performance of an application and/or computer system on niche applications.

This thesis investigates the quantitative and qualitative impacts of configurable coprocessors on the performance of a computer system. A transaction pair model of a configurable coprocessor operation is introduced. This model enables the development of a complete performance model that predicts the behaviour of a coupled configurable computer system. To quantify the parameters used by this model, a series of experiments were conducted. These experiments demonstrate that it is possible to transform an existing computer system into a coupled configurable computer system without the knowledge of the end user. Furthermore, the performance model can be used to predict when it is likely to be advantageous to use a configurable coprocessor to enhance an application and a computer system. The experiments show that the major factors impacting performance of an application are processing time, memory utilization delays, bus utilization delays, and operating system behaviour delays. These findings suggest that a tightly-coupled configurable computer architecture is better suited to mainstream software applications than a loosely-coupled configurable computer architecture.

Acknowledgements

I wish to thank my supervisor, Dr. Wayne Loucks, for his patience, his guidance, and his insight. This research would not have been possible without his support and the support of his family. I know Wayne had many sleepless nights throughout the duration of my thesis research. I only hope that I did not contribute significantly to his insomnia.

I wish to thank all of the funding agencies and corporations that supported my research through financial support and gifts-in-kind. In particular, I wish to thank the Natural Sciences and Engineering Research Council (NSERC) of Canada, the Ontario Graduate Scholarship (OGS) Program of the Province of Ontario, and Communications and Information Technology Ontario (CITO), formerly the Information Technology Research Centre (ITRC). I also wish to thank Altera Corporation, Mentor Graphics, Mesquite Software, and Nortel Networks for donating and/or subsidizing hardware and software for the purpose of my research.

Of course, my friends deserve thanks as well. Over the past seven years, I have made many new friendships that will last a lifetime. I particularly wish to thank all of the past and present members of the Parallel and Distributed Systems Research Group at the University of Waterloo.

Most of all, I wish to thank my parents for giving me the support and encouragement that I needed to successfully complete this degree. This thesis would not have been possible without them.

Trademarks

[1] SPARCstation is a trademark of SPARC International, Inc. and is licensed exclusively to Sun Microsystems, Inc.. Sun and SunOS are registered trademarks of Sun Microsystems, Inc..

[2] Altera, MAX, MAX+PLUS, Quartus, FLEX, and SignalTap are registered trademarks of Altera Corporation. AHDL, MAX+PLUS II, Quartus II, FLEX 10K, EPF10K50, APEX, Mercury, Stratix, Cyclone, FastTrack, MegaLAB, Avalon, Nios, RIPP-10, ARC-PCI, and Excalibur are trademarks of Altera Corporation.

[3] Xilinx, XACT, XC4005, and XC3090 are registered trademarks of Xilinx. XC95108, XC6264, XC6216, XC4025, XC4013, XC4010, XC4003, XC4002, XC3042, XC3030, XC3020, X-BLOX, Configurable Logic Cell, LCA, and Logic Cell are trademarks of Xilinx.

[4] Synopsys and Synopsys VHDL Compiler are registered trademarks of Synopsys, Inc.. DC Expert, DC Professional, Design Analyzer, Design Compiler, FPGA Compiler, Library Compiler, VHDL Compiler, Synopsys Graphical Environment, VHDL System Simulator, VSS Expert, and VSS Professional are trademarks of Synopsys, Inc..

[5] Hardware Object Technology, H.O.T., H.O.T. Works, H.O.T. I, H.O.T. II, and H.O.T. III are trademarks of Virtual Computer Corporation.

[6] Configurable Array Logic, CAL and CAL1024 are trademarks of Algotronix.

[7] PAL and PALASM are registered trademark of Advance Micro Devices, Inc..

[8] ABEL is a trademark of Data I/O Corporation.

[9] Tri-state is a registered trademark of National Semiconductor Corporation.

[10] Verilog is a registered trademark of Cadence Design Systems, Inc..

[11] X Windows System is a trademark of the Massachusetts Institute of Technology.

[12] UNIX is a trademark of AT&T Technologies, Inc..

[13] IBM and AT are registered trademarks of International Business Machines Corporation and PC/XT and PC/AT are trademarks of International Business Machines Corporation.

[14] Intel, Pentium, and i486 are registered trademarks of Intel Corporation. VTune is a trademark of Intel Corporation.

[15] Microsoft, MS, and MS-DOS are registered trademarks of Microsoft Corporation and MS-DOS, MS-Windows, Windows 95, Windows 98, Windows NT, Windows 2000, Windows XP and Win32s are trademarks of Microsoft Corporation.

[16] HP is a registered trademark of the Hewlett Packard Company.

Contents

1 Introduction 1

1.1 Motivation . . . 1

1.2 Configurable Computing . . . 3

1.3 Statement of Thesis . . . 5

1.4 Thesis Contributions . . . 6

1.5 Outline of the Thesis . . . 7

2 Introduction to Configurable Computing 8

2.1 Programmable Logic Devices . . . 8

2.1.1 Overview . . . 9

2.1.2 Programmable Logic Technologies . . . 9

2.1.3 Statically, Dynamically, and Partially Programmable Logic Devices . . . 11

2.1.4 SPLDs (Simple Programmable Logic Devices) . . . 12

2.1.5 PIDs (Programmable Interconnect Devices) . . . 15

2.1.6 HDPLDs (High-Density Programmable Logic Devices) . . . 16

2.1.7 RPUs (Reconfigurable Processing Units) . . . 22

2.2 Introduction to Modern Computer Architecture . . . 24

2.2.1 von Neumann Computer Architecture . . . 24

2.2.2 Personal Computer Architecture . . . 25

2.3 Configurable Computer Architectures . . . 29

2.3.1 Types of Configurable Computers . . . 31

2.4 Configurable Computing Platforms . . . 33

2.4.1 State-of-the-Art Configurable Computing Machines . . . 33

2.5 Benefits of Configurable Computing 35
2.6 Challenges of Configurable Computing 36

3 Models of Configurable Computing **38**
3.1 Introduction to Configurable Computer Models 38
3.2 The Transaction Pair Model 39
3.2.1 Comparison of Transaction Performance 43
3.2.2 Summary of the Transaction Pair Model 46
3.3 The Configurable Computer System Performance Model 46
3.3.1 Configuration Delays 51
3.3.2 Memory Utilization Delays 52
3.3.3 Bus Utilization Delays 53
3.3.4 Operating System Behaviour Delays 54
3.3.5 Processing Times 55
3.3.6 Evaluating the Net Performance Impact 56
3.4 Performance Model Scenarios 58
3.4.1 General Comments 60

4 Configurable Computing Platforms **62**
4.1 Platform I: PC + ARC-PCI Board 62
4.1.1 The ARC-PCI Board 63
4.1.2 ARC-PCI Development Kit 67
4.1.3 Configurable Computer Architecture 69
4.1.4 Application Programming Interface 69
4.1.5 Device Driver 71
4.1.6 Controller Design 75
4.1.7 User Designs 80
4.1.8 Comments on PCI Compliance 84
4.1.9 Comments on Performance 84
4.2 Platform II: Nios Embedded Processor Development Board 88
4.2.1 The Nios Embedded Processor Development Board 89
4.2.2 Nios Embedded Processor Development Kit 89
4.2.3 Configurable Computer Architecture 92

4.2.4 Nios Embedded Processor . . . 92
4.2.5 User Peripheral Designs . . . 94
4.2.6 Other Peripherals . . . 94
4.3 Platform III: Sun Workstation . . . 94
4.4 Platform Comparison . . . 95

5 Application 1: CSIM **97**
5.1 Introduction to Discrete-Event Simulation . . . 97
5.1.1 Discrete-Event Simulation Terminology . . . 98
5.1.2 Discrete-Event Simulation Tools and Libraries . . . 99
5.1.3 Accelerating Discrete-Event Simulation . . . 99
5.2 The CSIM Discrete-Event Simulation Library . . . 100
5.2.1 The Choice of CSIM . . . 100
5.2.2 Modeling Systems with CSIM . . . 101
5.2.3 Applications of CSIM . . . 102
5.2.4 Profiling the Performance of CSIM . . . 102
5.3 Enhancing CSIM . . . 104
5.3.1 Pseudo-Random Number Generation in CSIM . . . 104
5.3.2 Interfacing with Platform I . . . 105
5.3.3 Performance Optimizations . . . 105
5.4 Experimental Method . . . 107
5.5 Platform I: Experimental Results . . . 107
5.5.1 Application Speedup . . . 108
5.5.2 Evaluation of System Impact . . . 109
5.6 Interesting Observations . . . 110
5.6.1 Performance . . . 110
5.6.2 Impact of Optimizations . . . 111
5.6.3 Transparency . . . 111

6 Application 2: Pseudo-Random Number Generation **112**
6.1 Pseudo-Random Number Generation . . . 112
6.1.1 Linear Congruential Generators . . . 113
6.1.2 The Choice of Pseudo-Random Number Generation . . . 114

6.2 Enhancing Pseudo-Random Number Generation . . . 114
6.2.1 Interfacing with Platform I . . . 115
6.2.2 Interfacing with Platform II . . . 117
6.2.3 Performance Optimizations . . . 117
6.2.4 Platform I Performance Optimizations . . . 118
6.2.5 Platform II Performance Optimizations . . . 118
6.3 Experimental Method . . . 118
6.4 Platform I: Experimental Results . . . 118
6.4.1 Unbuffered Test Results . . . 119
6.4.2 Buffered Test Results . . . 119
6.4.3 Unbuffered Test Results on an Uncached System . . . 120
6.4.4 Buffered Test Results on an Uncached System . . . 120
6.4.5 Measuring the Impact of Caching . . . 122
6.5 Platform II: Experimental Results . . . 122
6.6 Interesting Observations . . . 123
6.6.1 Bus Utilization Delays . . . 123
6.6.2 Memory Utilization Delays . . . 124
6.6.3 Relative Performance of Processors . . . 124

7 Application 3: Minheap Management **125**
7.1 Introduction to Minheap Management . . . 125
7.1.1 The Choice of Minheap Management . . . 126
7.2 Enhancing Minheap Management . . . 126
7.2.1 Interfacing with Platform I . . . 128
7.2.2 Interfacing with Platform II . . . 128
7.2.3 Hardware Optimizations . . . 128
7.2.4 Performance Optimizations . . . 129
7.3 Experimental Method . . . 129
7.4 Platform I: Experimental Results . . . 129
7.4.1 Unbuffered Test Results . . . 130
7.4.2 Buffered Test Results . . . 130
7.4.3 Unbuffered Test Results on an Uncached System . . . 130

7.4.4 Buffered Test Results on an Uncached System . . . 132
7.4.5 Measuring the Impact of Caching . . . 132
7.5 Platform II: Experimental Results . . . 133
7.6 Interesting Observations . . . 133
7.6.1 Algorithm Complexity . . . 135
7.6.2 Memory Utilization Delays . . . 135
7.6.3 Hardware Optimizations vs. Coprocessor Optimizations . . . 135

8 Model Validation **137**
8.1 Timing Parameter Estimation . . . 137
8.1.1 Assumptions . . . 138
8.1.2 Platform I Configuration Delays . . . 139
8.1.3 Platform II Configuration Delays . . . 140
8.1.4 Summary of Configuration Delays . . . 141
8.1.5 Memory Utilization Observations . . . 141
8.1.6 Platform I Memory Utilization Delays . . . 142
8.1.7 Platform II Memory Utilization Delays . . . 144
8.1.8 Summary of Memory Utilization Delays . . . 145
8.1.9 Bus Utilization and Operating System Behaviour Observations . . . 145
8.1.10 Platform I Lumped Delays . . . 146
8.1.11 Platform II Lumped Delays . . . 146
8.1.12 Summary of Lumped Delays . . . 146
8.1.13 Processing Time Observations . . . 146
8.1.14 Platform I Processing Times . . . 147
8.1.15 Platform II Processing Times . . . 148
8.1.16 Summary of Processing Times . . . 148
8.1.17 Platform Comparison . . . 148
8.2 Comparison of Theoretical Performance with Actual Performance . . . 150
8.3 Observations . . . 151
8.3.1 Pre-Processing and Post-Processing . . . 151
8.3.2 System Profiling . . . 151
8.3.3 Processing Times . . . 152

8.3.4 Application Impact vs. System Impact . . . 152
8.3.5 Performance Model Suitability . . . 152
8.4 Application Implications . . . 153
8.4.1 Course Computation Granularity . . . 153
8.4.2 Opportunities to Exploit Parallelism . . . 153
8.4.3 Transaction I/O . . . 154
8.5 Architectural Implications . . . 154
8.5.1 Configuration Delays . . . 154
8.5.2 Memory Utilization Delays . . . 154
8.5.3 Lumped Delays . . . 155
8.5.4 Processing Times . . . 155
8.5.5 Summary of Implications . . . 156

9 Conclusions **157**
9.1 Thesis Contributions . . . 157
9.1.1 The Performance Model . . . 158
9.1.2 Challenges of Developing Configurable Coprocessors . . . 158
9.1.3 Mainstream Software Application Speedups . . . 159
9.1.4 Mainstream Software Application Delays . . . 160
9.1.5 Desirable Properties of Mainstream Software Applications . . . 160
9.1.6 Desirable Features of Configurable Computer Architectures . . . 161
9.1.7 Reference Design for the ARC-PCI Board . . . 161
9.1.8 Configuration Delays . . . 161
9.2 Potential for Future Research . . . 162
9.2.1 Profiling of Mainstream Software Applications . . . 162
9.2.2 Analysis of the Impact of System Load . . . 163
9.2.3 Platform FPGAs and RPUs . . . 163
9.2.4 Translation of Software Algorithms to Hardware Designs . . . 164
9.3 Thesis Applicability . . . 164

Bibliography **165**

A Experimental Results for Transfer Rates **173**

A.1 Platform I: Windows NT Results 173
A.2 Platform I: Linux Results 174

B Experimental Results for Pseudo-Random Number Generation 176
B.1 Platform I Results 176
B.2 Platform II Results 178
B.3 Platform III Results 180

C Experimental Results for Minheap Management 181
C.1 Platform I Results 181
C.2 Platform II Results 182
C.3 Platform III Results 185

D CSIM M/M/1 Queue Simulation Model 186

List of Tables

2.1 Comparison of Programmable Logic Technologies 10
2.2 Comparison of Bus Throughput 28
2.3 Comparison of Device Throughput 29
2.4 Configurable Computer Taxonomy 33
2.5 Configurable Computing Systems, Boards, and Devices 34

3.1 Transaction Pair Model Timing Parameters 41
3.2 Performance Model Timing Parameters for System 1 48
3.3 Performance Model Timing Parameters for System 2 49
3.4 Timing Parameter Estimates for a Non-Coprocessed System 58
3.5 Timing Parameter Estimates for a Coprocessed System 59
3.6 Estimated Impact of Timing Parameters 60

4.1 Summary of API Functions 71
4.2 Summary of Supported Device Driver IOCTLs 73
4.3 Base Address Regions 76
4.4 Altera FLEX 10K50 Device Configuration Timing 79
4.5 User Design Handshaking Signals 83
4.6 Windows Transfer Comparisons 85
4.7 Windows Transfer Times 87
4.8 Linux Transfer Times 88
4.9 Altera APEX 20K200E Device Configuration Timing 91
4.10 Summary of Computing Platform Processors 95
4.11 Summary of Computing Platform Coprocessors 95

5.1 CSIM Object Classes . . . 101
5.2 CSIM Profiling Results . . . 103
5.3 CSIM M/M/1 Performance Results . . . 107
5.4 CSIM M/M/1 Speedups . . . 108
5.5 System Profiling Raw Results . . . 109
5.6 System Profiling Percentages . . . 110

6.1 Unbuffered Test Results . . . 119
6.2 Buffered Test Results . . . 120
6.3 Unbuffered Test Results on an Uncached System . . . 121
6.4 Buffered Test Results on an Uncached System . . . 121
6.5 Impact of Caching on Performance . . . 122
6.6 Platform II Test Results . . . 123

7.1 Unbuffered Test Results . . . 130
7.2 Buffered Test Results . . . 131
7.3 Unbuffered Test Results on an Uncached System . . . 131
7.4 Buffered Test Results on an Uncached System . . . 132
7.5 Impact of Caching on Performance . . . 133
7.6 Platform II Blocking Test Results . . . 134
7.7 Platform II Non-Blocking Test Results . . . 134

8.1 Summary of Configuration Delays . . . 141
8.2 Summary of Memory Utilization Delays . . . 145
8.3 Summary of Lumped Delays . . . 147
8.4 Processing Time Summary . . . 148
8.5 Non-Coprocessed Timing Summary . . . 149
8.6 Coprocessed Timing Summary . . . 149
8.7 Estimated Bounds on System Speedups . . . 150
8.8 Actual Application Speedups . . . 150

A.1 Windows Device Driver Execution Times - Software Timed (WSPEED1A) . . . 173
A.2 Windows Device Driver Execution Times - Hardware Timed (WSPEED1B) . . . 174

A.3 Windows Unbuffered Application Execution Times (WSPEED2) 174
A.4 Windows Buffered Application Execution Times (WSPEED3) 174
A.5 Linux Device Driver Execution Times (LSPEED1) 175
A.6 Linux Unbuffered Application Execution Times (LSPEED2) 175
A.7 Linux Buffered Application Execution Times (LSPEED3) 175

B.1 PRAND1 177
B.2 PRAND2 177
B.3 PRAND3 177
B.4 PRAND1NC 178
B.5 PRAND2NC 178
B.6 PRAND3NC 179
B.7 ERAND1 179
B.8 ERAND2 179
B.9 SRAND1 180

C.1 PMIN1 181
C.2 PMIN2 182
C.3 PMIN3 182
C.4 PMIN1NC 183
C.5 PMIN2NC 183
C.6 PMIN3NC 183
C.7 EMIN1 184
C.8 EMIN2 184
C.9 EMIN3 184
C.10 SMIN1 185

List of Figures

1.1 Classification of Computing Machines 4

2.1 Classification of Programmable Logic Devices 10
2.2 Output Macrocell from a Philips P3Z22V10 13
2.3 Architecture of a Philips P3Z22V10 14
2.4 A Portion of an Aptix FPIC AX1024R 15
2.5 Actel ACT1 Series FPGA Architecture 18
2.6 Actel ACT1 Series FPGA Logic Module 19
2.7 Xilinx 4K Series FPGA Architecture 19
2.8 Xilinx 4K Series Combinational Logic Block 20
2.9 Xilinx 4K Series Input/Output Block 21
2.10 Altera Flex 10K Series CPLD Architecture 23
2.11 von Neumann Computer Architecture 25
2.12 Intel 80x86 Processor Family Personal Computer Architecture 26
2.13 Types of Configurable Computer Architectures 30
2.14 Types of Configuration Modes 31
2.15 Types of Configuration Contexts 32

3.1 Transaction Pair Model 40
3.2 Transactions Without Parameters 42
3.3 Transactions Without Return Values 43
3.4 Comparison of Transaction Performance 45
3.5 Comparisons of Execution Times 50

4.1 Illustration of Platform I 64

4.2 Photograph of the ARC-PCI Board 65
4.3 ARC-PCI Board Bus Architecture 68
4.4 ARC-PCI System Components 70
4.5 Configuration Timing for Altera FLEX 10K Series Devices 78
4.6 ARC-PCI Board Configuration on a Windows NT Platform 81
4.7 ARC-PCI Board Configuration on a Linux Platform 82
4.8 Hardware Development Flow 83
4.9 Buffered vs. Unbuffered Transactions 86
4.10 Illustration of Platform II 90
4.11 Nios Embedded Processor Development Board 91
4.12 Nios Embedded Processor System Components 93

5.1 Coprocessed CSIM Application 106

6.1 Pseudo-Random Number Generator Finite State Machine 116

7.1 Minheap Interface Finite State Machine 127

Chapter 1

Introduction

This thesis investigates use of configurable computers for the purpose of enhancing mainstream software applications. Previous research has shown that configurable computers can improve the performance of niche applications. The goal of this research is to quantify, model, and analyze the performance of configurable computers with respect to mainstream software applications. This research explores the feasibility of building a library of configurable hardware components to enhance tasks commonly required by software applications.

1.1 Motivation

Recent advances in the programmable logic device industry have made new approaches to computing feasible. These advances include the following:

Increased Gate Capacity – Modern CPLDs (Complex Programmable Logic Devices) and FPGAs (Field Programmable Gate Arrays) provide as many as 8,000,000 usable gates using in excess of 100 million transistors. By comparison, the Pentium II processor introduced in May of 1997 consisted of 7.5 million transistors [Int02].

Improved Performance – Devices have been shown to handle clock frequencies in excess of 250 MHz [Von97] and many devices are capable of achieving clock frequencies in excess of

66 MHz without the need for floorplanning[1].

Reduced Cost – In large quantities, programmable logic devices cost about the same as a modern processor. Devices such as the Cyclone [Cor02a] [Cor02b] are significantly less expensive than general-purpose processors.

Dynamic Configuration – Dynamic (run-time) configuration has increased the flexibility and usefulness of programmable logic devices.

Partial Configuration – Partial configuration has enabled a portion of a logic device to be configured while the remainder of the device continues normal operation, thus reducing the impact of configuration delays.

Advanced CAD Tools – Innovations in CAD (Computer-Aided Design) tools and the proliferation of HDLs (Hardware Description Languages) have simplified the task of building reusable, custom digital circuits.

In 1997, industry trends indicated that high-speed, partially configurable logic devices with 1,000,000 usable gates would be in production by 2001 [RH97]. This turned out to be a conservative estimate. By the 2nd quarter of 2000, the first commercially available logic device with 1,000,000 usable gates was released. The availability of such devices has spurred interest in the field of configurable computing. The study of configurable computing focuses on the design, implementation, and use of configurable computers. A configurable computer is a computing machine that possesses the ability to adapt its hardware architecture in real-time to the computation at hand. While configurable computers need not utilize programmable logic devices, the existence of such devices has simplified the development of configurable computers.

Configurable computing is based on the principle that a custom computing machine is likely to perform a particular computation more efficiently than a comparable general-purpose machine. A configurable computer's hardware may be customized for a particular computation to exploit parallelism. Traditionally, researchers have focused on the use of configurable computers for bit-serial algorithms exhibiting fine-grained parallelism. Research has shown that programmable logic devices are capable of achieving orders of magnitude of performance improvements compared to

[1] Floorplanning is the manual assignment of logic resources and routing resources by a hardware designer. As development tools have improved, the need for this low-level assignment of resources has diminished.

software implementations for certain applications including image processing [AA95], cryptography [VBR+96], and hardware emulation [DGJ+95]. These niche applications of configurable computing are important to some computer users. However, the majority of computer users rarely have a need for applications such as image processing and hardware emulation.

A few researchers have investigated configurable computing as a means of mainstream computing. The BRASS (Berkeley Reconfigurable Architectures, Systems, and Software) Research Group has investigated the use of programmable logic devices as configurable microprocessors [Deh94]. The goal of the BRASS Research Group was to create one or more microprocessors within a single configurable logic device and then customize the functional units within these microprocessors to the instruction stream. This research has the potential to revolutionize mainstream computing but the complexities of the approach have prevented it from becoming a mainstream computing technique. The most significant problem with this approach is that existing technologies still do not provide sufficient memory bandwidth to support the frequent dynamic configurations required.

Rather than focus on a niche application or struggle with the problems of rapid run-time configuration, this research investigates the development of configurable coprocessors to accelerate common software components. By choosing mainstream software tasks most suitable for acceleration, it is possible to avoid the problem of frequent configuration. The underlying belief is that the achievement of a modest speedup applicable to a broad range of software is more significant to mainstream computing than the achievement of a large speedup applicable only to a single software program. For configurable computing to become a mainstream computing technique, it must be shown that configurable computers are useful for more than just niche applications.

1.2 Configurable Computing

Configurable computing is not a new area of research. The field of configurable computing evolved from research into the reprogrammable and restructurable computers of the 1950's and 1960's. In the early 1960's, Estrin pioneered the development of the first restructurable computer [EBTB63]. His belief was that hardware specialization could enhance the performance of software while simplifying the task of programmers [EBTB63]. Modern configurable computing is based on essentially the same principles although the technology has improved substantially.

The terms *reprogrammable*, *restructurable*, *microprogrammable* and *configurable* are not interchangeable. A *reprogrammable* computer may be programmed to perform different tasks using software. Reprogrammable computers may use a fixed hardware architecture since programmability is provided using software. Virtually all modern computer architectures fall into this class of computers. A *restructurable* computer uses a network of datapaths to exploit multiple processing units for faster execution of a particular software task. The datapaths of a restructurable computer may be modified at run-time but the control logic elements (i.e., processing units) are fixed. A *microprogrammable* computer may be programmed at a low-level to emulate a particular instruction set. This provides some control logic flexibility. A *configurable* computer permits the creation/replacement of one or more processing units as well as modification of the datapaths. This class of computer permits both the configuring of datapaths and control logic at run-time.

Figure 1.1 classifies computing machines into reprogrammable, restructurable, configurable, and microprogrammable machines on the basis of datapath and control logic flexibility. It should be noted that a particular computing machine may fall into two or more classifications.

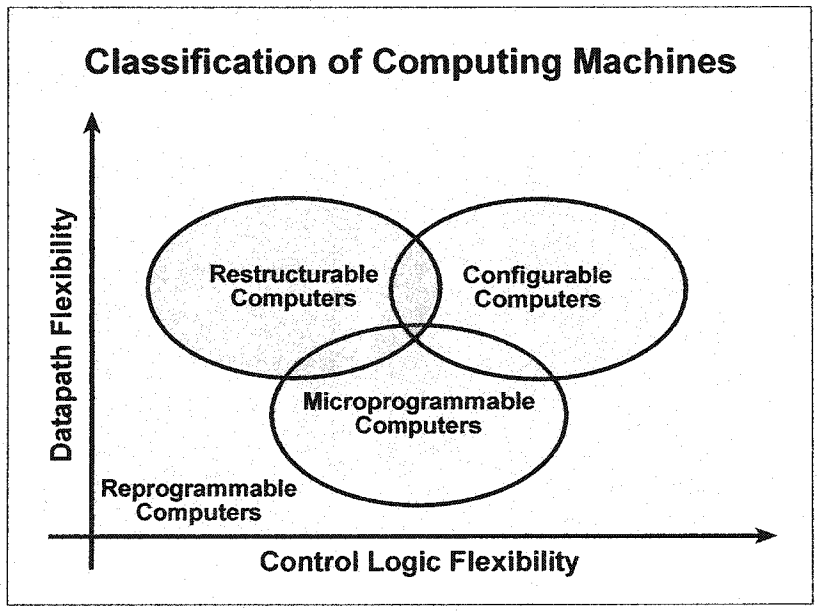


Figure 1.1: Classification of Computing Machines

The term, configurable computer, is now most often applied to computing machines using dynamically configurable logic. In other words, configurable computers are now synonymous

with a class of computing devices incorporating configurable logic devices that can be configured in-system, often at run-time. This definition is consistent with the use of the term in recent literature.

1.3 Statement of Thesis

It is my thesis that programmable logic devices have the potential to enhance mainstream computing platforms in ways previously unimagined. Programmable logic devices can be integrated into modern personal computers and workstations to provide a means of achieving the benefits of specialized hardware without incurring the costs typically associated with dedicated hardware accelerators. These devices open new possibilities for tasks or algorithms that can benefit from specialized hardware. Even tasks that only marginally benefit from hardware acceleration are potentially suitable for acceleration using a configurable coprocessor.

Tasks commonly implemented in software can be replaced with equivalent configurable hardware components. The tasks to be implemented as configurable hardware components can be determined by profiling target applications. It is likely that tasks such as memory management, queueing, and sorting are good candidates for replacement. Once developed, configurable hardware components, in the form of bitstream files for a particular configurable logic device, may be dynamically loaded into a configurable logic board or configurable logic devices mounted on a customized motherboard. By tightly coupling a configurable coprocessor with the processor of the host computer system, enhanced performance is possible. Software can take advantage of the configurable computing engine by calling specialized functions which communicate with the custom hardware.

In some respects, the approach and its objectives are similar to those of firmware [Opl67]. However, configurable computing is capable of much more than just the resequencing of a set of instructions. Configurable logic permits an entire computer architecture, including the control logic and datapaths, to be configured. Complete hardware specialization is possible. As a result, configurable computing provides flexibility without the degradation of performance sometimes associated with microprogramming-based approaches.

Even a modest improvement in performance may be justifiable for mainstream computing. It

is unreasonable to expect that this approach can achieve orders of magnitudes of performance improvements for mainstream computing applications. Rather than attempt to achieve a huge improvement in a small subset of applications, this research attempts to achieve a modest improvement in a broad range of applications. If configurable logic is ever to enter into mainstream computing, this will be the path required.

1.4 Thesis Contributions

This thesis makes the following contributions to the existing body of literature on configurable computing for mainstream software applications:

1. introduces, explains, and validates a novel configurable computing performance model that predicts both application performance and system performance,
2. illustrates several of the challenges associated with developing configurable coprocessors,
3. provides experimental results demonstrating speedups for two mainstream software applications (pseudo-random number generation and minheap management) executing on a tightly-coupled configurable computer,
4. quantifies memory utilization delays, bus utilization delays, and operating system behaviour delays for a mainstream software application (CSIM),
5. summarizes several desirable properties of mainstream software applications for configurable coprocessing,
6. identifies desirable features of configurable computer architectures to ensure adequate performance,
7. describes a reference design for an ARC-PCI Board with hardware support for dynamic configuration and high-speed I/O, and
8. shows that configuration delays are not the most significant source of delay for computationally-intensive applications.

1.5 Outline of the Thesis

Chapter 2 introduces the subject of configurable computing by presenting a discussion of the devices, tools, and configurable computing platforms that are commonly used. This chapter also describes some of the challenges of configurable computing. Chapter 3 describes the need for a model of configurable computers. A novel performance model of configurable computing is introduced and described in detail. Chapter 4 describes the test platforms used to obtain the experimental results presented in this thesis. The development process for each configurable computing platform is presented. Chapter 5, Chapter 6, and Chapter 7 describe experiments conducted on mainstream software applications. The first application, CSIM, is a discrete-event simulation library. The second application presented is pseudo-random number generation. The third and final application studied is minheap management, a task not normally implemented in hardware. Chapter 8 validates the performance model using the experimental results. This chapter also notes several interesting observations. Chapter 9 concludes the thesis with a discussion of the thesis contributions and interesting possibilities for future research.

Chapter 2

Introduction to Configurable Computing

A *configurable computer* is a computing device that provides hardware that may be modified at run-time to efficiently compute a set of tasks. Research has indicated price-performance improvements of a factor of 1,000 can be achieved using configurable computing techniques for computation intensive applications [Xil97]. Configurable computing is a relatively new area of research, despite the fact that programmable logic devices have been in existence for several decades. Technological advances now make it possible to exploit the dynamic nature of programmable logic devices. This chapter introduces the technology behind configurable computing and the terminology associated with this new area of research.

2.1 Programmable Logic Devices

The recent interest in configurable computing is largely due to the existence of high-speed, high-density programmable logic devices. The programmable logic device is the basic building block of a modern configurable computer. It provides the ability to modify both the control logic and datapaths of a portion of a computer in real-time. Since modern programmable logic devices evolved from earlier forms of programmable logic devices, no discussion of configurable computing

would be complete without a thorough introduction to programmable logic devices.

2.1.1 Overview

A *PLD (Programmable Logic Device)* is an integrated circuit that implements a digital circuit designed and programmed by a user. The first programmable logic devices became popular in the 1970's as a replacement for SSI (Small-Scale Integration) and MSI (Medium-Scale Integration) logic [BR96]. These early devices were ideal for implementing interface (also known as "glue") logic and other simple circuits. As programmable logic technology advanced, new forms of PLDs were introduced. The most important of these devices, FPGAs (Field Programmable Gate Arrays) and CPLDs (Complex Programmable Logic Devices), provided the size, performance, and ease of use necessary for experiments in configurable computing.

Several classifications of PLDs have been proposed [BR96] [Alt96] [Jen94] [Tri94]. Building upon these classifications, the one shown in Figure 2.1 is proposed. This classification enhances previous ones through the addition of a class for RPUs (Reconfigurable Processing Units). In addition, some new part names recently proposed (Platform FPGAs, and PSIDs) are included in the classification. Each class of PLD is discussed in detail later in this chapter.

2.1.2 Programmable Logic Technologies

Not all programmable logic devices are created equal. The characteristics of a programmable logic device are largely determined by the technology used to construct the programmable logic. Among the technologies used for building a programmable logic device are PROM (Programmable Read-Only Memory), EPROM (Erasable Programmable Read-Only Memory), EEPROM (Electrically Erasable Programmable Read-Only Memory), PLICE (Programmable Low-Impedance Circuit Element) Antifuse, ViaLink Antifuse, Flash SRAM (Static Random Access Memory) and SRAM. These technologies are discussed in detail in the book, "Principles of CMOS VLSI Design: A Systems Perspective" [WE93]. Table 2.1 provides a brief comparison of the features of these technologies.

Of these technologies, only SRAM-based devices with unlimited programming lifetimes are suitable for use in a configurable computing system. All other programmable logic device tech-

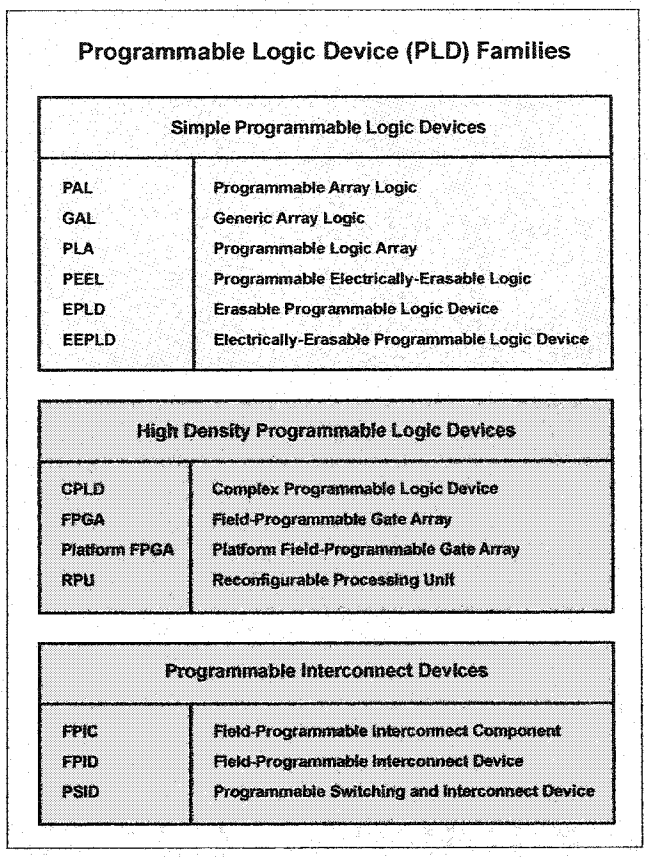


Figure 2.1: Classification of Programmable Logic Devices

Technology	Volatile	Programming		Density	Speed
		Method	Lifetime in Cycles		
PLICE Antifuse	No	External	1	High	High
ViaLink Antifuse	No	External	1	High	High
PROM	No	External	1	High	High
EPROM	No	External	10,000 Typical	High	High
EEPROM	No	External †	10,000 Typical	Moderate	Moderate
Flash SRAM	No	In-System	10,000 Typical	Moderate	Moderate
SRAM	Yes	In-System	Unlimited	Low	Moderate

† Some EEPROM devices can support in-system reprogramming if the circuit is supplied a programming voltage in addition to the normal supply voltage.

Table 2.1: Comparison of Programmable Logic Technologies

nologies lack the ability to be reprogrammed an unlimited[1] number of times. Any PLD that can be reprogrammed an unlimited number of times is considered to be a *configurable logic device.*

2.1.3 Statically, Dynamically, and Partially Programmable Logic Devices

It is sometimes useful to classify PLDs based on the reconfiguration methods the devices support. PLDs are commonly classified into the following three categories: statically-programmable (one-time programmable), dynamically-configurable, and partially-configurable logic devices.

Statically-programmable logic devices may be configured a finite number of times (e.g., 1,000-10,000 times). Although these devices are not suitable for use in a configurable computing system due to their limited lifetime, these devices can be used for computing systems that do not require dynamic (run-time) configuration. Devices based on EEPROM and Flash SRAM technology are examples of statically-programmable logic devices. These devices are suitable for applications such as rapid prototyping and hardware emulation. Although each prototype of the system requires a configuration cycle, it is unlikely that the number of prototypes tested over the usable lifetime of the part will exceed the number of configuration cycles supported by the device. A statically-programmable device is unsuitable for the implementation of a system requiring run-time configuration.

Dynamically-configurable devices may be fully-configured an unlimited number of times. These devices permit the implementation of systems that require configuration at run-time. All dynamically-configurable devices incorporate SRAM technology. A dynamically-configurable device may be used to implement a system that requires frequent configuration.

Partially-configurable devices refer to a subset of dynamically-configurable devices that may be configured one portion at a time. While a subset of the device is configured, the rest of the device continues normal operation. Partially-configurable devices can be used to reduce the amount of time spent waiting for the completion of the configuration of a device at run-time. Partial configuration helps to reduce the delays associated with the configuration of a device.

[1]Strictly speaking, the lifetime of a device is limited since all devices eventually fail. A configurable logic device is simply a device whose lifetime is independent of the number of configuration cycles.

This capability is an extremely useful feature for configurable computing.

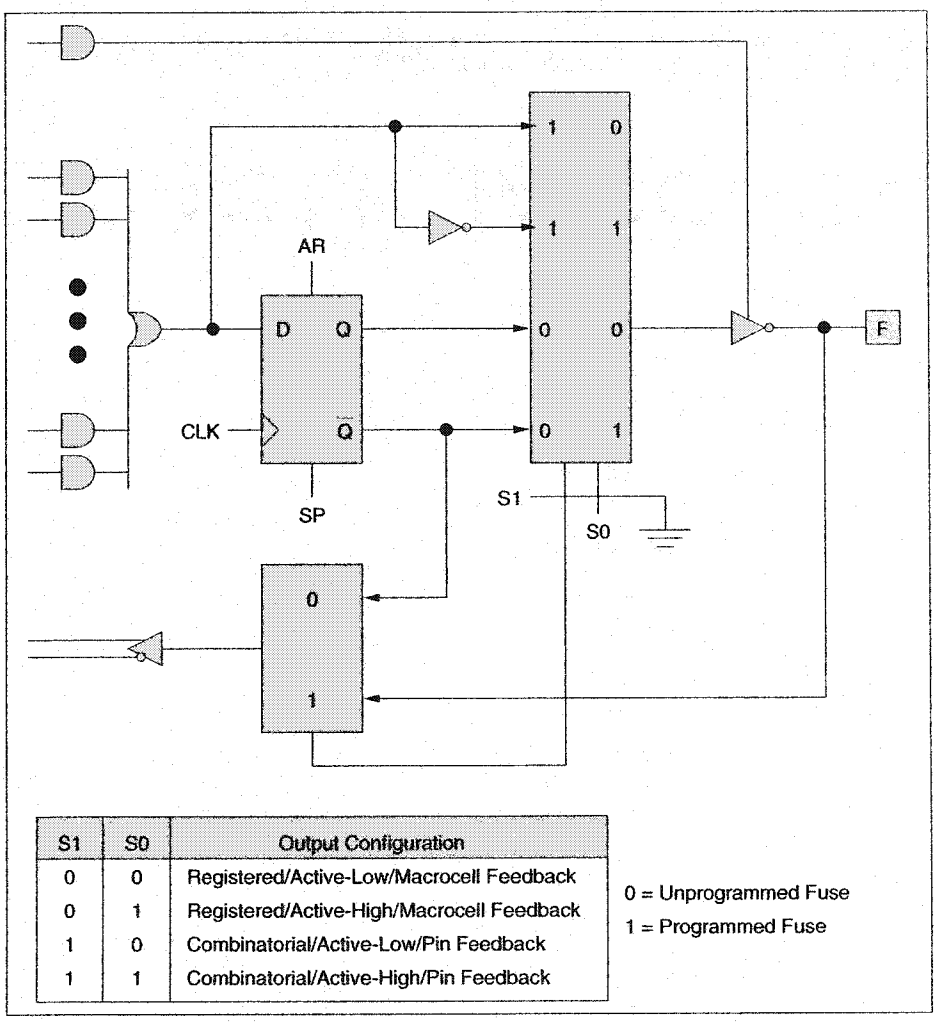
2.1.4 SPLDs (Simple Programmable Logic Devices)

The term, SPLD, is frequently used to refer to the simplest class of PLDs. The two basic types of SPLDs are PLAs (Programmable Logic Array) and PALs (Programmable Array Logic). The PLA was the first PLD to be manufactured commercially [BR96]. The first PLAs were introduced in the early 1970's but due to high manufacturing costs and poor performance, these devices did not become popular. PLAs are composed of a programmable AND-plane and a programmable OR-plane. These planes provide the equivalent of AND and OR gates using wired logic. Combinations of AND and OR gates may be used to implement any small digital logic circuit within a PLA [Man91].

PALs, the second type of SPLD, are also referred to as GALs (Generic Array Logic) or PEELs (Programmable Electrically Erasable Logic), depending upon the manufacturer of the devices. The basic building block of a PAL is called a macrocell. PALs are composed of a set of macrocells connected to an AND-plane. The set of macrocells is sometimes referred to as a fixed OR-plane since each input to a macrocell must pass through an OR gate. PALs are less complex than PLAs yet provide nearly as much flexibility. A PAL may be configured to compute combinatorial and/or registered functions of a small number of inputs. The exact number of inputs varies by part. The number of macrocells in a device often corresponds with the number of outputs leaving the device. An example of an output macrocell from a Philips P3Z22V10 [Phi97] is shown in Figure 2.2.

PALs typically incorporate EPROM or EEPROM technology for performance reasons. EPROM and EEPROM technology are often capable of supporting higher clock frequencies than SRAM technology. A few SPLDs based on SRAM technology exist. A common PAL is the 22V10. This device consists of 10 output macrocells and is therefore capable of generating up to 10 output signals. The architecture of a Philips P3Z22V10 [Phi97], a modern version of an EEPROM-based 22V10, is shown in Figure 2.3.

Due to the use of wide programmable planes (i.e., the AND plane in Figure 2.3), SPLDs are limited in the number of macrocells that can be used effectively. This architecture restricts the size of circuits that can be implemented in a single SPLD. Although SRAM-based SPLDs can



S1	S0	Output Configuration
0	0	Registered/Active-Low/Macrocell Feedback
0	1	Registered/Active-High/Macrocell Feedback
1	0	Combinatorial/Active-Low/Pin Feedback
1	1	Combinatorial/Active-High/Pin Feedback

0 = Unprogrammed Fuse
1 = Programmed Fuse

Figure 2.2: Output Macrocell from a Philips P3Z22V10

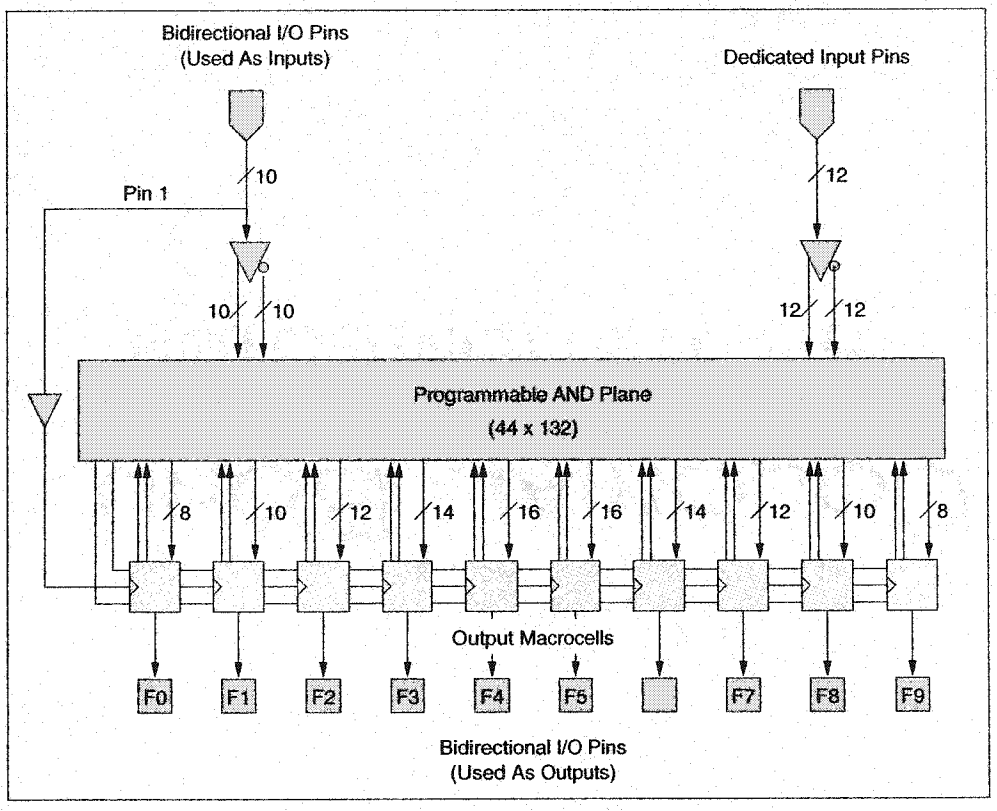


Figure 2.3: Architecture of a Philips P3Z22V10

be used for small configurable logic designs, they are unsuitable for configurable computing tasks requiring a substantial amount of programmable logic.

2.1.5 PIDs (Programmable Interconnect Devices)

PIDs are a class of PLD specifically designed for switching and routing signals. Although these devices use programmable logic technology, they do not have the capability to compute combinatorial or sequential functions of inputs. Their sole purpose is to route signals from input pins to one or more output pins.

The two best examples of PIDs are FPICs (Field Programmable Interconnect Components) and FPIDs (Field Programmable Interconnect Devices). FPICs are trademarks of Aptix [Apt93] and FPIDs are trademarks of I-Cube [I-C94]. Functionally, FPICs and FPIDs are identical. FPICs and FPIDs use a non-blocking switch matrix based on SRAM technology to route signals from input pins to the appropriate output pins. A portion of an Aptix FPIC architecture is shown in Figure 2.4. These devices use segmented channel routing to connect one or more I/O pads.

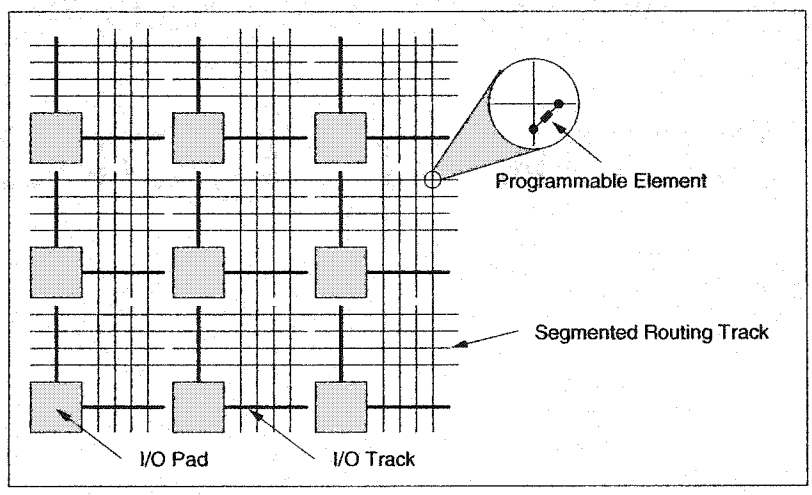


Figure 2.4: A Portion of an Aptix FPIC AX1024R

The performance of PIDs has typically been limited to clock frequencies less than 20 MHz [Apt95]. To address this limitation, I-Cube introduced a device known as a PSID (Programmable Switching and Interconnect Device) [I-C97]. PSIDs target applications that require high-speed

switching. PSIDs support frequencies up to 233 MHz [I-C97] through the use of a non-blocking switch matrix and design enhancements that simplify the routing of high-speed busses.

Although PIDs have obvious uses for networking applications, they are also useful for configurable computing. With a PID, it is possible to create a programmable circuit board that allows the routing of signals on the board to be user-defined. Aptix FPCBs (Field Programmable Circuit Boards) [Apt95] are examples of programmable circuit boards. These FPCBs are extremely useful for the rapid prototyping of systems involving multiple PLDs. Since PIDs are based on SRAM technology, the signal routing on the circuit board may be configured as often as required. Given this property, designers of PLD circuits need not fix pin locations for a prototype circuit board early in the design process. This approach simplifies and expedites the prototyping process.

2.1.6 HDPLDs (High-Density Programmable Logic Devices)

As the density of PLD devices increased, new HDPLD (High-Density Programmable Logic Device) architectures were adopted to permit more efficient placement and routing of macrocells. These advanced HDPLD (High-Density Programmable Logic Device) architectures include devices more commonly referred to as FPGAs (Field Programmable Gate Arrays) and CPLDs (Complex Programmable Logic Devices). Although FPGA and CPLD architectures can be quite different, both devices target similar applications (large-scale programmable digital circuits).

Gate arrays and FPGAs (Field Programmable Gate Arrays) are integrated circuits based on the use of a regular array of gates connected by user-defined routing. In the case of an ordinary gate array, the designer is only responsible for creating the masks for the gate array routing (i.e., the metalization layers). This approach reduces the NRE (Non-Recurring Engineering) costs associated with the fabrication of an ASIC device. FPGAs further reduce NRE costs by allowing the designer to route (program) the FPGA after fabrication. FPGAs are generic gate array devices that may be programmed in the field.

FPGAs can be classified by programming technology and by cell topology. The programming technology impacts device density, performance, and reprogrammability. The cell topology impacts device density, performance, and routability. The four FPGA cell topologies are row-oriented, structured array, hierarchical, and sea-of-gate topologies.

The Actel ACT1 Series FPGA, shown in Figure 2.5, is an example of a row-oriented topology. This device consists of horizontal rows of logic modules. The general-purpose logic module shown in Figure 2.6 may be used to implement simple combinational and sequential logic gates. Between each row, a channel of routing segments is used to route internal signals horizontally. Vertical routing segments (not shown in Figure 2.5 unless used to make a connection) permit the connection of logic modules with horizontal routing segments. The perimeter of a row-oriented device is surrounded by input/output modules connecting internal signals to the I/O pins of the device via buffers.

Row-oriented topologies are often used to implement PLDs based on Antifuse technology. The use of Antifuse technology permits the vertical interconnect to run directly through logic modules. An example of a logic module is shown in Figure 2.6. The ability to run vertical interconnect results in space-efficient, high-performance devices. However, recall that Antifuse technology is not suitable for configurable computing.

The Xilinx 4K Series FPGA, shown in Figure 2.7, is an example of a structured array topology. The basic building block of this type of array is a CLB (Combinational Logic Block) shown in Figure 2.8. In this architecture, CLBs consisting of LUTs (Look-Up Tables) and flip-flops are arranged in a matrix. This FPGA architecture is sometimes called island-style since each CLB is surrounded by horizontal and vertical routing channels. At the perimeter of the device, IOBs (I/O Blocks) are placed to connect internal signals to the I/O pins. An example of an IOB is shown in Figure 2.9.

Island-style FPGA topologies are common. FPGAs based on structured array topologies are simple to understand and easily expandable. The trade-off between density and routability may be adjusted by simply adding or removing routing segments. Similarly, device size may be increased by simply adding new rows and/or columns of CLBs along with the appropriate routing segments.

CPLDs (Complex Programmable Logic Devices) are an example of a hierarchical PLD. Most, but not all, CPLDs consist of a hierarchy of smaller PLDs. For example, a hierarchy of 22V10 devices may be connected to form a CPLD. Devices such as the Altera MAX 7000 Series are an example of this type of CPLD. However, CPLDs also include devices that might otherwise be classified as FPGAs. An example of a CPLD architecture is shown in Figure 2.10. The Altera Flex10K architecture resembles that of an island-style FPGA. This device is considered to be a

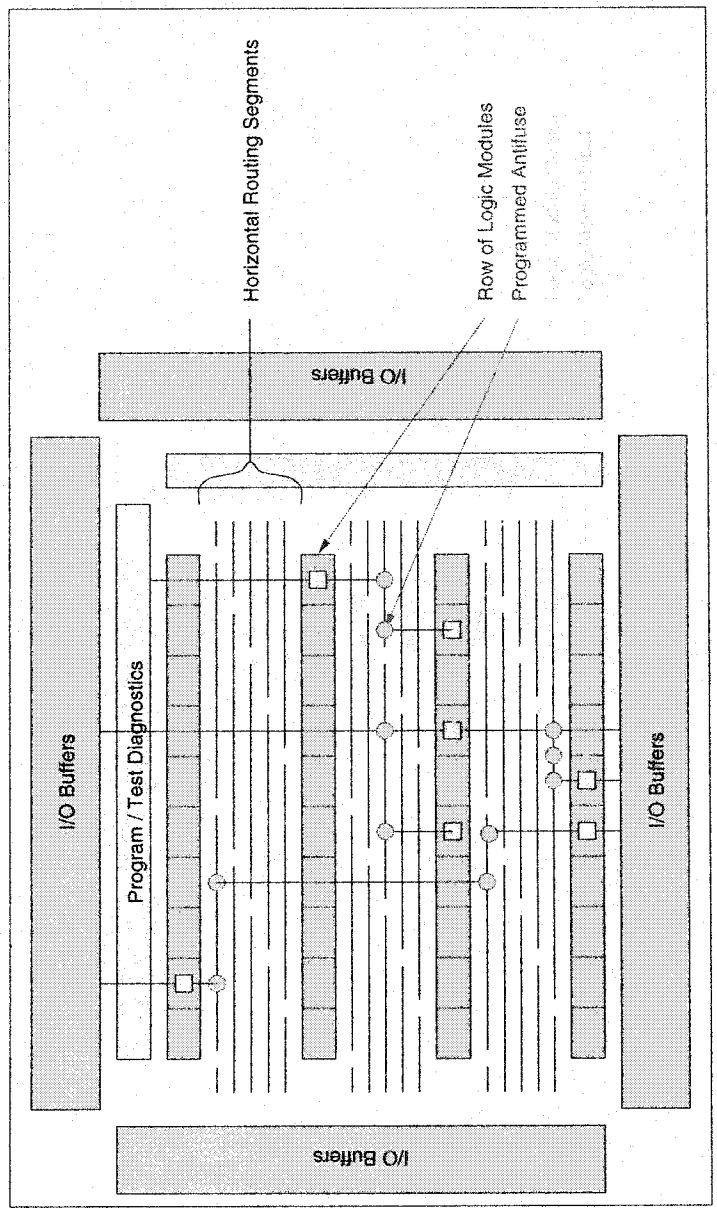


Figure 2.5: Actel ACT1 Series FPGA Architecture

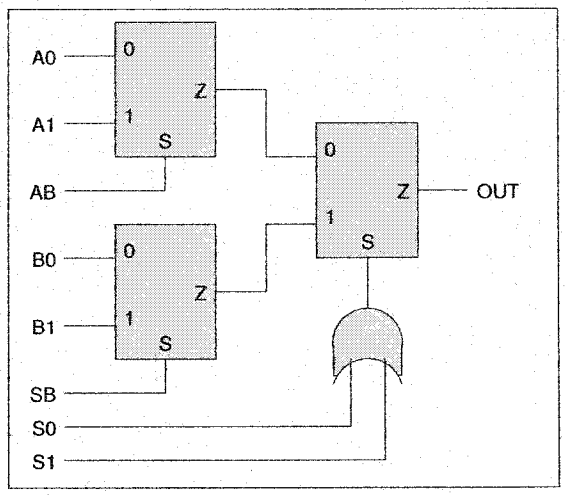


Figure 2.6: Actel ACT1 Series FPGA Logic Module

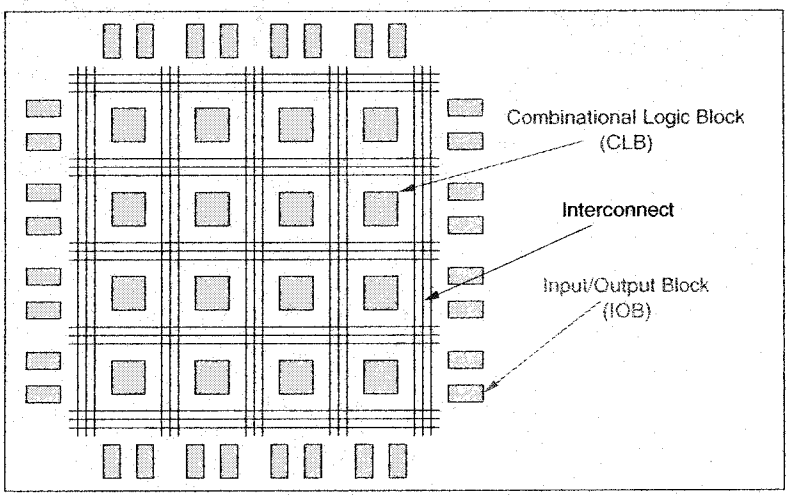


Figure 2.7: Xilinx 4K Series FPGA Architecture

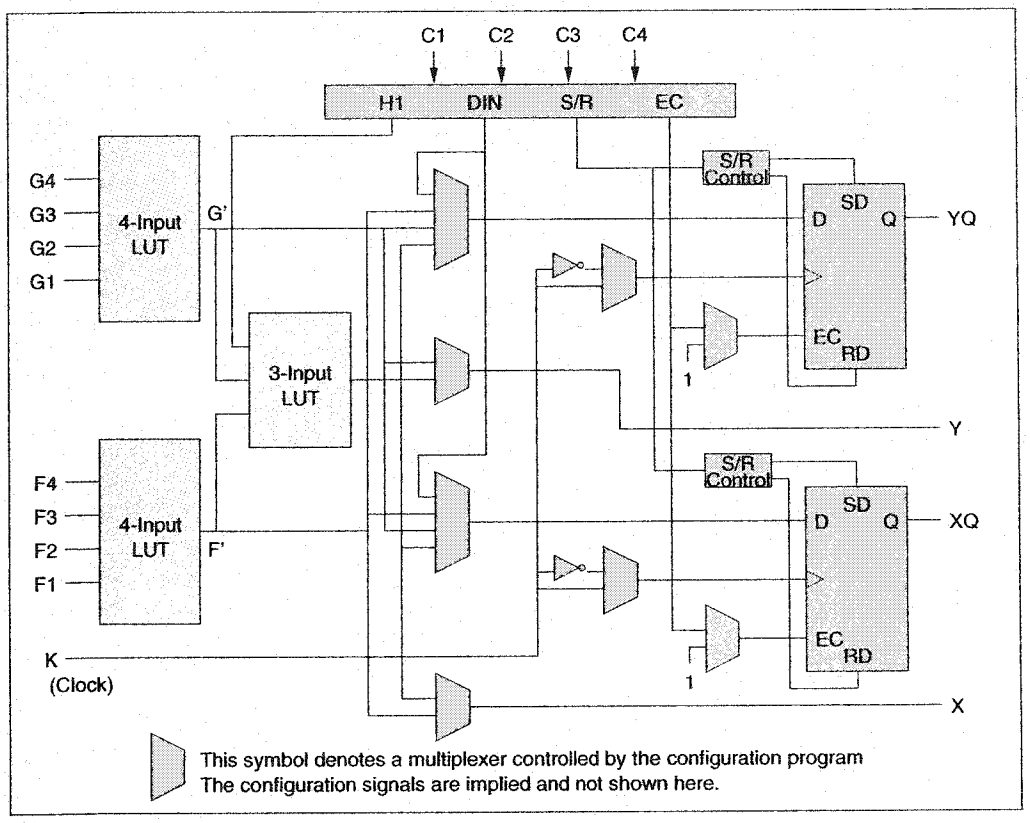


Figure 2.8: Xilinx 4K Series Combinational Logic Block

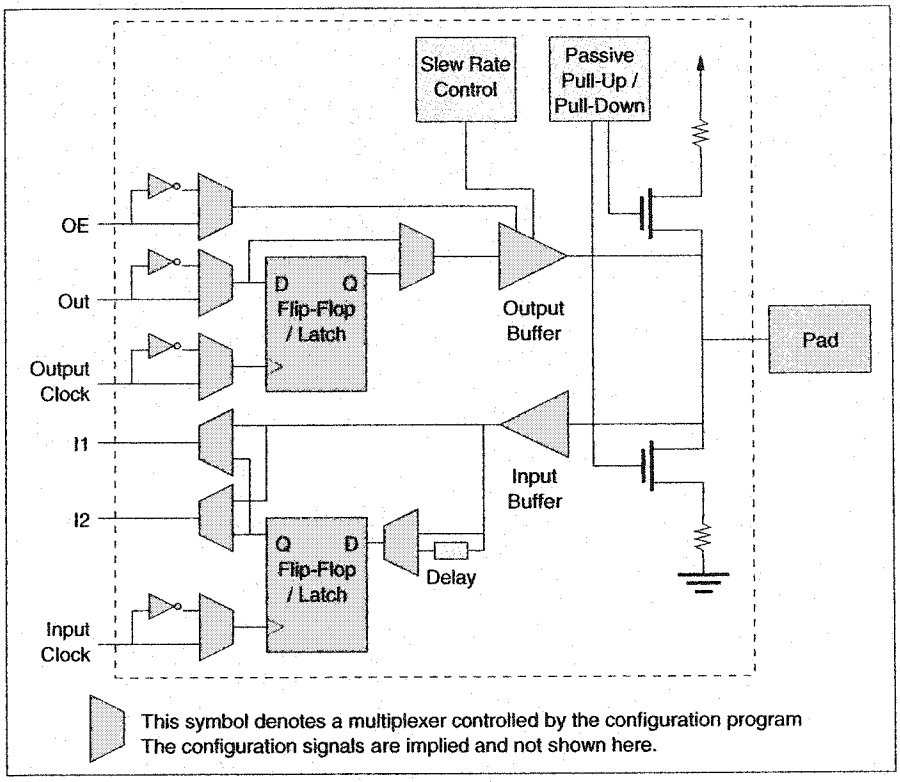


Figure 2.9: Xilinx 4K Series Input/Output Block

CPLD for the following reasons:

1. It consists of a hierarchy of LABs (Logic Array Blocks) that resemble SPLDs.
2. The ratio of flip-flops to combinational logic gates is not nearly as high as the ratio found in a typical FPGA.
3. It directly supports functions of a large number of inputs.
4. The performance of the device is more predictable than the performance of a typical FPGA due to its non-segmented routing.

The sea-of-gates topology was popular in the late 1980's prior to the merger of Algotronix and Xilinx. The Algotronix CAL1024 device is an example of a sea-of-gates device. It is no longer commercially produced. As device densities increased, the sea-of-gates topology was phased out of development. The routing of a sea-of-gates device posed a difficult problem for modern CAD tools due to the amount of flexibility provided by the architecture. More restrictive topologies were found to be much easier to place-and-route.

Regardless of the topology, FPGAs and CPLDs are the highest density PLDs on the market with gate counts approaching 8,000,000 usable gates. Very complex devices incorporating built-in functional units and high-speed serial I/O have been introduced. These technological advances further improve the density and performance of devices. SRAM-based HDPLDs are the building blocks of modern configurable computers.

2.1.7 RPUs (Reconfigurable Processing Units)

The term RPU was first proposed by Steve Casselman of Virtual Computing Corporation [Cas96]. This class of PLD includes FPGAs and CPLDs with special enhancements to support partially configurable computing. These devices provide mechanisms for high-speed partial configuration. Hardware components may be swapped into and out of a single device in real-time. In addition, these devices provide built-in functional units to improve the performance of common operations such as addition and bit-shifting. Instead of building upon arrays of combinational logic blocks and logic elements, RPUs often build upon arrays of simple processing units. Over 20 different variations of RPUs have been documented [Har01].

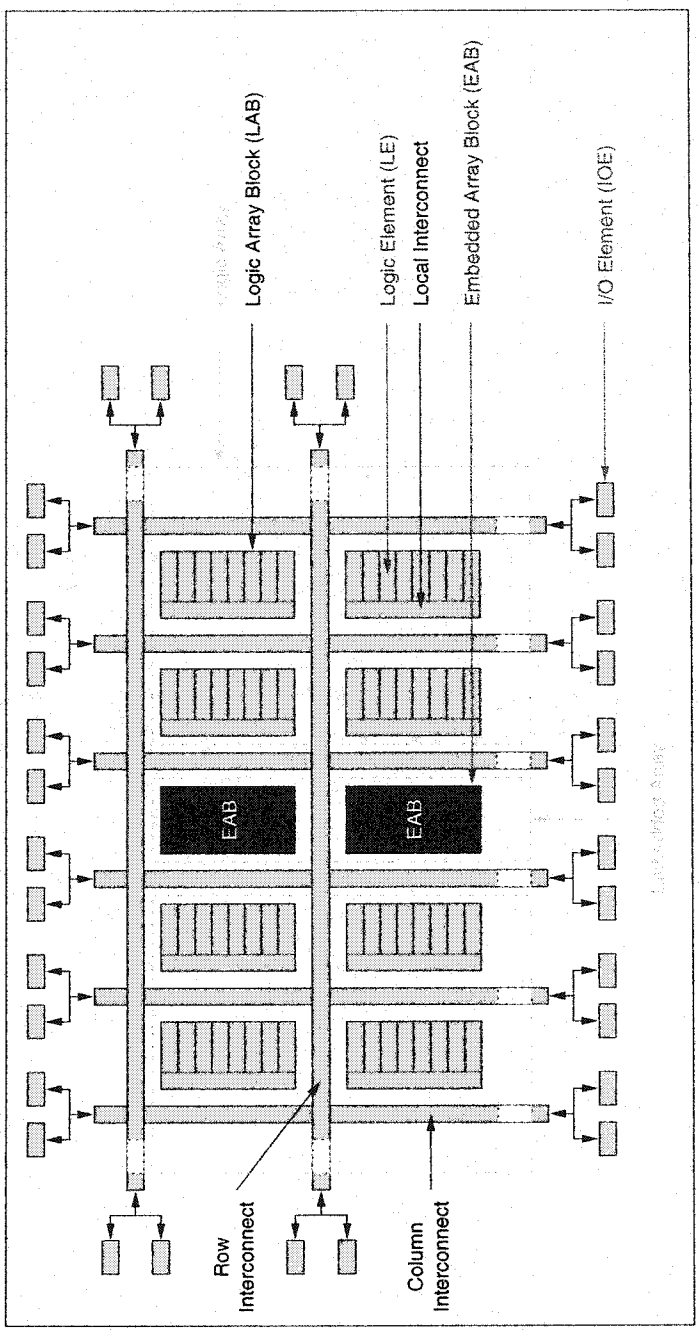


Figure 2.10: Altera Flex 10K Series CPLD Architecture

The DPGA (Dynamic Programmable Gate Array) [Deh94] proposed by André Dehon at MIT was one of the first RPU designs. This chip included a small on-chip instruction memory to assist with context-switching. Based on this device, the TSFPGA Time-Switched Field Programmable Gate Array [Deh96] and the MATRIX [MD96] RPUs were developed. The Colt device developed by Peter Athanas is another example of an RPU. It tackles the problem of context-switching using a technique known as Wormhole Run-Time Reconfiguration [BAM96].

The Xilinx XC6200 Series devices are the only commercial devices on the market that may be classified as RPUs. These devices support partial configuration and provide dedicated logic for the implementation of important functions. The Xilinx XC6200 Series devices are also the first commercial devices to have a public-domain bitstream format. All other PLDs use proprietary bitstream formats for device programming. For this reason, these devices have been widely adopted for research into configurable computing. Research conducted by Xilinx in August 1997 indicated that more than 100 research institutions around the world had acquired Xilinx XC6200 Series devices for research [Xil97].

2.2 Introduction to Modern Computer Architecture

Modern computer architecture has evolved since the early days of computing. Technological advances have permitted the development of complex computer architectures. However, the fundamentals of computer architecture remain the same. All computers consist of processing units, storage devices, input/output devices, and interconnection structures. The number of processing units and storage devices supported, the types of processing units and storage devices supported, and the interconnections among processing units and storage devices may differ but the building blocks remain the same.

2.2.1 von Neumann Computer Architecture

John von Neumann introduced a fundamental model of computation in 1945 when he drafted a report [vN45] on the EDVAC (Electronic Discrete Variable Automatic Computer). The von Neumann Computer Architecture [God93a] has become the foundation of modern computer architecture. The von Neumann Computer Architecture consists of a Central Control [Unit], a Central

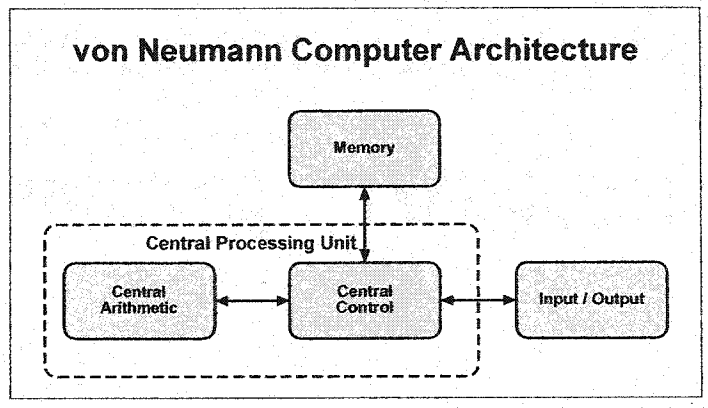


Figure 2.11: von Neumann Computer Architecture

Arithmetic [Unit], a Memory, and Input / Output [Devices] as shown in Figure 2.11 [God93b]. The Central Control [Unit] and the Central Arithmetic [Unit] combine to form a CPU (Central Processing Unit). The von Neumann Computer Architecture is based on the stored program concept. A program is a sequence of instructions to be executed. Prior to execution, the Memory is loaded with a program. Upon execution, the Central Control [Unit] reads an instruction in Memory, decodes the instruction, and performs the tasks associated with each instruction. Although simple in concept and design, the von Neumann Computer Architecture provides the flexibility necessary to build a general-purpose computer capable of performing a complex sequence of computations.

2.2.2 Personal Computer Architecture

Personal computers based on the Intel 80x86 Processor Family have dominated the computer marketplace for the past two decades. Personal computer architecture has evolved from an architecture quite similar to the von Neumann Computer Architecture into the complex architecture shown in Figure 2.12. Enhancements and technological innovations have been incorporated into the architecture while maintaining backward compatibility with legacy devices.

One key difference between the von Neumann Computer Architecture and the Intel 80x86 Processor Family Personal Computer Architecture is that the CPU does not connect directly with all other functional units and devices. A hierarchy of busses and bridges allow the interconnection

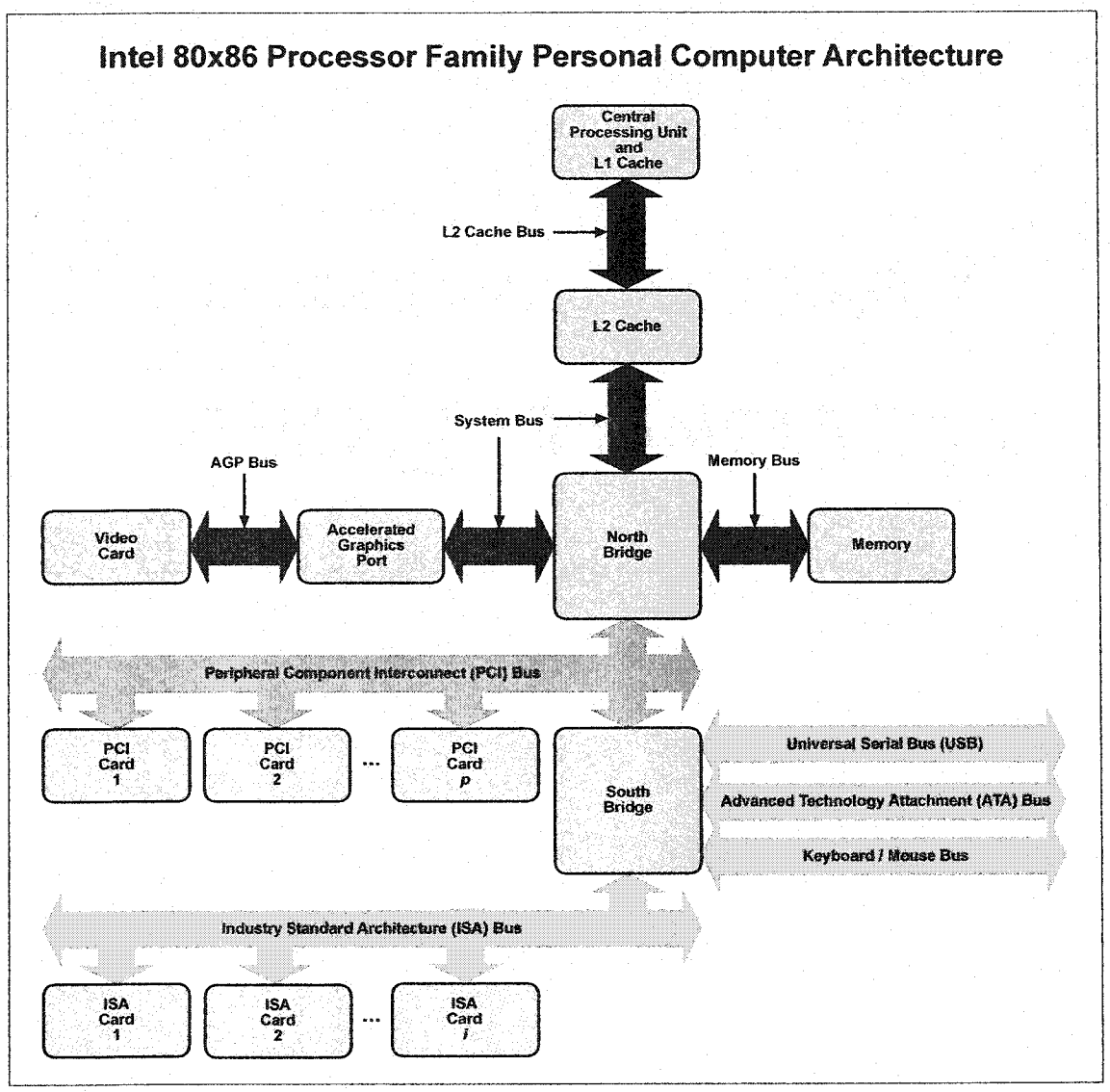


Figure 2.12: Intel 80x86 Processor Family Personal Computer Architecture

of both high-speed devices and legacy devices. This hierarchy provides more opportunities for the CPU to work on one task while devices work independently on other tasks. This architectural feature is an example of low-level parallelism. The CPU operates on a single instruction stream but the completion of an instruction may cause a device to start performing a task while the CPU continues executing its instruction stream.

This noticeable change in computer architecture results from a desire for high system performance while maintaining backwards compatibility with legacy devices. The bus hierarchy allows slower, legacy devices to connect to a high-speed processor without significantly reducing the performance of the CPU on other tasks. The CPU, the busses, and the devices often operate at different clock frequencies. The CPU may eventually be forced to wait for a slow device to complete an operation. However, if the CPU is capable of executing instructions while waiting for a slow device to respond to a request, overall system performance improves.

Clearly, the introduction of independence between functional units and devices within a computer blurs the line between sequential and parallel processing. Personal computers are capable of computing several tasks in parallel. However, this low-level parallelism does not change the sequential nature of the instruction stream. All of the low-level parallelism may be hidden from the end user[2]. For this reason, modern computer architectures are considered to be examples of sequential computers. The term parallel computer is commonly associated with computing machines that expose the existence of two or more processing units.

Performance Limitations in Personal Computers

There are limits to the performance benefits that can be achieved by adding more devices and busses to a computer architecture. A bus hierarchy introduces synchronization points. When two busses wish to share information, synchronization between the busses is necessary. Bridges attempt to alleviate synchronization bottlenecks by scheduling bus transactions and buffering responses to reduce the impact of synchronization. Bridges effectively tradeoff bus latency for increased throughput. By using the time spent waiting for a slower bus to respond to a request effectively, it is possible to improve overall system performance.

[2]Operating systems often expose some of the parallelism through the use of kernel functions and application libraries to allow programmers to exploit the hardware more aggressively.

Comparison of Bus Throughput

Name	Maximum Data Width (bits)	Peak Frequency (MHz)	Peak Throughput (Mbits / s)	Performance Comparison Relative to ISA Bus
System Bus	32	533	17056	107
AGP Bus (4X)	128	66	8448	53
PCI Bus	64	66	4224	26
Firewire Bus	1	3200	3200	20
ATA/100 Bus	32	25	800	5
SCSI-3 Bus	16	40	640	4
USB 2.0 Bus	1	480	480	3
ISA Bus	16	10	160	1

Table 2.2: Comparison of Bus Throughput

As devices and busses are added, the control logic increases in complexity. This additional complexity can restrict performance. Personal computer architectures have traditionally sacrificed performance for backwards compatibility and flexibility. For example, support for the 16-bit ISA (Industry Standard Architecture) Bus still exists in many personal computers. The throughput of the ISA Bus is approximately three orders of magnitude less than that of a modern CPU. The issue rate for a CPU can be two 32-bit instructions per clock cycle at a clock rate of 2 GHz. This is effectively a rate of 32-bits per 250 ps. Legacy ISA Cards operate on 16-bit instructions per bus cycle at a bus rate of 8 MHz[3]. This is effectively a rate of 32-bits per 250 ns. Therefore, a modern CPU issues instructions approximately 1000 times faster than the ISA Bus. This throughput difference can result in significant delays when synchronization between two busses occurs. The bridges connecting busses in the bus hierarchy attempt to minimize these delays by buffering and reordering bus transactions. However, these delays can never be eliminated. For example, a PCI bus transaction always impacts the system bus. A comparison of the throughput of busses is provided in Table 2.2 and a comparison of the throughput of devices is provided in Table 2.3.

[3]Strictly speaking, the clock frequency of an ISA card can range from 6 MHz to 10 MHz, depending upon the CPU clock frequency

Comparison of Device Throughput

Name	Maximum Data Width (bits)	Peak Frequency (MHz)	Peak Throughput (Mbits / s)	Performance Comparison Relative to CD-ROM
Pentium IV CPU	64	2600	166400	2667
Ethernet	1	1000	1000	16
Hard Disk	1	352	352	6
CD-ROM (52X)	8	8	62	1

Table 2.3: Comparison of Device Throughput

2.3 Configurable Computer Architectures

Configurable computers use configurable hardware structures to provide a level of hardware flexibility not found in ordinary computer architectures. Configurable computer architectures typically consist of a fixed, general-purpose hardware component and a configurable, application-specific hardware component. The general-purpose hardware component may be a processor core or a complete general-purpose computer system. The application-specific hardware component is typically one or more HDPLDs configured for a particular task. The two components may be coupled at the instruction level, the system bus level, or the peripheral bus level as shown in Figure 2.13. A computer that uses configurable hardware at the instruction-level is referred to as a *reconfigurable processor unit.* A computer that couples configurable hardware with a CPU at the system bus level is referred to as a *tightly-coupled configurable computer.* A computer that couples programmable hardware with a CPU at the peripheral bus level is referred to as a *loosely-coupled configurable computer.*

Loosely-coupled and tightly-coupled configurable computer architectures are prevalent. These coupled configurable computer architectures can be built by integrating off-the-shelf components. The configurable hardware is often referred to as a configurable coprocessor. A configurable coprocessor may be treated as just another device in the system. Alternatively, a configurable coprocessor may be treated as another, arguably less powerful, processor. Regardless, the introduction of a configurable coprocessor presents opportunities for increased parallel computation and increased throughput.

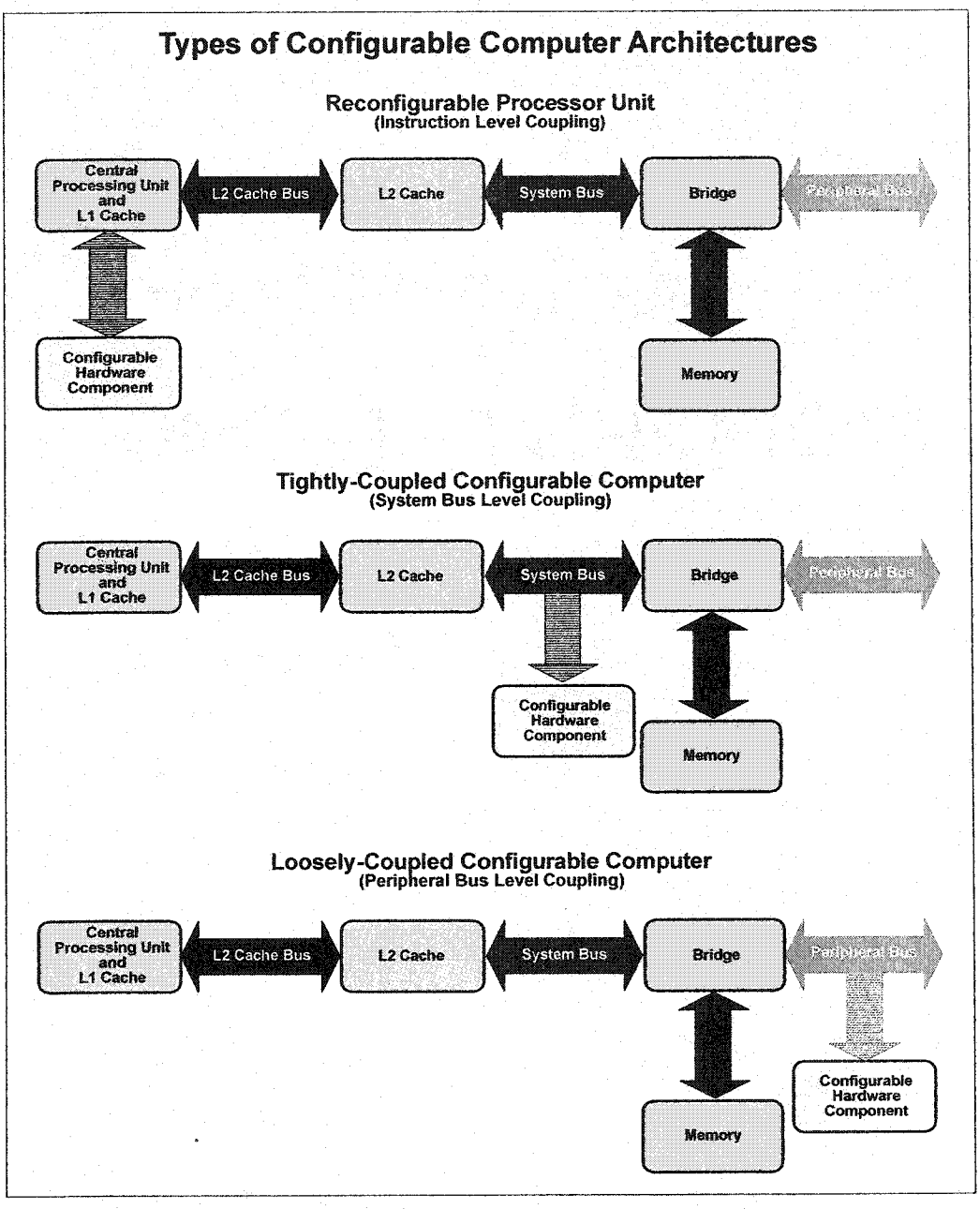


Figure 2.13: Types of Configurable Computer Architectures

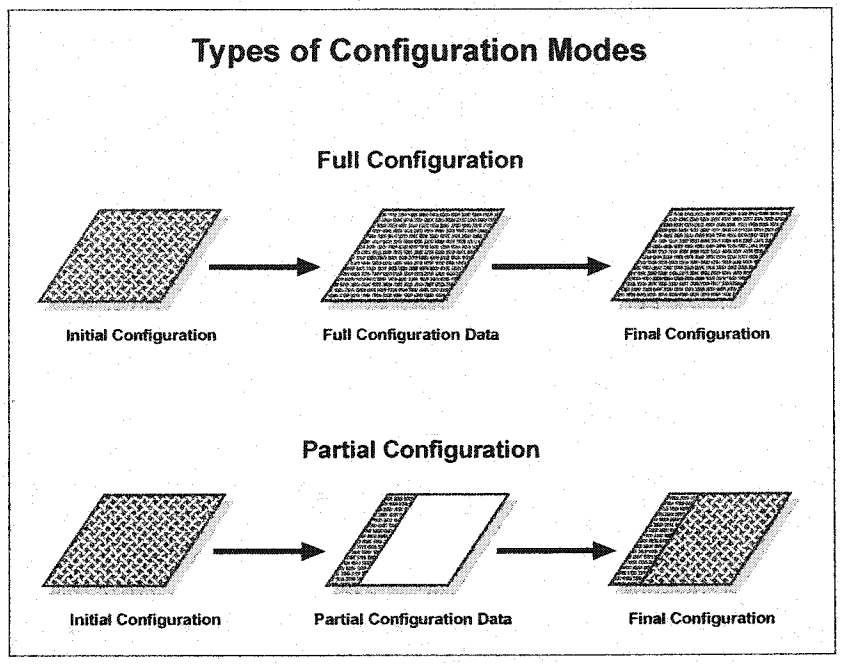


Figure 2.14: Types of Configuration Modes

2.3.1 Types of Configurable Computers

It is possible to classify configurable computers using configuration modes and configuration contexts. A configuration mode refers to the method used to store a new or modified design into the configurable hardware. A configuration context refers to the storage of one or more active designs within the configurable hardware.

Two types of configuration modes exist. Configurable devices may be either fully-configured or partially configured. Full configuration requires that the entire configurable device be configured at one time. Partial configuration permits a portion of a configurable device to be configured without the need to configure the entire device. Partial configuration requires much more sophisticated development tools and devices. The two configuration modes are illustrated in Figure 2.14.

Three types of configuration contexts exist. Configurable devices may be classified as single context, multiple context, or pipeline context. Single context devices can only hold a single design that is either in a state of configuration or active use. Multiple context and pipeline context devices support an active design as well as several inactive designs, one of which may be in a state of

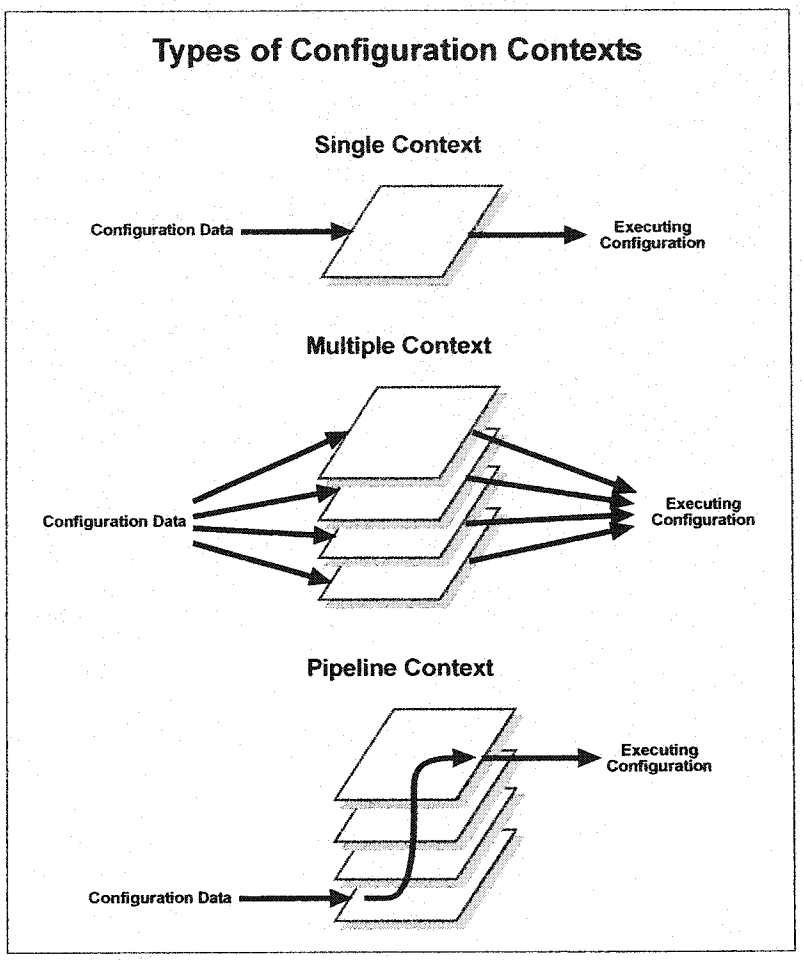


Figure 2.15: Types of Configuration Contexts

configuration. In a multiple context device, any inactive design can be switched with the active design. In a pipeline context device, the first design in the pipeline is always the active design. The three configuration contexts are illustrated in Figure 2.15.

Using the configuration modes and contexts to classify configurable hardware devices, it is possible to develop a taxonomy for configurable computers. This taxonomy is shown in Table 2.4. It should be noted that a partially configurable, pipelined device does not yet exist. Such a device would be very complex and it would require an equally complex toolset to support the design and configuration of the device. However, it is theoretically possible that such a device will be

Configurable Computer Taxonomy

		Configuration Mode	
		Full	Partial
Configuration Context	Single	SF	SP
	Multiple	MF	MP
	Pipeline	PF	PP

Table 2.4: Configurable Computer Taxonomy

introduced as toolsets improve.

2.4 Configurable Computing Platforms

The variety of programmable logic devices available has led to a variety of approaches to configurable computing. There are three approaches to configurable computing that have become popular. The first approach is to use a programmable logic device to implement one or more simple processors within a computer system. The second approach is to build an entire computer system out of programmable logic devices. The last, and by far most popular, approach is to build a programmable logic board and then use it as a custom hardware-based accelerator.

The use of programmable logic devices as hardware-based accelerators has resulted in the development of a number of custom computing machines. Table 2.5 categorizes some interesting configurable computing machines. The majority of the machines listed are examples of programmable logic boards suitable for coupled configurable computing.

2.4.1 State-of-the-Art Configurable Computing Machines

State-of-the-art configurable computing machines share the following characteristics:

1. support for multiple programmable devices (FPGAs, CPLDs, or RPUs),
2. high usable gate capacity,
3. high-speed bus interface (e.g., SBus, PCI, or custom),

Configurable Computing Systems / Boards / Devices	Year	Citation	Approach	Maximum Device Capacity	Configuration		Purpose
					Context	Mode	
APS X-84	1996	[Ass96]	ISA Board	1-XC4010	Single	Full	Education
Anyboard	1993	[VMT+92]	ISA Board	5-XC3042	Single	Full	Research
ARC-PCI	1997	[Alt97a]	PCI Board	3-EPF10K50	Single	Full	Research
BORG	1992	[Cha94]	ISA Board	2-XC3030 + 2-XC3042	Single	Full	Education
BORG-II	1994	[Cha94]	ISA Board	2-XC4003 + 2-XC4002	Single	Full	Education
Chameleon	1992	[HF92]	System	7-CAL1024	Single	Full	Research
CHAMP	1994	[Box94]	VME Board	16-XC4013	Single	Full	Research
CM-2X	1993	[CR93]	System	16-XC4005	Single	Full	Research
EVCI	1995	[CTS95]	SBus Board	1-XC4013	Single	Full	Commercial
Nios Development Board	1999	[Cor02c]	System	1-EPF20K200E	Single	Full	Commercial
Ganglion	1992	[CB92]	VME Board	24-XC3090	Single	Full	Research
H.O.T. I	1996	[CTS96]	PCI Board	1 XC6216 + 1 XC-4013	Single	Partial	Commercial
H.O.T. II	1998	[VCC98]	PCI Board	1 XCS40-4 + 1 XC95108-15	Single	Partial	Commercial
H.O.T. II-XL	1998	[VCC98]	PCI Board	1 XC4062XLT-1 + 1 XC95108-15	Single	Partial	Commercial
MORPH-ISA	1992		ISA Board	6-XC4025	Single	Full	Research
Nano Processor	1994	[WHG94]	ISA Board	2-XC3090	Single	Full	Commercial
PCI Pamette V1	1996	[VBR+96]	PCI Board	5-XC4013	Single	Full	Research
PeRLe-0	1989	[BRV89]	VME Board	25-XC3020	Single	Full	Research
PeRLe-1	1993		DEC Board	24-XC3090	Single	Full	Research
PRISM	1991	[WAL+93]	System	4-XC3090	Single	Full	Research
PRISM-II	1993	[WAL+93]	System	60-XC4010	Single	Full	Research
P-Series Virtual Computer	1993	[Cas93]	System	52-XC4013	Single	Full	Commercial
Rasa	1993	[TAS93]	ISA Board	3-XC4010	Single	Full	Research
RIPP-10	1994	[Alt94]	ISA Board	8-EPF81188 + 1-EPF8452	Single	Full	Research
RPM	1995	[DGJ+95]	SCSI Device	63-XC4013	Single	Full	Research
SPACE	1993	[MCMB93]	System	16-CAL1024	Single	Full	Research
SPACE 2	1996	[Mil96]	System	8-XC6264	Single	Partial	Research
Spectrum	1996	[Cor96]	System	32-XC4013 + 2-XC4010	Single	Full	Commercial
Splash	1989	[GHK+91]	VME Board	32-XC3090	Single	Full	Research
Splash 2	1992	[ABD92]	SBus Board	16-XC4010	Single	Full	Research
Spyder	1993	[IS93]	VME Board	5-XC4003 + 2-A1280	Single	Full	Research
Stratix Development Board	2003	[Mic02]	System	1-EPS125	Single	Full	Commercial
Teramac	1995	[ACC+95]	SCSI Device	108-PLASMAs	Single	Full	Research
Transmogrifier-1	1994	[GKC+94]	System	4-XC4010	Single	Full	Research
Transmogrifier-2	1997	[LGv+97]	System	32-EPF10K50	Single	Full	Research
UP 1 Education Board	1997	[Alt97b]	System	1-EPF10K20 + 1-EPM7128S	Single	Full	Education
Virtex-II Pro Prototype Platform	2002	[Xil02]	System	1-XC2VP50	Single	Full	Commercial
XS40	1997	[X E97]	System	1-XC4010	Single	Full	Education
XS95	1997	[Cor97]	System	1-XC95108	Single	Full	Education

Table 2.5: Configurable Computing Systems, Boards, and Devices

4. partial configurability, and

5. availability of tools to assist with development and debugging.

Some configurable computers that have these characteristics include VCC's H.O.T. (Hardware Object Technology) II Development System [VCC98], Altera's ARC-PCI Board [Alt97a], the University of Toronto's Transmogrifier II [LGv+97] and the Giga Operations Spectrum System [Cor96]. All of these development systems are well-suited for advanced research into configurable computing.

2.5 Benefits of Configurable Computing

Configurable computers have the rather unique property that they provide a means of exploiting fine-grain parallelism without sacrificing the ability to support other forms of parallelism. Bit-wise parallel algorithms and other algorithms that exhibit fine-grain parallelism are ideally suited for implementation in programmable logic devices. However, programmable logic devices are equally adept at exploiting medium or large-grain parallelism. Configurable computers can adapt to the problem at hand.

A programmable logic device can be programmed to operate like a SIMD (Single-Instruction, Multiple Data) parallel computer, or a MIMD (Multiple-Instruction, Multiple Data) parallel computer. With the advent of dynamically programmable logic, it is even possible to interchange from a SIMD architecture to a MIMD architecture in real-time. This is unlike any SIMD/MIMD hybrid architecture ever proposed [Dun90].

Configurable computers also present an opportunity for a significant savings in terms of cost. It is possible to envision a single configurable computing device replacing a set of dedicated hardware components. Similarly, common software tasks can be accelerated using a single programmable logic device. Software tasks that are used infrequently by the general population might be suitable for coprocessing using configurable computing techniques.

It is unknown whether the full benefits of configurable computing will ever be realized. While configurable computing has the potential revolutionize computing, the challenges of configurable computing are complex. These challenges must be addressed for configurable computing to become

a viable technique.

2.6 Challenges of Configurable Computing

Configurable computing is a new paradigm for computing. There are a number of challenges faced by designers of configurable computer architectures and applications. The challenges are a direct result of the parallelism inherent in hardware.

The challenges of configurable computing are similar to those faced by designers of parallel computing systems. The reason for this is simple. Systems of digital logic are inherently parallel. Unless a designer uses a mechanism, such as a finite state machine, to force a sequencing of operations, all hardware operations occur in parallel. Because of this fact, the most significant challenge of configurable computing is the effective exploitation of the parallelism of the configurable hardware. The translation of a sequential algorithm to a concurrent algorithm is a very difficult task.

Designs of configurable hardware components must tackle some difficult issues related to synchronization, communication, hardware limitations, and software interfacing. These hardware / software codesign issues have been resolved by hardware designers with some success [KS02] [BL97] [KL95] [BTA93] for certain applications. These issues still remain a significant design challenge for mainstream computing. While CAD (Computer Aided Design) tools may eventually address hardware limitations effectively [RH97], synchronization and communication will always be important issues to consider. As long as HDLs (Hardware Description Languages) permit the expression of parallelism, synchronization and communication will be issues for designers to resolve.

Dynamically configurable computing presents an additional challenge to designers in the form of time partitioning. Not only must configurable hardware components be partitioned to fit in the available space in the programmable logic devices, the components must also be partitioned with respect to time. In a dynamically configurable computer, hardware components may be swapped in and out of the programmable logic devices when deemed appropriate. This process is not unlike the operation of virtual memory. However, the problem is much more difficult since swapping of a hardware component physically changes the computer itself. The problem of time partitioning

is a popular area of configurable computing research [BAM96] [Deh96] [PB99].

All forms of computing pose challenges. While the complications associated with configurable computing are significant, it is possible that the benefits of this new computing paradigm outweigh the costs in certain situations. To predict when this may occur, an accurate model of configurable computing is necessary.

Chapter 3

Models of Configurable Computing

To analyze the performance of a coupled configurable computer, it is useful to model the system as a heterogeneous multiprocessor using techniques applicable to parallel and distributed systems. The increased opportunities for parallel computation should be taken into account when evaluating the performance of a configurable computer. These new opportunities for parallelism permit application-specific configurable hardware to deliver additional processing resources to the system. By permitting a host computer to focus on other operations, configurable hardware can enhance system performance, even if the configurable hardware is fundamentally slower than the computer that hosts it.

3.1 Introduction to Configurable Computer Models

A configurable coprocessor is an independent processing unit within a computer system. Due to their reliance on SRAM technology, configurable coprocessors execute at slow clock frequencies compared to modern processors. However, configurable coprocessors provide application-specific circuitry that often allows them to outperform a general-purpose processor. This performance benefit is particularly true for niche applications.

It is important to understand that it is possible to accelerate a processor by adding another processor, regardless of the relative speeds of the two processors if synchronization is not an issue. When a slow processor is added to a system with a fast processor, more tasks can be performed provided that synchronization between the processors is not a bottleneck. As long as the processors can compute in parallel, speedup is possible. Using this reasoning, it is possible for a configurable coprocessor to accelerate a host computer system with a fast processor. To investigate the realities of such an approach, a detailed model of a coupled configurable computer system is needed.

A model of configurable coprocessing has already been proposed [CMQ02]. However, this model does not explicitly address the impact of memory utilization delays, bus utilization delays, and operating system behaviour delays. Therefore, a more detailed model is necessary to analyze these aspects of the performance of a coupled configurable computer system.

3.2 The Transaction Pair Model

It is possible to model a program executing on a loosely-coupled or tightly-coupled configurable computer system as a sequence of transactions between a host computer and a configurable coprocessor. In the context of a coupled configurable computing system, transactions represent atomic, coprocessor operations. The use of a transaction model is common in the field of parallel and distributed systems, particularly for systems that require high reliability [Jal94] [Tan95].

Transactions require synchronization between the processor and the coprocessor twice during every transaction. The processor first communicates parameters to the coprocessor. After computing results, the coprocessor then communicates the results back to the processor. Both of these communications require synchronization between the processor and the coprocessor. The transaction concept [Gra81] may be enhanced by developing a model that focuses on the synchronization between the processor and the coprocessor. The advantage of such a model is that it provides a mechanism to support long transactions without introducing a large performance bottleneck. Rather than model a program as a sequence of transactions, a sequence of transaction pairs is used. The transaction pairs represent synchronization points between the processor and the coprocessor. This model, referred to as the *transaction pair model* for the purpose of this thesis,

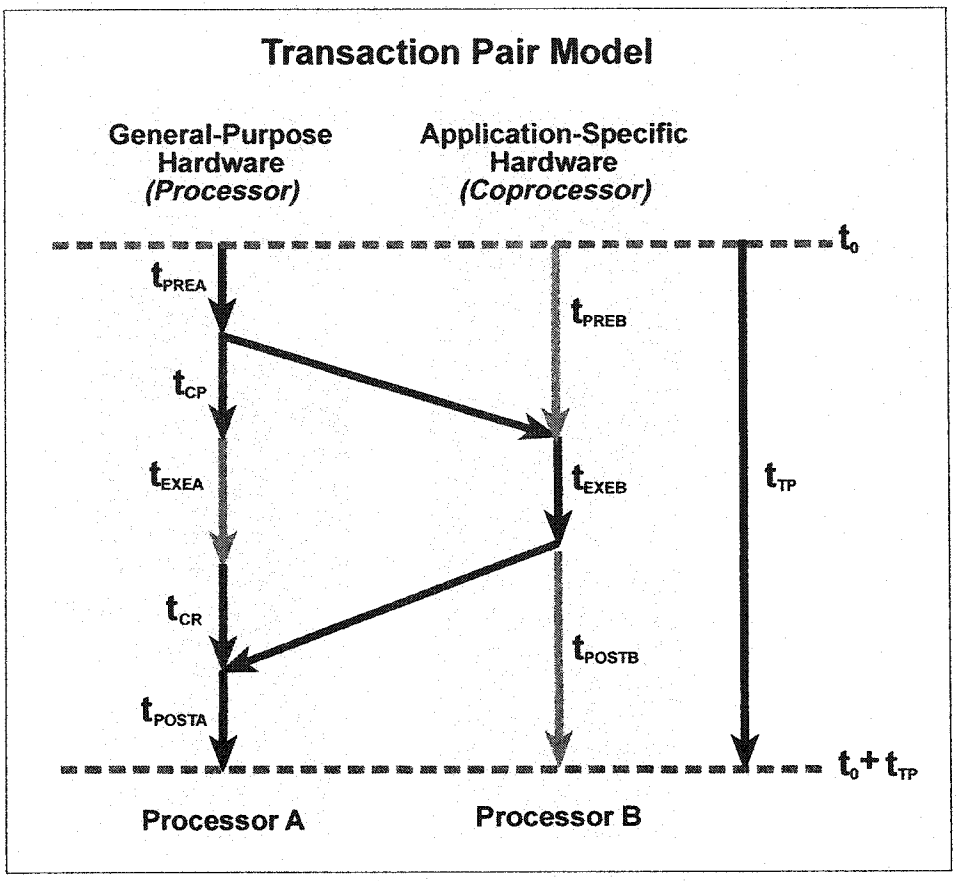


Figure 3.1: Transaction Pair Model

accounts for pre-processing, communication of parameters, parallel execution, communication of results, and post-processing.

Figure 3.1 illustrates the transaction pair model. It should be noted that Figure 3.1 is just one possible view of a transaction pair. The processor is assumed to be the master of the transaction pair and the coprocessor is assumed to be the slave. Some operations may require more processing time from one of the two processing units. If the processor requires more processing time, the coprocessor may only utilize a portion of t_{PREB} and t_{POSTB}. If the coprocessor requires more processing time, the processor might only be able to utilize a portion of t_{EXEA}.

Table 3.1 defines the timing parameters used by the transaction pair model. The processor is referred to as Processor A and the configurable coprocessor is referred to as Processor B. For all

timing parameters, the subscripts A and B have been used to denote timing parameters specific to each processor.

Timing Parameter	Definition
t_{PREA}	This is the time required by Processor A to produce parameters for the transaction to Processor B.
t_{CP}	This is the time required by Processor A to send parameters for the transaction to Processor B.
t_{EXEA}	This is the time available to Processor A to execute instructions while waiting for a response from Processor B.
t_{CR}	This is the time required by Processor A to receive return values for the transaction from Processor B.
t_{POSTA}	This is the time required by Processor A to consume return values for the transaction from Processor B.
t_{PREB}	This is the time available to Processor B to execute instructions while waiting for the start of a new transaction from Processor A.
t_{EXEB}	This is the time required by Processor B to execute a transaction initiated by Processor A.
t_{POSTB}	This is the time available to Processor B to execute instructions while waiting for the conclusion of the current transaction from Processor B.
t_{TP}	This is the total time required for the execution of a transaction pair. This is simply the sum of t_{PREA}, t_{CP}, t_{EXEA}, t_{CR}, and t_{POSTA}.
t_0	This is the start time of the transaction pair.
$t_0 + t_{TP}$	This is the end time of the transaction pair.

Table 3.1: Transaction Pair Model Timing Parameters

The transaction pair model provides a starting point for further analysis of the performance of coupled configurable computer systems. Each transaction pair represents the execution of a single operation on a configurable coprocessor. An operation may be very simple (e.g., a single instruction) or very complex (e.g., a complete algorithm). It is possible to model any application as a sequence of transaction pairs. Therefore, the transaction pair model can be applied to any application program.

For operations that do not require the communication of parameters from the processor to the configurable coprocessor, the transaction pair model reduces to the model shown in Figure 3.2. The first communication step and pre-processing are not required. In practice, this situation is unlikely to occur since synchronization is often required at the start of a coprocessor operation. However, it is important to show that this situation can be handled by this model.

For operations that do not require the communication of return values from the configurable

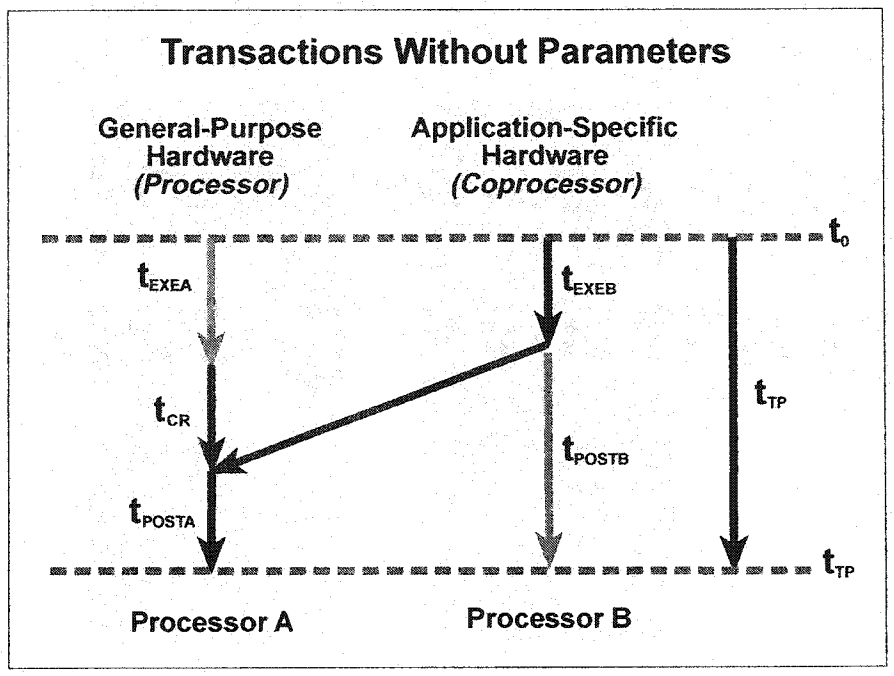


Figure 3.2: Transactions Without Parameters

coprocessor to the processor, the transaction pair model reduces to the model shown in Figure 3.3. The second communication step and post-processing are not required. In practice, this situation is unlikely to occur since synchronization is often required at the conclusion of a coprocessor operation. However, it is important to show that this situation can be handled by this model.

For some operations, it may be necessary to initiate several data transfers between the processor and the coprocessor. It should be noted that the transaction pair model supports the communication of large amounts of data. The timing parameter, t_{CP}, can represent the time required to communicate a single parameter or multiple parameters. Similarly, the timing parameter, t_{CR}, can represent the time required to communicate a single return value or multiple return values.

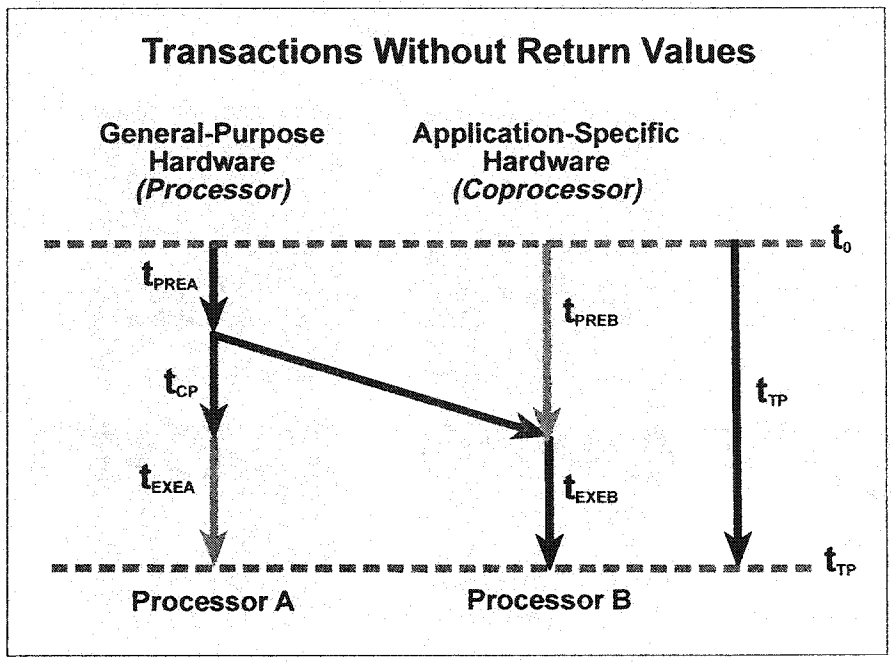


Figure 3.3: Transactions Without Return Values

3.2.1 Comparison of Transaction Performance

The transaction pair model can be used to compare the performance of a multiprocessor system with the performance of a configurable coprocessor system. Using the transaction pair model, an application is simply a sequence of transaction pairs. Assume that an application exists that requires t_{APP} to process all transaction pairs on a computer with a single processor. Since the transaction pairs execute on a single processor, no time is required to communicate transaction parameters and return values. Hence, the total execution time is t_{APP}.

Now, consider a multiprocessor system with two processors identical to the processor in the single processor computer. If the computations associated with the transaction pairs are distributed to the processors equally, the execution time of the application becomes $\frac{t_{APP}}{2}$, neglecting the time required to communicate transaction parameters and return values. Application and system speedup are limited to a factor of 2, neglecting secondary effects such as changes in cache coherency and operating system behaviour.

Next, consider a configurable coprocessor system that consists of a single processor and a

configurable coprocessor. Assume that the processor is identical to the processor in the single processor computer. Further assume that the coprocessor is a factor of F_{EXE} times faster than the processor at executing operations associated with transactions. If the computation associated with the transaction pairs is distributed to the processor and the coprocessor on the basis of relative processing capability, the execution time of the application becomes $\frac{t_{APP}}{(F_{EXE}+1)}$, neglecting the time required to communicate transaction parameters and return values. Application and system speedup are limited to a factor of $F_{EXE}+1$, neglecting secondary effects such as changes in cache coherency and operating system behaviour.

For the purpose of this example, the time required to communicate transaction parameters and return values has been assumed to be negligible. The analysis presented may still be valid if this assumption is not true. If the times required to communicate transaction parameters and return values are approximately equal for both systems, the communication overhead reduces application performance by a fixed amount on both systems. Even if more time is required to communicate transaction parameters and return values by the configurable coprocessor system, it is unlikely that the communication overhead is sufficient to cause the multiprocessor system to outperform a fast, configurable coprocessor system.

Figure 3.4 illustrates the performance comparison between a multiprocessor system and a configurable coprocessor system. The configurable coprocessor system has the potential to outperform the multiprocessor system if X is greater than 1. If this is the case, it is possible for the coprocessor to process transactions faster than the processor.

This comparison only considers the relative processing power of the systems. The comparison is simply performed to illustrate the limitations of multiprocessor systems. The speedup associated with a multiprocessor system is limited to the number of processors in the system since the processors are identical.[1] This is not the case for a configurable coprocessor system. A fast, configurable coprocessor may speedup a system to a much greater degree.

[1]Strictly speaking, the speedup can exceed the number of processors if the influence of secondary effects is positive. For example, additional cache memory in a multiprocessor system may result in better cache coherency and thus, better system performance.

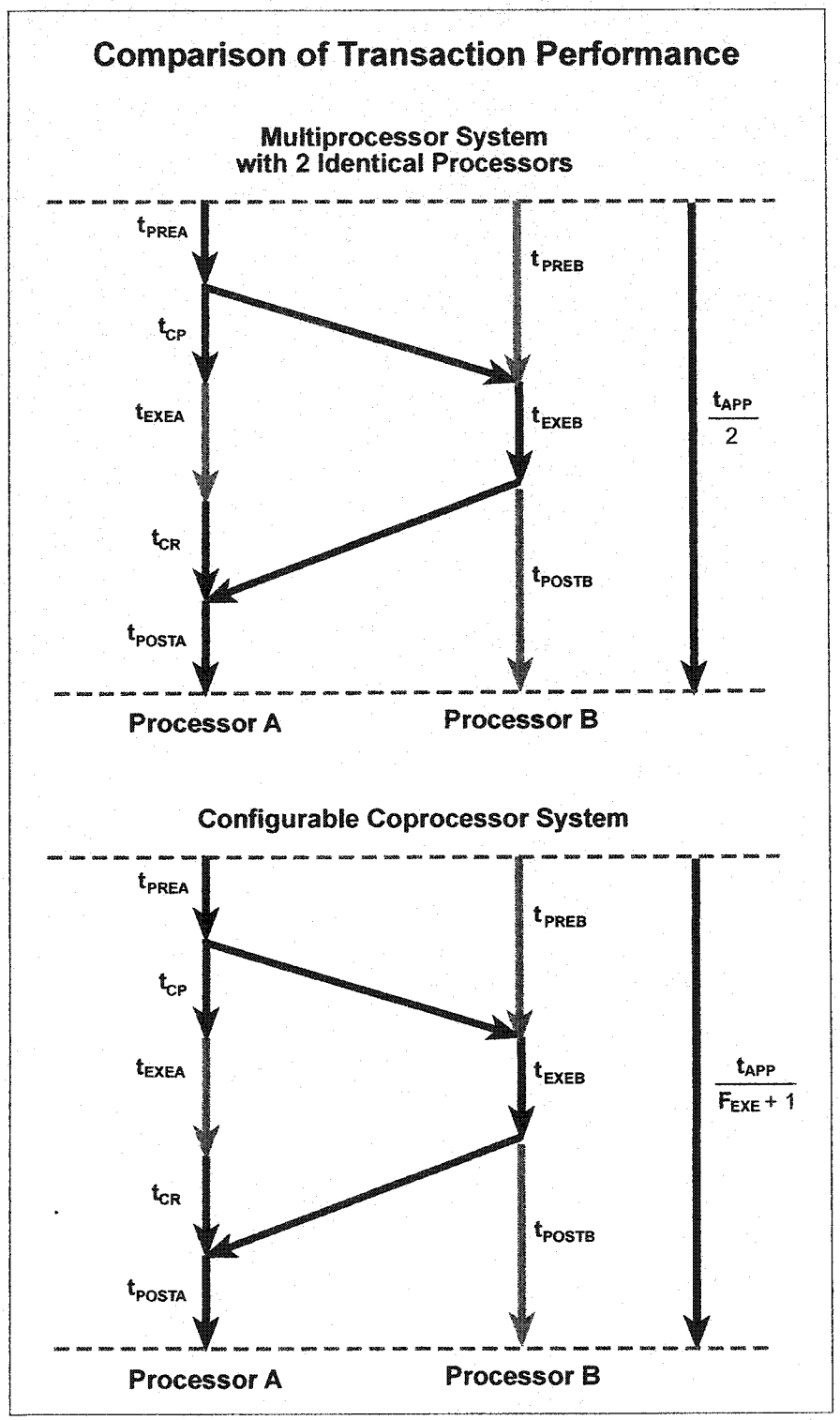


Figure 3.4: Comparison of Transaction Performance

3.2.2 Summary of the Transaction Pair Model

A transaction pair provides a model of a single operation. A transaction pair sequence provides a model of an entire application. However, extensions to the transaction pair model are necessary to model the performance of an entire configurable computer system. A change in the behaviour of an application changes the behaviour of the system and other applications executing on the system. These secondary effects must be considered if an accurate model of an entire configurable computer system is desired.

3.3 The Configurable Computer System Performance Model

A performance model of a configurable computer system must account for changes in the behaviour of the system that result from the use of a configurable coprocessor by an application. The use of a configurable coprocessor introduces opportunities for parallel computation and additional delays associated with hardware use and configuration. Coprocessor use impacts memory utilization, bus utilization and operating system behaviour. The cumulative impact may be positive or negative. In fact, for a multi-tasking computer system, it is possible for a coprocessor to have a positive impact on the performance of an application and a negative impact on the performance of the system. The opposite can also be true. Thus, it is important to have a model capable of predicting the performance a coprocessed application and the impact of the coprocessed application on the entire system.

By extending the transaction pair model, it is possible to develop a performance model of a configurable computer system. This extended model is a function of the execution time of an application and the changes in the behaviour of the system. To simplify the model, system execution times are divided into a fixed time component and several time components that vary based on system changes. The fixed time component represents the execution time required in the ideal scenario. The variable time components represent the execution time variations due to changes in specific aspects of the behaviour of the system. This extended model can be used to investigate both application performance and system performance. For the purpose of this thesis, this extended model is referred to as the *performance model.*

The goal of the performance model is to predict when application and / or system speedups are achievable using configurable coprocessing. It is not meant to accurately predict the execution time of an application after coprocessing. It is simply meant to give an indication of whether speedup is possible or highly unlikely.

Several timing parameters are defined to model the constant time and variable time components of system execution time. For a non-coprocessed system, the timing parameters introduced in Table 3.2 are used to model the system. For a coprocessed system, the timing parameters introduced in Table 3.3 are used to model the system. For the purpose of this thesis, the non-coprocessed system is referred to as *System 1* and the coprocessed system is referred to as *System 2*. Accordingly, timing parameters specific to System 1 are denoted by the subscript 1 and timing parameters specific to System 2 are denoted by the subscript 2. Similarly, for System 2, timing parameters specific to Processor A are denoted by the subscript A and timing parameters specific to Processor B are denoted by the subscript B.

For the purpose of using the performance model, worst case times should be used for all timing parameters. Worst case estimates can be used to determine a conservative estimate of speedup. Best case estimates can be used to determine an optimistic estimate of speedup. There exists a possibility that the average and the best case values for these timing parameters may not be directly proportional to the worst case values. If the distribution of values is skewed by a large amount, the performance model may predict the incorrect outcome. For the purpose of this model, the probability of this scenario is assumed to be very small. This assumption is not bad since the accuracy of any model in the presence of large statistical variations is likely very poor.

Figure 3.5 compares system execution times in a non-coprocessed computer system to those in a coupled configurable computer system. This figure uses the timing parameters introduced in Table 3.2 and Table 3.3. It is clear from Figure 3.5 that the configurable computer system performs more tasks than a non-coprocessed computer system. Configuration delays, pre-processing, and post-processing are not performed in a non-coprocessed computer system. Despite this overhead, system speedup is possible due to the availability of additional processing resources provided by the coprocessor.

Timing Parameter	Definition
t_{MEM1}	This is the total difference between the typical time required by all applications executing in System 1 for memory accesses and the ideal time required by all applications executing in System 1 for memory accesses. The ideal time is the minimum time required to perform all memory accesses assuming ideal memory utilization. The actual time and ideal time differ due to changes in memory utilization. Some examples of these changes include additional memory accesses, reduced cache system performance, and increased virtual memory system paging.
t_{BUS1}	This is the total difference between the typical time required by all applications executing in System 1 for system bus transactions and the ideal time required by all applications executing in System 1 for system bus transactions. The ideal time is the minimum time required to perform all system bus transactions assuming ideal bus utilization. The actual time and ideal time differ due to changes in bus utilization. Some examples of these changes include additional bus transactions, increased bus latency, and reduced bus availability.
t_{OS1}	This is the total difference between the typical time required by all applications executing in System 1 for operating system kernel operations and the ideal time required by all applications executing in System 1 for operating system kernel operations. The ideal time is the minimum time required to perform all operating system kernel operations assuming ideal operating system behaviour. The actual time and ideal time differ due to changes in operating system behaviour. Some examples of these changes include additional context switching, additional interrupt servicing, and additional kernel operations.
t_{EXE1}	This is the total processing time required by all applications executing in System 1 neglecting time included in t_{MEM1}, t_{BUS1}, and t_{OS1}.
t_{SYS1}	This is the total system execution time required by all applications executing in System 1. This is simply the sum of t_{MEM1}, t_{BUS1}, t_{OS1}, and t_{EXE1}.

Table 3.2: Performance Model Timing Parameters for System 1

Timing Parameter	Definition
t_{CFGA2}	This is the total configuration time delay experienced by Processor A as a result of all configurations of Processor B in System 2.
t_{MEM2}	This is the total difference between the typical time required by all applications executing in System 2 for memory accesses and the ideal time required by all applications executing in System 1 for memory accesses. The ideal time is the minimum time required to perform all memory accesses assuming ideal memory utilization. The actual time and ideal time differ due to changes to the memory system to support Processor B.
t_{BUS2}	This is the total difference between the typical time required by all applications executing in System 2 for system bus transactions and the ideal time required by all applications executing in System 1 for system bus transactions. The ideal time is the minimum time required to perform all system bus transactions assuming ideal bus utilization. The actual time and ideal time differ due to changes to the bus system to support Processor B.
t_{OS2}	This is the total difference between the typical time required by all applications executing in System 2 for operating system kernel operations and the ideal time required by all applications executing in System 1 for operating system kernel operations. The ideal time is the minimum time required to perform all operating system kernel operations assuming ideal operating system behaviour. The actual time and ideal time differ due to changes in operating system behaviour to support Processor B.
t_{PREA2}	This is the total pre-processing time of all applications executing on Processor A in System 2.
t_{EXEA2}	This is the total processing time of all applications executing on Processor A in System 2 neglecting pre-processing (t_{PREA2}) and post-processing (t_{POSTA2}) as well as neglecting t_{CFGA2}, t_{MEM2}, t_{BUS2}, and t_{OS2}. Unlike t_{EXEA} in the transaction pair model discussed previously, this time is the actual processing time rather than the time available for processing.
t_{POSTA2}	This is the total post-processing time of all applications executing on Processor A in System 2.
t_{PREB2}	This is the total pre-processing time of all applications executing on Processor B in System 2. Unlike t_{PREB} in the transaction pair model discussed previously, this time is the actual pre-processing time rather than the time available for pre-processing.
t_{EXEB2}	This is the total processing time of all applications executing on Processor B in System 2 neglecting pre-processing (t_{PREA2}) and post-processing (t_{POSTA2}).
t_{POSTB2}	This is the total post-processing time of all applications executing on Processor B in System 2. Unlike t_{POSTB} in the transaction pair model discussed previously, this time is the actual post-processing time rather than the time available for post-processing.
t_{CFGB2}	This is the total configuration time delay experienced by Processor B as a result of all configurations of Processor B in System 2.
t_{EXE2}	This is the total processing time required by all applications executing in System 2 neglecting time included in t_{MEM2}, t_{BUS2}, t_{OS2}, and t_{CFGA2}.
t_{SYS2}	This is the total system execution time required by all applications executing in System 1. This is simply the sum of t_{CFGA2}, t_{MEM2}, t_{BUS2}, t_{OS2}, and t_{EXE2}. It is assumed that Processor A waits for the completion of transactions and configuration by Processor B if necessary.

Table 3.3: Performance Model Timing Parameters for System 2

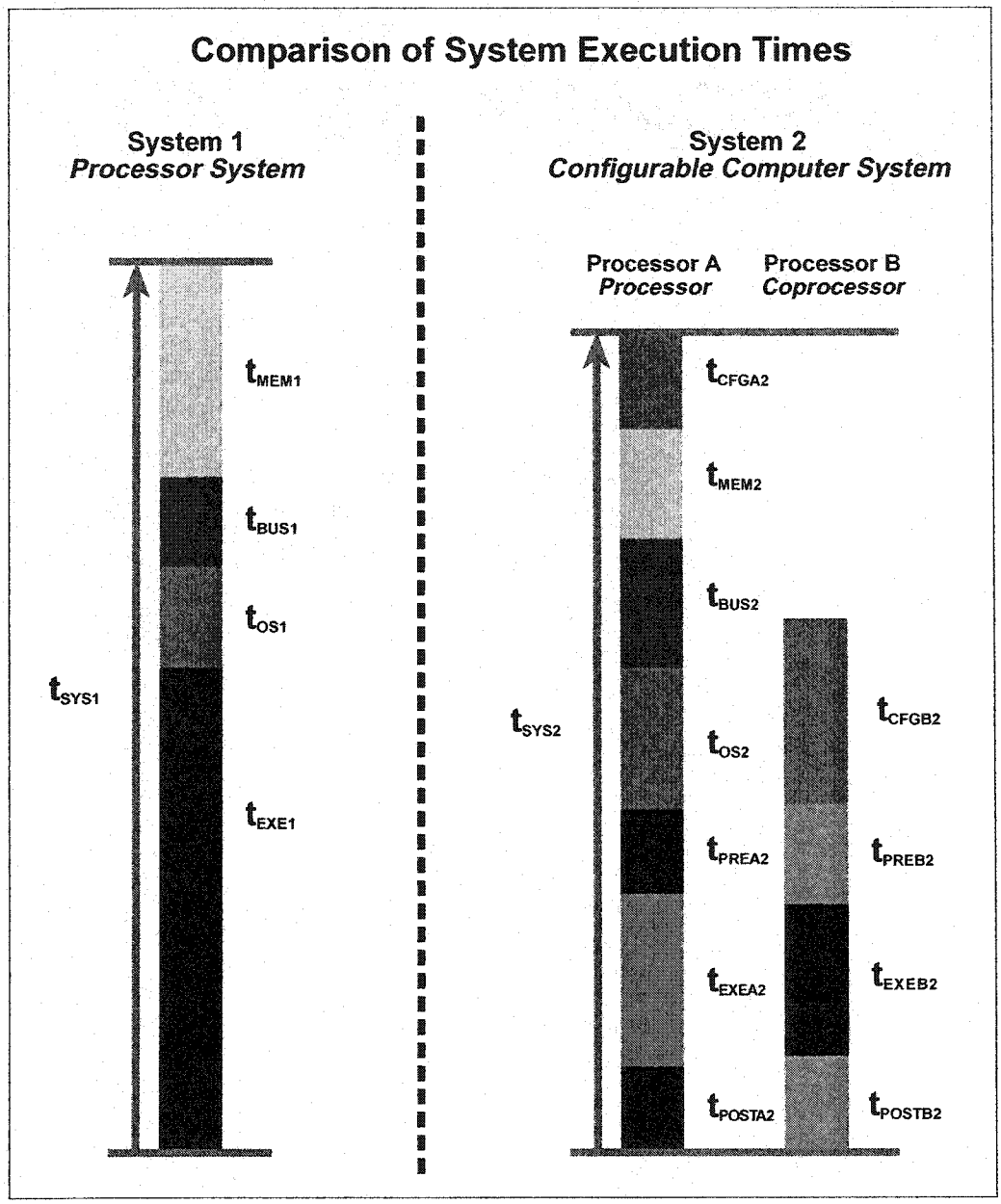


Figure 3.5: Comparisons of Execution Times

3.3.1 Configuration Delays

Configuration delays result from the time required to prepare a configurable coprocessor for processing. The configuration delays model the time required to configure the programmable logic devices within the system. These delays are specific to a configurable computer system. However, each delay impacts the processor and the coprocessor in different ways. For this reason, two configuration delay parameters exist in the performance model. The first delay, t_{CFGA2}, models the delay experienced by Processor A. The second delay, t_{CFGB2}, models the delay experienced by Processor B. The configuration delays do not take into account the time and effort required to develop, synthesize, and map hardware designs onto the configurable coprocessor. The time required to produce configurable coprocessors is a form of non-recurring delay. It does not impact the execution time of an application so it is beyond the scope of the performance model.

Configuration delays can be negligible or significant. If configuration can be performed without impeding the operation of the system, then the delay is negligible. This is true if configuration occurs frequently and the system contains a configurable logic device that supports multiple contexts or if configuration does not occur frequently. Using a configurable logic device that supports multiple contexts, an unused context can be configured while the active context performs tasks. Such a device allows fast context switching if the context is already loaded, but requires approximately the same amount of time as a single-context device if the context must be loaded. If the configuration process disrupts the operation of the system, the configuration delay can be significant.

The architecture of a configurable coprocessor has a tremendous impact upon the significance of configuration delays. The architecture can facilitate configuration in parallel with computation or it can hinder it. The architecture also defines the number of simultaneous coprocessor designs that may be in use as well as the time required to configure the hardware with a new design.

The operating system executing on the processor is in the best position to determine the coprocessor design necessary for pending computations. It may be desirable for the processor to initiate the configuration of the coprocessor hardware. To do this, a scheduler is required to determine the next hardware design to be configured and used. This scheduler is analogous to a process scheduler in a multitasking operating system. The key difference is that the scheduler configures a hardware design instead of swapping processes. The choice of scheduler has a signifi-

cant impact upon the probability of a configuration being available and thus, the performance of a configurable coprocessor system.

A detailed discussion of scheduling strategies is beyond the scope of this thesis. However, it is worthy of future research. It may be the case that process scheduling strategies work well for scheduling hardware configurations. Alternatively, it may be the case that a new scheduling strategy is required for efficient configurable computing. Regardless of the scheduling strategy, it is possible to model the performance of the system provided an estimate of the configuration delays can be determined and the frequency of configuration can be estimated.

3.3.2 Memory Utilization Delays

The time required for memory operations varies based on the state of the system. In a heavily loaded system, an application may have to spend more time on memory operations due to high memory utilization and poor cache performance. Memory utilization delays express the excess time required to perform memory operations over and above the ideal amount of time required for all memory operations in an ideal situation. In other words, memory utilization delays represent the variable portion of time devoted to memory operations. All non-ideal applications have non-zero memory utilization delays.

Although the total memory required by a configurable coprocessor system may actually be larger than a non-coprocessed system due to the additional state information that must be stored, there is usually more total memory to exploit in a configurable coprocessor system. Both the Processor A and Processor B typically have local memory available. The efficient distribution of memory accesses across multiple memory systems can improve the performance of the system. Cache performance on Processor A may improve or degrade based on changes in locality. Also, it may be possible to write applications that avoid unnecessary paging of the virtual memory system. Thus, changes in memory utilization can result in performance gains and losses.

It is difficult to estimate the magnitude of the impact of memory utilization delays on the performance of a system. The impact depends on the relative bandwidths of the memory systems in use. For example, Processor A's memory hierarchy is likely to be much faster than Processor B's memory hierarchy if there is a cache present in Processor A but not present in Processor

B. The impact of memory utilization changes, denoted as t_{MEM}, can be measured as shown in Equation 3.1. If t_{MEM} is positive, the impact of distributing memory is positive. If t_{MEM} is negative, the impact of distributing memory is negative.

$$t_{MEM} = t_{MEM1} - t_{MEM2} \tag{3.1}$$

There is not much that can be done to improve the impact of memory utilization. It is a characteristic of the application and the hardware/software partitioning. It may be possible to more effectively distribute memory accesses by repartitioning the application into hardware and software components. Further research on this issue is necessary to determine if this is practical.

3.3.3 Bus Utilization Delays

The time required for bus operations varies based on the state of the system. Bus operations are necessary to communicate with hardware devices, external interfaces, and main memory. In a heavily-loaded system, an application may have to spend more time on bus operations due to high bus utilization and poor cache performance. Bus utilization delays express the excess time required to perform bus operations at all levels of the bus hierarchy. This is the time over and above the amount of time required for all bus operations in an ideal situation. In other words, bus utilization delays represent the variable portion of time devoted to bus operations. All applications executing in a multitasking environment have non-zero bus utilization delays.

The use of a configurable coprocessor may require more or less extensive use of communication over the bus hierarchy. This need can cause the overall performance of the system to decrease or increase. Other bus transactions may be delayed or accelerated by the use of the coprocessor. One bus transaction at the top-level of the bus hierarchy may cause delays at other levels of the bus hierarchy. A single change may introduce a signficant amount of bus latency in the system. The impact of this change in bus latency must be considered by the model. Given the timing parameters in the performance model, it is possible to express the impact of this change as shown in Equation 3.2. t_{BUS} denotes the impact of changes in bus utilization within the system. A positive value indicates a positive impact.

$$t_{BUS} = t_{BUS1} - t_{BUS2} \tag{3.2}$$

Improving the impact of bus utilization is difficult. A positive impact can only occur if bus utilization is reduced by the partitioning of applications between Processor A and Processor B. A positive impact can be accomplished by selecting operations to coprocess that already use the bus hierarchy heavily. In other words, the use of an external hardware device is replaced with the use of a configurable coprocessor. Partitioning of the system is key to improving performance.

3.3.4 Operating System Behaviour Delays

The time required for kernel operations varies based on the state of the system. In a heavily loaded system, an application may have to spend more time on kernel operations due to increased context switching, resource blocking, and other operating system scheduling issues. Operating system behaviour delays express the excess time required to perform kernel operations over and above the amount of time required for all kernel operations in an ideal situation. In other words, operating system behaviour delays represent the variable portion of time devoted to kernel operations. All non-ideal applications executing in a multitasking environment have non-zero operating system behaviour delays.

The impact of processing scheduling and context switching on a system is extremely difficult to predict. However, it is clearly something to be considered. The probability of the operating system initiating a context switch increases with the use of any additional hardware device. Both MS Windows NT and Linux require a kernel call to access an external hardware device. The execution of this kernel call is likely to result in a context switch. For a non-coprocessed application that does not normally require a context switch, the addition of a configurable coprocessor results in context switches that are "unnecessary". Since context switches can take a significant amount of time, this change in operating system behaviour can have a severe negative impact upon performance. This impact can be quantified as shown in Equation 3.3.

$$t_{OS} = t_{OS1} - t_{OS2} \tag{3.3}$$

Communication with a configurable coprocessor is likely to result in a context switch by the operating system. It is common for an operating system to start performing other tasks after initiating communication with a hardware resource such as a configurable coprocessor. For this reason, configurable coprocessors should only be used for computations that normally require context switches during execution. Otherwise, the delay associated with context switching significantly limits the performance of the system. Furthermore, if it is assumed that context switches always occur during communication with a hardware resource, it is possible to provide an upper bound on the acceptable level of change in operating system behaviour.

3.3.5 Processing Times

Ideally, a coprocessed system is capable of processing operations quicker than a non-coprocessed system. The processing impact, t_{EXE}, can be quantified as shown in Equation 3.4. This impact depends upon the relative speeds of the processors and the complexity of the coprocessor hardware. Clock frequency is not the sole determining factor in processing times. Application execution time is a function of the clock frequency (or propagation delay in the case of asynchronous designs) as well as the number of cycles per instruction (or operations). While a configurable coprocessor may have a slower clock frequency, it may be capable of executing more complex instructions or many instructions at one time to provide greater processing capability and effectively reduce processing times.

$$t_{EXE} = t_{EXE1} - t_{EXE2} \tag{3.4}$$

Equation 3.4 uses absolute processing times to determine the impact of coprocessing. Given the time required to execute an application on System 1 (t_{EXE1}) and the time required to execute an application on System 2 (t_{EXE2}), the impact is simply the difference. This time difference represents the improvement (or degradation) in processing time that results from the addition of a configurable coprocessor. It can only be determined if the processing times on both systems are known or can be accurately estimated.

3.3.6 Evaluating the Net Performance Impact

Equation 3.5 expresses the system execution time for a non-coprocessed system (System 1). This time is a function of memory utilization delays (t_{MEM1}), bus utilization delays (t_{BUS1}), operating system behaviour delays (t_{OS2}), and processing times (t_{EXE1}). All of the quantities in this expression are strictly positive. The fixed portion of system execution time for this non-coprocessed system is included in t_{EXE1}.

$$t_{SYS1} = t_{MEM1} + t_{BUS1} + t_{OS1} + t_{EXE1} \tag{3.5}$$

Equation 3.6 expresses the Processor A component of the system execution time for a coprocessed system (System 2). This time is a function of configuration time (t_{CFGA2}), memory utilization delays (t_{MEM2}), bus utilization delays (t_{BUS2}), operating system behaviour delays (t_{OS2}), pre-processing time (t_{PREA2}), processing time (t_{EXEA2}), and post-processing time (t_{POSTA2}). It is important to note that t_{MEM2}, t_{BUS2}, and t_{OS2} may be negative quantities since they are expressed relative to System 1. The remaining quantities (t_{PREA2}, t_{EXEA2}, t_{POSTA2}, and t_{CFGA2} are strictly positive.

$$t_{A2} = t_{PREA2} + t_{EXEA2} + t_{POSTA2} + t_{OS2} + t_{BUS2} + t_{MEM2} + t_{CFGA2} \tag{3.6}$$

Equation 3.7 expresses the Processor B component of the system execution time for a coprocessed system (System 2). This time is a function of pre-processing time, execution time, post-processing time and configuration time. Memory utilization delays, bus utilization delays, and operating system behaviour delays are not included in the expression for the Processor B component of the system execution time. Processor A is assumed to be the master of Processor B. Processor A is responsible for initiating transaction pairs (when appropriate) and is also responsible for ensuring the completion of transaction pairs. Hence, memory utilization delays bus utilization delays, and operating system behaviour delays are not included in the expression for the Processor B component of the system execution time.

$$t_{B2} = t_{PREB2} + t_{EXEB2} + t_{POSTB2} + t_{CFGB2} \tag{3.7}$$

Equation 3.8 expresses the system execution time for a coprocessed system (System 2) using the knowledge that $t_{A2} > t_{B2}$. This relationship must be true. Otherwise, Processor A overruns Processor B. If this was not known, the system execution time would need to be expressed as the maximum of the two processor component execution times.

$$t_{SYS2} = t_{A2} \tag{3.8}$$

Equation 3.9 represents the speedup (or slowdown) of the system that occurs when a configurable coprocessor is introduced. This equation is based on the performance model of a coupled configurable coprocessor system. If t_{SYS2} is less than t_{SYS1}, the configurable coprocessor improves the performance of the system ($Speedup > 1$). If t_{SYS1} is less than t_{SYS2}, the configurable coprocessor degrades the performance of the system ($Speedup < 1$). If t_{SYS2} equals t_{SYS1}, the configurable coprocessor does not have an impact on the performance of the system ($Speedup = 1$).

$$Speedup = \frac{t_{SYS1}}{t_{SYS2}} \tag{3.9}$$

Given Equation 3.9, it is possible to solve for any one of the timing parameters necessary to achieve speedup given the other timing parameters. For example, the processing time of System 2 necessary to obtain a speedup of 2 can be calculated using Equation 3.10. This equation can be derived from Equation 3.6, Equation 3.8, and Equation 3.9 using the fact that t_{EXE2} is the sum of t_{PREA2}, t_{EXEA2}, and t_{POSTA2}.

$$t_{EXE2} = \frac{t_{SYS1}}{2} - (t_{OS2} + t_{BUS2} + t_{MEM2} + t_{CFG2}) \tag{3.10}$$

Alternatively, it is possible to express the absolute impact of coprocessing in terms of the impact expressions previously developed. The total impact, t_{TOTAL} of coprocessing is given by Equation 3.11. A positive t_{TOTAL} indicates a positive impact upon the system. The impact is expressed as the time saved by coprocessing the application.

$$t_{TOTAL} = t_{MEM} + t_{OS} + t_{BUS} + t_{EXE} - t_{CFGA2} \tag{3.11}$$

3.4 Performance Model Scenarios

Although every application system is slightly different, it is possible to substitute reasonable estimates of the timing parameters to predict the performance of a system using the performance model. This permits an examination of the impact of each timing parameter on the performance of the system. For example, consider the four identical sets of timing parameters given in Table 3.4. For the purpose of this example, assume these timing parameters correspond with a mainstream software application running on a non-coprocessed system.

Timing Parameter	Loosely Coupled		Tightly Coupled	
	Scenario 1 (in s)	Scenario 2 (in s)	Scenario 3 (in s)	Scenario 4 (in s)
t_{MEM1}	300	300	300	300
t_{BUS1}	100	100	100	100
t_{OS1}	100	100	100	100
t_{EXE1}	500	500	500	500
t_{SYS1}	1000	1000	1000	1000

Table 3.4: Timing Parameter Estimates for a Non-Coprocessed System

Consider the following four scenarios:

Scenario 1:

- Loosely-coupled coprocessor system.
- Memory utilization delays do not vary with coprocessing.
- Bus utilization delays increase by a factor of 4 due to coprocessing.
- Operating system behaviour delays increase by a factor of 4 due to coprocessing.
- Coprocessor computes application results 20% faster than a general-purpose processor

Scenario 2:

- This is the same as Scenario 1 with the exception that the coprocessor computes application results 80% faster than a general-purpose processor

Scenario 3:

- Tightly-coupled coprocessor system.
- Memory utilization delays do not vary with coprocessing.
- Bus utilization delays increase by a factor of 2 due to coprocessing.
- Operating system behaviour delays increase by a factor of 2 due to coprocessing.
- Coprocessor computes application results 20% faster than a general-purpose processor

Scenario 4:

- This is the same as Scenario 3 with the exception that the coprocessor computes application results 80% faster than a general-purpose processor

The timing parameters presented in Table 3.5 for a coprocessed system correspond with the four scenarios described. For the purpose of this example, memory utilization delays have remained constant. Also, it should be noted that the bus utilization delays and the operating system behaviour delays are often related to the coupling of the system. For the loosely-coupled scenarios, the same timing parameters are used for both bus utilization delays and operating system behaviour delays. The same is true for the tightly-coupled scenarios.

Timing Parameter	Loosely Coupled		Tightly Coupled	
	Scenario 1 (in s)	Scenario 2 (in s)	Scenario 3 (in s)	Scenario 4 (in s)
t_{CFGA2}	0.01	0.01	0.01	0.01
t_{CFGB2}	0.01	0.01	0.01	0.01
t_{MEM2}	300	300	300	300
t_{BUS2}	400	400	200	200
t_{OS2}	400	400	200	200
t_{EXE2}	400	100	400	100
t_{SYS2}	1500	1200	1100	800

Table 3.5: Timing Parameter Estimates for a Coprocessed System

Using the timing parameter estimates presented in Table 3.4 and Table 3.5, it is possible to estimate the impact of the sources of delays. Table 3.6 shows the impact of the delays quite clearly. Negative values represent performance degradations (i.e., increased delays) and positive values represent performance improvements (i.e., decreased delays).

Timing Parameter	Loosely Coupled		Tightly Coupled	
	Scenario 1 (in s)	Scenario 2 (in s)	Scenario 3 (in s)	Scenario 4 (in s)
t_{MEM}	0	0	0	0
t_{BUS}	-300	-300	-100	-100
t_{OS}	-300	-300	-100	-100
t_{EXE}	100	400	100	400
t_{TOTAL}	-500	-200	-100	200

Table 3.6: Estimated Impact of Timing Parameters

It should be noted that if t_{SYS1} is known and t_{TOTAL} is known, it is possible to calculate t_{SYS2} using Equation 3.12. This expression uses the fact that the impacts are relative to the original (non-coprocessed) system. Hence, the system execution times are related.

$$t_{SYS2} = t_{SYS1} - t_{TOTAL} \tag{3.12}$$

Based on the timing parameter estimates presented and the calculated impact of these timing parameter estimates, the tightly-coupled systems outperform the loosely-coupled systems. This is due to the fact that smaller bus utilization delays and operating system behaviour delays were estimated for the tightly-coupled systems. The timing parameter estimates are examined in more detail later in this thesis.

3.4.1 General Comments

For the purpose of this example, each scenario is assumed to represent a single application with one type of operation being coprocessed. The system execution time is assumed to be 1000 s prior to the introduction of a coprocessor. Since it is assumed that the system is only running a single application, the system execution time is also the application execution time. After the introduction of the coprocessor, the application execution time (t_{SYS2}) may be calculated by subtracting t_{CFGA2}, t_{MEM}, t_{BUS}, t_{OS}, and t_{EXE} from the application execution time (t_{SYS1}) of the non-coprocessed system.

The timing parameters in Table 3.5 assume that the hardware is ready for configuration and use at the start of the application's execution. Furthermore, only one configuration of the hardware

is assumed. This results in small configuration times estimated to be 0.01 s and rounded down to 0 s for the calculation of the system execution time. It is possible for configuration delays to dominate the execution time of the coprocessed application and it is also possible for configuration delays to be negligible. For the purpose of this example, the configuration delays are negligible.

For loosely-coupled systems, bus utilization delays (t_{BUS}) can have a large negative impact upon the system. Even if the bus is used sparingly, the performance impact on the entire bus hierarchy can be substantial. The reason for this influence is that the coprocessor in the loosely-coupled system is attached to a peripheral bus rather than the system bus. In other words, the coprocessor lies near the bottom of the bus hierarchy. A change in the utilization of the peripheral bus indirectly impacts a large portion of the bus hierarchy and hence, can have a large negative impact upon bus utilization delays. The increase in bus utilization delays also impacts the operating system behaviour delays (t_{OS}). The impact of memory utilization delays (t_{MEM}) is application dependent. The impact of memory utilization delays has been ignored for the purpose of this example. Memory utilization delays are considered later in this thesis.

For tightly-coupled systems, bus utilization delays (t_{BUS}) can have a small negative impact. Less of the bus hierarchy is impacted by an increase in bus transactions since the coprocessor is tightly-coupled to the processor. Similarly, operating system behaviour delays (t_{OS}) can have a small negative impact for tightly-coupled systems. Memory utilization delays (t_{MEM}) are application dependent. They have been ignored for the purpose of this example.

Clearly, it is possible to construct scenarios for the performance model that predict both positive and negative performance impacts. The next step in the development of the performance model is to determine estimates for real-world timing parameters. These estimates can be obtained by profiling the execution of configurable coprocessor systems on mainstream software applications. Given real-world timing parameters, it is possible to use the performance model to predict when the impact of a configurable coprocessor is positive or negative for a particular application.

Chapter 4

Configurable Computing Platforms

Two types of configurable computing platforms are investigated in this thesis. The first platform is an example of a loosely-coupled configurable computer. This platform consists of a PC (Personal Computer) and a configurable coprocessor mounted on a peripheral board. For the purpose of this thesis, this platform is referred to as *Platform I.* The second platform is an example of a tightly-coupled configurable computer. This platform consists of a Nios Embedded Processor Development Board [Cor02c] configured with a Nios processor connected directly to a coprocessor. For the purpose of this thesis, this platform is referred to as *Platform II.* A non-configurable platform is also investigated for the purpose of comparison. This platform consists of a Sun Workstation. For the purpose of this thesis, this platform is referred to as *Platform III.*

4.1 Platform I: PC + ARC-PCI Board

Platform I consists of a development workstation and a test workstation. The test workstation is a PC with an Intel Pentium III with a 450 MHz processor, 512 MB (MegaBytes) of SDRAM (Synchronous Dynamic Random Access Memory), and 1 GB (GigaByte) of swap space. This workstation runs Windows NT 4.0 with Service Pack 4. An ARC-PCI (Altera Reconfigurable

Computer - Peripheral Component Interconnect) Board connects to this workstation via a PCI (Peripheral Component Interconnect) Bus and a serial port connection. The ARC-PCI board provides the configurable hardware necessary for experiments into configurable computing. A second PC serves as a development platform. This PC consists of an Intel Pentium II with a 300 MHz processor, 512 MB of SDRAM, and 1 GB of swap space. This workstation runs Windows NT 4.0 with Service Pack 4, the hardware development tool (MAX+PLUS II), and the software development tools (Visual C++ 5.0, Windows Software Development Kit, and Windows Driver Development Kit). The use of a separate development workstation is essential for device driver development since a bug in a device driver can corrupt the file system of the test workstation. The test and development workstations are connected using ethernet and serial connections as shown in Figure 4.1.

4.1.1 The ARC-PCI Board

The ARC-PCI (Altera Reconfigurable Computer - Peripheral Component Interconnect) Board is a peripheral board designed by Altera Corporation to serve as a research platform for experiments in configurable computing. This board may be used as a configurable coprocessor in any computing platform that provides an empty PCI bus slot. The ARC-PCI Board incorporates Altera FLEX (Flexible Logic Element matriX) 10K Series devices, SRAM (Static Random Access Memory) modules, a 32-bit PCI bus interface and an external interface. This combination of programmable hardware, memory, and high-speed interfaces makes the board suitable for use in a wide variety of computing and interfacing tasks. Figure 4.2 shows a photograph of an ARC-PCI Board.

Programmable Logic

The ARC-PCI Board incorporates three Altera EPF10K50RC240-3 devices [Alt96]. One device serves as a *controller device* and the remaining two devices serve as *user devices*. A portion of the controller device implements the PCI bus interface. The remainder of the controller device manages the user devices and performs application-specific computations. The user devices provide the configurable hardware to enable the implementation of complex application-specific coprocessors.

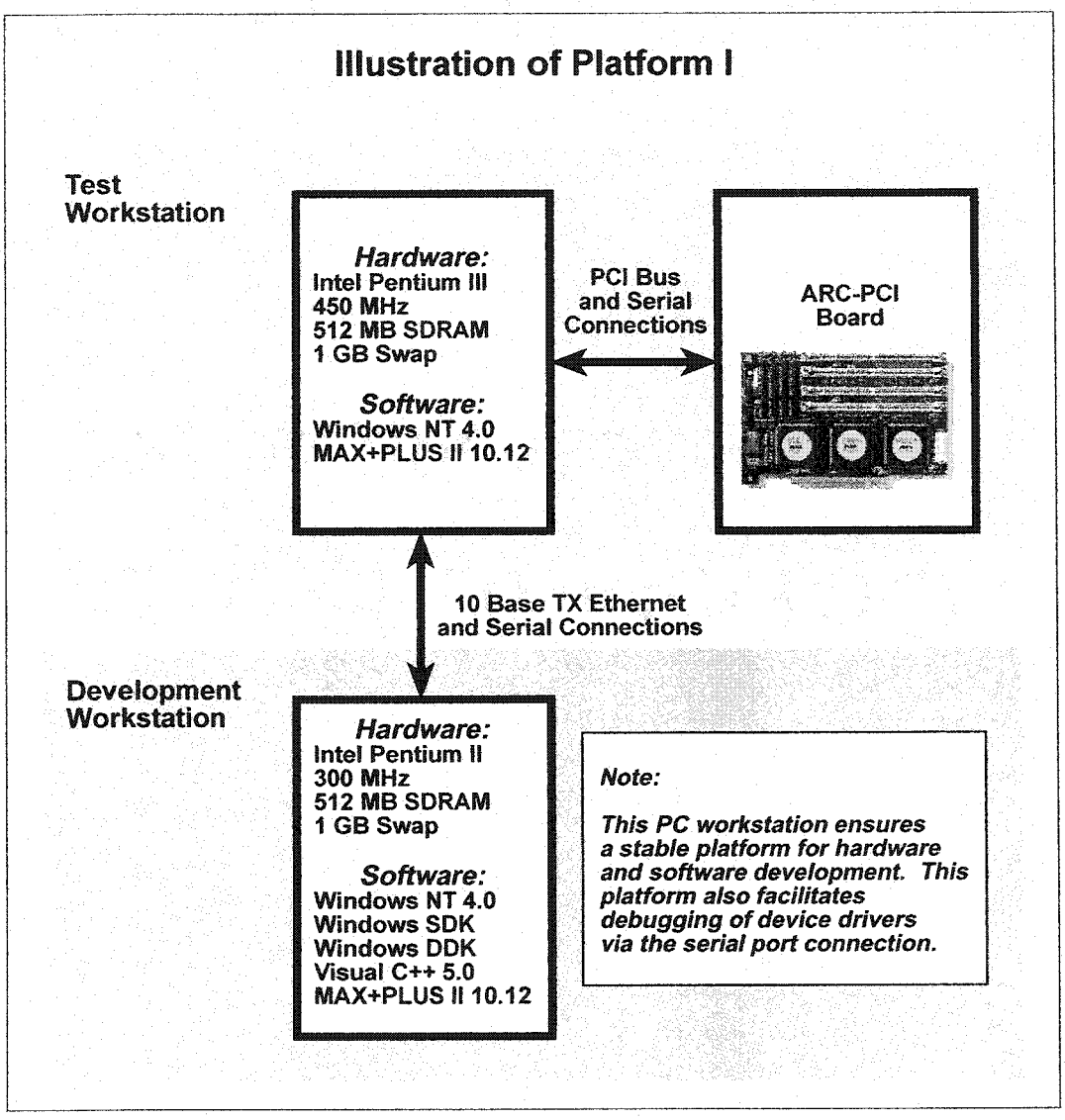


Figure 4.1: Illustration of Platform I

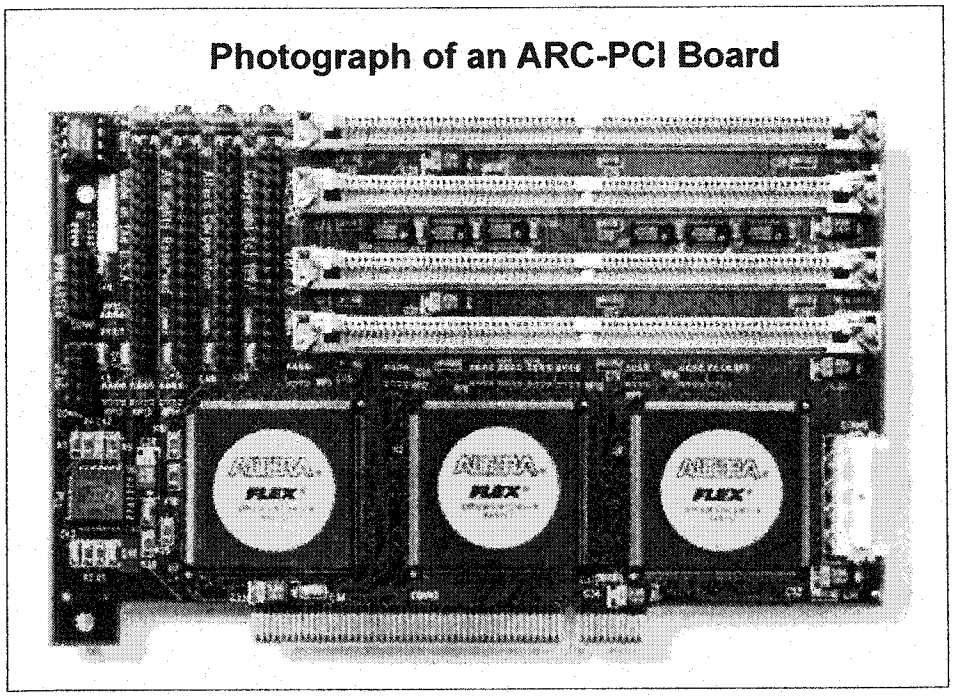


Figure 4.2: Photograph of the ARC-PCI Board

The EPF10K50RC240-3 is a 240 pin device in a RQFP (poweR Quad Flat Pack) package. This device is an example of a HDPLD. This device uses configurable SRAM LEs (Logic Elements) that support ICR (In-Circuit Reconfiguration) to provide dynamically configurable hardware. This device provides 36,000 logic gates[1], 20,480 RAM bits, and 189 usable I/O pins. This device provides the equivalent of 116,000 usable logic gates if all embedded RAM bits are fully utilized for logic. Altera estimates this device provides the equivalent of 50,000 usable logic gates for typical applications.

In total, the ARC-PCI Board provides 8,640 LEs (Logic Elements). Approximately 1,100 LEs are required to implement the PCI bus interface using an Altera PCI/MT32 MegaFunction [Alt98a]. This amounts to less than 13% of the LEs available on the board. The remaining 87% of the LEs may be used for application-specific hardware designs.

Memory Modules

The ARC-PCI Board provides 4 SIMM slots for memory devices. Two of the SIMM slots are used for memory that is shared by the 3 devices. The remaining two slots are used for cache memory. Each user device connects to a single cache memory slot. Each SIMM slot accommodates standard, 72-pin static SIMMs with support for memory modules as large as 1 M $\times$ 32 bits (4 MB). The cache memory devices are typically used for data caching by a particular user device while the shared memory devices are typically used for the centralized storage of configuration data and interface data.

ARC-PCI Bus Architecture

The programmable hardware devices and memory devices are connected by a complex network of busses. These busses are illustrated in Figure 4.3. The *Clock Bus* distributes a low-skew clock signal to the dedicated clock pins on each of the devices on the ARC-PCI Board. The *PCI Bus* provides access to a subset of the full PCI bus connections supplied by the host computer. The *Fast Bus* consists of unidirectional signals that can be used by the controller device to signal the

[1] As indicated in Chapter 2, logic gate estimates are only rough estimations of the actual logic capacity of a programmable logic device. The logic capacity of a programmable logic device depends upon the application, the quality of the development tools, the utilization of embedded memory blocks, the type of logic elements used, and the quality of the design.

start of a transaction. The *Memory Busses* provide access to the shared memory devices. These busses can also be used to transfer data between the 3 devices. The *Cache Busses* provide access to the cache memory devices. These busses can also be used to communicate via the external interface provided by the I/O connector.

4.1.2 ARC-PCI Development Kit

Altera provides a development kit for the ARC-PCI Board that includes all of the hardware and software components required to start designing configurable computing systems using the board. The ARC-PCI Development Kit consists of the following items:

- ARC-PCI Board with 2 Integrated Device Technology 7MP4060 128 K $\times$ 32 SRAM Modules
- License to use MAX+PLUS II
- License to use a PCI MegaFunction (either the Altera PCIT1 32-Bit PCI target interface or the Altera PCI/MT32 32-Bit PCI master/target controller)
- ARC-PCI Board schematics and documentation
- Sample AHDL (Altera Hardware Description Language) designs for the controller and user devices
- Generic device driver for Windows 95, Windows 98, and Windows NT
- Sample C language application code
- TTF2MAPP executable for translating TTF (Tabular Text Files) to MAPP (MAssively Parallel Programming) files

This development kit provides a satisfactory starting point for developing simple designs. However, the kit lacks the flexibility and performance necessary for developing complex designs. The sample AHDL code is well-written but difficult to understand and modify due to the use of proprietary AHDL code. The supplied device driver is not very useful in practice. It is a generic PCI device driver based upon a 3rd party DLL (Dynamic Link Library). The use of a 3rd party DLL makes the driver difficult to modify. The driver also performs poorly on dynamic

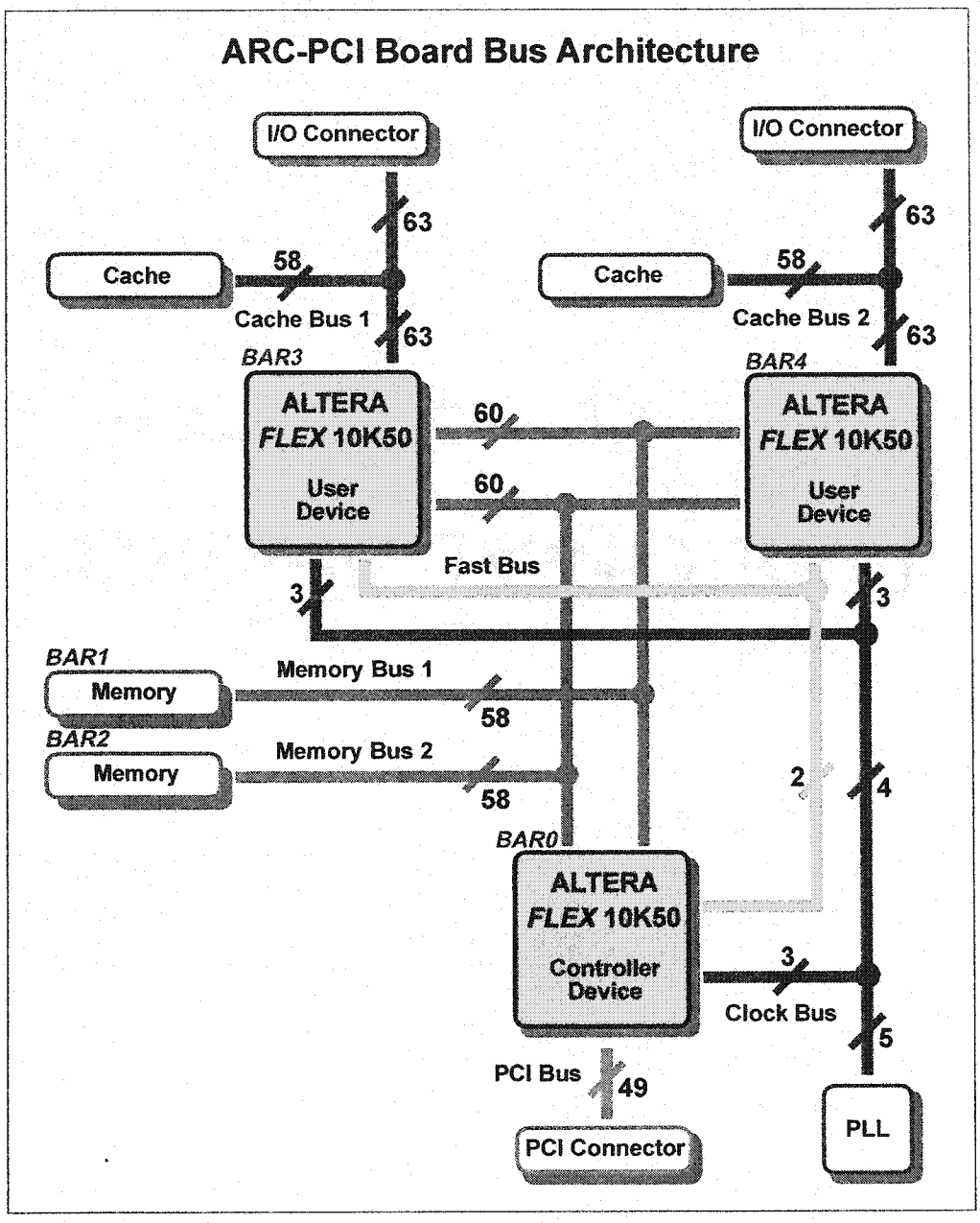


Figure 4.3: ARC-PCI Board Bus Architecture

configuration since the configuration data is not buffered on the ARC-PCI Board. Configuration times typically exceed 1 s due to PCI bus latencies and operating system behaviour. Custom device drivers and controller designs are necessary to obtain 33 MHz performance and correct system behaviour.

4.1.3 Configurable Computer Architecture

The combination of an ARC-PCI Board with a PC results in a loosely-coupled configurable computer architecture. Custom hardware and software components transform this computer architecture into a powerful computing system. The primary components of a complete system are the application software, the device driver, the controller design, and the user designs as illustrated in Figure 4.4. The application software communicates with the device driver using an API (Application Programming Interface) that translates library calls into IOCTLs (Input / Output Control Codes). The device driver communicates with the controller design on the ARC-PCI Board via the HAL (Hardware Abstraction Layer) provided by the operating system. The HAL translates commands into PCI bus transactions. The controller design communicates with the user designs using simple memory-mapped I/O (Input / Output).

It is common for transactions to be initiated by the application software. Applications request a transaction by calling a library function from the API. If required, the device driver initiates one or more PCI bus transactions to transfer data to or from the controller design. The controller design may respond to the PCI bus transactions directly or initiate communication with the user designs via memory-mapped I/O. The precise behaviour of the system depends upon the transaction and the custom hardware components utilized.

4.1.4 Application Programming Interface

Application software typically uses an API (Application Programming Interface) to communicate with a device driver. An API provides a set of library functions that simplify the task of interfacing with the device driver and the associated hardware device. The API for the ARC-PCI Board provides functions that read and write the memory-mapped I/O regions corresponding to the ARC-PCI Board. There are nine essential API functions. These API functions are summarized

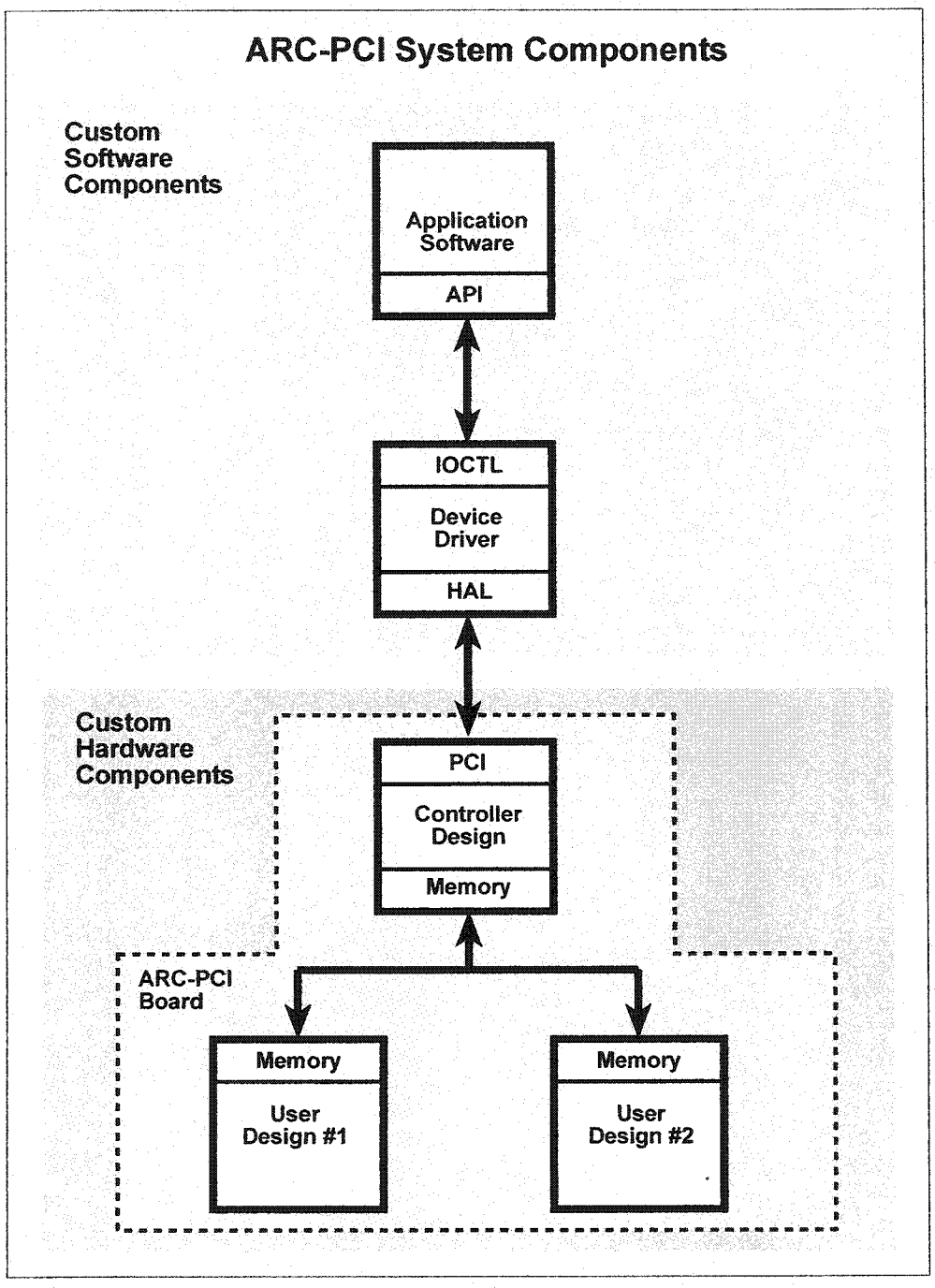


Figure 4.4: ARC-PCI System Components

in Table 4.1. The API developed for the ARC-PCI is the same for both the Windows and Linux operating systems. This API allows the development of portable software applications that utilize an ARC-PCI Board.

API Function	Description
arcpci_open( )	Establishes a connection to the ARC-PCI device driver
arcpci_close( )	Terminates a connection to the ARC-PCI device driver
arcpci_initialize( )	Initializes the ARC-PCI Board
arcpci_settings( )	Determines the features supported by the ARC-PCI Board
arcpci_store_MAPP( )	Stores a MAPP mode configuration file in the SRAM on the ARC-PCI Board
arcpci_read_bar( )	Reads 32 bits from a BAR (Base Address Region)
arcpci_write_bar( )	Writes 32 bits to a BAR (Base Address Region)
arcpci_block_read_bar( )	Reads a block of data (of maximum size 32 × 32 bits) from a BAR (Base Address Region)
arcpci_block_write_bar( )	Writes a block of data (of maximum size 32 × 32 bits) to a BAR (Base Address Region)

Table 4.1: Summary of API Functions

4.1.5 Device Driver

A device driver facilitates communication between application software executing within an operating system and a hardware device. A device driver can be considered an extension to an operating system to support peripheral devices. For Platform I, the device driver permits application software executing on a host PC to communicate with custom hardware executing on an ARC-PCI Board.

It is a complex task to design a device driver for an ARC-PCI Board. This board incorporates a PCI bus interface that supports plug and play operation. PCI devices provide a configuration space that permits the control of I/O regions, memory regions and interrupt vectors. This configuration space is set during a process referred to as PCI bus enumeration. All PCI-compliant devices are probed and configured by PCI bus enumeration. Provided that hardware configurations remain constant over time, PCI bus enumeration prevents hardware conflicts from occurring. However, configurable computing boards such as the ARC-PCI Board may change their hardware configuration over time. This unique issue must be considered carefully during device driver development.

The controller device on the ARC-PCI Board must be configured properly prior to the start of the device driver. The configuration of the controller device may be performed using a PROM (Programmable Read-Only Memory) or using MAX+PLUS II to drive signals onto the JTAG (Joint Test Action Group) port of the board via a serial connection. If the controller device has not yet been configured, the board is not detected by the operating system. If the controller device has been configured but is subsequently configured after the start of the device driver, the configuration space is not set appropriately for the most recently configured controller device. This scenario likely results in a failure of the operating system unless PCI bus enumeration is performed again and the device driver is stopped and restarted.

Device drivers may be started at boot time or dynamically as applications require them. It is often simpler to design a device driver that starts at boot time. However, for configurable computing, it is more practical to develop device drivers that start and stop dynamically. Otherwise, a reboot of the operating system is required whenever the hardware configuration of the board changes. A system reboot is undesirable so the ability to start and stop device drivers dynamically is essential.

Windows NT 4.0 Device Driver

A kernel mode device driver for Windows NT 4.0 [DN99] [VM99] was developed for the ARC-PCI Board. This device driver supports dynamic starting and stopping. The application software starts and stops the device driver via the SCM (Service Control Manager) only when required. The device driver initiates PCI bus enumeration on startup ensuring that the device driver configuration matches with the most recent hardware configuration of the controller device on the ARC-PCI Board. However, it also means that the startup of the device driver is quite time consuming. Depending upon the hardware configuration of the PC, several seconds may be required for PCI bus enumeration.

IOCTLs (Input / Output Control Codes) are a mechanism used to communicate data between an application and a hardware device. The ARC-PCI device driver supports a large set of generic IOCTLs that permit the transfer of data. These IOCTLs are described in Table 4.2.

The device driver is written entirely in the C language [KR88]. C++ language [Str94] device driver development is not fully supported for the Windows NT 4.0 operating system. In total,

IOCTL	Description
0x0A00	Initializes the ARC-PCI Board
0x0A01	Determines the features supported by the ARC-PCI Board
0x0A10	Reads from BAR_0 – the controller interface
0x0A11	Writes from BAR_0 – the controller interface
0x0A12	Block reads BAR_0 – the controller interface
0x0A13	Block writes BAR_0 – the controller interface
0x0A20	Reads BAR_1 – memory bus 1
0x0A21	Writes BAR_1 – memory bus 1
0x0A22	Block reads BAR_1 – memory bus 1
0x0A23	Block writes BAR_1 – memory bus 1
0x0A30	Reads BAR_2 – memory bus 2
0x0A31	Writes BAR_2 – memory bus 2
0x0A32	Block reads BAR_2 – memory bus 2
0x0A33	Block writes BAR_2 – memory bus 2
0x0A40	Writes a MAPP mode configuration to SRAM on the ARC-PCI Board
0x0A60	Reads BAR_3 – user design 1 interface
0x0A61	Writes BAR_3 – user design 1 interface
0x0A62	Block reads BAR_3 – user design 1 interface
0x0A63	Block writes BAR_3 – user design 1 interface
0x0A64	Reads BAR_4 – user design 2 interface
0x0A65	Writes BAR_4 – user design 2 interface
0x0A66	Block reads BAR_4 – user design 2 interface
0x0A67	Block writes BAR_4 – user design 2 interface
Others	Undefined and/or application-specific IOCTLs

Table 4.2: Summary of Supported Device Driver IOCTLs

the device driver consists of approximately 1,000 commented lines of code. The device driver may be customized to provide application-specific IOCTLs. However, this is unnecessary. The device driver provides all the IOCTLs required to build a fully-functional configurable computing system using an ARC-PCI Board.

The device driver incorporates advanced features including fast I/O dispatching, file I/O, and event logging. Fast I/O dispatching reduces the latency associated with device driver calls. File I/O allows the device driver to read a MAPP mode configuration file directly off the file system. Event logging allows a software designer to trace the execution of the device driver. These advanced features result in a device driver that is both efficient and easy to use.

Linux Device Driver for v2.4 Series Kernels

The Windows Driver Model [One99] and the Linux Driver Model [Rub98] are not identical. Differences in PCI bus enumeration, file I/O, and device driver loading / unloading complicate the task of porting a device driver from Windows to Linux. In 2001, the ARC-PCI device driver was successfully ported to Linux for v2.4 series kernels [Gra01].

Linux v2.4 series kernels enumerate PCI devices once, at boot time[2]. The kernel builds a linked list of all PCI devices in the system at this time. If the controller device on the ARC-PCI Board has been configured prior to booting, the ARC-PCI Board is detected. Otherwise, the board is ignored by the system. The device driver may be loaded at boot time or it may be loaded dynamically. However, it can only be loaded if the device has been successfully detected at boot time.

Boot time PCI bus enumeration poses a significant challenge for the ARC-PCI Board. The simplest solution involves ensuring that the controller device on the board is configured prior to booting Linux. The user devices can be configured after the device driver has been loaded and started. However, a configuration of the controller device will result in a kernel panic. For this reason, it is very important to have a reference controller design that can be used to communicate with a wide variety of user designs. This reference controller design can be programmed into a PROM so that the ARC-PCI Board is detected at boot time. This approach solves the PCI bus

[2]Some Linux v2.5 series kernels support hot-swappable PCI devices. Using these kernels, bus enumeration is not limited to once at boot time.

enumeration problem provided the configuration space of the controller design is not modified by a subsequent hardware configuration.

Due to security considerations, only certain classes of device drivers are allowed to directly access the file system. For this reason, the device driver may not read MAPP mode configuration files directly from the file system. These files are instead read by an application and transferred to the device driver via a memory transfer. This simplifies the design of the device driver but complicates the design of the application. However, it also allows the device driver to support MAPP mode configuration without the need for direct access to the file system.

In Linux, dynamic loading and unloading of device drivers is accomplished through the use of kernel modules. Since the dynamic loading and unloading of kernel modules does not solve the problem of PCI bus enumeration under Linux, there is no value to supporting this feature. The complexity of implementing the device driver as a loadable kernel module could not be justified and was not supported.

4.1.6 Controller Design

The controller device of the ARC-PCI Board is the only device on the board hardwired to the PCI bus interface. It acts as a bridge between the user devices and the PCI bus controller. It is responsible for monitoring the PCI bus and responding to PCI bus requests. All communication between the host computer and the ARC-PCI Board routes through the controller device. It is also responsible for managing the use of the memory busses to ensure that bus contention does not occur. A reference controller design provides the essential controller functionality. This design exceeds the 33 MHz PCI bus speed requirement and simplifies the task of building a working system.

The Reference Controller Design

The reference controller design incorporates a PCI bus interface, a configuration control circuit, and a small set of control and status registers. The controller design consists of a top-level graphical design file that connects control logic written in VHDL [Ins93] to an Altera PCI/MT32 MegaFunction [Alt98a]. This controller design synthesizes using MAX+PLUS II and exceeds the

performance requirements of the PCI bus. The design is sufficient for many applications and it provides a good starting point for more advanced controller designs.

The Altera PCI/MT32 MegaFunction [Alt98b] provides a 32-bit master/target PCI interface suitable for use with the ARC-PCI Board. The PCI/MT32 supplies an interface that is compatible[3] with several PCI bus controllers. This core uses memory-mapped I/O to communicate with the PCI bus controller. Each distinct memory-mapped I/O region corresponds to a BAR. A maximum of six BARs may be supported by this core.

For the purpose of the reference controller design, the Altera PCI/MT32 MegaFunction creates five BARs with two mapping to the shared memory devices via the memory busses, two mapping to the user devices via the memory busses, and one mapping directly to the controller device. The primary function of the control logic is bridging requests between the PCI bus and the devices. The use of five BARs simplifies the address decode logic within the controller design. This substantially improves the overall performance of the controller design. A summary of the five BARs is provided in Table 4.3.

BAR	Description	Size
0	Accesses control/status registers in the controller design	1 KB
1	Redirects access to SRAM on memory bus 1	512 KB
2	Redirects access to SRAM on memory bus 2	512 KB
3	Redirects access to user design 1	1 KB
4	Redirects access to user design 2	1 KB

Table 4.3: Base Address Regions

The control logic is responsible for responding to PCI bus transactions, managing the operation of the busses on the ARC-PCI Board, and configuring the user devices. The control logic uses one finite-state machine to respond to PCI bus transactions and another finite-state machine to configure the user devices. The control logic is split into two FSMs to reduce the number of state bits required to encode states. This optimization is required to permit the operation of the controller at the desired clock frequency of 33 MHz.

All functions of the controller interface are accessed by the software via a set of control and status registers. The reference controller design implements a 32-bit control register named GPR_0

[3]Compatibility and compliance are not equivalent. The ARC-PCI Board is not PCI-compliant but it is compatible with many PCI bus controllers.

that corresponds with address $0x000$ of BAR_0. Bit 0 of this register enables MAPP (MAssively Parallel Programming) mode configuration of the user devices. Bit 1 of this register enables the configuration of user device 1. Bit 2 of this register enables the configuration of user device 2. The remaining bits of GPR_0 may be used for temporary storage. The reference controller design also implements a 32-bit address register named GPR_1 that corresponds with address $0x004$ of BAR_0. All 32 bits are automatically loaded directly from the address lines during a transfer to BAR_1. Hence, GPR_1 may not be used for temporary storage.

Dynamic Configuration

MAPP mode configuration is a high-speed configuration protocol developed by Altera for use with the Altera 10K50 devices on the ARC-PCI Board. The inner workings of MAPP mode configuration have not been published by Altera but the following details have been revealed about the protocol:

1. TTF (Tabular Text Files) may be converted to MAPP programming files using the TTF2MAPP program supplied by Altera with the ARC-PCI Board.
2. Configuration words for Altera 10K50 devices are 44 bits wide.
3. Configuration files for Altera 10K50 devices consist of 30552 configuration words.
4. Configuration files contain a device ID but no other form of error-checking.
5. Multiple devices on the ARC-PCI Board can be configured identically at the same time.
6. MAPP mode configuration timing is similar to the timing used by passive serial configuration protocol.

A timing diagram for configuration of Altera FLEX 10K series devices is shown in Figure 4.5. This timing diagram is similar to the one provided in Altera's Application Note 116 [Cor99]. This timing diagram can be used for passive serial configuration, passive parallel configuration, and MAPP mode configuration of Altera FLEX 10K50 devices.

Passive serial configuration programs a device by transmitting configuration files as a series of bits. Passive parallel configuration uses a 8 bit configuration word that is clocked in serially.

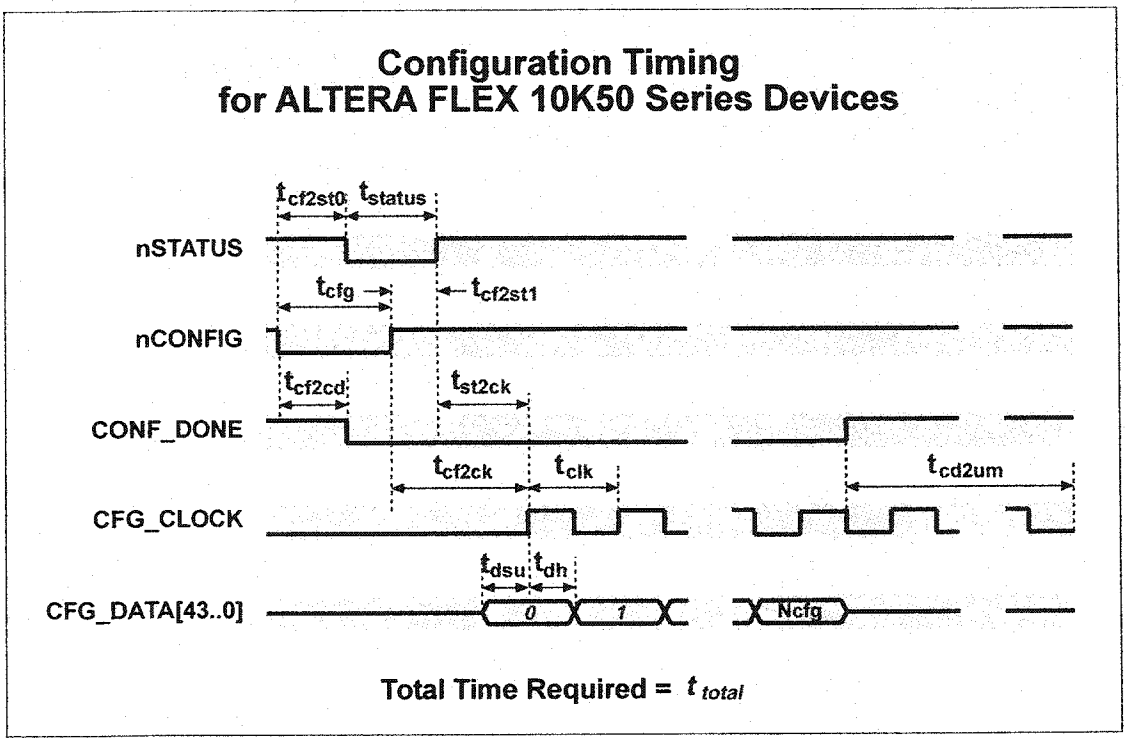


Figure 4.5: Configuration Timing for Altera FLEX 10K Series Devices

MAPP mode configuration uses a 44 bit configuration word to program the device. The timing parameters associated with these configuration modes are shown in Table 4.4. In theory, MAPP mode configuration is approximately 20 times faster than passive serial configuration.

Altera FLEX 10K50 Device Configuration Timing

Timing Parameter	PS (in ns)	PPS (in ns)	MAPP Mode	
			Ideal (in ns)	Actual (in ns)
t_{CF2CD}	200	200	200	200
t_{CF2ST0}	200	200	200	200
t_{CF2ST1}	4,000	4,000	4,000	100
t_{CFG}	2,000	2,000	2,000	300
t_{STATUS}	1,000	1,000	1,000	1,000
t_{CF2CK}	5,000	5,000	5,000	2,520
t_{ST2CK}	1,000	1,000	1,000	1,620
t_{DSU}	10	10	10	150
t_{DH}	0	0	0	30
t_{CLK}	60	15	60	180
t_{CD2UM}	600	600	600	360
n_{CFG}	621,240	77,655	30,552	30,552
t_{TOTAL}	37,282,000	9,326,200	1,840,720	5,502,540

Table 4.4: Altera FLEX 10K50 Device Configuration Timing

In practice, it is impossible to implement a controller design on the ARC-PCI Board that can write 44 bits of configuration data every 60 ns. The configuration pins are connected to memory bus 2 on the ARC-PCI Board. The memory connected to memory bus 2 cannot be accessed during a configuration without disrupting the configuration process. This restriction means that all configuration data must be stored in the memory connected to memory bus 1. Several wait states must be introduced to read the configuration data from memory, allow for the setup time on the configuration signals, and transmit the configuration word. A clock period of 180 ns is used instead of 60 ns for t_{CLK}. Experimentation also revealed that it is possible to reduce t_{CF2CK} by almost 50%. These timing differences limit the best possible MAPP mode configuration time to approximately 5.5 ms. This configuration timing has been successfully tested on three ARC-PCI Boards.

Caution must be exercised when experimenting with MAPP mode configuration. It is possible to damage the device if MAPP mode configuration fails since checksums are not used to validate the configuration information. MAPP mode configuration does not provide the same level of

safety as other configuration modes.

The reference controller design implements a finite-state machine that manages MAPP mode configuration. Hardware control of configuration substantially improves configuration performance. As many as seven user designs may be stored in memory on the ARC-PCI Board. These user designs may be swapped into a user device in just 5.5 ms. This is an extremely important feature of the reference controller design. This feature distinguishes this controller design from the one supplied by Altera in the ARC-PCI development kit.

Figures 4.6 and Figures 4.7 illustrate a significant difference between the way configuration is handled using a Windows NT Platform and the Linux Platform. Under Windows NT, the device driver may directly access the filesystem. As a result, only one IOCTL is required to initiate the configuration of a user device as depicted in Figure 4.6. Under Linux, the device driver must receive its data via IOCTLs. As a result, many IOCTLs are required to transfer the MAPP mode configuration data to the ARC-PCI Board.

4.1.7 User Designs

The configuration of the user devices with user designs is optional. If the application-specific design circuitry is simple, it is sometimes possible to fit this circuitry in the controller design. For more complex applications, user designs are required to implement the circuitry. The hardware development flow for a user design is illustrated in Figure 4.8.

User designs communicate with the controller design using the memory busses. The fast bus, the clock bus, and the memory bus of the ARC-PCI Board are used to signal memory bus activity to user designs. Figure 4.3 illustrates the bus hierarchy. In total, nine handshaking signals are used. These signals are shown in Table 4.5

The `system_clock` signal controls the timing of transactions. Transactions start on the rising-edge of the `system_clock`. The address bus and data bus enable signals allow user designs to assume mastership of the memory busses when the controller design is not using them. This feature is particularly useful if a user design needs to access the memory devices connected to the shared memory busses. Separate handshaking signals are provided for each of the user designs so that the designs may operate independently.

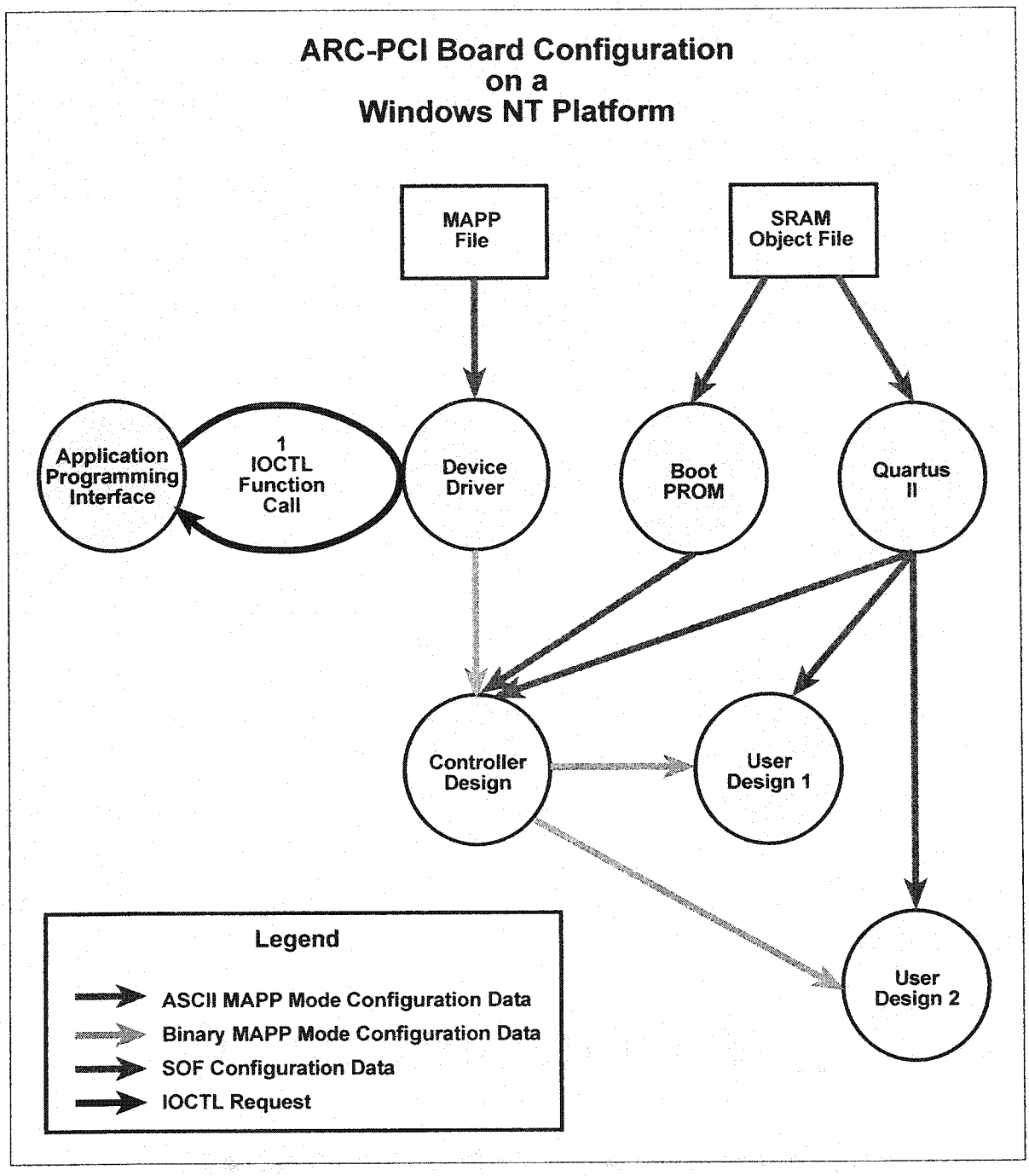


Figure 4.6: ARC-PCI Board Configuration on a Windows NT Platform

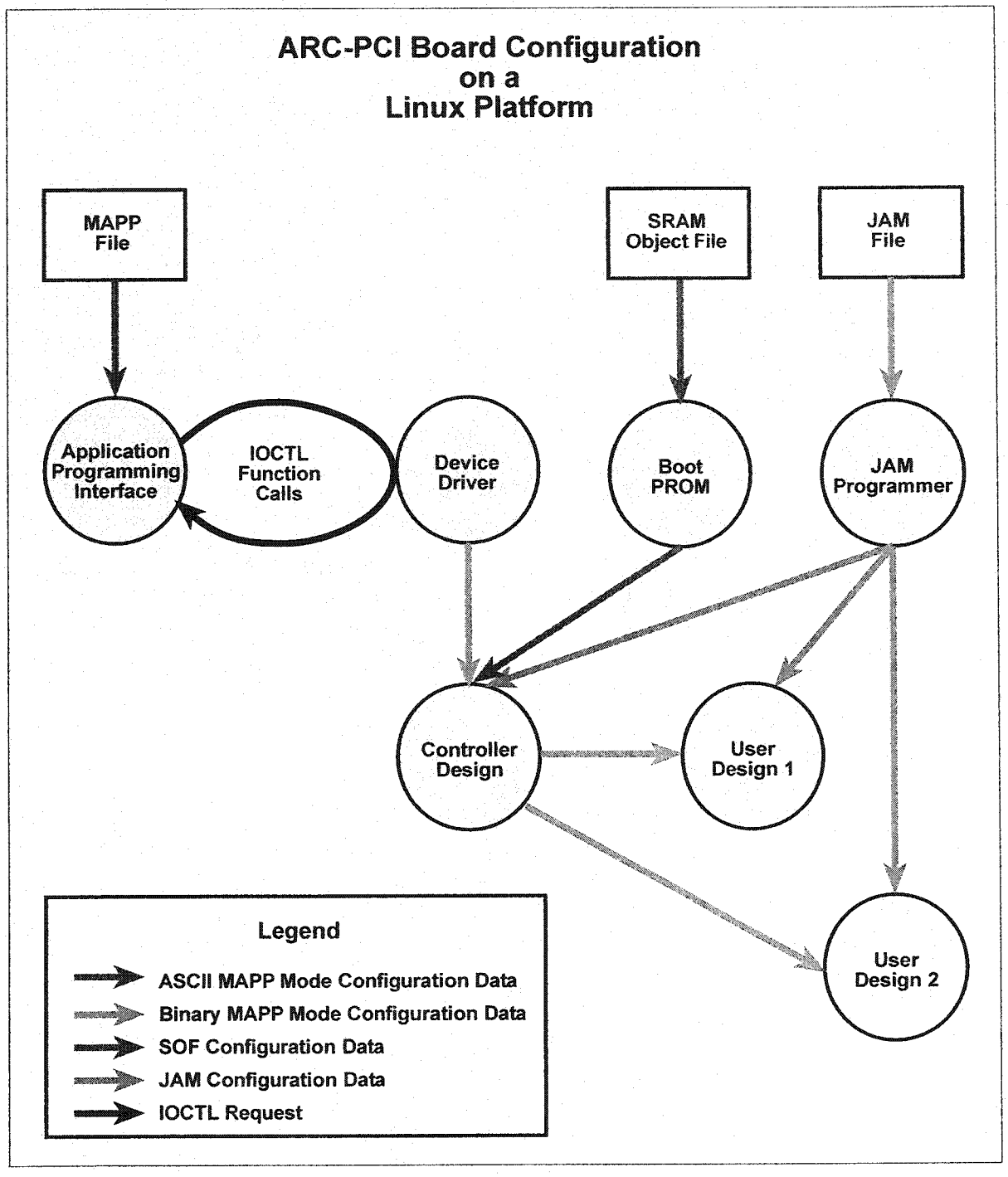


Figure 4.7: ARC-PCI Board Configuration on a Linux Platform

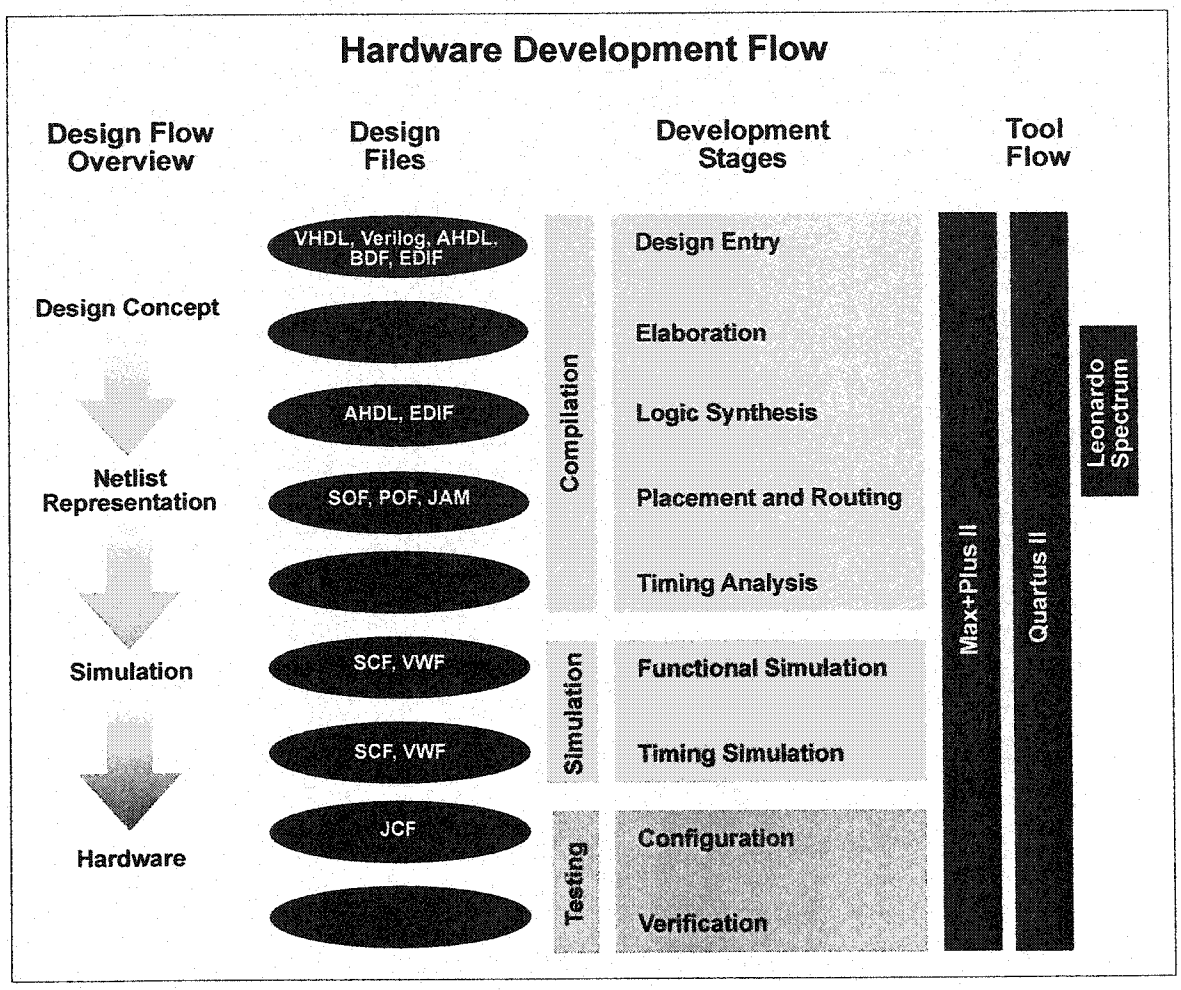


Figure 4.8: Hardware Development Flow

ARC-PCI Board Signal	Description
system_clock	33 MHz clock signal used by the controller for transaction timing
bus1_pld1_mion	Active-low read enable for user design 1
bus1_pld2_mion	Active-low read enable for user design 2
bus1_pld1_fastn	Active-low address enable for user design 1
bus1_pld2_fastn	Active-low address enable for user design 2
bus2_pld1_mion	Active-low write enable for user design 1
bus2_pld2_mion	Active-low write enable for user design 2
bus2_pld1_fastn	Active-low data bus enable for user design 1
bus2_pld2_fastn	Active-low data bus enable for user design 2

Table 4.5: User Design Handshaking Signals

User designs must continuously monitor the handshaking signals to determine when it is appropriate to read or write data on the memory busses. User designs are expected to respond to these signals within one clock cycle. Wait states are not permitted. The performance of user designs is not constrained in any other way.

4.1.8 Comments on PCI Compliance

The ARC-PCI board is not PCI-compliant. Information on PCI-compliant boards can be found in the book, PCI System Architecture [SA95]. The ARC-PCI Board can be connected to a PCI bus but there is no guarantee it works with all PCI bus controllers for the following reasons:

1. The Altera FLEX 10K series devices do not support PCI I/O signaling levels.
2. The ARC-PCI Board uses a PLL to reduce skew on the clock traces. The use of a PLL does not conform to the original PCI Specification. A signal driven by a PLL does not operate correctly when the PCI clock is single-stepped. In practice, the use of a PLL does not introduce any significant incompatibilities. In fact, recent updates to the PCI Specification permit the use of PLLs.
3. The ARC-PCI Board has a separate power connection to supply the devices with power. The peak power consumption of the ARC-PCI Board exceeds the power that can be delivered by the PCI bus. The use of independent supply voltages can result in problems.

4.1.9 Comments on Performance

The Altera FLEX 10K series devices can be clocked at 33 MHz but great care must be taken to successfully build designs to achieve this clock frequency. The pinouts of the devices on the ARC-PCI Board are fixed. The pinout constraints limit the flexibility and performance of hardware designs. It is often necessary to limit the complexity of the controller design and the user designs to successfully meet all timing requirements.

Software timing results were obtained to analyze the performance of the API, the device driver and the reference controller design. First, an experiment was conducted to verify that time estimates reported by Windows accurately reflect the elapsed time of a sequence of operations

as observed by the ARC-PCI Board. A simple user design that implemented a hardware timer was developed. A comparison of the software timing and hardware timing results is presented in Table 4.6. These results show that the software time estimates reported by software timer routines provided by Windows are reasonably accurate. Software timing is used for all experimental results presented later in this thesis.

Windows Timing Comparisons			
Transaction Name	Device Driver Execution Times (in ns)		
	Software Timer	Hardware Timer	Difference
Controller Design Read	545	544	1
Controller Design Write	221	221	0
Memory Read	530	530	1
Memory Write	220	220	0
User Design Read	546	545	1
User Design Write	221	220	0

Table 4.6: Windows Transfer Comparisons

A series of experiments were conducted to examine the average time required to perform a single transaction using the reference design on the ARC-PCI Board. Each transaction represents a read or write operation to a device on the ARC-PCI Board. The three classes of devices of interest are the controller interface, the shared memory devices, and the user designs. A simple user design that implements a single general-purpose register was created for use in these experiments. An application program was written to execute a sequence of transactions and report the execution time.

Three types of transactions were evaluated during each experiment. Device driver transaction times report the time required by the device driver to perform a read or a write. Buffered transaction times report the time required by the application to perform 16 reads or 16 writes as a block[4] of data buffered in the the device driver. Unbuffered transaction times report the time required by the application to perform a read or a write. Figure 4.9 illustrates the difference between buffered and unbuffered transactions. Each type of transaction was repeated 10,000,000 times during each experiment. The average transaction times were recorded. The complete set of

[4]Although not presented in this thesis, experimentation with different block sizes was performed. A block size of 16 was chosen to balance memory usage with performance. Larger block sizes did not substantially improve transaction performance.

transfer rate test results is provided in Appendix A

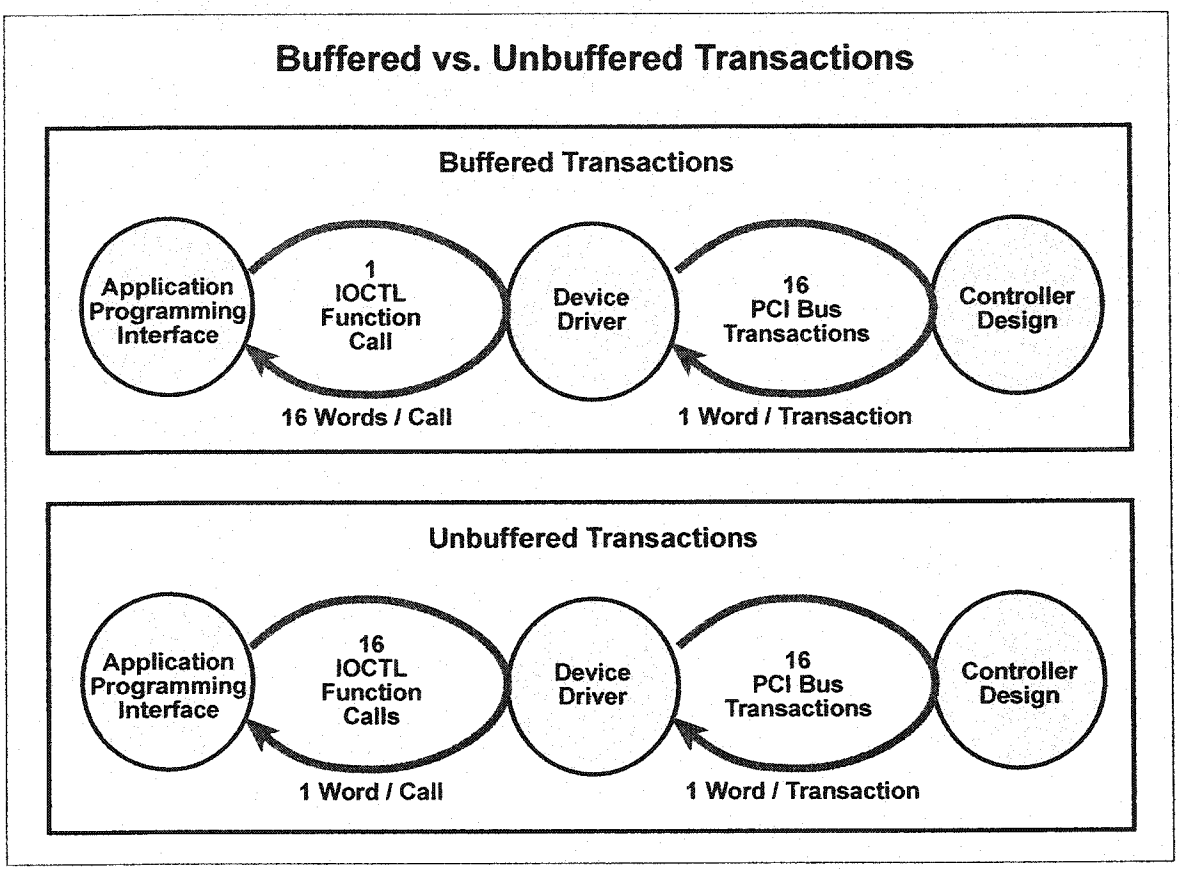


Figure 4.9: Buffered vs. Unbuffered Transactions

Table 4.7 shows the average transaction times using Windows for 32-bit transfers initiated by a device driver, buffered transfers initiated by an application, and unbuffered transfers initiated by an application. Buffering significantly reduces the average transaction time experienced by an application. The grouping of transactions into blocks consisting of 16 words reduces the number of IOCTLs executed. This effectively reduces the number of context switches initiated by the operating system. Assuming that every IOCTL results in a context switch, the IOCTL overhead represents the time required for a context switch in Windows. This figure is approximately 2000 ns, regardless of the transaction type.

It is useful to compare the observed performance with the theoretical performance expected.

Windows Transaction Times				
Transaction Name	Device Driver Transaction Times (in ns)	Buffered Transaction Times (in ns)	Unbuffered Transaction Times (in ns)	IOCTL Overhead (in ns)
Controller Design Read	545	579	2509	1964
Controller Design Write	221	303	2201	1980
Memory Read	530	579	2537	2007
Memory Write	220	302	2177	1957
User Design Read	546	581	2541	1995
User Design Write	221	301	2175	1954

Table 4.7: Windows Transfer Times

Using the PCI/MT32 MegaFunction, each PCI bus read transaction requires at least 8 clock cycles or 240 ns. The reference design performs buffered read transactions in approximately 580 ns. Using the PCI/MT32 MegaFunction, each PCI bus write transaction requires at least 9 clock cycles or 270 ns. The reference design performs buffered write transactions in approximately 300 ns. Synchronization delays have less of an impact upon the performance of PCI bus write transactions. The observed performance of the reference design is very close to the theoretical performance of the hardware.

Using the transaction times shown in Table 4.7, it is possible to calculate the amount of time required to transmit a MAPP mode configuration file to memory on the ARC-PCI Board. A MAPP mode configuration file requires the transmission of 30,552 configuration words. Each configuration word is 44-bits wide. The transmission of a configuration word requires two 32-bit PCI bus write transactions. In total, the equivalent of 61,104 transactions are necessary. The MAPP mode configuration file is read directly by the device driver under Windows. The time per buffered write transaction for a device driver is approximately 221 ns. Therefore, the transmission of a MAPP mode configuration file to the ARC-PCI Board requires approximately 13.5 ms. This assumes that the file is cached in memory. In practice, additional delays are encountered as the file is read from the file system. These delays are difficult to estimate since the delays depend upon the state of the computer system and its workload.

Table 4.8 shows the average transaction times using Linux for 32-bit transfers initiated by a device driver, buffered transfers initiated by an application, and unbuffered transfers initiated by an application. The results are similar to those observed for Windows. However, the IOCTL

overhead is smaller for Linux on average. Assuming that this overhead represents the time required for a context switch, Linux has a context switch time of approximately 1800 ns. It is worth noting that there is a slight difference between the overhead experienced by read and write transactions. Read transactions experience an additional delay of approximately 135 ns.

Linux Transaction Times				
Transaction Name	Device Driver Transaction Times (in ns)	Buffered Transaction Times (in ns)	Unbuffered Transaction Times (in ns)	IOCTL Overhead (in ns)
Controller Design Read	465	523	2310	1845
Controller Design Write	210	242	1920	1710
Memory Read	465	524	2309	1844
Memory Write	210	242	1919	1709
User Design Read	465	523	2309	1845
User Design Write	210	242	1919	1709

Table 4.8: Linux Transfer Times

Subtle timing differences have been observed between the Windows and Linux implementations as illustrated in Table 4.7 and Table 4.8. The Windows transfer times are slightly slower than those observed for Linux. The observed difference is relatively small. It could be related to the relative performance of the operating systems or it could be related to the load on the operating systems at the time of testing. Windows is used for all experimental results presented later in this thesis.

4.2 Platform II: Nios Embedded Processor Development Board

Platform II consists of a development workstation and a test workstation. The test workstation is a Nios Embedded Processor Development Board [Cor02c] configured with a 33 MHz Nios processor [Cor02d], 256 KB of SRAM, and no swap space. No operating system is run on this workstation. Application software directly accesses the hardware devices using memory-mapped I/O (Input/Output). This board provides sufficient programmable logic resources to permit the development and use of configurable coprocessors. A PC serves as a development platform. This PC consists of an Intel Pentium III with a 450 MHz processor, 512 MB of SDRAM, and 1 GB

of swap space. This workstation runs Windows 2000, the hardware development tools (Quartus II and the Nios Embedded Processor Development Kit), and the software development tools (GnuPro Toolkit and the Nios Embedded Processor Development Kit). The test and development workstations are connected using parallel and serial connections as shown in Figure 4.10.

4.2.1 The Nios Embedded Processor Development Board

The Nios Embedded Processor Development Board [Cor02c] is a standalone platform for embedded system development using a Nios processor. This board may be used as a configurable computing platform although it is most typically used to prototype embedded systems. As a configurable computing platform, it serves as a tightly-coupled configurable computer formed by connecting a soft core Nios processor to one or more configurable coprocessors. Figure 4.11 shows a photograph of a Nios Embedded Processor Development Board.

Programmable Logic

The Nios Embedded Processor Development Board uses an Altera APEX 20K200EFC484-2X device to implement an embedded system. This device provides 8,320 LEs and 104 KB of SRAM. It is not known whether this device supports MAPP mode configuration. However, it does support passive serial mode configuration. The configuration timing shown in Table 4.9 applies to APEX 20KE series devices. A total of 35 ms is required to configure the APEX 20KE series device on the Nios Embedded Processor Development Board.

4.2.2 Nios Embedded Processor Development Kit

It is beyond the scope of this thesis to document the entire Nios Embedded Processor Development Kit. This kit is very well documented [Cor02c] [Cor02d] [Cor02e] [Alt02a] [Alt02c] [Alt02b]. It provides all of the hardware, cores, software libraries, and tools necessary to build complex embedded systems based on the Nios processor.

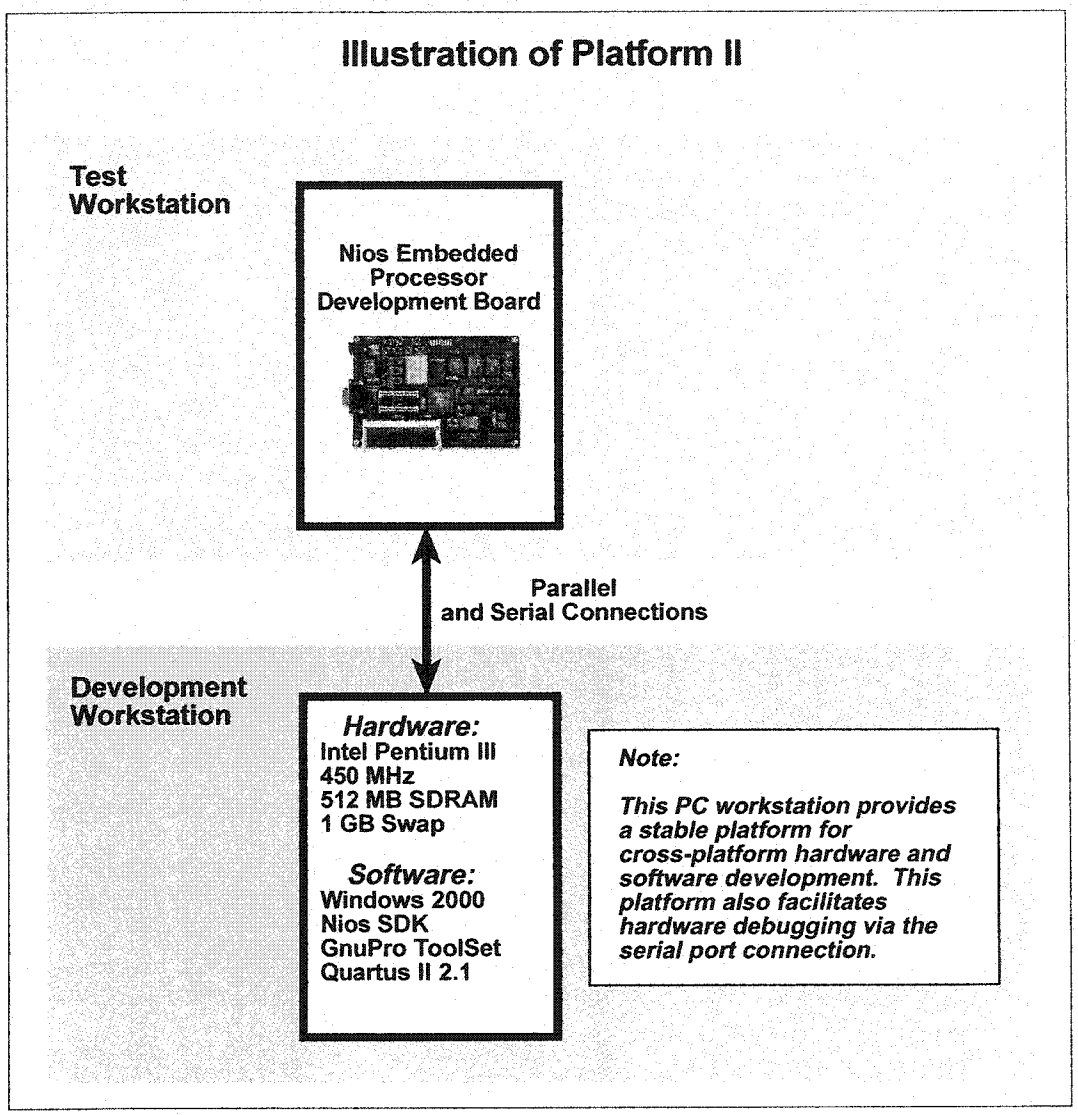


Figure 4.10: Illustration of Platform II

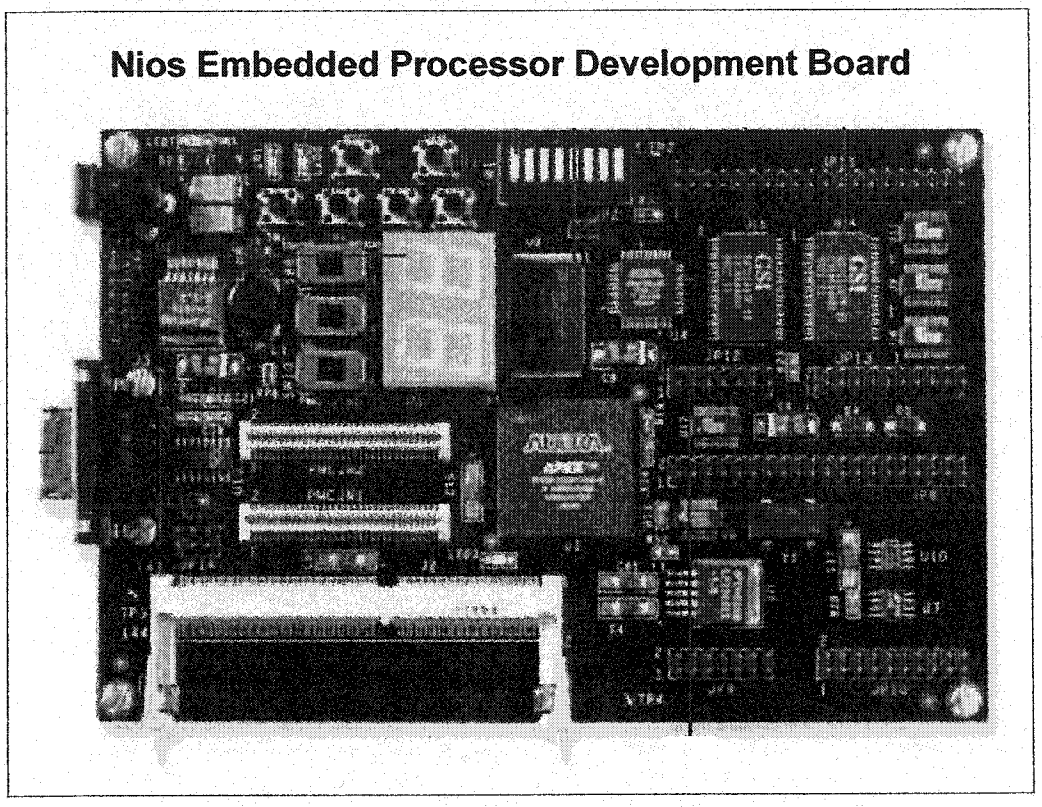


Figure 4.11: Nios Embedded Processor Development Board

Altera APEX 20K200E Device Configuration Timing

Timing Parameter	PS (in ns)
t_{CF2CD}	200
t_{CF2ST0}	200
t_{CF2ST1}	1,000
t_{CFG}	8,000
t_{STATUS}	10,000
t_{CF2CK}	40,000
t_{ST2CK}	1,000
t_{DSU}	10
t_{DH}	0
t_{CLK}	18
t_{CD2UM}	2,000
n_{CFG}	1,968,016
t_{TOTAL}	34,490,280

Table 4.9: Altera APEX 20K200E Device Configuration Timing

4.2.3 Configurable Computer Architecture

The Nios Embedded Processor System consists of application software, a Nios processor, and a set of peripherals as shown in Figure 4.12. Application software communicates with the Nios embedded processor using API functions that are translated into one or more memory-mapped I/O transactions. The Nios embedded processor communicates with peripherals via a proprietary Avalon bus. This bus is a 32-bit bus that clocks at the same frequency as the processor. The clock frequency is nominally 33 MHz for a Nios processor.

It is possible to create a tightly-coupled configurable computer by defining one or more user peripherals that attach directly to the Avalon Bus of the Nios processor. However, such a system differs from a more traditional configurable computer system with respect to dynamic configuration. Since the Nios processor is implemented using the same programmable logic device as the user peripherals, the processor cannot initiate a dynamic configuration of the user peripherals. As a result, the user peripherals are fixed for the duration of the system[5].

4.2.4 Nios Embedded Processor

Nios embedded processors can be optimized for performance or area. A 32-bit Nios processor optimized for performance is used for all experimental results presented in this thesis pertaining to Platform II. The processor clocks at 33 MHz, the nominal clock frequency for a processor on the Nios Embedded Processor Development Board. Hardware multiplication and all other performance optimizations are enabled. These optimizations are discussed in greater detail in Chapters 6 and 7.

For the purpose of this thesis, the Nios embedded processor is the Avalon bus master. Bus mastering peripherals are not used. This avoids the delays associated with bus arbitration. However, it also forces all peripherals to be slaves to the processor.

[5]This would not necessarily be the case if partial configuration was supported by the programmable logic device. Also, this limitation could be avoided by adding another programmable logic device to the Nios Embedded Processor Development Board

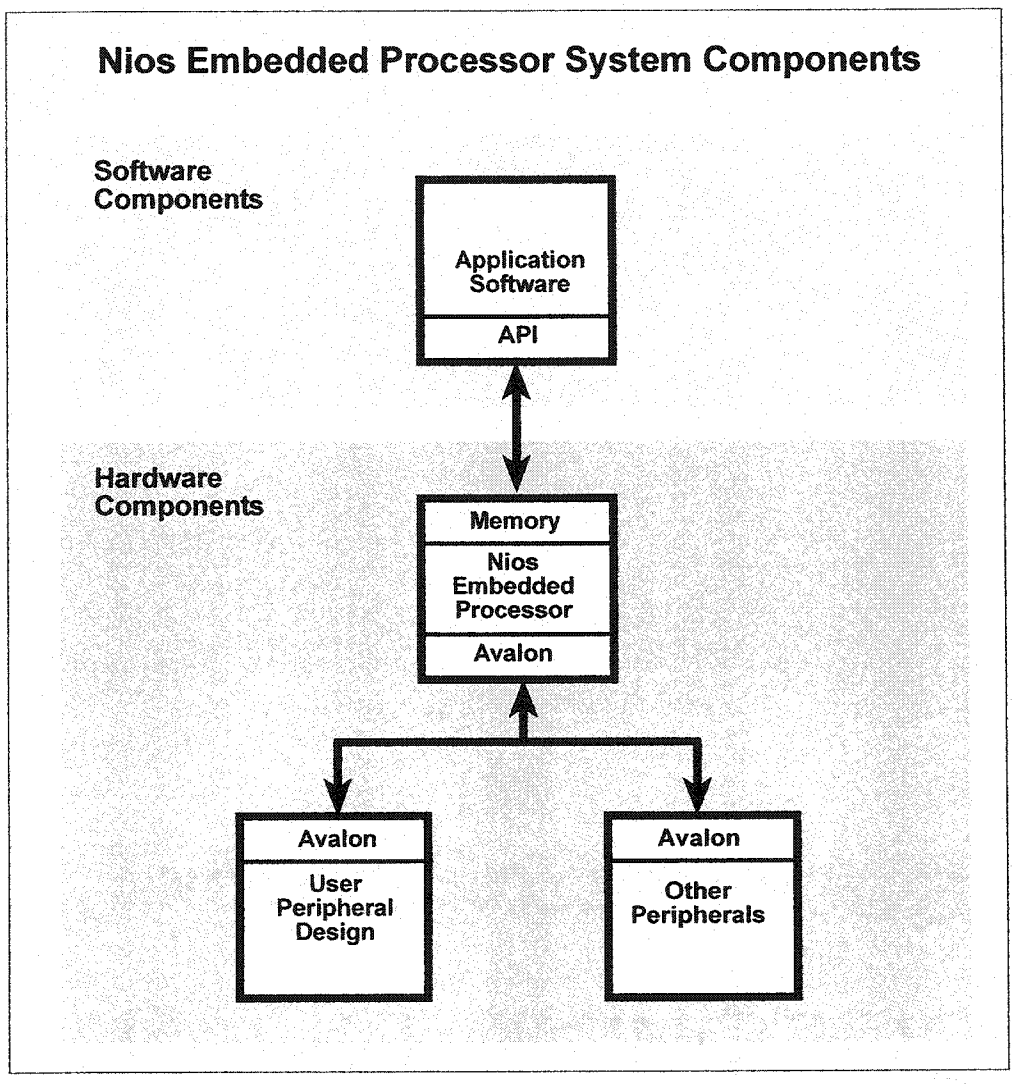


Figure 4.12: Nios Embedded Processor System Components

4.2.5 User Peripheral Designs

Altera provides a wizard to simplify the development of custom peripherals that attach to the Avalon bus of the Nios processor. Using the wizard, each peripheral is assigned a unique memory-mapped I/O region within the address space of the processor. Optionally, peripherals may support interrupts. The timing of the interface is flexible. Wait states can be inserted to allow extra time for peripherals to respond to requests. If wait states are not used, peripherals must be capable of responding to a bus transaction once every clock period.

User peripheral designs can implement any hardware device that fits in the available programmable logic resources provided that the device meets the timing requirements of the Avalon bus. These constraints are very weak given the large amount of programmable logic resources available within the device and the flexibility of the bus interface. The Altera APEX 20K200E device is a powerful platform for the development of complex user designs.

4.2.6 Other Peripherals

For the purpose of the experimental results presented in this thesis, a system configuration similar to the one documented in the Nios Tutorial [Cor02e] is used. A standard set of peripheral interfaces must be instantiated to build a complete system. RAM, ROM and UARTs are required to run application software on the embedded system. Other peripherals such as pushbutton inputs, LED outputs, and timers are used to simplify the task of testing and debugging the system.

4.3 Platform III: Sun Workstation

Platform III consists of a Sun Ultra 1 with a 167 MHz processor, 128 MB of RAM, and 1 GB of swap space. This platform does not provide any programmable logic resources. It is investigated to verify that the PC platform results are consistent with those found in modern workstations. The Solaris 2.6 Operating System is installed on this workstation.

4.4 Platform Comparison

Table 4.10 summarizes the key features of the processors of each platform discussed in this thesis and Table 4.11 summarizes the key features of the coprocessors of each platform discussed in this thesis. Platforms I and II are very similar in terms of quantity of programmable logic resources and RAM. However, there is a substantial difference in the quality of the programmable logic resources and RAM provided. The LEs available in an Altera APEX series device are more powerful than those found in an Altera FLEX series device. They are faster and can compute more complex functions of inputs. Similarly, the RAM available in an Altera APEX series device is much faster than the RAM available in an Altera FLEX series device.

Feature	Platform I	Platform II	Platform III
Processor Type	450 MHz Pentium III	33 MHz Nios	167 MHz Ultra 1
On-Chip (Cache) Memory Capacity	512 KB SRAM	N/A	512 KB SDRAM
On-Chip (Cache) Memory Latency	4.4 ns	N/A	12 ns
Off-Chip Memory Capacity	512 MB SDRAM	256 KB SRAM	128 MB SDRAM
Off-Chip Memory Latency	70 ns	20 ns	60 ns
Virtual Memory Capacity	1 GB	N/A	1 GB

Table 4.10: Summary of Computing Platform Processors

Feature	Platform I	Platform II	Platform III
Coprocessor Type	FLEX 10K50	APEX 20K200E	N/A
Coprocessor Logic Resources	8,640 LEs	8,320 LEs	N/A
Off-Chip Memory Capacity	1 MB SRAM	N/A	N/A
Off-Chip Memory Latency	20 ns	N/A	N/A
On-Chip Memory Capacity	60 KB SRAM	104 KB SRAM	N/A
On-Chip Memory Latency	17 ns	4.2 ns	N/A
Coprocessor Bus	32-Bit PCI Bus	32-Bit Avalon Bus	N/A
Coprocessor Bus Bandwidth	100 MBytes / second	133 MBytes / second	N/A

Table 4.11: Summary of Computing Platform Coprocessors

Another significant difference between Platforms I and II is the speed of the processor. The soft core Nios processor used in Platform II runs at a much slower clock frequency than the Pentium III processor used in Platform I. This clock frequency difference means that the difference in bandwidth between the processor and the bus in Platform I is much higher than the difference in Platform II.

Platform III differs from both Platforms I and II quite significantly. Platform III is simply included in this research to allow comparisons between the performance of a PC and a Sun workstation. These comparisons establish that it does not make a significant difference whether a PC or a Sun workstation is used for configurable computing research. The two platforms deliver similar application performance.

Chapter 5

Application 1: CSIM

This chapter describes the first of three sets of experiments into configurable computing applications. The goals of this experiment were to identify and quantify the major factors influencing the performance of a loosely-coupled configurable computer on a mainstream software application. CSIM, a discrete-event simulation library, was chosen as the mainstream software application. The execution of CSIM was profiled using a FIFO queue as a simulation benchmark. Using the results of this analysis, a configurable coprocessor for CSIM was developed for the purpose of generating pseudo-random numbers. This coprocessor reproduces the exact algorithm used for pseudo-random number generation within CSIM. A series of experiments were conducted on Platform I using this configurable coprocessor. The experimental results demonstrate the difficulties associated with accelerating a mainstream software application using a loosely-coupled configurable computer.

5.1 Introduction to Discrete-Event Simulation

Discrete-event simulation is a computational technique for predicting the dynamic behaviour of a complex system. Common applications of discrete-event simulation include the simulation of assembly lines, military operations, supply chains [SP00], air traffic, computer architectures, and wireless communication systems. In a discrete-event simulation, the state of the system is updated

only when events of interest occur. Unlike continuous simulation, only the portions of the system state related to events of interest are updated. All discrete systems and many continuous systems can be simulated accurately using discrete-event simulation.

5.1.1 Discrete-Event Simulation Terminology

A number of terms have a specific meaning within the context of discrete-event simulation. For the purpose of this thesis, systems, models, and discrete-events are defined in the following subsections. These terms are used throughout this chapter.

System

The *system* is the physical system to be simulated. The system represents a real-world process or problem to be studied. Typically, simulation is used when it is impossible, impractical, or inappropriate to study a problem using an experimental or analytical method. For example, simulation is used to study chemical reactions since it is often impractical to conduct repeated experiments using expensive materials. A chemical reaction is an example of a system.

Model

The *model* is a computational representation of the behaviour of a system. A model is typically written as a set of communicating logical processes. These processes predict the dynamic behaviour of the active entities of the system. As these processes compute, they generate a sequence of events in time. These events may represent the production / consumption of resources, communication of information, or other events of interest within the system. An accurate model investigates all events of interest.

Discrete-Event

A *discrete-event* is a time-stamped event that represents some activity within the system. As discrete-events are generated by processes, they are placed in an event queue. During each cycle of a discrete-event simulation, one event is removed from the event queue and processed. As

each event is processed, the current simulation time is updated based on the time-stamp of the most recently processed event. The amount of time required to simulate a system is directly proportional to the number of events processed.

5.1.2 Discrete-Event Simulation Tools and Libraries

Discrete-event simulation tools and libraries aid in the development of complex discrete-event simulations. Discrete-event simulation tools and libraries differ in their approach towards achieving the goal of modeling and predicting the behaviour of a system. Tools provide a way of describing a system graphically and simulating its behaviour. Libraries provide a set of data structures and methods that permit describing a system textually and simulating its behaviour. Tools offer ease of use while libraries offer flexibility and performance.

Discrete-event simulation tools assist with the graphical construction and evaluation of a model. Discrete-event simulation tools provide a graphical user interface that enables the development of complex models using a graphical modeling language. Knowledge of a programming language is not required to use a discrete-event simulation tool. Examples of tools include SIMUL8 [HP02] and Powersim [Pow01].

Discrete-event simulation libraries assist with the textual construction and evaluation of a model. Discrete-event simulation libraries provide functions that enable the development of complex models using a programming language. Examples of discrete-event simulation libraries include CSIM [Mes98a] [Mes98b], Parsimony [PW99], Maisie [Bag91] [BL94], and Parsec [BMT+98].

5.1.3 Accelerating Discrete-Event Simulation

PDES (Parallel Discrete-Event Simulation) [Fuj90] and DDES (Distributed Discrete-Event Simulation) [Fuj93] are two popular techniques for accelerating discrete-event simulations. A parallel discrete-event simulator uses a parallel computer to compute portions of the entire simulation in parallel. A distributed discrete-event simulator uses a distributed computing platform such as a network of workstations to compute portions of the entire simulation in parallel.

Exploiting parallelism in discrete-event simulation is not a trivial task. If a single event queue is used to manage all events, a communication bottleneck results when a logical process tries to

access the event queue. If each logical process is associated with a distinct event queue, time synchronization across logical processes poses a communication bottleneck.

Researchers have investigated more aggressive techniques for exploiting parallelism and minimizing communication bottlenecks [PML92]. It is possible to build a parallel or distributed discrete-event simulator that does not attempt to process events in order. Such simulators optimistically process events and if it turns out that an event should not have been executed, the simulator rolls back the change to the system state.

5.2 The CSIM Discrete-Event Simulation Library

CSIM is a discrete-event simulation library that permits the rapid development of process-oriented simulations using the C [KR88] and C++ [Str94] programming languages. CSIM began as a research project [Sch86] at the University of Texas at Austin in 1986. Since then, the CSIM discrete-event simulation library has matured into a commercial software package. Today, CSIM is used worldwide by programmers faced with the challenge of simulating complex systems.

5.2.1 The Choice of CSIM

The source code for CSIM v18 was obtained under a non-disclosure agreement. CSIM was chosen for this research for the following reasons:

1. CSIM is an example of a mainstream software application.
2. CSIM is a computationally-intensive application.
3. CSIM is a commercial software package.
4. CSIM is widely used in both industry and academia.
5. CSIM is a mature software package.
6. CSIM is well-documented.

Other applications could have been studied. Several simulation libraries, mathematical libraries, and other software packages were considered. CSIM was chosen based primarily on availability and familiarity.

5.2.2 Modeling Systems with CSIM

Using CSIM, systems are modeled as a set of processes that communicate and interact at discrete points in time. Version 18 of the CSIM C++ Library [Mes98a] [Mes98b] provides a set of classes and member functions that permit the creation, use, and destruction of CSIM objects. These objects form the basic building blocks for a simulation program. Table 5.1 describes the object classes that CSIM provides.

CSIM Object Classes	Description
Processes	CSIM processes encapsulate the active entities in the discrete-event simulation model. A process manager is responsible for scheduling the execution of processes and restoring their context. The four states of process execution are actively computing, ready to start computing, holding until a specified simulation time, or waiting for an event to occur. Several instances of a process may exist simultaneously.
Facilities	CSIM facilities model resources in a simulated system. A facility is a combination of a request queue and one or more servers that service the request queue. When a process requests service from a facility, the request is added to the request queue. The ordering of requests in the queue can vary but typically requests are sorted by process priority. Once the servicing of a request is complete, the process releases the facility. Statistics on the utilization of facilities are automatically maintained by the object class.
Storages	CSIM storages represent resources that can be partially allocated to a requesting process. Storages consist of a request queue and a counter. The counter keeps track of the amount of available storage.
Events	Synchronize process activities
Mailboxes	Facilitate a primitive form of interprocess communication
Table Structures	Collect statistical data during the execution of a simulation
Process Classes	Calculate and report statistics on the execution of a simulation
Streams	Generate streams and statistical distributions of pseudo-random numbers

Table 5.1: CSIM Object Classes

5.2.3 Applications of CSIM

CSIM permits designers of simulations to use native C++ objects and methods to model the behaviour of complex systems. The use of native C++ objects and methods permits CSIM to simulate systems quickly and accurately. CSIM has been successfully used to model Application Specific Integrated Circuit (ASIC) designs, communication systems, transportation systems, and other complex systems.

5.2.4 Profiling the Performance of CSIM

The performance of CSIM was investigated using VTune, a profiling tool developed by Intel. VTune monitors the execution of an application using a set of dedicated hardware registers present in Intel Pentium III processors. VTune can perform both intrusive and non-intrusive tests. Non-intrusive tests were used to sample function execution. For the purpose of analysis, functions were grouped into the following categories:

Process Management - Process management includes all functions related to the creation, scheduling, and completion of processes.

Streams and Distribution Generation - Streams and distribution generation consists of functions to generate pseudo-random numbers and statistical distributions of pseudo-random numbers.

Event Management - Event management functions focus on the creation, scheduling, and completion of events including interprocess communication.

Statistics Generation - Statistics generation includes all functions related to the statistical analysis of the behaviour of simulation runs.

Storage Management - Storage management functions provide a means of allocating, managing, and releasing all resources.

Miscellaneous Activites - Miscellaneous activities includes functions that cannot easily be categorized above.

A simulation model of a simple M/M/1 queue [LK91] with a service time of 2 and an interarrival time of 2 was profiled using VTune. This M/M/1 queue model is provided in Appendix D. This

M/M/1 queue model served as a stress test for the simulation library. By adjusting the total number of arrivals, it was possible to adjust the number of processes, events, and pseudo-random numbers used during the simulation. For the purpose of profiling, a simulation with 1,000,000 arrivals was run three times. Table 5.2 shows a summary of the average results of the profiling analysis.

Functionality	**Percentage of Total Execution Time**
Process Management	46.43%
Streams and Distributions Generation	25.06%
Event Management	11.25%
Statistics Generation	9.74%
Storage Management	5.34%
Miscellaneous Activities	2.18%

Table 5.2: CSIM Profiling Results

For this simulation, CSIM spent 46% of its execution time scheduling and managing its processes. Although process management accounts for a large percentage of the total execution time, each process management task consumes very little time. Process management involves the use of highly-optimized function calls, assembly language routines, and direct register manipulation. As a result, process management is not a good candidate for coprocessing.

Streams and distribution generation was chosen as a candidate for coprocessing. Streams and distribution generation represents approximately 25% of the execution time of CSIM on the simulation investigated. The generation of pseudo-random numbers was targeted for coprocessing. CSIM's pseudo-random number generator is not an ideal algorithm for coprocessing for the following reasons:

1. The algorithm is simple. Pseudo-random number generation is easily implemented on a CISC (Complex Instruction Set Computer) processor so speedup is unlikely.

2. The algorithm uses double-precision floating point arithmetic. While it is possible to implement double-precision floating point arithmetic using a configurable logic device, the difficulty of accomplishing this task is high.

3. The algorithm requires very little execution time. It is not clear whether a configurable logic

device is capable of computing the result faster than a general-purpose processor.

However, CSIM's pseudo-random number generator was a suitable candidate for coprocessing for the following reasons:

1. The algorithm is used often during simulations.
2. The algorithm is easy to understand.
3. The algorithm is self-contained.

Event management was also considered as a possible candidate for coprocessing. It was not chosen due to the fact that it would require substantial modifications to the CSIM library. Approximately 15% of the total lines of source code for CSIM would be directly impacted by the coprocessing of event management functions. The complexity of the event management code also made it a poor candidate for coprocessing. The ARC-PCI Board does not provide sufficient programmable logic resources to implement a large subset of the CSIM event management functions.

5.3 Enhancing CSIM

Pseudo-random number generators are often implemented in hardware. VHDL designs of pseudo-random number generators are publically available on the internet. However, the use of a publically available pseudo-random number generator is unsuitable for the purpose of enhancing CSIM. It is desirable to show that it is possible to obtain the same results with a coprocessed version of CSIM. For this reason, the software algorithm used by CSIM was translated into a hardware implementation specified in VHDL.

5.3.1 Pseudo-Random Number Generation in CSIM

CSIM uses several different pseudo-random number generators of varying complexity. The simplest pseudo-random number generator used by CSIM is a Mixed Linear Congruential Generator [Leh51]. The pseudo-random number generator produces a double precision floating point

value in the range $U(0,1)$. The details of its implementation are not described due to the non-disclosure agreement.

5.3.2 Interfacing with Platform I

A VHDL implementation of CSIM's pseudo-random number generator was designed and tested using MAX+PLUS II. This algorithm was then modified to interface with the reference controller design described in the previous chapter. The device driver and the API for the ARC-PCI Board were modified to add a new function named `arcpci_get_random()`. This function initiates the computation and retrieval of a double precision floating point value from the configurable coprocessor.

Additional hardware and software components are necessary to utilize a configurable coprocessor. A non-coprocessed CSIM application consists of C++ software components. A coprocessed CSIM application consists of C++ software, C software, and VHDL specified hardware components as illustrated in Figure 5.1.

5.3.3 Performance Optimizations

Coprocessor optimizations were investigated to improve the performance of the configurable coprocessor. The coprocessor design for the pseudo-random number generator exploits the fact that pseudo-random numbers are predictable. Rather than generate the pseudo-random number when requested to do so, the number is pre-calculated by the coprocessor and stored in a temporary register. This optimization permits the pseudo-random number to be returned without the need for wait states. Due to the latency associated with PCI bus cycles, there is always sufficient time to calculate the next pseudo-random number prior to receiving the next request. The use of a status register and a tight polling loop is unnecessary. This pre-calculation saves three PCI bus cycles since each random number takes three clock cycles to generate. Using this optimization, performance improves by a factor of 1.375.

Device driver optimizations were also investigated. Two versions of the `arcpci_get_random` function were implemented. The unoptimized version retrieved a single pseudo-random number from the coprocessor. The optimized version retrieved a block of pseudo-random numbers from

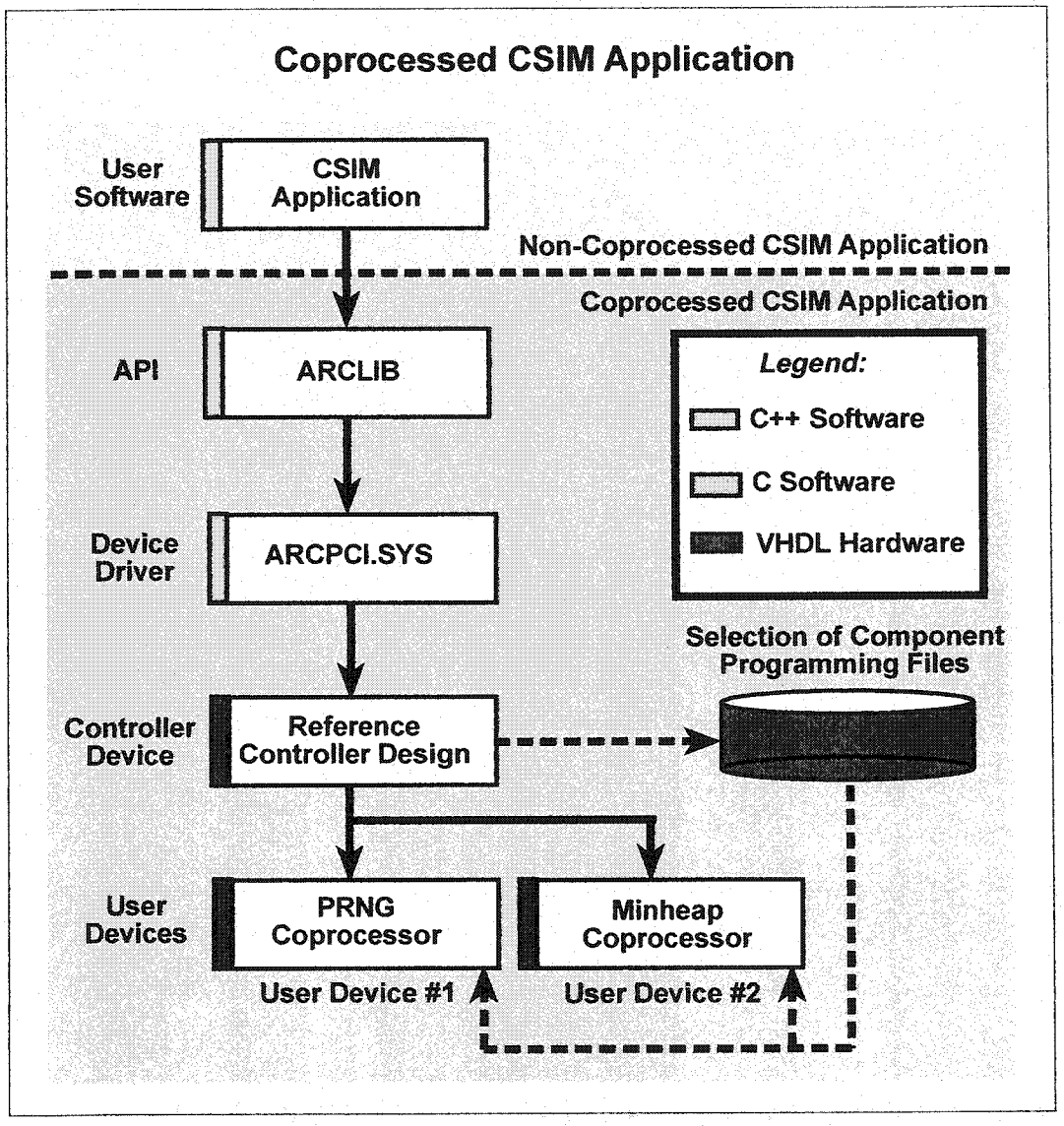


Figure 5.1: Coprocessed CSIM Application

the coprocessor and buffered them accordingly. The optimized version also included support for fast I/O dispatching.

5.4 Experimental Method

A series of experiments were conducted to examine the average time required to simulate the M/M/1 queue model. Three different implementations of the CSIM library were tested. The first implementation used CSIM without any enhancements or modifications. The second implementation used CSIM with an unoptimized configurable coprocessor system for pseudo-random number generation. The third implementation used CSIM with an optimized configurable coprocessor system for pseudo-random number generation. The optimized configurable coprocessor system provided support for fast I/O dispatching and the buffering of transactions. Each version was tested on simulations of varying sizes.

5.5 Platform I: Experimental Results

During testing, the test workstation was isolated from the network and configured as a standalone workstation. The load on the workstation did not vary due to external factors. Small deviations in the execution times of the routines were observed. These deviations were due to variations in the performance of the L1 and L2 caches over time. The average results of the experiments are shown in Table 5.3.

Number of Arrivals	Random Numbers Generated	Normal CSIM Execution Time (in s)	Unoptimized Coprocessed CSIM Execution Time (in s)	Optimized Coprocessed CSIM Execution Time (in s)
1	2	0.010	0.932	0.221
10,000	20,000	0.140	0.952	0.370
100,000	200,000	1.352	6.139	1.893
1,000,000	2,000,000	13.410	57.993	17.305

Table 5.3: CSIM M/M/1 Performance Results

The results for the coprocessed versions of CSIM presented in Table 5.3 include the time required to load, start, stop, and unload the device driver. For small simulation runs, the time required to startup and shutdown the device driver dominates the execution time. For large simulation runs, the time to startup and shutdown the device driver is insignificant. Using the result for the smallest simulation run, it is possible to quantify the time required to startup and shutdown the device driver. Approximately 0.2 s is consumed by this activity.

Device driver optimizations improve performance by a factor of 3.35. This substantial improvement shows that bus and operating system delays are significant[1]. The computation time is constant for both the unoptimized and optimized versions. The computation time represents a small percentage of the total execution time.

5.5.1 Application Speedup

Using the experimental results presented in Table 5.3, it is possible to calculate the application speedups associated with the coprocessed versions of CSIM. For the coprocessed versions of CSIM, the fractional application speedups represent a degradation in performance. These degradations in performance are shown in Table 5.4.

Number of Arrivals	Random Numbers Generated	Normal CSIM Speedup	Unoptimized Coprocessed CSIM Speedup	Optimized Coprocessed CSIM Speedup
1	2	1.000	0.011	0.045
10,000	20,000	1.000	0.147	0.378
100,000	200,000	1.000	0.220	0.714
1,000,000	2,000,000	1.000	0.231	0.773

Table 5.4: CSIM M/M/1 Speedups

Although the performance degradations decrease as the size of the simulation run increases, there is no reason to believe that an application speedup in excess of 1 is possible. In fact, it can be easily shown that application speedup is impossible using this configurable coprocessor. Using the results of Table 4.7, the time required to perform two buffered 32-bit read transactions can be calculated to be approximately 1,160 ns on average. A quick test of the CSIM pseudo-random

[1]The impact of fast I/O dispatching upon performance was not quantified. In practice, fast I/O dispatching would always be used by a device driver to minimize IOCTL overhead.

number generation algorithm indicates that random numbers can be produced in substantially less than 1,160 ns on average[2]. Based on these observations, application speedup is impossible in this case.

5.5.2 Evaluation of System Impact

An evaluation of the system impact was performed using VTune. The execution of all processes was noted and recorded. VTune reported every function call and kernel primitive observed executing during the test runs. The function calls were grouped into CSIM application calls, Windows NT kernel calls, and other application calls to study the system. The raw results of this profiling for the three implementations of CSIM are shown in Table 5.5.

Process	Clock Ticks of Normal CSIM Execution Time	Clock Ticks of Unoptimized Coprocessed CSIM Execution Time	Clock Ticks of Optimized Coprocessed CSIM Execution Time
CSIM Application	13,774	13,550	13,672
Windows NT Kernel	285	3,490	3,076
Other Applications	63	259	177
Totals	14,122	17,299	16,925

Table 5.5: System Profiling Raw Results

Table 5.6 shows the results of system profiling as percentages. It is interesting to note that kernel primitives account for a much larger percentage of clock cycles for the coprocessed systems. The CSIM application consumes more than 75% of all clock cycles available to the system. Thus, this system is referred to as a lightly-loaded system.

These results show that as CSIM spends more time communicating with the configurable coprocessor, the amount of time dedicated to other applications in the system increases. This increase is actually quite substantial considering the fact that the other applications in the system are running in the background. Overall system performance depends upon the mix of active applications. Other applications may benefit indirectly from the coprocessing of an application.

[2]The exact figure has been left out of this thesis to avoid disclosing the details of the algorithm used by CSIM. For purposes of comparison, the pseudo-random number generator presented in the next chapter requires approximately 22.5 ns on average.

Process	Percentage of Normal CSIM Execution Time	Percentage of Unoptimized Coprocessed CSIM Execution Time	Percentage of Optimized Coprocessed CSIM Execution Time
CSIM Application	97.54%	78.33%	80.78%
Windows NT Kernel	2.02%	20.17%	18.17%
Other Applications	0.45%	1.50%	1.05%

Table 5.6: System Profiling Percentages

5.6 Interesting Observations

There are three interesting observations that can be made with respect to this application system and the experimental results obtained. The performance of the application degrades but it is unclear whether the performance of the system improves or degrades. Software optimizations have a greater impact upon the performance of the application than hardware optimizations. Finally, a configurable coprocessor can be used transparently. These observations provide some insight into the complexity of loosely-coupled configurable computing.

5.6.1 Performance

Despite the fact that the application experiences performance degradation, it is unclear whether the host computer experiences an overall performance degradation or not. The use of a configurable coprocessor causes the host computer to perform additional transactions that were not previously required. However, the computations performed by the coprocessor are no longer performed by the host computer. If the time required by the host computer to perform additional transactions is less than the time originally required to perform the computations, the host computer experiences a performance improvement. More computing power is delivered to other applications executing on the host computer as it waits for the coprocessor to perform its computations. The fact that the execution time of the application increases does not necessarily mean that the performance of the entire system degrades. The application may spend more time waiting for the completion of memory and bus transactions but at the same time, other processes in the system may be able to execute.

5.6.2 Impact of Optimizations

The software optimizations performed had a greater impact on performance than hardware optimizations performed. The computation of a pseudo-random number sequence is a simple task. Communication of the result to the host computer is much more complicated. Two I/O operations are required to retrieve a pseudo-random number from the coprocessor. The ratio of communication to computation in this application is very high. By introducing device driver optimizations such as buffering and fast I/O dispatching, performance improved by a factor of 3.35. Coprocessor optimizations only improved performance by a factor of 1.375. This observation regarding the impact of optimizations can likely be extended to other simple calculations with high communication to computation ratios.

5.6.3 Transparency

It is also important to note that the introduction of the configurable coprocessor was completely transparent to the end user. The application source code for the simulation model was the same for all implementations of CSIM tested. The changes to the system were hidden within the CSIM library. All tests produced identical simulation results. Only the performance of the three CSIM implementations differed. This application demonstrated that it is possible to successfully replace a software algorithm with a configurable coprocessor. In addition, this application demonstrated that an end user need not know when a configurable coprocessor is used.

Chapter 6

Application 2: Pseudo-Random Number Generation

This chapter describes the second of three sets of experiments into configurable computing applications. The goal of this experiment was to compare the performance of a simple mainstream software application on loosely-coupled and tightly-coupled configurable computers. A configurable coprocessor for a pseudo-random number generator similar to the one used in CSIM was designed and tested. A series of experiments were conducted on Platforms I and II using this coprocessor. Experimental results illustrate the large performance gap between a loosely-coupled and a tightly-coupled configurable computer system. The coprocessor design degrades the application performance for Platform I but enhances the application performance for Platform II.

6.1 Pseudo-Random Number Generation

Pseudo-Random Number Generators (PRNGs) are algorithms for efficiently obtaining a reproducible sequence of numbers that are distributed uniformly on the interval [0,1]. By definition, it is impossible for any deterministic algorithm to generate a sequence of truly random numbers. The goal of an effective pseudo-random number generator is to produce a sequence of numbers that approximate a realization of Independent and Identically Distributed (IID) $U(0,1)$ random

variables. Given such a sequence, it is possible to construct non-uniform distributions and approximations of random processes. Uniform distributions, non-uniform distributions, and random processes are essential to the numerical modeling of complex systems. For this reason, pseudo-random number generators are an essential component of any discrete-event simulation tool or library.

Perhaps sequences of pseudo-random numbers were best defined by D. H. Lehmer in 1949 when he described them as the following:

> "...a vague notion embodying the idea of a sequence in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests traditional with statisticians and depending somewhat on the use to which the sequence is to be put."

6.1.1 Linear Congruential Generators

Linear Congruential Generators (LCGs) are a popular type of pseudo-random number generator. Introduced by D. H. Lehmer in 1949, LCGs [Leh51] are based on the iterative equation shown in Equation 6.1.

$$Z_i = (aZ_{i-1} + c)(\text{mod } m) \tag{6.1}$$

The choice of constants m, a, c, and Z_0 impact the effectiveness of the algorithm. Typically, m is set to 2^b where b is the number of bits in the integer representation. If a and c are chosen judiciously, it is possible to ensure the LCG has a full period. This property means every value between 0 and $m-1$ occurs exactly once in the first m iterations of the LCG. In 1964, Hull and Dobell [HD62] proved Theorem 1 that provides constraints on appropriate choices of a and c.

Theorem 1 (Hull and Dobell) *The LCG defined in Equation 6.1 has a full period if and only if the following three conditions hold:*

1. *The only positive integer that (exactly) divides both m and c is 1.*
2. *If q is a prime number (divisible by only itself and 1) that divides m, then q divides $a-1$.*

3. If 4 *divides* m, *then* 4 *divides* $a - 1$.

If the increment c is non-zero, the LCG is referred to as a Mixed Linear Congruential Generator. Mixed LCGs can have a full period since c is non-zero. If the increment c is 0, the LCG is referred to as a Multiplicative Linear Congruential Generator. Multiplicative LCGs have reduced computational requirements since the addition of c is not required.

6.1.2 The Choice of Pseudo-Random Number Generation

Pseudo-random number generation was chosen as an application for this research for the following reasons:

1. Pseudo-random number generation is an algorithm used by many mainstream software applications including web browsers, spreadsheet packages, database managers, simulators, and operating systems.
2. Pseudo-random number generation is easy to understand.
3. Pseudo-random number generation can be performed successfully using the programmable logic devices available on the target platforms

Pseudo-random number generation is not an ideal application for configurable coprocessing. Although well suited to implementation in programmable hardware, the ratio of communication to computation required is high. Results must be communicated frequently between the coprocessor and the host computer. If the communication latency is large, it is impossible to improve the performance of an application.

6.2 Enhancing Pseudo-Random Number Generation

The pseudo-random number generator used in this application differed from the one used in the CSIM tests. A Mixed Linear Congruential Generator was implemented to produce a signed 32-bit pseudo-random number. This generator used the following constants:

$$m = 2^{32} \tag{6.2}$$

$$a = 1103515245 \tag{6.3}$$

$$c = 12345 \tag{6.4}$$

$$Z_0 = -2075277215 \tag{6.5}$$

The Mixed Linear Congruential Generator coprocessor implemented for the CSIM tests was reused to reduce the amount of time required to develop the coprocessor hardware. The design used in the CSIM experiments was modified to use the constants shown above. Also, the hardware to convert the result to a double precision floating point representation was removed. The modified design was specified in a subset of VHDL suitable for synthesis for both target platforms.

The VHDL design uses a single finite state machine to interface with the processor (via the PCI bus for Platform I and via the Avalon bus for Platform II) and precalculate pseudo-random numbers one at a time. The finite state machine activates when the interface of the coprocessor is read. At the end of each transaction, the finite state machine waits for the read signal to become deasserted. The finite state machine for the pseudo-random number generator is shown in Figure 6.1.

6.2.1 Interfacing with Platform I

For Platform I, the configurable coprocessor did not require any special modifications to interface with the host computer via the reference controller design. The coprocessor design was very similar to the design used for the CSIM experiments. However, unlike the previous design, this design only required a single 32-bit register to store the pseudo-random number. The number is stored as an unsigned 32-bit value rather than a double-precision floating point value. This meant that the address did not need to be fully decoded. All interface reads resulted in the same function being performed.

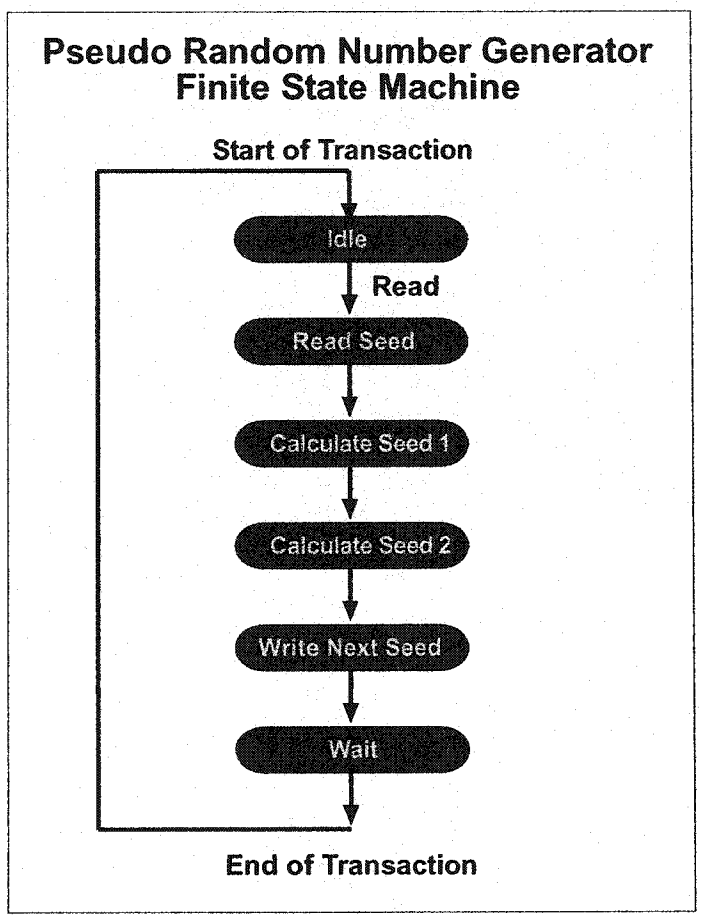


Figure 6.1: Pseudo-Random Number Generator Finite State Machine

6.2.2 Interfacing with Platform II

For Platform II, several hardware modifications to the configurable coprocessor were necessary. First, the pins were renamed to work with the Avalon bus of the Nios processor. Next, several pins were deleted from the coprocessor design since the Avalon bus is simpler than the bus hierarchy provided on the ARC-PCI Board of Platform I. Finally, the read signal used to indicate the start of a coprocessor transaction for Platform I was replaced with equivalent handshaking signals for Platform II.

From a software standpoint, each API function call used in Platform I was replaced with a memory-mapped I/O transaction to a control/status register for Platform II. These modifications were straightforward. The API used for Platform I could have been ported to Platform II. This was not deemed necessary since the software modifications required to accommodate the Avalon bus were simple.

Testing revealed some subtle timing issues. These timing problems were debugged using the built-in logic analyzer support provided by Quartus II. In several cases, flip-flop triggering was changed to falling-edge triggering from rising-edge triggering to satisfy the timing constraints of the Avalon bus. The use of both clock edges effectively halves the permissible clock period. The use of both clock edges was possible since the programmable logic device used in Platform II is significantly faster than the devices used in Platform I.

The interfacing differences between Platform I and Platform II were relatively small since both platforms use memory-mapped I/O. The differences might have been much more noticeable if the reference controller design of Platform II did not use memory-mapped I/O. The complications of PCI bus interfacing were hidden by the reference controller design used in Platform I. The handshaking signals provided by the reference controller design used in Platform I were easily mapped onto equivalent signals on the Avalon bus used in Platform II. This was a coincidence.

6.2.3 Performance Optimizations

The coprocessor design consists of a simple finite state machine that reads the current seed, calculates the next seed, and stores the next seed. Pseudo-random numbers are precalculated to save clock cycles. Approximately 4 clock cycles are saved by precalculating results since 4 clock

cycles are consumed by calculating the next seed and storing it. This optimization allowed the coprocessor to respond to transactions within 2 clock cycles. This optimization was applied on both platforms.

6.2.4 Platform I Performance Optimizations

For Platform I, transactions were buffered to improve performance. A new function had to be added to the API to support buffering. This function, `arcpci_get_random2()`, retrieves the next pseudo-random number in the sequence from a local buffer in memory. If the local memory is empty, the function requests the next 16 pseudo-random numbers using one buffered transaction.

6.2.5 Platform II Performance Optimizations

Due to the simplicity of the system, no Platform II specific optimizations were performed on the coprocessor design. Each call to the coprocessor required only a single line of C code. The coprocessor design was already optimized.

6.3 Experimental Method

A series of experiments were conducted to examine the average time required to generate pseudo-random numbers on Platform I and Platform II. For Platform I, the impacts of caching and buffering upon application performance were also investigated. For Platform II, caching and buffering were not applicable since the platform did not incorporate a cache or an operating system.

6.4 Platform I: Experimental Results

The following experimental results represent averages of five test runs of the application system on Platform I. For the entire set of test results, refer to Appendix B. Unless otherwise noted, results shown in gray represent estimates of performance rather than actual performance results.

Estimates are provided for simulation runs that could not be performed due to excessive execution times. These estimates were obtained by doing a linear regression based on the data points available. The Microsoft Excel FORECAST function was used to perform this regression when necessary.

6.4.1 Unbuffered Test Results

Table 6.1 summarizes the average time required to generate a set of pseudo-random numbers on Platform I without the use of buffering. For pseudo-random number generation, buffering would always be used to reduce average latency. However, buffering is not possible for all applications so it is interesting to examine the impact of buffering on the performance of the application.

PC RAND Test Results With Caching			
PRNG Iterations	PRAND1 Software (in s)	PRAND2 Unbuffered (in s)	Unbuffered Speedup PRAND1 / PRAND2
500000	0.013	1.274	0.010
1000000	0.023	2.544	0.009
2500000	0.057	6.349	0.009
5000000	0.113	12.696	0.009
10000000	0.224	25.409	0.009
25000000	0.561	63.517	0.009
50000000	1.125	127.015	0.009
100000000	2.257	254.041	0.009
250000000	5.628	635.093	0.009
500000000	11.270	1270.060	0.009

Table 6.1: Unbuffered Test Results

6.4.2 Buffered Test Results

Table 6.2 summarizes the average time required to generate a set of pseudo-random numbers on Platform I with the use of buffering. This buffering effectively reduces the average transaction time by reducing the number of context switches performed by the operating system.

PC RAND Test Results With Caching			
PRNG Iterations	PRAND1 Software (in s)	PRAND3 Buffered (in s)	Buffered Speedup PRAND1 / PRAND3
500000	0.013	0.290	0.046
1000000	0.023	0.589	0.040
2500000	0.057	1.456	0.039
5000000	0.113	2.912	0.039
10000000	0.224	5.820	0.038
25000000	0.561	14.559	0.039
50000000	1.125	29.122	0.039
100000000	2.257	58.236	0.039
250000000	5.628	145.589	0.039
500000000	11.270	291.167	0.039

Table 6.2: Buffered Test Results

6.4.3 Unbuffered Test Results on an Uncached System

Table 6.3 summarizes the average time required to generate a set of pseudo-random numbers on Platform I without the use of buffering and without the use of L1 and L2 caches. This table can be compared with Table 6.1 to determine the impact of caching upon this system. In practice, caching is always desirable. However, these results provide some insight into the significance of the effect of caching. The disabling of the cache also reduces the memory bandwidth available to the processor so that it is more comparable to the memory bandwidth available to the coprocessor. These results can be used to quantify the difference between the memory bandwidth available to the processor and the memory bandwidth available to the coprocessor.

6.4.4 Buffered Test Results on an Uncached System

Table 6.4 summarizes the average time required to generate a set of pseudo-random numbers on Platform I with the use of buffering and without the use of L1 and L2 caches. This table can be compared with Table 6.2 to determine the impact of caching upon this system.

PC RAND Test Results Without Caching			
PRNG Iterations	PRAND1NC Software (in s)	PRAND2NC Unbuffered (in s)	Unbuffered Speedup PRAND1NC / PRAND2NC
500000	2.931	143.913	0.020
1000000	5.822	283.289	0.021
2500000	14.654	706.378	0.021
5000000	30.076	1412.521	0.021
10000000	59.308	2825.106	0.021
25000000	148.500	7062.061	0.021
50000000	295.722	14125.656	0.021
100000000	574.249	28252.845	0.020
250000000	1428.407	70634.414	0.020
500000000	2856.381	141270.362	0.020

Table 6.3: Unbuffered Test Results on an Uncached System

PC RAND Test Results Without Caching			
Maximum Entries	PRAND1NC Software (in s)	PRAND3NC Buffered (in s)	Buffered Speedup PRAND1NC / PRAND3NC
500000	2.931	5.792	0.506
1000000	5.822	11.575	0.503
2500000	14.654	29.577	0.495
5000000	30.076	58.434	0.515
10000000	59.308	117.483	0.505
25000000	148.500	293.153	0.507
50000000	295.722	586.537	0.504
100000000	574.249	1172.632	0.490
250000000	1428.407	2931.880	0.487
500000000	2856.381	5861.775	0.487

Table 6.4: Buffered Test Results on an Uncached System

Impact of L1 and L2 Caching on Test Results									
PRNG Iterations	PRAND1NC Uncached Software (in s)	PRAND1 Cached Software (in s)	Performance Gain PRAND1NC / PRAND1	PRAND2NC Uncached Unbuffered (in s)	PRAND2 Cached Unbuffered (in s)	Performance Gain PRAND2NC / PRAND2	PRAND3NC Uncached Buffered (in s)	PRAND3 Cached Buffered (in s)	Performance Gain PRAND3NC / PRAND3
500000	2.931	0.013	219.825	143.913	1.274	112.979	5.792	0.290	19.946
1000000	5.822	0.023	249.500	283.289	2.544	111.373	11.575	0.589	19.658
2500000	14.654	0.057	258.606	706.378	6.349	111.255	29.577	1.456	20.311
5000000	30.076	0.113	265.379	1412.521	12.696	111.255	58.434	2.912	20.065
10000000	59.308	0.224	265.164	2825.106	25.409	111.187	117.483	5.820	20.185
25000000	148.500	0.561	264.864	7062.061	63.517	111.183	293.153	14.559	20.136
50000000	295.722	1.125	262.864	14125.656	127.015	111.213	586.537	29.122	20.141
100000000	574.249	2.257	254.468	28252.845	254.041	111.214	1172.632	58.236	20.136
250000000	1428.407	5.628	253.789	70634.414	635.093	111.219	2931.880	145.589	20.138
500000000	2856.381	11.270	253.457	141270.362	1270.060	111.231	5861.775	291.167	20.132

Table 6.5: Impact of Caching on Performance

6.4.5 Measuring the Impact of Caching

Table 6.5 compares the application execution times observed on Platform I. This table quantifies the impact of caching upon each of the application systems.

It should be noted that caching plays a more significant role in an application system that does not use a configurable coprocessor. The use of a configurable coprocessor partitions the application data into two different memory spaces. Only a portion of the application data can be cached. Overall, the caching of application data is less effective. Also, systems with a configurable coprocessor spend less time accessing local memory and more time performing I/O operations. Caching plays a smaller role in the overall performance of a system with a configurable coprocessor.

6.5 Platform II: Experimental Results

The experimental results in Table 6.6 represent averages of five test runs of the application system on Platform II. For the entire set of test results, refer to Appendix B. The lock-step tests perform the generation of one pseudo-random number at a time. The next computation does not start until the previous calculation has concluded. Barrier synchronization is used by the software.

These experimental results show that a speedup of 2.691 is obtained for pseudo-random number generation on Platform II. This speedup does not vary from one run to the next. The constant

Excalibur RAND Test Results			
PRNG Iterations	ERAND1 Software (in s)	ERAND2 Lock-Step (in s)	Lock-Step Speedup ERAND1 / ERAND2
500000	1.382	0.513	2.691
1000000	2.763	1.027	2.691
2500000	6.909	2.567	2.691
5000000	13.817	5.134	2.691
10000000	27.634	10.269	2.691
25000000	69.086	25.671	2.691
50000000	138.173	51.343	2.691
100000000	276.345	102.686	2.691
250000000	690.862	256.714	2.691
500000000	1381.724	513.428	2.691

Table 6.6: Platform II Test Results

speedup results from the fact that this platform only runs a single program at a time and caches are not used. There is no reason for the performance to vary from one run to the next.

6.6 Interesting Observations

There are three interesting observations that can be made with respect to this application system and the experimental results obtained. Bus utilization delays, memory utilization delays, and the relative performance of processors are factors that play a significant role in determining the performance of a coprocessed application. For a loosely-coupled configurable system, it is difficult if not impossible to overcome all of these factors, particularly for a simple application. For a tightly-coupled configurable system, the impact of these factors is less and speedup of a simple application is possible.

6.6.1 Bus Utilization Delays

Platform I has a high communication latency and Platform II has a low communication latency. The communication latency is proportional to the strength of the coupling between the host

computer and the configurable coprocessor board. For simple coprocessed algorithms, a high communication latency dominates the execution time. Platforms with a low communication latency are suitable for coprocessing simple algorithms. Platforms with a high communication latency are unsuitable for coprocessing simple algorithms. Higher communication latency leads to higher bus utilization delays.

6.6.2 Memory Utilization Delays

Caching plays a major role in the performance of applications. Platform I provides a processor cache that is an order of magnitude faster than the memory available to the coprocessor. This difference in memory bandwidth is very significant. A comparison of the experimental results with caching disabled to those with caching enabled provides an indication of the magnitude of this difference. Platform II lacked a processor cache but registers gave the equivalent of cache performance. Effectively, the bandwidth available to the coprocessor was approximately 5 times that available to the processor. Memory bandwidth gaps between the processor and the coprocessor must be offset by an equivalent performance improvement resulting from increased hardware specialization and increased parallelism.

6.6.3 Relative Performance of Processors

The relative performance of the processor to the configurable coprocessor is also a significant factor in determining application performance. The clock rate of the processor in Platform I is an order of magnitude faster than the clock rate of the configurable coprocessor design. Unless the configurable coprocessor design uses fewer than one tenth the clock cycles required by the processor to compute the same result, a performance gap exists between the processor and the configurable coprocessor. A performance gap between the processor and the configurable coprocessor must be offset by an equivalent performance improvement resulting from increased parallelism and/or other sources of performance improvements. For Platform II, the processor and the configurable coprocessor clock at the same frequency. The configurable coprocessor outperforms the processor if it can compute the same result in fewer clock cycles without introducing other sources of delays. The configurable coprocessor is more likely to outperform the processor on Platform II.

Chapter 7

Application 3: Minheap Management

This chapter describes the third of three sets of experiments into configurable computing applications. The goal of this experiment was to compare the performance of a complex mainstream software application executing on a loosely-coupled and a tightly-coupled configurable computer. A configurable coprocessor for minheap management was designed and tested. This coprocessor provides the capability to insert an item into a minheap and remove the minimum item from the minheap. A series of experiments were conducted on Platforms I and II using this coprocessor. Experimental results illustrate the large performance gap between loosely-coupled and tightly-coupled configurable computer systems. The results also indicate that the complex task of minheap management can potentially benefit substantially from configurable coprocessing. The coprocessor design degrades the application performance of Platform I under most circumstances but significantly enhances the application performance of Platform II.

7.1 Introduction to Minheap Management

A minheap [Pre99] is a data structure that can be used to maintain a partially ordered list of items. The data structure is a special form of a binary tree that always stores the item with the

smallest key value at the root of the tree. Unlike a linked list, a minheap performs a balancing step upon insertion so that the item with the smallest key value can be retrieved in constant time. After each retrieval, the minheap must perform a balancing step. The worst-case performance of a balancing step is logarithmic. Minheaps are often used in mainstream software applications. Minheaps can be used to create event queues, sort lists of items, schedule processes, and store data efficiently.

A special class of minheap is the binary minheap. A binary minheap is a heap-ordered binary tree that always forms a complete tree. This class of minheap can be implemented using an array. A binary minheap uses memory resources as efficiently as possible. For the purpose of this thesis, the term minheap will be used to refer to a binary minheap.

7.1.1 The Choice of Minheap Management

Minheap management was chosen as an application for this research for the following reasons:

1. Minheaps are data structures used by many mainstream software applications including spreadsheet packages, word processors, database managers, simulators, and operating systems.
2. Minheaps use memory resources efficiently
3. Minheaps are relatively complex to manage

7.2 Enhancing Minheap Management

Successfully implementing a configurable coprocessor design for minheap management was a significant challenge. The VHDL design required approximately twelve months of effort to produce a design that was suitable for both target platforms. Platform I posed the biggest challenge. A functionally correct design was created in a few days. However, this design did not meet the timing requirements of the PCI bus. The fixed pinouts of the ARC-PCI Board make it difficult to obtain 33 MHz performance for user designs. After several attempts, a set of optimizations was found that enabled the development of a working design for Platform I. This design consisted of

three interacting finite state machines. The top-level finite state machine interfaced the minheap management functions to the processor. This finite state machine is shown in Figure 7.1.

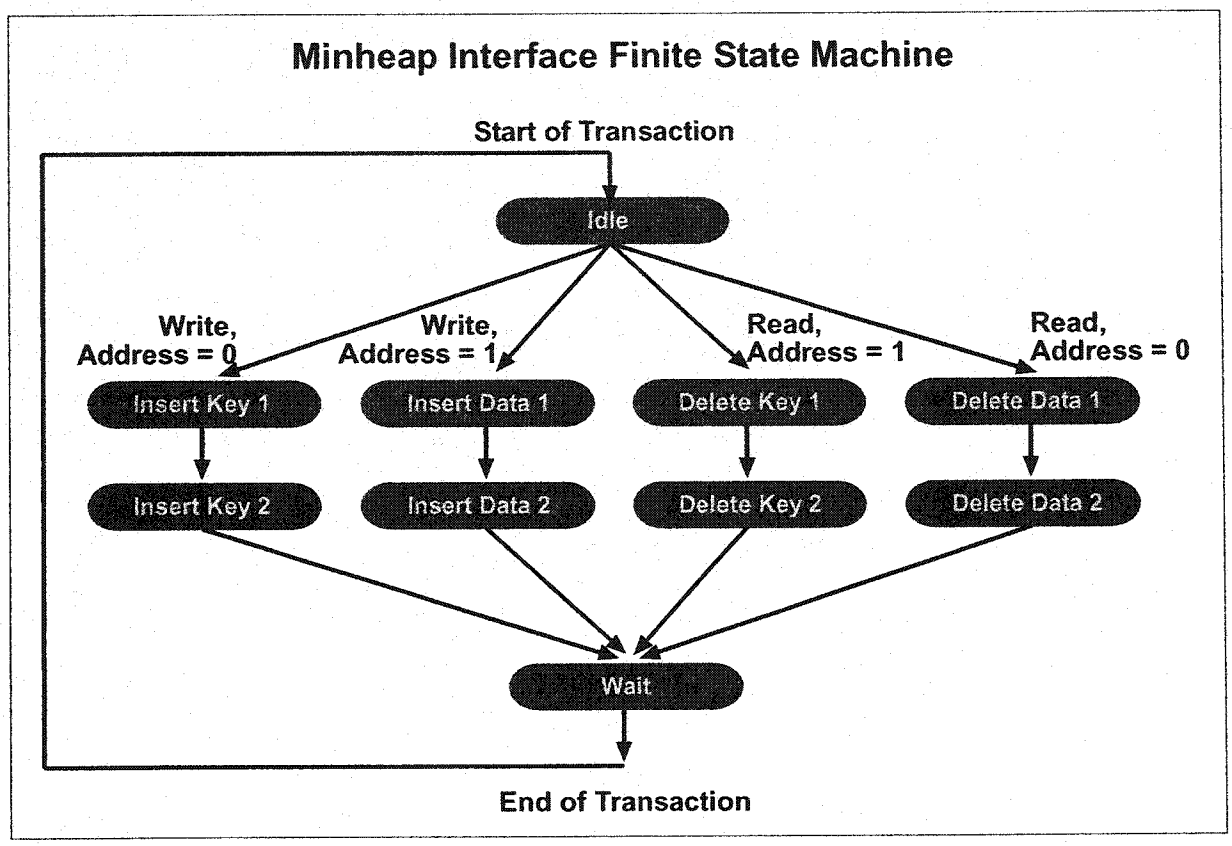


Figure 7.1: Minheap Interface Finite State Machine

Two minheap management functions were implemented. Each one was associated with a finite state machine. The insertion finite state machine consisted of 14 states. The deletion finite state machine consisted of 20 states. These finite state machines were developed by translating a C implementation of minheap insertion and deletion algorithms into a sequence of atomic actions. Each action was assigned to a state. A dataflow approach was then used to merge as many states as possible.

7.2.1 Interfacing with Platform I

The minheap coprocessor was designed to interface with Platform I. The reference controller design simplified this task. Two control / status registers were implemented in the user design. The first register mapped onto the key of the heap entry and the second register mapped onto the data of the heap entry. During an insertion, the key is written to the first control register. Then, the data is written to the second control register and the insertion process automatically starts. During a deletion the key is read from the first status register. Then the data is read from the second control register and the deletion process automatically starts. The deletion process ensures that the next entry to delete is stored in the control registers at the conclusion of the deletion process.

7.2.2 Interfacing with Platform II

One very important hardware change to the coprocessor interface was necessary. During testing, it became apparent that the Nios processor was capable of generating requests to insert and delete minheap entries faster than the coprocessor could process the requests. A status register had to be added to the design to allow the Nios processor to check if the previous request had been processed successfully. Also, it was necessary to ensure that the software delayed for 1 clock cycle after initiating a request. If the status register was polled immediately following the generation of a request, it was possible for the Nios processor to assume that the previous request had completed successfully. This timing problem was extremely difficult to debug.

7.2.3 Hardware Optimizations

The most significant hardware optimization was the use of three communicating finite state machines rather than a single monolithic finite state machine. Without this hardware optimization, it would have been impossible to achieve 33 MHz performance on Platform I. Despite this significant optimization, hand coding of the state assignments was necessary to ensure adequate performance from one compilation to the next. One-hot state assignments did not deliver adequate performance. Several hand coded assignments had to be compiled and tested before a suitable set of assignments was found. This hand coding was tedious.

7.2.4 Performance Optimizations

Parallelism was exploited through two simple optimizations. The insertion process completes the bus write prior to the actual completion of the insertion process. Effectively, much of the work associated with the transaction is handled by post-processing (t_{POSTB2}). The minheap is balanced after the bus write completes. This permits the software to resume its computations (t_{EXEA1}) while the coprocessor continues its computations. Similarly, the deletion process completes the bus read prior to the actual completion of the deletion process. The minheap is balanced in a post-processing step (t_{POSTB2}) after the bus read completes. This permits the software to compute in parallel (t_{EXEA1}) with the coprocessor.

7.3 Experimental Method

A series of experiments were conducted to examine the average time required to insert and delete heap entries on Platform I and Platform II. For Platform I, the impacts of caching and buffering upon application performance were investigated. For Platform II, caching and buffering were not applicable since the platform did not incorporate a cache or an operating system.

Insertion and deletion were not investigated separately. Instead, the minheap management functions were tested in conjunction to simplify the repeated testing of the system. Entries were inserted into the minheap until a specified maximum number of entries was reached. Then, all of the entries were deleted and the process was repeated. In total, this sequence of insertions and deletions was performed a total of 5,000,000 times to produce execution times in a reasonable range.

7.4 Platform I: Experimental Results

The following experimental results represent an average of five test runs of the application system. For the entire set of test results, refer to Appendix C. Unless otherwise noted, results shown in gray represent estimates of performance rather than actual performance results. Estimates are provided for simulation runs that could not be performed due to insufficient memory.

PC MIN Test Results With Caching			
Maximum Entries	PMIN1 Software (in s)	PMIN2 Unbuffered (in s)	Unbuffered Speedup PMIN1 / PMIN2
50	0.192	2.447	0.079
100	0.415	5.141	0.081
250	1.184	13.756	0.086
500	2.580	28.878	0.089
1000	5.562	60.473	0.092
2500	15.448	155.259	0.099
5000	33.981	313.236	0.108
10000	74.878	629.190	0.119
25000	211.602	1577.052	0.134
50000	462.583	3156.822	0.147

Table 7.1: Unbuffered Test Results

7.4.1 Unbuffered Test Results

Table 7.1 summarizes the average time required to execute the test application on Platform I without the use of buffering.

7.4.2 Buffered Test Results

Table 7.2 summarizes the average time required to execute the test application on Platform I with the use of buffering. This buffering effectively reduces the average transaction time by reducing the number of context switches performed by the operating system.

7.4.3 Unbuffered Test Results on an Uncached System

Table 7.3 summarizes the average time required to execute the test application on Platform I without the use of buffering and without the use of L1 and L2 caches. This table can be compared with Table 7.1 to determine the impact of caching upon this system.

PC MIN Test Results With Caching			
Maximum Entries	PMIN1 Software (in s)	PMIN3 Buffered (in s)	Buffered Speedup PMIN1 / PMIN3
50	0.192	1.969	0.098
100	0.415	3.689	0.112
250	1.184	9.275	0.128
500	2.580	19.941	0.129
1000	5.562	41.922	0.133
2500	15.448	107.868	0.143
5000	33.981	217.777	0.156
10000	74.878	437.595	0.171
25000	211.602	1097.049	0.193
50000	462.583	2196.139	0.211

Table 7.2: Buffered Test Results

PC MIN Test Results Without Caching			
Maximum Entries	PMIN1NC Software (in s)	PMIN2NC Unbuffered (in s)	Unbuffered Speedup PMIN1NC / PMIN2NC
50	39.477	139.320	0.283
100	87.668	279.472	0.314
250	247.468	698.038	0.355
500	540.587	1399.040	0.386
1000	1164.146	2798.021	0.416
2500	3216.345	6994.966	0.460
5000	6858.937	13989.874	0.490
10000	14583.783	27979.690	0.521
25000	39241.845	69949.138	0.561
50000	84881.336	139898.218	0.607

Table 7.3: Unbuffered Test Results on an Uncached System

PC MIN Test Results Without Caching			
Maximum Entries	PMIN1NC Software (in s)	PMIN3NC Buffered (in s)	Buffered Speedup PMIN1NC / PMIN3NC
50	39.477	21.621	1.826
100	87.668	38.097	2.301
250	247.468	87.512	2.828
500	540.587	175.012	3.089
1000	1164.146	349.104	3.335
2500	3216.345	871.381	3.691
5000	6858.937	1741.842	3.938
10000	14583.783	3482.764	4.187
25000	39241.845	8705.530	4.508
50000	84881.336	17410.140	4.875

Table 7.4: Buffered Test Results on an Uncached System

7.4.4 Buffered Test Results on an Uncached System

Table 7.4 summarizes the average time required to execute the test application on Platform I with the use of buffering and without the use of L1 and L2 caches. This table can be compared with Table 7.2 to determine the impact of caching upon this system.

7.4.5 Measuring the Impact of Caching

Table 6.5 compares the application execution times observed on Platform I. This table quantifies the impact of caching upon each of the application systems. Caching improves the performance of the non-coprocessed system by a factor of 209 for the 1000 maximum entries test. Caching impacts the optimized coprocessed system by a factor of 8 for the 1000 maximum entries test. Caching plays a much more significant role in the performance of a non-coprocessed application.

Impact of L1 and L2 Caching on Test Results									
Maximum Entries	PMIN1NC Uncached Software (in s)	PMIN1 Cached Software (in s)	Performance Improvement PMIN1NC / PMIN1	PMIN2NC Uncached Unbuffered (in s)	PMIN2 Cached Unbuffered (in s)	Performance Improvement PMIN2NC / PMIN2	PMIN3NC Uncached Buffered (in s)	PMIN3 Cached Buffered (in s)	Performance Improvement PMIN3NC / PMIN3
50	39.477	0.192	205.182	139.320	2.447	56.926	21.621	1.969	10.982
100	87.668	0.415	211.452	279.472	5.141	54.357	38.097	3.689	10.327
250	247.468	1.184	209.080	698.038	13.756	50.745	87.512	9.275	9.435
500	540.587	2.580	209.546	1399.040	28.878	48.447	175.012	19.941	8.777
1000	1164.146	5.562	209.303	2798.021	60.473	46.269	349.104	41.922	8.327
2500	3216.345	15.448	208.202	6994.966	155.259	45.053	871.381	107.868	8.078
5000	6858.937	33.981	201.847	13989.874	313.236	44.662	1741.842	217.777	7.998
10000	14583.783	74.878	194.768	27979.690	629.190	44.469	3482.764	437.595	7.959
25000	39241.845	211.602	185.451	69949.138	1577.052	44.354	8705.530	1097.049	7.935
50000	84881.336	462.583	183.494	139898.218	3156.822	44.316	17410.140	2196.139	7.928

Table 7.5: Impact of Caching on Performance

7.5 Platform II: Experimental Results

The experimental results in Table 7.6 represent averages of five test runs of the application system on Platform II using an unoptimized application. For the entire set of test results, refer to Appendix C.

These experimental results show that a speedup of slightly more than an order of magnitude was achieved for large minheaps. This speedup depends upon the size of the minheap tested.

The experimental results in Table 7.7 represent averages of five test runs of the application system on Platform II using an optimized application. This application exploited parallelism aggressively by computing the next heap entry prior to verifying the completion of the previous coprocessor transaction.

The exploitation of parallelism resulted in a substantial improvement in performance. A performance improvement of approximately 40% was obtained. This permitted a speedup by a factor of almost 14 for large minheaps.

7.6 Interesting Observations

Three interesting observations can be made with respect to this application system and the experimental results obtained. The algorithm complexity, memory utilization delays, and optimizations

Excalibur MIN Test Results			
Maximum Entries	EMIN1 Software (in s)	EMIN2 Lock-Step (in s)	Lock-Step Speedup EMIN1 / EMIN2
50	14.765	1.612	9.157
100	32.133	3.361	9.559
250	88.767	8.921	9.950
500	190.644	18.710	10.189
1000	407.414	39.191	10.396
2500	1105.996	100.634	10.990
5000	2342.113	203.039	11.535
10000	4944.313	407.849	12.123
25000	13224.603	1022.279	12.936
50000	27025.086	2046.329	13.207

Table 7.6: Platform II Blocking Test Results

Excalibur MIN Test Results			
Maximum Entries	EMIN1 Software (in s)	EMIN3 Parallel (in s)	Parallel Speedup EMIN1 / EMIN3
50	14.765	1.469	10.054
100	32.133	2.934	10.950
250	88.767	7.336	12.100
500	190.644	14.681	12.986
1000	407.414	29.797	13.673
2500	1105.996	75.146	14.718
5000	2342.113	150.727	15.539
10000	4944.313	301.889	16.378
25000	13224.603	755.375	17.507
50000	27025.086	1511.185	17.883

Table 7.7: Platform II Non-Blocking Test Results

play a significant role in determining the performance of a coprocessed application. For a more complex application such as minheap management, both loosely-coupled and tightly-coupled configurable computers are potentially suitable.

7.6.1 Algorithm Complexity

As the algorithm complexity increases, the benefit of a configurable coprocessor increases provided that the processor can continue to compute in parallel with the coprocessor. Insertions and deletions on large minheaps require more computations than insertions and deletions on small minheaps. As the size of the minheap increases, so does the benefit of the configurable coprocessor. It is unclear if this benefit eventually diminishes due to other factors. Very large minheaps would need to be tested to see if the performance improvements eventually reach a limit. Larger minheaps could not be tested due to insufficient memory on the target platforms.

7.6.2 Memory Utilization Delays

Caching plays a major role in the performance of applications. When the cache is disabled, the configurable coprocessor can actually provide a speedup on Platform I. This was unexpected. Part of the reason for this performance improvement is the fact that the processor thrashes. The processor performs additional operations that would be unnecessary if caching had been enabled. However, there is another reason for this performance improvement. The disabling of the L1 and L2 cache effectively equates the memory bandwidth of the processor with the memory bandwidth associated with the coprocessor. Given similar memory bandwidth, memory utilization delays do not significantly impact the performance of the coprocessor. With caching enabled, memory utilization delays play a significant role in the performance of the coprocessed system.

7.6.3 Hardware Optimizations vs. Coprocessor Optimizations

Hardware optimizations are often made out of necessity. The design must be optimized at the hardware level to provide an adequate level of performance. These optimizations actually have little impact upon the overall performance of the system. However, coprocessor optimizations can have a profound impact upon the overall performance of the system. A small improvement

to the software portion of the system can result in a significant performance improvement. This difference is demonstrated by the 40% performance improvement achieved on Platform II simply by reordering the instruction sequence.

Chapter 8

Model Validation

Using the experimental results presented in the previous chapters, it is possible to estimate suitable values for the performance model timing parameters introduced in Chapter 3. These estimates permit the validation of the performance model. The strengths and weaknesses of the performance model are revealed by the validation process. Using the validated model, it is possible to assess the desired features of coupled configurable coprocessors.

8.1 Timing Parameter Estimation

The execution time of an application is rarely fixed. Caching, multitasking, input data, and user interaction are some of the events that lead to changes in the behaviour of an application. The performance model introduces a set of timing parameters to estimate the execution time of an application. In particular, this model permits the evaluation of the benefit of coprocessing on a particular application. The timing parameters depend on the application being studied, the state of other applications in the system, and the state of the system. Estimates can be made for minimum, maximum and typical values for a particular application. These estimates provide insight into the execution of the coprocessed application and its impact upon the system.

8.1.1 Assumptions

For the purpose of estimating the parameters for the performance model, the following assumptions are made:

1. A non-coprocessed application requires 1000 s of system execution time (t_{SYS1}). This includes the lumped effects of memory utilization delays, bus utilization delays, and operating system behaviour delays.

2. Configuration delays are fixed. The complexity of the coprocessor does not impact the time required to configure the coprocessor.

3. The lumped effects of memory utilization delays, bus utilization delays, and operating system behaviour delays account for approximately 160 s of the system execution time, with the following breakdown:

   - Memory utilization delays account for more time than bus utilization delays and operating system behaviour delays since the computer system is assumed to be lightly-loaded.
   - Memory utilization delays (t_{MEM1}) account for 100 s of the system execution time.
   - Bus utilization delays (t_{BUS1}) account for 30 s of the system execution time.
   - Operating system behaviour delays (t_{OS1}) account for 30 s of the system execution time.

4. Coprocessors may compute results faster or slower than a processor. A coprocessor may be an order of magnitude slower ($t_{SYS2} >> t_{SYS1}$) or faster ($t_{SYS2} << t_{SYS1}$), depending upon the application.

The choice of 1000 s for system execution time is arbitrary. If a smaller or larger value is chosen, the delays assumed for memory utilization, bus utilization, and operating system behaviour must be scaled appropriately. The choice of 160 s for the lumped effects of memory utilization delays, bus utilization delays, and operating system behaviour delays is based on the system profiling results obtained for CSIM as indicated in Table 5.6. This table shows the introduction of a coprocessor increases kernel usage by 16% on a lightly-loaded computer. Therefore, it has been

assumed that these lumped delays account for approximately 16% of the system execution time. This percentage does vary from one application to another.

To account for variations in system performance, the validation of the performance model considers minimum, maximum, and typical values for all timing parameters. A large range of minimum and maximum values are presented to show the impact of the large variability associated with application performance. Alternatively, variables could have been used for the purpose of this validation. However, the use of variables would make it more difficult to illustrate how predicted results compare with actual results.

It should be noted that since the system is assumed to be lightly-loaded, the system execution time is effectively equal to the application execution time. The system overhead is extremely small. The lightly-loaded assumption seems reasonably given the system profiling results presented in Table 5.6. These results show that the non-coprocessed CSIM system spends 98% of all clock cycles on application related activities.

8.1.2 Platform I Configuration Delays

For Platform I, the configuration delays experienced by the processor and the coprocessor are different. As indicated in Table 4.4, a programmable logic device on the ARC-PCI Board requires 5.5 ms to be configured. This duration is the typical configuration delay experienced by the coprocessor (t_{CFGB2}). Assuming that the configuration data is already stored on the ARC-PCI Board, the typical delay experienced by the processor is simply the time required to initiate and complete a configuration cycle. Initiation of a configuration cycle requires a single memory write transaction to set the configuration bit in the interface control register. Completion of a configuration cycle requires a single memory write transaction to clear the configuration bit in the interface control register after waiting for a fixed configuration delay. Since memory write transactions typical require 300 ns as indicated in Table 4.7, initiation and completion of a configuration cycle typically requires 600 ns of processor time (t_{CFGA2}). If acknowledgment of the completion of the configuration is necessary, additional time is required to poll the controller design to determine the status of the coprocessor. It is assumed that acknowledgment of a successful configuration is not required.

Configuration of a device may not always be required. If configuration of a device is not required, the minimum configuration delay for Platform I is effectively 0 s. This case is true for both the processor (t_{CFGA2}) and the coprocessor (t_{CFGB2}), provided that the processor does not need to check if the device has already been configured. For example, a system that only utilizes a single coprocessor design does not require dynamic configuration. It can be assumed that the boot time configuration is appropriate.

The maximum configuration delay is dependent upon the system load and application behaviour. However, a reasonable estimate of the maximum configuration delay experienced by the coprocessor is the time required to transfer a configuration file to the ARC-PCI Board plus the time required to configure one of the devices. As indicated in Chapter 4, the transfer of a MAPP mode configuration file requires a minimum of 13.5 ms and the configuration of a FLEX 10K50 device requires approximately 5.5 ms resulting in a total configuration delay of approximately 19 ms. This value is an estimate of the maximum configuration delay for the coprocessor (t_{CFGB2}).[1] The processor only experiences delays resulting from the transfer of a configuration file, the initiation of a configuration cycle, and the completion of a configuration cycle. Therefore, the maximum configuration delay experienced by the processor (t_{CFGA2}) is approximately 13.5 ms.

8.1.3 Platform II Configuration Delays

For Platform II, the processor and the coprocessor are implemented using the same programmable logic device. As a result, the configuration delays experienced by the processor (t_{CFGA2}) and the coprocessor (t_{CFGB2}) are identical. The time required to configure a device on the Nios Embedded Processor Development Board is approximately 34.5 ms as indicated in Table 4.9. This value serves as an estimate of the typical configuration delay experienced by both the processor and the coprocessor for Platform II.

Like Platform I, the device used in Platform II may not require dynamic configuration. Therefore, the minimum time required for configuration is effectively 0 s. This value serves as an estimate of the minimum configuration delay experienced by both the processor (t_{CFGA2}) and the

[1] The maximum configuration delay is unbounded. It is possible for the delay to be larger if the PCI bus is heavily utilized or if the operating system is heavily-loaded. The estimate provided is reasonable given the assumption of a lightly-loaded system.

coprocessor (t_{CFGB2}).

Unlike Platform I, the configuration files for Platform II are not transferred to the coprocessor prior to use. As a result, the maximum configuration delays are equivalent to the typical configuration delays for Platform II. A value of 34.5 ms is used as an estimate for the maximum configuration delays experienced by both the processor (t_{CFGA2}) and the coprocessor (t_{CFGB2}).

8.1.4 Summary of Configuration Delays

Table 8.1 summarizes the estimates of minimum, maximum, and typical values for configuration delays experienced by the processor (t_{CFGA2}) and the coprocessor (t_{CFGB2}). Estimates of the total time required to configure a coprocessor are shown for both Platform I and Platform II. These estimates are used in the validation of the performance model. The validation assumes that devices only need to be configured once per application. This assumption is fair if sufficient contexts are available. If this is not the case, the estimates shown should be multiplied by the frequency of configurations required.

Timing Parameter	Platform I			Platform II		
	Minimum (in s)	Maximum (in s)	Typical (in s)	Minimum (in s)	Maximum (in s)	Typical (in s)
t_{CFGA2}	0	0.0135	0.0000006	0	0.0345	0.0345
t_{CFGB2}	0	0.019	0.0055	0	0.0345	0.0345

Table 8.1: Summary of Configuration Delays

8.1.5 Memory Utilization Observations

Based on the experimental results obtained throughout the course of this research, it is clear that memory bandwidth and memory utilization can play a significant role in the performance of an application. The impact is particularly large for mainstream software applications that use memory extensively. For example, sorting algorithms read data items at least once and perhaps many times during execution. Changes to the distribution and accessibility of data impact algorithm performance.

When an algorithm is coprocessed, application data can be distributed across additional mem-

ory systems. The presence of additional memory systems effectively increases the available memory bandwidth in the system. This increased bandwidth may lead to performance improvements in the system. However, the use of additional memory systems introduces overhead associated with data transfer and data duplication. Also, the distribution of memory across additional memory systems can cause changes in the performance of cache subsystems. Additional memory offers more locations to cache data. However, changes in the distribution of application data can influence the locality of reference. These changes may increase or decrease the benefit of caching application data. Clearly, changes in memory utilization patterns and data distribution due to coprocessing can positively or negatively impact the performance of an application.

8.1.6 Platform I Memory Utilization Delays

The impact of caching upon an application can be quantified. This impact serves as a worst case estimate of memory utilization delays upon a coprocessed system. Based on the experimental results obtained for pseudo-random number generation on Platform I, it is possible to estimate the impact of caching upon the performance of the system. Table 6.5 shows the performance gain of caching for the non-coprocessed system and the coprocessed system. For the largest test run (5,000,000 PRNG iterations), the non-coprocessed system demonstrates a performance improvement of 262 with caching enabled. For the same test, the coprocessed system demonstrates a performance improvement of 20 with caching enabled. The ratio of these performance improvements indicates that the distribution of memory negatively impacts the system by a factor of 13.

Based on the experimental results obtained for minheap management on Platform I, it is possible to determine a second estimate of the impact of caching upon the performance of the system. Table 7.5 shows the performance gain of caching for the non-coprocessed system and the coprocessed system. For the largest test (1,000 entries maximum), the non-coprocessed system demonstrates a performance improvement of 209 with caching enabled while the coprocessed system demonstrates a performance improvement of 8 with caching enabled. The ratio of these performance improvements indicates that the distribution of memory negatively impacts the system by a factor of 26.

Clearly, a large negative impact can result from the distribution of application memory across

memory subsystems. However, it must be noted that Platform I lacks a cache subsystem on the ARC-PCI Board. If the ARC-PCI Board possessed a cache subsystem comparable to the PC's L1 and L2 caches, the impact of memory distribution would be quite different. The impact of caching would not be the dominant component of memory utilization delays in this case. This might mean that the impact of memory utilization delays would actually be positive rather than negative if Platform I possessed a cache subsystem on the ARC-PCI Board.

It is not a surprise that caching has more impact upon the performance of minheap management in a loosely-coupled configurable computer system. Minheap insertion and deletion algorithms access large amounts of application data frequently. A large cache can significantly improve the performance of minheap insertion and deletion algorithms. On the other hand, pseudo-random number generators require very little storage for application data. All of the application data required by a pseudo-random number generator can be stored in registers within a coprocessor. As a result, caching plays a small role in the performance of a pseudo-random number generator.

Using the peak performance improvement figures, it is estimated that memory utilization delays negatively impact application performance by a factor (F_{MEM}) of 10 to 30 times for mainstream software applications coprocessed using Platform I. This factor allows for the performance impacts associated with pseudo-random number generation ($F_{MEM} = 13$) and minheap management ($F_{MEM} = 26$). Both the use of slower memory devices and the distribution of data contribute to this loss of performance.

To estimate the timing parameters for the performance model, it is important to establish how much time is spent by an application on memory transactions. Once this is known, it is possible to calculate minimum, maximum, and typical values for memory utilization delays. Based on the assumption that a non-coprocessed application spends approximately 100 s of its execution time on memory utilization delays[2], it is possible to estimate the memory utilization delays for a coprocessed application. Using the factor F_{MEM} to estimate the changes in these delays, memory utilization delays (t_{MEM2}) are estimated to account for a minimum of 1000 s and a maximum of 3000 s in a coprocessed application. To estimate the typical value, the average value of 2000 s is used.

[2]It is difficult to estimate the delays associated with memory utilization. These delays depend upon the state of the system. The estimate that 10% of the execution time is related to these delays is partially based on results of CSIM profiling. However, this figure could be much smaller or larger for other applications.

8.1.7 Platform II Memory Utilization Delays

For tightly-coupled configurable computer systems, changes in data distribution and memory utilization can positively impact the performance of a system. The delays associated with data transfers are much smaller in a tightly-coupled configurable computer system. The benefit of data distribution can exceed the cost of data transfer and data duplication. This effect is an important difference between a tightly-coupled configurable computer system and a loosely-coupled configurable computer system.

For Platform II, the memory access times associated with the processor are 20 ns as indicated in Table 4.10 while the memory access times associated with the coprocessor are 4.2 ns as indicated in Table 4.11. In other words, the memory associated with the coprocessor is approximately 5 times faster than the memory associated with the processor. The use of the coprocessor on this platform increases the available memory bandwidth by a substantial amount. More memory is available for use and the additional memory is faster. A factor (F_{MEM}) of 0.2 should be used to take into account the relative speeds of the memory systems for the purpose of estimating the memory utilization delays for the performance model.

However, the memory utilization delays (t_{MEM1}) are 0 s for the non-coprocessed application since caching is not an issue and the platform does not have an operating system to influence application behaviour. The lack of an operating system means that a multiplying factor cannot be used to establish an estimate of t_{MEM2}. Clearly, the value of t_{MEM2} is a negative value indicating that the memory utilization delays impact is positive. An upper bound on the impact of memory utilization delays can be established using the results of the pseudo-random number generation experiments. These results demonstrate a speedup of 2.691 as indicated in Table 6.6. A portion of this speedup results from the impact of memory utilization delays and the remainder results from a reduction in processing times. Given the simplicity of the pseudo-random number generation algorithm, it can be argued that most, if not all, of the speedup recorded for this application results from the impact of the faster memory on the system. If the system execution time is reduced by a factor of 2.691, this translates to a reduction of approximately 630 s. Thus, a value of -630 s is used as the estimate of the minimum value of memory utilization delays. The maximum value of memory utilization delays is estimated to be 0 s since in the worst case, external memory is used for all memory accesses. The average of the two values is used as the typical value

for memory utilization delays.

8.1.8 Summary of Memory Utilization Delays

Table 8.2 summarizes the estimates of minimum, maximum, and typical values for memory utilization timing experienced by the processor and the coprocessor. Estimates are shown for both Platform I and Platform II. It should be noted that the factor (F_{MEM}) used to estimate memory utilization delays depends upon whether the platform is loosely-coupled or tightly-coupled.

Timing Parameter	Platform I			Platform II		
	Minimum (in s)	Maximum (in s)	Typical (in s)	Minimum (in s)	Maximum (in s)	Typical (in s)
t_{MEM1}	100	100	100	0	0	0
Multiplying Factor (F_{MEM})	10	30	20	N/A	N/A	N/A
$t_{MEM2} = t_{MEM1} \times F_{MEM}$	1000	3000	2000	-630	0	-315
$t_{MEM} = t_{MEM1} - t_{MEM2}$	-900	-2900	-1900	630	0	315

Table 8.2: Summary of Memory Utilization Delays

8.1.9 Bus Utilization and Operating System Behaviour Observations

Bus utilization delays and operating system behaviour delays are a small component of the execution time of a non-coprocessed mainstream application. This observation is particularly true if the system is lightly-loaded. The use of a coprocessor introduces new bus transactions that must be performed. These additional bus transactions increase bus utilization and thus, the delays that result from bus transactions. The use of a coprocessor also changes the way processes are scheduled in a multitasking operating system. This change is small, but significant, particularly if additional context switches are required by the operating system. For the purpose of this validation, the bus utilization delays and the operating system behaviour delays are lumped together since they are closely related. These delays are referred to as the lumped delays of bus utilization and operating system behaviour.

Using the results of the CSIM system profiling analysis provided in Table 5.6, it is observed that kernel usage increases dramatically with the introduction of a coprocessor. Kernel usage increase by a factor between 9 and 10. This large increase is partly due to the large number of

bus transactions that must be performed in a coprocessed system and partly due to changes in operating system behaviour.

8.1.10 Platform I Lumped Delays

It is possible to estimate the minimum, maximum, and typical timing parameters for bus utilization delays and operating system behaviour delays for Platform I. Using a factor (F_{LUMP}) that ranges from 9 to 10 and the assumption that 60 s of non-coprocessed system execution time corresponds with the lumped delays ($t_{LUMP1} = t_{BUS1} + t_{OS1}$), it is possible to estimate the lumped delays ($t_{LUMP2} = t_{BUS2} + t_{OS2}$) for a coprocessed application. The coprocessed application experiences lumped delays (t_{LUMP2}) ranging from 540 s to 600 s. An average of 570 s is used as the estimate for the typical lumped delays.

8.1.11 Platform II Lumped Delays

Platform II is unusual in the sense that it does not run an operating system. Also, the number of bus cycles required by Platform II is fixed since the external memory and the coprocessor reside on the same bus. As a result, bus utilization delays and operating system behaviour delays do not occur. The minimum, maximum, and typical timing parameters estimated for these lumped delays are 0 s in all cases for Platform II.

8.1.12 Summary of Lumped Delays

Table 8.3 summarizes the estimates of minimum, maximum, and typical values for the lumped delays experienced by the processor and the coprocessor. Estimates are shown for both Platform I and Platform II. Since the lumped delays associated with Platform II are 0 s in all cases, the factor (F_{LUMP}) is not applicable (N/A) for this platform.

8.1.13 Processing Time Observations

The time required to process transactions by a processor is different than the time required to process transactions by a coprocessor. Typically, the application-specific nature of the coprocessor

Timing Parameter	Platform I			Platform II		
	Minimum (in s)	Maximum (in s)	Typical (in s)	Minimum (in s)	Maximum (in s)	Typical (in s)
t_{LUMP1}	30	30	30	0	0	0
Multiplying Factor (F_{LUMP})	9	10	9.5	N/A	N/A	N/A
$t_{LUMP2} = t_{LUMP1} \times F_{LUMP}$	540	600	570	0	0	0
$t_{LUMP} = t_{LUMP1} - t_{LUMP2}$	-510	-570	-540	0	0	0

Table 8.3: Summary of Lumped Delays

permits it to process transactions faster than the general-purpose processor. For the purpose of this discussion, processing time refers to the maximum total time required for pre-processing, execution, and post-processing by either the processor or coprocessor. The separation of pre-processing, execution, and post-processing times is beyond the scope of this validation.

8.1.14 Platform I Processing Times

For the purpose of this analysis, it is assumed that a coprocessor can improve performance by a factor (F_{EXE}) ranging from 0.1 to 10, depending upon the relative speeds of the processor and the coprocessor.[3] A factor of 0.1 represents a full order of magnitude of performance improvement. A factor of 10 represents a full order of magnitude of performance degradation. The values in Table 8.4 show estimates of the raw processing time for both coprocessor platforms. For the typical values, it is assumed that the application-specific coprocessor can outperform the processor on the application-specific task by a factor of 2. Hence, a factor (F_{EXE}) of 0.5 is used for the estimation of the typical processing time values.

For a non-coprocessed application, the processing times equal the application execution times less the memory utilization delays and the lumped delays. The non-coprocessed application processing times (t_{EXE1}) for Platform I are 840 s in all cases. Therefore, the coprocessed application processing times (t_{EXE2}) range from 84 s to 8400 s for Platform I. Based on the assumption that the coprocessor processes transactions faster, a value of 420 s is used as the estimate of the typical processing time values for Platform I.

[3]Some coprocessors can achieve several orders of magnitude of improvement. The value of ten is conservative.

8.1.15 Platform II Processing Times

Since the lumped delays associated with Platform II are 0 s in all cases, processing times represent a larger fraction of system execution time on Platform II. The non-coprocessed application processing times (t_{EXE1}) for Platform II are 900 s in all cases. Therefore, the coprocessed application processing times (t_{EXE2}) range from 90 s to 9000 s for Platform II. Based on the assumption that the coprocessor processes transactions faster, a value of 450 s is used as the estimate of the typical processing time values for Platform II.

8.1.16 Summary of Processing Times

Table 8.4 summarizes the estimates of minimum, maximum, and typical values for the processing times experienced by the processor and the coprocessor. Estimates are shown for both Platform I and Platform II.

Timing Parameter	Platform I			Platform II		
	Minimum (in s)	Maximum (in s)	Typical (in s)	Minimum (in s)	Maximum (in s)	Typical (in s)
t_{EXE1}	840	840	840	900	900	900
Multiplying Factor (F_{EXE})	0.1	10	0.5	0.1	10	0.5
$t_{EXE2} = t_{EXE1} \times F_{EXE}$	84	8400	420	90	9000	450
$t_{EXE} = t_{EXE1} - t_{EXE2}$	756	-7560	420	810	-8100	450

Table 8.4: Processing Time Summary

8.1.17 Platform Comparison

Table 8.5 summarizes the timing estimates for a non-coprocessed application given the assumptions made previously. Regardless of the platform, the system execution time (t_{SYS1}) is assumed to be 1000 s. This permits a comparison of the two platforms with respect to the benefits of coprocessing. It should be noted that lumped delays are not applicable to Platform II. Hence, the behaviour of the application on Platform II is slightly more predictable.

Table 8.6 summarizes the estimates of minimum, maximum, and typical system execution times for Platforms I and II. The estimates show that for Platform I, an example of a loosely-

Timing Parameter	Platform I			Platform II		
	Minimum (in s)	Maximum (in s)	Typical (in s)	Minimum (in s)	Maximum (in s)	Typical (in s)
t_{MEM1}	100	100	100	100	100	100
t_{LUMP1}	60	60	60	0	0	0
t_{EXE1}	840	840	840	900	900	900
t_{SYS1}	1000	1000	1000	1000	1000	1000

Table 8.5: Non-Coprocessed Timing Summary

coupled configurable computing platform, performance gains for the application are theoretically impossible. The minimum system execution time is greater than 1000 s. However, for Platform II, an example of a tightly-coupled configurable computing platform, performance gains are theoretically possible. The minimum system execution time is less than 1000 s.

Timing Parameter	Platform I			Platform II		
	Minimum (in s)	Maximum (in s)	Typical (in s)	Minimum (in s)	Maximum (in s)	Typical (in s)
t_{CFGA2}	0	0.0135	0.0000006	0	0.0345	0.0345
t_{CFGB2}	0	0.019	0.0055	0	0.0345	0.0345
t_{MEM2}	1000	3000	2000	-630	0	-315
t_{LUMP2}	540	600	570	0	0	0
t_{EXE2}	84	8400	420	90	9000	435
t_{SYS2}	1624	12000	2990	-540‡	9000	120

‡Using the minimum estimates, the model indicates a speedup greater than ∞ for Platform II. This anomaly is easily explained. The minimum estimates for t_{MEM2} and t_{EXE2} never happen at the same time. An application is either memory bound or computation bound.

Table 8.6: Coprocessed Timing Summary

It is important to note that the configuration delays are negligible compared to the other delays in the system. The coprocessor could be configured thousands of times without impacting overall system performance significantly. The configuration delays become significant as the granularity of the transactions approaches the time required to configure the device. Also, as the frequency of configuration increases, the significance of configuration delays increases. For any application that is computationally-intensive, configuration delays are negligible.

8.2 Comparison of Theoretical Performance with Actual Performance

Table 8.7 provides a summary of the estimated bounds on system speedups based on the performance model. The speedups for Platform I range from a low of 0.083 to a high of 0.616. In other words, Platform I is not predicted to accelerate a system under any circumstances. Note that this does not mean that loosely-coupled configurable computing is hopeless. It simply means that the ARC-PCI Board is unlikely to provide any speedup on a mainstream software application given the architecture of the board. The speedups for Platform II range from a low of 0.111 to a high of ∞. Platform II is typically capable of accelerating an application. The tightly-coupled architecture of Platform II is more suitable for configurable coprocessing.

Platform	Minimum Speedup	Maximum Speedup	Typical Speedup
Platform I	0.083	0.616	0.334
Platform II	0.111	∞	8.333

Table 8.7: Estimated Bounds on System Speedups

Table 8.8 summarizes the actual system speedups obtained for the three application systems described in this thesis. System speedups could not be obtained for Application 1 (CSIM) on Platform II since the application code is not portable to this platform.

Application	Platform I		Platform II	
	Minimum Speedup	Maximum Speedup	Minimum Speedup	Maximum Speedup
Application 1 (CSIM)	0.231	0.773	N/A	N/A
Application 2 (RAND)	0.009	0.039	2.691	2.691
Application 3 (MIN)	0.092	0.133	10.054	13.673

Table 8.8: Actual Application Speedups

8.3 Observations

Comparing the estimated application speedups shown in Table 8.7 with the actual application speedups shown in Table 8.8, it appears that the performance model correctly predicts that Platform II outperforms Platform I. The performance model predictions can account for all test results with the exception of the Application 2 (RAND) tests on Platform I. The order of magnitude slowdown for processing times is simply too optimistic in this case. Pseudo-random number generation is not a complex computation. The processor inside the PC is capable of generating pseudo-random numbers much faster than the coprocessor (approximately 50 times faster). Also, it is possible that the lumped delays are not large enough for Platform I. More profiling is recommended to refine the timing parameters for Platform I. This profiling would improve the accuracy of the model.

8.3.1 Pre-Processing and Post-Processing

Although pre-processing and post-processing are not explicitly validated, it is an important part of the performance model. An indepth investigation of the effects of pre-processing and post-processing would be an interesting topic for future research. Pre-processing and post-processing allows the processor and the coprocessor to compute in parallel. As shown in Chapter 7, the reordering of computations can lead to a significant performance improvement (approximately 40% for Application 3). Given this result, the order of magnitude of performance improvement estimated for processing times may be a bit conservative. Further research of the effects of pre-processing and post-processing is recommended.

8.3.2 System Profiling

Some of the differences between the predicted performance and the actual performance result from the fact that the validation uses the profiling of Application 1 (CSIM) to estimate the factors for the other applications. While these factors may be suitable for Application 1, they may not be suitable for modeling the other applications. Profiling of Application 2 (RAND) and Application 3 (MIN) might produce a different range of predictions. This profiling can be done in future research to determine if this is a significant source of errors.

8.3.3 Processing Times

The estimates of processing times assume a fixed range of speedups associated with coprocessing. This range is based on the fact that an order of magnitude of performance increase is common for niche applications of configurable computing. The value of one order of magnitude is a very conservative estimate for niche applications. It is unclear whether performance improvements in excess of an order of magnitude are realistic for mainstream software applications. Future research into the characterization of the processing times of mainstream software applications is recommended.

Also, the performance model predicts the impact of a coprocessor on the performance of the system, assuming a lightly-loaded system. For a heavily-loaded system, the application represents a smaller fraction of the processing in the entire system. It would not be possible to use the same factors presented to predict processing times for a heavily-loaded system. Either the model would need to be modified to isolate application processing times from system processing times or the factors would need to be adjusted according to Amdahl's Law [HP90].

8.3.4 Application Impact vs. System Impact

The performance model predicts the impact of a coprocessor on the performance of the system. The application impact might differ from the system impact in some situations. Using a subset of the performance model, it is possible to predict the application impact. Future research into when the application impact is likely to differ from the system impact is recommended. This is particularly important for heavily-loaded systems. The study of heavily-loaded systems is beyond the scope of this thesis.

8.3.5 Performance Model Suitability

It appears that the performance model is a rough model of system performance. It is a good starting point for future research into more advanced models of configurable computing. As currently specified, the performance model could be used to predict whether configurable coprocessing is a viable technique for accelerating a mainstream software application. It is not suitable

for accurately predicting the exact speedup obtained using configurable coprocessing. However, it provides a way of evaluating whether there exists potential for speedup or whether a speedup is unlikely.

8.4 Application Implications

It is possible to use the performance model to predict the benefit of coprocessing on a particular application. Based on the performance model, an ideal application has the following properties:

- course computation granularity,
- opportunities to exploit parallelism, and
- limited I/O per transaction.

8.4.1 Course Computation Granularity

Course computation granularity is important to justify the overhead associated with configuration delays. Course computation granularity reduces the need for frequent configuration cycles. Also, applications with course computation granularity are more likely to benefit from application-specific hardware. Simple computations such as pseudo-random number generation do not benefit from coprocessing. The time required by an application to start a transaction and receive a response exceeds the time required to compute the computation.

8.4.2 Opportunities to Exploit Parallelism

Opportunities to exploit parallelism are important. Large performance gains are possible using pre-processing and post-processing. The speedup of an application (and thus, the system) is limited if pre-processing and post-processing is impossible. Pre-processing and post-processing permit a slow application-specific coprocessor to accelerate a computation. Without opportunities to exploit parallelism, the coprocessor must compute its results significantly faster than the processor to permit speedup.

8.4.3 Transaction I/O

Transactions that require the communication of large amounts of data are not good candidates for coprocessing. Transactions require use of the bus hierarchy. As the amount of data communicated between the processor and the coprocessor increases, so does the amount of bus traffic. Since bus transactions tend to be slower than memory accesses, the impact of this bus traffic can be dramatic. Applications that can be divided into transactions requiring little, if any, I/O are preferable.

8.5 Architectural Implications

It is interesting to observe the implications of the performance model on the architecture of a coprocessing system. Clearly, tightly-coupled configurable computing platforms have the potential to deliver high performance. The tight integration offered by such a platform mitigates the communication and synchronization bottlenecks. The use of a loosely-coupled configurable computing platform is unlikely to deliver any performance benefits.

8.5.1 Configuration Delays

It should be noted that configuration delays are not a determining factor in the performance of a coprocessed application that is computationally-intensive. Many configuration cycles are necessary for configuration delays to become a significant factor in the overall performance of a computationally-intensive application. However, for applications with extremely small execution times and systems that require frequent configuration, the configuration delays may be significant. Efforts should always be made to minimize the impact of configuration delays by reducing the number of configuration cycles if possible.

8.5.2 Memory Utilization Delays

It is apparent that memory utilization delays are a determining factor in the performance of a coprocessed application. Thus, every effort should be made to reduce memory latency and to ensure that the coprocessor has sufficient bandwidth to memory. A slow memory device can

influence the performance of a coprocessor significantly. Ideally, coprocessors should have access to memory bandwidth at least equivalent to that associated with processors. In a tightly-coupled configurable coprocessing system, it is likely that the memory bandwidth available to the processor will be close to that available to the coprocessor. The tight coupling indirectly ensures similar bus speeds and memory access speeds since it is very difficult to tightly couple processing units that execute at dramatically different speeds. However, the sharing of a memory system can lead to bus utilization and operating system behaviour delays so the coprocessor should have a local, dedicated memory for storage, if possible.

8.5.3 Lumped Delays

In a loosely-coupled system, the loose coupling between the processor and the coprocessor inevitably introduces synchronization delays. The busses that connect the processor to the coprocessor will be impacted by these synchronization delays. Also, these delays contribute to changes in operating system behaviour. A tightly-coupled coprocessing architecture reduces these delays. For some tightly-coupled platforms, such as Platform II, these lumped delays may be negligible. Tightly-coupled platforms offer increased system predictability. There is no compelling reason for choosing a loosely-coupled architecture over a tightly-coupled architecture. If feasible, a tightly-coupled architecture should be used.

8.5.4 Processing Times

The dominant factor in the success or failure of any coprocessing system is the increased performance of a coprocessor due to its application-specific design. The application to be coprocessed must be suitable for coprocessing. It must be possible to implement the necessary transactions in configurable hardware. This is not an easy task in some cases. For example, complex algorithms, such as minheap deletion, can be very difficult to design, build, and test. Also, it is very important to remember that although processing time is the dominant component of a non-coprocessed application, this may not be true for a coprocessed application. This observation is a very significant implication of the performance model.

8.5.5 Summary of Implications

Clearly, if enhanced performance of an application is desired, a tightly-coupled configurable coprocessor architecture is necessary. A balanced approach is best for application performance. Unless all sources of delay are considered during the design of a coprocessed application, the performance of the coprocessed application will be less than desirable. A small component of the execution time of a non-coprocessed application can become the dominant component of the execution time of a coprocessed application.

Chapter 9

Conclusions

Configurable computers offer increased control logic flexibility and datapath flexibility. Application-specific coprocessors may be built for mainstream software applications using configurable logic devices. It is possible to accelerate common data structures and algorithms using a configurable coprocessor. These data structures and algorithms can benefit a wide range of mainstream software applications, given a suitable configurable computer architecture.

9.1 Thesis Contributions

The thesis presents an empirical study of configurable coprocessing based on experimental results obtained using three mainstream software applications. The applications include two simple pseudo-random number generation algorithms and a complex minheap insertion and deletion algorithm. The results of this empirical study provide some insight into the complexity of configurable computing. This thesis lays a foundation for future research by making the following contributions:

- introduces, explains, and validates a novel configurable computing performance model that predicts both application performance and system performance,
- illustrates several of the challenges associated with developing configurable coprocessors,

- provides experimental results demonstrating speedups for two mainstream software applications (pseudo-random number generation and minheap management) executing on a tightly-coupled configurable computer,
- quantifies memory utilization delays, bus utilization delays, and operating system behaviour delays for a mainstream software application (CSIM),
- summarizes several desirable properties of mainstream software applications for configurable coprocessing,
- identifies desirable features of configurable computer architectures to ensure adequate performance,
- describes a reference design for an ARC-PCI Board with hardware support for dynamic configuration and high-speed I/O, and
- shows that configuration delays are not the most significant source of delay for computationally-intensive applications.

9.1.1 The Performance Model

The performance model introduced in Chapter 3 and validated in Chapter 8 expands upon previous research into the modeling of configurable computer systems. The performance model attempts to quantify the impact of memory utilization delays, bus utilization delays, and operating system behaviour delays by placing bounds on the delays. Previously published models of configurable computer systems have not attempted to quantify these sources of delays. In addition, the performance model provides insight into the performance of the application as well as the performance of the system.

9.1.2 Challenges of Developing Configurable Coprocessors

As indicated in Chapter 5, Chapter 6, and Chapter 7, the development of configurable coprocessors is a difficult and time-consuming task. Any application that can be written as a finite state machine can be implemented in a configurable coprocessor provided that sufficient resources (e.g.,

memory bits, logic elements, etc.) are available. However, if resources are limited, hardware development can be difficult if not impossible. The process of translating software algorithms into hardware designs requires both skill and intuition. The most obvious difference between hardware and software designs is that software designs do not need to meet the strict minimum clock frequency requirements of hardware designs.

It is important to note that the hardware / software interface plays a significant role in the performance of a configurable coprocessor. This is demonstrated by the performance gains that are achieved through the use of buffering, pre-processing, post-processing, and fast I/O dispatching. These performance enhancements make a significant difference in the overall performance of the system. Special care must be taken to ensure that the hardware /software interface is tailored for the application. Hardware / software codesign techniques help with this process but more tools are needed to assist with exploring interface alternatives. Hand-coded interfaces and reference designs are a good way to ensure adequate performance at the interface.

9.1.3 Mainstream Software Application Speedups

This research has shown that tightly-coupled configurable computers are suitable for coprocessing mainstream software applications. On the tightly-coupled configurable computer studied, the coprocessed pseudo-random number generator outperforms the non-coprocessed algorithm by a factor of 2.6. For minheap insertion and deletion, speedups approaching a factor of 14 can be achieved on a tightly-coupled configurable computer. These speedups are noticeable improvements.

Furthermore, it has been shown that coprocessing may be done transparently, without any impact upon the quality of the results. This was shown by coprocessing CSIM, a commercial discrete-event simulation library. Although performance improvements were not achieved in this experiment, this was mostly due to the use of a loosely-coupled configurable computer architecture. Such architectures are not likely to be suitable for coprocessing mainstream software applications although they are potentially suitable for niche applications. In particular, this research has shown that Platform I is unlikely to produce any significant speedup on a mainstream software application.

9.1.4 Mainstream Software Application Delays

Memory utilization delays, bus utilization delays, and operating system delays are significant factors influencing the performance of coupled configurable computer systems. The act of coprocessing transactions causes application data to be distributed and perhaps duplicated for local access. This distribution of data impacts the performance of caching and introduces additional synchronization delays. It also causes new bus transactions to be performed to facilitate communication between the processor and the coprocessor. This additional overhead changes the behaviour of the operating system further impacting both application performance and system performance.

Of course, no performance improvement is possible if the coprocessor is unable to compute its results efficiently. Thus, the ratio of the processing power of the coprocessor to the processor is an important consideration. A balanced approach that attempts to minimize all sources of delays and distribute computations on the basis of processing resources appears to be best. Based on the model, it is clear that minimizing one source of delay does not maximize the performance of the system.

9.1.5 Desirable Properties of Mainstream Software Applications

Configurable coprocessing is most effective if the mainstream software application exhibits the following properties:

- course computation granularity,
- opportunities to exploit parallelism, and
- limited I/O per transaction.

While speedup is possible for applications that do not exhibit all three of these properties, speedup is more probable for applications that do exhibit these properties. It should also be noted that some niche applications have become mainstream software applications. For example, cryptography is now commonly used by a wide variety of software applications. Security is a major consideration in any application that utilizes confidential data. The use of a configurable

coprocessor for cryptographic applications could result in a significant performance improvement in such mainstream software applications.

9.1.6 Desirable Features of Configurable Computer Architectures

It is clear that tight coupling of a processor with a coprocessor is beneficial to the performance of an application. Loosely-coupled systems suffer from large communication latencies for bus traffic. Mainstream software applications manipulate large quantities of data. If the coupling is loose, many new bus transactions are often required to communicate data and time is not used most effectively. Tight coupling is particularly important for simple computations. Very complex computations can overcome large communication latencies. One way to reduce communication latency is to use a platform without an operating system. The use of an operating system introduces additional delays that may be unnecessary. Unless required by the application or other applications in the system, operating systems should be avoided.

9.1.7 Reference Design for the ARC-PCI Board

Chapter 4 describes a reference design for the ARC-PCI Board. This reference design consists of an application programming interface, a device driver, and a reference controller design. This reference design provides the following features that make it unique:

- built-in hardware management of MAPP mode configuration,
- simple memory-mapped I/O interface for user designs,
- near optimal I/O performance,
- clear separation of the controller logic from the user logic, and
- platform support for Windows NT and Linux.

9.1.8 Configuration Delays

Previous research into configurable computing has focused on the overhead of configuration delays. While configuration delays can be significant, this research has shown that for computationally-

intensive applications, configuration delays are negligible compared to other sources of delays. It is possible to configure a coprocessor thousands of times without significantly impacting the performance of the application or the system. Furthermore, a good configurable computer architecture can help to minimize the overhead associated with configuration. Storing configuration data locally can reduce configuration times and bus utilization delays that result from transferring configuration data over the bus hierarchy.

9.2 Potential for Future Research

This work has revealed the need for future research into the following areas:

- profiling of other mainstream software applications executing on configurable computer systems,
- analysis of the impact of system load on configurable computer performance,
- investigation of platform FPGA architectures that incorporate both general-purpose processors and configurable logic elements,
- study of other types of configurable computer architectures for inclusion in the performance model, and
- translating software algorithms into hardware designs that exploit parallelism.

9.2.1 Profiling of Mainstream Software Applications

For the purpose of this research, three applications were studied. Research into other applications is necessary to ensure that the performance model is an adequate representation of the performance of mainstream software applications. In particular, system level profiling could reveal more information about the nature of memory utilization delays, bus utilization delays, and operating system behaviour delays. A thorough understanding of these delays should help in the refinement of the performance model.

9.2.2 Analysis of the Impact of System Load

For the purpose of this research, the system was lightly-loaded for all tests. This effectively meant that system execution time was equivalent to application execution time. In a heavily-loaded system, this assumption is invalid. As system load increases, configurable coprocessors should theoretically have less impact upon the performance of an application. It is unclear if system performance benefits more or less from a configurable coprocessor if the system is heavily-loaded. An investigation of the impact of system load on the performance of a configurable coprocessor is recommended.

9.2.3 Platform FPGAs and RPUs

Alternative hardware architectures for configurable computing should be studied. Both Platform FPGAs and RPUs offer very tight coupling between the processor and the coprocessor. Such devices might be capable of achieving performance speedups otherwise unattainable. It is quite possible that these devices may yield good performance for mainstream software applications.

Platform FPGAs, such as the Xilinx Virtex II Pro devices, incorporate general-purpose processors and configurable logic elements. These devices provide a very tight coupling between the processor and the coprocessor. Two (or more) processing units share a common system bus. These devices represent the next generation of configurable coprocessors. They are likely to be suitable for high-end embedded systems.

RPUs (Reconfigurable Processor Units) for configurable computing are also worth investigating. The latest version of the Nios processor supports the creation of custom instructions using VHDL. Effectively, the Nios processor is a form of RPU. These devices effectively merge the processor and the coprocessor into a single design. As a starting point for future research, it is recommended that Application 2 (RAND) be implemented as a custom instruction for the Nios processor. Application 2 is the simplest of the applications presented in this thesis. It is also the one most suitable for implementation as a custom instruction. The other two applications would require the implementation of several custom instructions.

9.2.4 Translation of Software Algorithms to Hardware Designs

During the development of the hardware designs for the three applications studied in this research, it became apparent that the translation of a software algorithm into a hardware design is a difficult, error-prone, and time-consuming process. Simple algorithms, such as pseudo-random number generation, can be implemented using combinational logic and a few registers. More complex algorithms, such as minheap insertion and deletion, are much more difficult to translate from software implementations to hardware implementations. While it is true that all finite sequential algorithms can be implemented using a finite state machine, this is not necessarily ideal for all software algorithms. This technique worked reasonably well for the minheap insertion and deletion algorithms studied but it would not have been feasible for more complex algorithms due to problems associated with state bit encoding for large state machines. Further research into automated techniques for translating software algorithms to hardware designs is recommended. Also, a study of hardware design techniques for complex, sequential logic should be performed.

9.3 Thesis Applicability

The research presented in this thesis has industrial applications. The cost of building and utilizing a configurable computer architecture could be justified for software applications that demand high performance. Also, the techniques presented in this thesis for enhancing mainstream software applications could be used to enhance embedded software systems and real-time systems. Finally, although the performance model is discussed in the context of configurable computing, it could be used to model multiprocessors and other types of coprocessing systems.

Bibliography

[AA95] Peter M. Athanas and A. Lynn Abbott. Addressing the Computational Requirements of Image Processing with a Custom Computing Machine: An Overview. In *Proceedings of the Ninth International Parallel Processing Symposium Special Workshop on Reconfigurable Architectures and Algorithms*, Santa Barbara, California, April 1995.

[ABD92] Jeffrey M. Arnold, Duncan A. Buell, and E. G. Davis. Splash 2. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–324, June 1992.

[ACC+95] R. Amerson, R. Carter, B. Culbertson, P. Kuekes, and G. Snider. Teramac–Configurable Custom Computing. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 32–38, Napa Valley, California, April 1995.

[Alt94] Altera Corporation, San Jose, California. *Reconfigurable Interconnect Peripheral Processor (RIPP10) Users Manual*, May 1994.

[Alt96] Altera Corporation, San Jose, California. *1996 Data Book*, June 1996.

[Alt97a] Altera Programmable Hardware Development Program. World Wide Web Document, January 1997.
`http://www.altera.com/html/programs/phd.html`.

[Alt97b] Altera Corporation, San Jose, California. *University Program Design Laboratory Package User Guide*, August 1997.

[Alt98a] Altera Corporation, San Jose, California. *PCI MegaCore Function User Guide*, November 1998.

[Alt98b] Altera Corporation, San Jose, California. *pcit1 PCI Target MegaCore Function*, July 1998.

[Alt02a] Altera Corporation, San Jose, California. *Nios Embedded Processor 32-Bit Programmer's Reference Manual*, April 2002.

[Alt02b] Altera Corporation, San Jose, California. *Nios Embedded Processor Peripherals Reference Manual*, September 2002.

[Alt02c] Altera Corporation, San Jose, California. *Nios Embedded Processor Software Development Reference Manual*, July 2002.

[Apt93] Aptix Corporation, San Jose, California. *Programmable Interconnect System Data Book*, 1993.

[Apt95] Aptix Corporation, San Jose, California. *FPCB Development System User's Manual*, 1995.

[Ass96] APS-X84 Development Kit Documentation. World Wide Web Document, December 1996.
`http://www.associatedpro.com/aps/x84.html`.

[Bag91] Rajive L. Bagrodia. Designing Efficient Simulations Using Maisie. In *Proceedings of the 1991 Winter Simulation Conference*, pages 243–247, Phoenix, Arizona, December 1991.

[BAM96] Ray A. Bittner Jr., Peter M. Athanas, and Mark D. Musgrove. Colt: An Experiment in Wormhole Run-Time Reconfiguration. In *Proceedings of the 1996 SPIE Photonics East Conference*, Boston, Massachussetts, November 1996.

[BL94] Rajive L. Bagrodia and Wen-Toh Liao. A Language for the Design of Efficient Discrete-Event Simulations. *IEEE Transactions on Software Engineering*, 20(4):225–238, April 1994.

[BL97] William D. Bishop and Wayne M. Loucks. A Heterogeneous Environment for Hardware/Software Cosimulation. In *Proceedings of the 30th Annual Simulation Symposium*, pages 14–22, Atlanta, Georgia, April 1997.

[BMT+98] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, B. Park, and H. Song. A Parallel Simulation Environment for Complex Systems. *IEEE Computer*, 31(10):77–85, October 1998.

[Box94] Brian Box. Field-Programmable Gate Array Based Reconfigurable Preprocessor. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 40–48, Napa Valley, California, April 1994.

[BR96] Stephen D. Brown and Jonathon Rose. FPGA and CPLD Architectures: A Tutorial. *IEEE Design & Test of Computers*, 13(2):42–57, 1996.

[BRV89] Patrice Bertin, Didier Roncin, and Jean Vuillemin. Introduction to Programmable Active Memories. Technical Report PRL Research Report #3, DEC Paris Research Laboratory, Paris, France, 1989.

[BTA93] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming Pin Limitations in FPGA-Based Logic Emulators. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 142–151, Napa Valley, California, April 1993.

[Cas93] Steve Casselman. Virtual Computing and the Virtual Computer. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 43–48, Napa Valley, California, April 1993.

[Cas96] Steve Casselman. Reconfigurable Processing Units (RPUs). Newsgroup Posting, 1996. `news:comp.arch.fpga`.

[CB92] C. E. Cox and W. E. Blanz. GANGLION–A Fast Field-Programmable Gate Array Implementation of a Connectionist Classifier. *IEEE Journal of Solid-State Circuits*, 27(3):288–299, March 1992.

[Cha94] Pak K. Chan. A Field-Programmable Prototyping Board: XC4000 BORG User's Guide. Technical Report UCSC-CRL-94-18, Board of Studies in Computer Engineering, University of California, Santa Cruz, Santa Cruz, California, April 1994.

[CMQ02] Stephen Charlwood, Jonathan Mangnall, and Steven Quigley. System-Level Modelling for Performance Estimation of Reconfigurable Coprocessors. In *Proceedings of the 12th International Conference Field Programmable Logic 2002*, Montpellier, France, September 2002.

[Cor96] Giga Operations Corporation. Spectrum Reconfigurable Computing Developers. World Wide Web Document, December 1996. `http://www.reconfig.com/giga/spectrc.htm`.

[Cor97] X Engineering Software Systems Corporation. XS95 Board. World Wide Web Document, January 1997. `http://www.xess.com/FPGA/ho02001.html`.

[Cor99] Altera Corporation. Configuring APEX 20K, FLEX 10K & FLEX 6000 Devices. Application Note A-AN-116-01, Altera Corporation, San Jose, California, August 1999.

[Cor02a] Altera Corporation. Cyclone Product Backgrounder. Product Backgrounder, Altera Corporation, San Jose, California, November 2002.

[Cor02b] Altera Corporation. Delivering RISC Processors in an FPGA for $2.00. White Paper, Altera Corporation, San Jose, California, November 2002.

[Cor02c] Altera Corporation. Nios Embedded Processor Development Board Data Sheet. Data Sheet DS-NIOSDEVBD-2.1, Altera Corporation, San Jose, California, April 2002.

[Cor02d] Altera Corporation. Nios Embedded Processor Getting Started User Guide. User Guide UG-NIOSGSG-2.2, Altera Corporation, San Jose, California, September 2002.

[Cor02e] Altera Corporation. Nios Tutorial. Tutorial TU-NIOSTTRL-1.1, Altera Corporation, San Jose, California, April 2002.

[CR93] Steven A. Cuccaro and Craig F. Reese. The CM-2X: A Hybrid CM-2/Xilinx Prototype. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 121–130, April 1993.

[CTS95] Steve Casselman, Michael Thornburg, and John Schewel. Hardware Object Programming on the EVC1–A Reconfigurable Computer. In *Proceedings of the 1995 SPIE Photonics East Conference*, Philadelphia, Pennsylvania, October 1995.

[CTS96] Steve Casselman, Michael Thornburg, and John Schewel. H.O.T. Works Development System. World Wide Web Document, December 1996. `http://www.vcc.com/products/pci6200.html`.

[Deh94] André Dehon. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 31–39, April 1994.

[Deh96] André Dehon. *Reconfigurable Architectures for General-Purpose Computing.* Ph.D. Dissertation, Massachussetts Institute of Technology, Cambridge, Massachussetts, October 1996.

[DGJ+95] Michel Dubois, Alain Gefflaut, Jaeheon Jeong, Adrian Moga, and Koray Oner. Multiprocessor Emulation with RPM: Early Experience. Technical Report CENG95-23, University of Southern California, Los Angeles, California, December 1995.

[DN99] Edward N. Dekker and Joseph M. Newcomer. *Developing Windows NT Device Drivers.* AddisonWesley Publishing Company, Reading, Massachussetts, first edition, 1999.

[Dun90] Ralph Duncan. A Survey of Parallel Computer Architectures. *IEEE Computer*, February 1990.

[EBTB63] G. Estrin, B. Bussell, R. Turn, and J. Bibb. Parallel Processing in a Restructurable Computer System. *IEEE Transactions on Electronic Computers*, 12:747–755, December 1963.

[Fuj90] Richard M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):31–53, October 1990.

[Fuj93] Richard M. Fujimoto. Parallel and Distributed Discrete Event Simulation. In *Proceedings of the Winter Simulation Conference*, pages 106–114, 1993.

[GHK+91] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti. Building and Using a Highly Parallel Programmable Logic Array. *IEEE Computer*, 24(1):81–89, January 1991.

[GKC+94] David Galloway, David Karchmer, Paul Chow, David Lewis, and Jonathan Rose. The Transmogrifier: The University of Toronto Field-Programmable System. Technical Report CSRI-306, Computer Systems Research Institute, University of Toronto, Toronto, Ontario, Canada, June 1994.

[God93a] Michael Godfrey. Introduction to "The First Draft Report on the EDVAC". *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[God93b] Michael Godfrey. The Computer as von Neumann Planned It. *IEEE Annals of the History of Computing*, 15(1):11–21, 1993.

[Gra81] Jim Gray. The Transaction Concept, Virtues and Limitations. In *Proceedings of the Seventh International Conference on Very Large Databases*, pages 144–151, Cannes, France, September 1981.

[Gra01] David Grant. An ARC-PCI Board Device Driver for the Linux V2.4 Series Kernel. URA Report, University of Waterloo, Waterloo, Ontario, Canada, December 2001.

[Har01] Reiner Hartenstein. A Decade of Reconfigurable Computing: A Visionary Retrospective. In *Proceedings of Design, Automation, and Test in Europe (DATE '01)*, pages 642–649, Munich, Germany, March 2001.

[HD62] T. E. Hull and A. R. Dobell. Random Number Generators. *SIAM Review*, 4:230–254, 1962.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., Palo Alto, California, 1990.

[HP92] B. Heeb and C. Pfister. Chameleon: A Workstation of a Different Colour. In *Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping. Second International Workshop on Field-Programmable Logic and Applications*, pages 152–161, Vienna, Austria, August 1992.

[HP02] Jaret W. Hauge and Kerrie N. Paige. *Learning SIMUL8: The Complete Guide.* Plain Vu Publishers, March 2002.

[I-C94] I-Cube, Inc., Santa Clara, California. *The FPID Family Data Sheet*, February 1994.

[I-C97] Digital Crosspoint Switching Solutions. Online Technical Seminar (`ftp://ftp.icube.com/pub/icubePDF/Technology/DigitalCrosspointSwitching.pdf`), September 1997.

[Ins93] Institute of Electrical and Electronics Engineers, New York, New York. *IEEE Std 1076-1993, IEEE Standard VHDL Language Reference Manual*, 1993.

[Int02] Intel Microprocessor Quick Reference Guide. World Wide Web Document, December 2002.
`http://www.intel.com/pressroom/kits/quickreffam.htm`.

[IS93] Christian Iseli and Eduardo Sanchez. Spyder: A Reconfigurable VLIW Processor Using FPGAs. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 17–24, Napa Valley, California, April 1993.

[Jal94] Pankaj Jalote. *Fault Tolerance in Distributed Systems.* Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1994.

[Jen94] Jesse H. Jenkins. *Designing with FPGAs and CPLDs.* Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1994.

[KL95] Asawaree Kalavade and Edward A. Lee. Hardware/Software Codesign Using Ptolemy. In Jerzy Rozenblit and Klaus Buchenrieder, editors, *Codesign: Computer-Aided Software/Hardware Engineering*, chapter 19, pages 397–413. IEEE Press, New York, New York, 1995.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, Inc., Englewood Cliffs, New Jersey, 2nd edition, 1988.

[KS02] Kenneth B. Kent and Micaela Serra. Context Switching in a Hardware/Software Co-Design of the Java Virtual Machine. In *Proceedings of Design Automation and Test in Europe*, March 2002.

[Leh51] Derrick Henry Lehmer. Mathematical Methods in Large-Scale Computing Units. In *Proceedings of the Second Symposium on Large-Scale Digital Computing Machinery, 1949*, volume 26, pages 141–146, Cambridge, Massachusetts, 1951. Harvard University Press.

[LGv+97] David M. Lewis, David R. Galloway, Marcus van Ierssel, Jonathon Rose, and Paul Chow. The Transmogrifier-2: A 1 Million Gate Rapid Prototyping System. In *Proceedings of the 1997 ACM Fifth International Symposium on Field-Programmable Gate Arrays*, pages 53–61, Monterey, California, February 1997.

[LK91] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis.* McGraw-Hill, Inc., New York, New York, second edition, 1991.

[Man91] M. Morris Mano. *Digital Design.* Prentice Hall, Inc., Englewood Cliffs, New Jersey, second edition, 1991.

[MCMB93] George Milne, P. Cockshott, G McCaskill, and P. Barrie. Realizing Massively Concurrent Systems on the SPACE Machine. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 26–32, Napa Valley, California, April 1993.

[MD96] E. Mirsky and André Dehon. MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In J. M. Arnold and K. L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 157–166, Napa Valley, California, April 1996.

[Mes98a] Mesquite Software, Inc., Austin, Texas. *Getting Started: CSIM18 Simulation Engine (C++ Version)*, 1998.

[Mes98b] Mesquite Software, Inc., Austin, Texas. *User's Guide: CSIM18 Simulation Engine*, 1998.

[Mic02] Stratix EP1S25 Development Kit. Preliminary Product Specification, December 2002.

[Mil96] George Milne. SPACE 2 Reconfigurable Computing Platform. World Wide Web Document, December 1996.
`http://www.cis.unisa.edu.au/acrc/projects/space2.html`.

[One99] Walter Oney. *Programming the Microsoft Windows Driver Model.* Microsoft Press, Redmond, Washington, first edition, 1999.

[Opl67] Ascher Opler. Fourth-Generation Software. *Datamation*, 13(1):22–24, January 1967.

[PB99] Gajjala Purna and Dinesh Bhatia. Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers. *IEEE Transactions on Computers*, 48(6):579–590, June 1999.

[Phi97] Philips Semiconductors, Sunnyvale, California. *P3Z22V10 Data Sheet - Preliminary Specification*, March 1997.

[PML92] Bruno R. Preiss, Ian D. MacIntyre, and Wayne M. Loucks. On the Trade-off between Time and Space in Optimistic Parallel Discrete-Event Simulation. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS-92)*, pages 33–42, Newport Beach, California, January 1992.

[Pow01] Powersim Corporation. Features in Powersim Studio 2001. White Paper, Powersim Corporation, December 2001.
`http://www.powersim.com/common/pdf/studio_2001_info.pdf`.

[Pre99] Bruno R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. John Wiley & Sons, Inc., New York, New York, first edition, 1999.

[PW99] Bruno R. Preiss and K. W. Carey Wan. The Parsimony Project: A Distributed Simulation Testbed in Java. In *Proceedings of the 1999 International Conference On Web-Based Modelling & Simulation*, San Francisco, California, January 1999.

[RH97] Jonathon Rose and Dwight Hill. Architectural and Physical Design Challenges for One-Million Gate FPGAs and Beyond. In *Proceedings of the 1997 ACM Fifth International Symposium on Field-Programmable Gate Arrays*, pages 129–132, Monterey, California, February 1997.

[Rub98] Alessandro Rubini. *Linux Device Drivers*. OŔeilly, Sebastopol, California, first edition, February 1998.

[SA95] Tom Shanley and Don Anderson. *PCI System Architecture*. AddisonWesley Publishing Company, Reading, Massachussetts, third edition, September 1995.

[Sch86] Herbert D. Schwetman. CSIM: A C-Based, Process-Oriented Simulation Language. In *Proceedings of the 1986 Winter Simulation Conference*, pages 387–396, December 1986.

[SP00] Daniel Schunk and Beth Plott. Using Simulation to Analyze Supply Chains. In *Proceedings of the 2000 Winter Simulation Conference*, Orlando, Florida, December 2000.

[Str94] Bjarne Stroustrup. *The C++ Programming Language*. AddisonWesley Publishing Company, Reading, Massachussetts, second edition, 1994.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1995.

[TAS93] Donald E. Thomas, Jay K. Adams, and Herman Schmit. A Model and Methodology for Hardware-Software Codesign. *IEEE Design & Test of Computers*, 10(3):6–15, 1993.

[Tri94] Stephen M. Trimberger, editor. *Field-Programmable Gate Array Technology*. Kluwer Academic Publishers, Norwell, Massachussetts, 1994.

[VBR+96] Jean E. Vuillemin, Patric Bertin, Didier Roncin, Mark Shand, Hervé H. Touati, and Philippe Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(1):56–69, March 1996.

[VCC98] Virtual Computer Corporation. H.O.T. II Development System. Data Sheet, Virtual Computer Corporation, Reseda, California, 1998.
`http://www.vcc.com/Papers/Hotii_DS.PDF`.

[VM99] Peter G. Viscarola and W. Anthony Mason. *Windows NT Device Driver Development.* MacMillan Technical Publishing, Indianapolis, Indiana, first edition, 1999.

[VMT+92] David E. Van Den Bout, Joseph N. Morris, Douglas Thomae, Scott Labrozzi, Scot Wingo, and Dean Hallman. AnyBoard: An FPGA-Based Reconfigurable System. *IEEE Design & Test of Computers*, 9(3):21–30, 1992.

[vN45] John von Neumann. First Draft of a Report on the EDVAC. Technical Report W-6700RD-492, Moore School of Electrical Engineering, University of Pennsylvannia, June 1945.

[Von97] Brian Von Herzen. Signal Processing at 250 MHz using High-Performance FPGA's. In *Proceedings of the 1997 ACM Fifth International Symposium on Field-Programmable Gate Arrays*, pages 62–68, Monterey, California, February 1997.

[WAL+93] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. PRISM-II Compiler and Architecture. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 9–16, Napa Valley, California, April 1993.

[WE93] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective.* AddisonWesley Publishing Company, Reading, Massachussetts, second edition, 1993.

[WHG94] Michael J. Wirthlin, Brad L. Hutchings, and Kent L. Gilson. The Nano Processor: A Low Resource Reconfigurable Processor. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 23–30, Napa Valley, California, April 1994.

[X E97] XS40 Board. World Wide Web Document, January 1997. `http://www.xess.com/FPGA/ho02002.html`.

[Xil97] Xilinx University Program Internet Seminar. Online Internet Seminar, July 1997. `http://www.enen.com/`.

[Xil02] Xilinx, Inc., San Jose, California. *Virtex-II Pro Prototype Platform User Guide*, October 2002.

Appendix A

Experimental Results for Transfer Rates

This appendix provides the complete set of experiment results for the transfer rates experiments summarized in Chapter 4. Each experiment consisted of 10,000,000 consecutive transfers. Experiments were run five times. The total execution time for each experimental run is shown. The average execution time and standard deviation are also shown.

A.1 Platform I: Windows NT Results

Table A.1 shows the device driver execution times observed on Platform I running Windows NT. Software timing was used.

Windows Device Driver Execution Times - Software Timed (WSPEED1A)							
Transaction Name	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
Interface Read	5.458	5.448	5.447	5.448	5.448	5.450	0.005
Interface Write	2.213	2.203	2.234	2.203	2.203	2.211	0.013
Memory Read	5.298	5.307	5.308	5.297	5.308	5.304	0.006
Memory Write	2.203	2.193	2.204	2.203	2.203	2.201	0.005
PLD Read	5.478	5.448	5.457	5.448	5.448	5.456	0.013
PLD Write	2.203	2.203	2.194	2.233	2.203	2.207	0.015

Table A.1: Windows Device Driver Execution Times - Software Timed (WSPEED1A)

Table A.2 shows the device driver execution times observed on Platform I running Windows NT. Hardware timing was used.

Table A.3 shows the unbuffered application execution times observed on Platform I running Windows NT. Software timing was used.

Windows Device Driver Execution Times - Hardware Timed (WSPEED1B)							
Transaction Name	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
Interface Read	5.451	5.440	5.435	5.444	5.434	5.441	0.007
Interface Write	2.206	2.199	2.226	2.199	2.202	2.206	0.011
Memory Read	5.291	5.294	5.300	5.293	5.299	5.295	0.004
Memory Write	2.198	2.193	2.198	2.195	2.203	2.197	0.004
PLD Read	5.467	5.440	5.445	5.447	5.435	5.447	0.012
PLD Write	2.198	2.197	2.194	2.228	2.196	2.203	0.014

Table A.2: Windows Device Driver Execution Times - Hardware Timed (WSPEED1B)

Windows Unbuffered Application Execution Times (WSPEED2)							
Transaction Name	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
Interface Read	25.086	25.086	25.086	25.086	25.086	25.086	0.000
Interface Write	22.031	22.022	22.022	21.991	21.972	22.008	0.025
Memory Read	25.366	25.357	25.346	25.427	25.376	25.374	0.031
Memory Write	21.821	21.752	21.751	21.791	21.752	21.773	0.032
PLD Read	25.397	25.447	25.406	25.397	25.396	25.409	0.022
PLD Write	21.751	21.742	21.751	21.741	21.751	21.747	0.005

Table A.3: Windows Unbuffered Application Execution Times (WSPEED2)

Table A.4 shows the buffered application execution times observed on Platform I running Windows NT. Software timing was used.

Windows Buffered Application Execution Times (WSPEED3)							
Transaction Name	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
Interface Read	5.798	5.789	5.788	5.788	5.789	5.790	0.004
Interface Write	3.105	3.044	3.004	3.015	3.004	3.034	0.043
Memory Read	5.789	5.788	5.798	5.779	5.798	5.790	0.008
Memory Write	3.024	3.014	3.025	3.024	3.014	3.020	0.006
PLD Read	5.798	5.788	5.799	5.828	5.828	5.808	0.019
PLD Write	3.004	3.015	3.004	3.014	3.005	3.008	0.006

Table A.4: Windows Buffered Application Execution Times (WSPEED3)

A.2 Platform I: Linux Results

Table A.5 shows the device driver execution times observed on Platform I running Linux. Software timing was used. These results were obtained by David Grant [Gra01]. They have been provided for the purpose of comparison.

Table A.6 shows the unbuffered application execution times observed on Platform I running Linux. Software timing was used. These results were obtained by David Grant [Gra01]. They

Linux Device Driver Execution Times (LSPEED1)							
Transaction Name	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
Interface Read	4.646	4.646	4.646	4.646	4.646	4.646	0.000
Interface Write	2.098	2.099	2.098	2.099	2.099	2.099	0.000
Memory Read	4.646	4.648	4.649	4.648	4.648	4.648	0.001
Memory Write	2.099	2.098	2.099	2.099	2.098	2.099	0.000
PLD Read	4.646	4.646	4.646	4.647	4.646	4.646	0.001
PLD Write	2.098	2.098	2.098	2.098	2.098	2.098	0.000

Table A.5: Linux Device Driver Execution Times (LSPEED1)

have been provided for the purpose of comparison.

Linux Unbuffered Application Execution Times (LSPEED2)							
Transaction Name	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
Interface Read	23.087	23.090	23.092	23.090	23.125	23.097	0.016
Interface Write	19.191	19.188	19.188	19.187	19.255	19.202	0.030
Memory Read	23.085	23.085	23.085	23.087	23.087	23.086	0.001
Memory Write	19.188	19.188	19.191	19.189	19.188	19.189	0.001
PLD Read	23.123	23.085	23.088	23.087	23.085	23.093	0.016
PLD Write	19.189	19.189	19.187	19.188	19.187	19.188	0.001

Table A.6: Linux Unbuffered Application Execution Times (LSPEED2)

Table A.7 shows the buffered application execution times observed on Platform I running Linux. Software timing was used. These results were obtained by David Grant [Gra01]. They have been provided for the purpose of comparison.

Linux Buffered Application Execution Times (LSPEED3)							
Transaction Name	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
Interface Read	5.233	5.232	5.232	5.232	5.233	5.233	0.000
Interface Write	2.417	2.417	2.417	2.418	2.418	2.417	0.000
Memory Read	5.240	5.238	5.237	5.237	5.237	5.238	0.001
Memory Write	2.418	2.417	2.417	2.417	2.417	2.417	0.000
PLD Read	5.228	5.228	5.228	5.228	5.228	5.228	0.000
PLD Write	2.417	2.417	2.417	2.417	2.417	2.417	0.000

Table A.7: Linux Buffered Application Execution Times (LSPEED3)

Appendix B

Experimental Results for Pseudo-Random Number Generation

This appendix provides the complete set of experiment results for the pseudo-random number generation experiments summarized in Chapter 6. Each experiment performed a specific number of pseudo-random number generation iterations on the test platform. Each iteration calculates one pseudo-random number. Experiments were run five times. The total execution time for each experimental run is shown. The average execution time and standard deviation are also shown.

B.1 Platform I Results

Table B.1 shows the application execution times observed on Platform I running Windows NT. Software timing was used. A configurable coprocessor was not used to obtain these results. The results serve a baseline for comparison.

Table B.2 shows the application execution times observed on Platform I running Windows NT. Software timing was used. The unoptimized configurable coprocessor system was used to obtain these results.

Table B.3 shows the application execution times observed on Platform I running Windows NT. Software timing was used. The optimized configurable coprocessor system was used to obtain these results.

Table B.4 shows the application execution times observed on Platform I running Windows NT with L1 and L2 caching disabled. Software timing was used. A configurable coprocessor was not used to obtain these results. The results serve a baseline for comparison of experimental results with the L1 and L2 cache disabled.

Table B.5 shows the application execution times observed on Platform I running Windows

PC Software RAND Tests (PRAND1)							
PRNG Iterations	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
500000	0.020	0.010	0.010	0.010	0.010	0.013	0.006
1000000	0.020	0.020	0.030	0.020	0.020	0.023	0.006
2500000	0.060	0.050	0.060	0.051	0.060	0.057	0.006
5000000	0.110	0.110	0.120	0.110	0.120	0.113	0.006
10000000	0.221	0.220	0.230	0.221	0.230	0.224	0.006
25000000	0.561	0.561	0.560	0.571	0.561	0.561	0.001
50000000	1.132	1.121	1.122	1.132	1.121	1.125	0.006
100000000	2.253	2.254	2.263	2.243	2.253	2.257	0.006
250000000	5.629	5.628	5.628	5.618	5.658	5.628	0.001
500000000	11.286	11.266	11.257	11.246	11.266	11.270	0.015

Table B.1: PRAND1

PC Hardware RAND Tests (PRAND2)							
PRNG Iterations	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
500000	1.281	1.272	1.272	1.272	1.272	1.274	0.004
1000000	2.533	2.544	2.554	2.543	2.544	2.544	0.007
2500000	6.349	6.349	6.349	6.350	6.349	6.349	0.000
5000000	12.698	12.698	12.698	12.699	12.688	12.696	0.005
10000000	25.467	25.396	25.397	25.386	25.397	25.409	0.033
25000000	63.491	63.561	63.472	63.491	63.571	63.517	0.045
50000000	127.013	127.033	127.022	126.973	127.033	127.015	0.025
100000000	254.005	254.095	254.075	253.986	254.045	254.041	0.046
250000000	635.093	635.073	635.134	635.183	634.983	635.093	0.075
500000000	1270.116	1270.087	1269.976	1270.086	1270.036	1270.060	0.055

Table B.2: PRAND2

PC Hardware RAND Tests (PRAND3)							
PRNG Iterations	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
500000	0.290	0.291	0.290	0.291	0.290	0.290	0.001
1000000	0.581	0.581	0.611	0.590	0.581	0.589	0.013
2500000	1.452	1.452	1.463	1.452	1.462	1.456	0.006
5000000	2.914	2.914	2.914	2.904	2.915	2.912	0.005
10000000	5.818	5.818	5.829	5.818	5.819	5.820	0.005
25000000	14.560	14.551	14.551	14.551	14.581	14.559	0.013
50000000	29.142	29.112	29.142	29.102	29.112	29.122	0.019
100000000	58.263	58.224	58.204	58.203	58.284	58.236	0.036
250000000	145.600	145.599	145.599	145.590	145.559	145.589	0.017
500000000	291.159	291.128	291.249	291.109	291.188	291.167	0.055

Table B.3: PRAND3

PC Software RAND Tests (PRAND1NC)							
PRNG Iterations	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
500000	2.904	2.985	2.904	2.964	2.984	2.931	0.047
1000000	5.658	6.099	5.708	5.829	5.908	5.822	0.241
2500000	14.411	14.761	14.791	14.812	14.491	14.654	0.211
5000000	31.174	29.964	29.091	29.252	29.303	30.076	1.046
10000000	58.894	60.938	58.093	58.505	58.634	59.308	1.467
25000000	149.155	149.234	147.112	149.635	148.704	148.500	1.203
50000000	297.488	297.808	291.870	292.240	293.772	295.722	3.340
100000000	578.662	572.043	572.042	571.993	569.569	574.249	3.822
250000000	1428.544	1428.134	1428.544	1429.796	1426.581	1428.407	0.237
500000000	2856.708	2856.237	2856.197	2856.888	2858.270	2856.381	0.284

Table B.4: PRAND1NC

PC Hardware RAND Tests (PRAND2NC)							
PRNG Iterations	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
500000	147.833	143.786	143.517	140.632	143.797	143.913	2.565
1000000	284.439	284.309	282.085	281.485	284.128	283.289	1.394
2500000	705.845	707.257	705.615	705.144	708.028	706.378	1.213
5000000	1410.408	1413.262	1413.212	1413.363	1412.361	1412.521	1.248
10000000	2827.305	2824.441	2823.370	2825.002	2825.413	2825.106	1.449
25000000	7061.094	7058.659	7062.486	7062.475	7065.590	7062.061	2.516
50000000	14129.177	14118.771	14122.357	14126.764	14131.209	14125.656	5.068
100000000	28265.343	28238.995	28242.099	28255.342	28262.447	28252.845	11.853
250000000	70673.841	70599.667	70601.325	70641.076	70656.161	70834.414	33.068
500000000	141354.671	141200.787	141200.035	141283.966	141312.351	141270.362	68.634

Table B.5: PRAND2NC

NT with L1 and L2 caching disabled. Software timing was used. The unoptimized configurable coprocessor system was used to obtain these results. Results shown in gray are estimates based on linear projections.

Table B.6 shows the application execution times observed on Platform I running Windows NT with L1 and L2 caching disabled. Software timing was used. The optimized configurable coprocessor system was used to obtain these results.

B.2 Platform II Results

Table B.7 shows the application execution times observed on Platform II. Software timing was used. A configurable coprocessor was not used to obtain these results. The results serve a baseline for comparison.

Table B.8 shows the application execution times observed on Platform II. Software timing was used. A configurable coprocessor system was used to obtain these results.

PC Hardware RAND Tests (PRAND3NC)							
PRNG Iterations	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
500000	5.658	5.809	5.988	5.528	5.979	5.792	0.201
1000000	11.476	11.637	11.777	11.296	11.687	11.575	0.190
2500000	29.222	31.045	29.382	29.392	28.842	29.577	0.851
5000000	58.834	59.276	58.233	57.944	57.883	58.434	0.603
10000000	118.110	117.579	117.098	118.320	116.308	117.483	0.811
25000000	292.640	293.542	293.142	292.791	293.652	293.153	0.446
50000000	585.843	586.423	586.894	587.024	586.503	586.537	0.464
100000000	1172.917	1172.336	1173.657	1172.446	1171.805	1172.632	0.696
250000000	2932.507	2930.844	2931.395	2933.368	2931.285	2931.880	1.033
500000000	5862.450	5863.221	5862.690	5861.138	5859.375	5861.775	1.545

Table B.6: PRAND3NC

Excalibur Software RAND Tests (ERAND1)							
PRNG Iterations	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
500000	1.381	1.382	1.382	1.381	1.382	1.382	0.001
1000000	2.764	2.763	2.763	2.764	2.763	2.763	0.001
2500000	6.909	6.909	6.908	6.909	6.908	6.909	0.001
5000000	13.818	13.817	13.817	13.817	13.818	13.817	0.001
10000000	27.634	27.635	27.634	27.635	27.634	27.634	0.001
25000000	69.086	69.087	69.086	69.086	69.086	69.086	0.001
50000000	138.173	138.172	138.173	138.172	138.172	138.173	0.001
100000000	276.345	276.345	276.345	276.345	276.345	276.345	0.000
250000000	690.862	690.862	690.862	690.863	690.862	690.862	0.000
500000000	1381.724	1381.725	1381.724	1381.724	1381.725	1381.724	0.001

Table B.7: ERAND1

Excalibur Hardware RAND Tests (ERAND2)							
PRNG Iterations	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
500000	0.513	0.513	0.514	0.513	0.514	0.513	0.001
1000000	1.026	1.027	1.027	1.027	1.027	1.027	0.000
2500000	2.567	2.567	2.567	2.567	2.568	2.567	0.000
5000000	5.134	5.134	5.134	5.135	5.134	5.134	0.000
10000000	10.269	10.268	10.269	10.268	10.269	10.269	0.001
25000000	25.671	25.672	25.671	25.671	25.672	25.671	0.001
50000000	51.343	51.343	51.342	51.343	51.343	51.343	0.000
100000000	102.686	102.685	102.686	102.685	102.686	102.686	0.001
250000000	256.714	256.714	256.714	256.714	256.714	256.714	0.000
500000000	513.428	513.429	513.428	513.428	513.428	513.428	0.000

Table B.8: ERAND2

Sun Software RAND Tests (SRAND1)							
PRNG Iterations	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
500000	0.146	0.148	0.145	0.146	0.146	0.146	0.002
1000000	0.293	0.292	0.287	0.290	0.299	0.291	0.003
2500000	0.723	0.731	0.733	0.736	0.733	0.729	0.005
5000000	1.465	1.474	1.470	1.482	1.455	1.470	0.005
10000000	2.949	2.927	2.949	2.991	3.022	2.942	0.013
25000000	7.344	7.316	7.391	7.326	7.399	7.350	0.038
50000000	14.685	14.650	14.747	14.758	14.742	14.694	0.049
100000000	29.825	29.538	29.475	29.505	29.377	29.613	0.187
250000000	73.518	73.630	74.265	77.960	73.524	73.804	0.403
500000000	148.181	147.057	147.115	151.683	147.349	147.451	0.633

Table B.9: SRAND1

B.3 Platform III Results

Table B.9 shows the application execution times observed on Platform III running Solaris. Software timing was used. A configurable coprocessor was not used to obtain these results. The results serve a baseline for comparison. These results show that the PC platform performs comparably with other workstations.

Appendix C

Experimental Results for Minheap Management

This appendix provides the complete set of experiment results for the minheap management experiments summarized in Chapter 7. Each experiment performed a total of 5,000,000 sequences of insertions and deletions to a specified minheap size. Experiments were run five times. The total execution time for each experimental run is shown. The average execution time and standard deviation are also shown.

C.1 Platform I Results

Table C.1 shows the application execution times observed on Platform I running Windows NT. Software timing was used. A configurable coprocessor was not used to obtain these results. The results serve a baseline for comparison.

PC Software MIN Tests (PMIN1)							
Heap Entries	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
50	0.191	0.200	0.190	0.191	0.190	0.192	0.004
100	0.420	0.411	0.411	0.420	0.411	0.415	0.005
250	1.182	1.191	1.182	1.182	1.181	1.184	0.004
500	2.584	2.574	2.594	2.573	2.574	2.580	0.009
1000	5.558	5.568	5.558	5.568	5.558	5.562	0.005
2500	15.432	15.432	15.433	15.442	15.502	15.448	0.030
5000	33.999	33.999	33.969	33.969	33.968	33.981	0.017
10000	74.908	74.888	74.908	74.847	74.838	74.878	0.033
25000	211.604	211.624	211.575	211.624	211.584	211.602	0.022
50000	462.625	462.586	462.615	462.515	462.575	462.583	0.043

Table C.1: PMIN1

PC Hardware MIN Tests (PMIN2)							
Heap Entries	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
50	2.443	2.484	2.443	2.434	2.433	2.447	0.021
100	5.148	5.147	5.138	5.137	5.137	5.141	0.006
250	13.740	13.780	13.770	13.749	13.740	13.756	0.018
500	28.882	28.871	28.882	28.871	28.882	28.878	0.006
1000	60.507	60.457	60.447	60.507	60.447	60.473	0.031

Table C.2: PMIN2

PC Hardware MIN Tests (PMIN3)							
Heap Entries	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
50	1.972	1.963	1.983	1.963	1.963	1.969	0.009
100	3.685	3.695	3.686	3.695	3.685	3.689	0.005
250	9.254	9.263	9.253	9.344	9.263	9.275	0.039
500	19.949	19.928	19.949	19.939	19.938	19.941	0.009
1000	41.911	41.970	41.920	41.910	41.901	41.922	0.027

Table C.3: PMIN3

Table C.2 shows the application execution times observed on Platform I running Windows NT. Software timing was used. The unoptimized configurable coprocessor system was used to obtain these results.

Table C.3 shows the application execution times observed on Platform I running Windows NT. Software timing was used. The optimized configurable coprocessor system was used to obtain these results.

Table C.4 shows the application execution times observed on Platform I running Windows NT with L1 and L2 caching disabled. Software timing was used. A configurable coprocessor was not used to obtain these results. The results serve a baseline for comparison of experimental results with the L1 and L2 cache disabled.

Table C.5 shows the application execution times observed on Platform I running Windows NT with L1 and L2 caching disabled. Software timing was used. The unoptimized configurable coprocessor system was used to obtain these results. Results shown in gray are estimates based on linear projections.

Table C.6 shows the application execution times observed on Platform I running Windows NT with L1 and L2 caching disabled. Software timing was used. The optimized configurable coprocessor system was used to obtain these results.

C.2 Platform II Results

Table C.7 shows the application execution times observed on Platform II. Software timing was used. A configurable coprocessor was not used to obtain these results. The results serve a baseline

PC Software MIN Tests (PMIN1NC)							
Heap Entries	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
50	38.766	41.089	39.507	38.876	39.147	39.477	0.945
100	89.418	86.514	86.605	89.398	86.405	87.668	1.590
250	247.186	248.377	247.606	246.634	247.535	247.468	0.637
500	537.773	537.152	539.786	545.645	542.580	540.587	3.533
1000	1165.706	1162.922	1164.435	1161.480	1166.187	1164.146	1.955
2500	3207.041	3238.336	3231.938	3203.646	3200.763	3216.345	17.445
5000	6844.522	6839.284	6848.648	6853.474	6908.755	6858.937	28.335
10000	14674.581	14563.912	14563.311	14562.911	14554.198	14583.783	50.915
25000	39313.069	39353.117	39261.355	39257.199	39024.484	39241.845	127.785
50000	93103.627	82763.308	82923.569	82768.660	82847.518	84881.336	4596.868

Table C.4: PMIN1NC

PC Hardware MIN Tests (PMIN2NC)							
Heap Entries	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
50	137.848	140.172	140.231	138.139	140.212	139.320	1.216
100	278.420	280.714	278.861	278.009	281.355	279.472	1.476
250	697.423	697.813	699.306	698.604	697.043	698.038	0.915
500	1400.093	1399.893	1396.548	1399.222	1399.442	1399.040	1.435
1000	2798.724	2795.599	2800.057	2797.002	2798.724	2798.021	1.735

Table C.5: PMIN2NC

PC Hardware MIN Tests (PMIN3NC)							
Heap Entries	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
50	23.524	21.100	20.960	21.311	21.210	21.621	1.072
100	37.334	37.073	39.988	37.674	38.415	38.097	1.171
250	86.535	88.817	87.216	88.727	86.264	87.512	1.202
500	175.152	176.254	174.481	173.679	175.493	175.012	0.981
1000	349.943	348.541	347.850	349.393	349.793	349.104	0.888

Table C.6: PMIN3NC

Excalibur Software MIN Tests (EMIN1)							
Heap Entries	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
50	14.765	14.765	14.766	14.765	14.766	14.765	0.001
100	32.133	32.133	32.133	32.133	32.133	32.133	0.000
250	88.767	88.767	88.766	88.767	88.767	88.767	0.000
500	190.644	190.644	190.644	190.644	190.644	190.644	0.000
1000	407.414	407.415	407.414	407.415	407.414	407.414	0.001
2500	1105.996	1105.996	1105.996	1105.996	1105.996	1105.996	0.000
5000	2342.114	2342.113	2342.113	2342.114	2342.113	2342.113	0.001
10000	4944.313	4944.313	4944.313	4944.313	4944.313	4944.313	0.000
25000	13224.603	13224.603	13224.603	13224.603	13224.603	13224.603	0.000

Table C.7: EMIN1

Excalibur Hardware MIN Tests (EMIN2)							
Heap Entries	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
50	1.612	1.613	1.612	1.613	1.612	1.612	0.001
100	3.362	3.361	3.361	3.361	3.362	3.361	0.001
250	8.921	8.921	8.921	8.921	8.921	8.921	0.000
500	18.710	18.710	18.710	18.710	18.710	18.710	0.000
1000	39.191	39.191	39.191	39.191	39.191	39.191	0.000

Table C.8: EMIN2

for comparison.

Table C.8 shows the application execution times observed on Platform II. Software timing was used. A configurable coprocessor system was used to obtain these results. An unoptimized version of the application was used to obtain these results. This version did not exploit parallelism aggressively.

Table C.9 shows the application execution times observed on Platform II. Software timing was used. A configurable coprocessor system was used to obtain these results. An optimized version of the application was used to obtain these results. This version exploited parallelism aggressively.

Excalibur Hardware MIN Tests (EMIN3)							
Heap Entries	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
50	1.468	1.469	1.469	1.468	1.469	1.469	0.001
100	2.934	2.935	2.934	2.935	2.934	2.934	0.001
250	7.336	7.337	7.336	7.336	7.336	7.336	0.000
500	14.681	14.681	14.681	14.682	14.681	14.681	0.000
1000	29.798	29.797	29.797	29.798	29.797	29.797	0.001

Table C.9: EMIN3

Sun Software MIN Tests (SMIN1)							
Heap Entries	Trial 1 (in s)	Trial 2 (in s)	Trial 3 (in s)	Trial 4 (in s)	Trial 5 (in s)	Average (in s)	Standard Deviation
50	0.590	0.677	0.473	0.879	0.802	0.684	0.162
100	1.819	1.427	1.738	1.049	0.829	1.372	0.429
250	2.289	2.289	2.286	2.274	2.274	2.282	0.008
500	4.895	4.935	4.900	4.898	4.901	4.906	0.016
1000	10.423	10.501	10.436	10.433	10.487	10.456	0.035
2500	28.400	28.427	28.709	29.859	28.467	28.772	0.620
5000	61.202	61.534	61.327	61.325	61.250	61.328	0.127
10000	136.992	132.959	132.394	132.427	136.870	134.328	2.387
25000	361.670	366.322	361.842	372.521	372.710	367.013	5.444
50000	868.259	793.701	798.169	794.263	796.450	810.168	32.523

Table C.10: SMIN1

C.3 Platform III Results

Table C.10 shows the application execution times observed on Platform III running Solaris. Software timing was used. A configurable coprocessor was not used to obtain these results. The results serve a baseline for comparison. These results show that the PC platform performs comparably with other workstations.

Appendix D

CSIM M/M/1 Queue Simulation Model

The M/M/1 queue simulation model used for the purpose of this research is a slightly modified version of the CSIM M/M/1 queue simulation model (Ex2cpp.cpp) to permit larger simulation runs. The modified version is shown below:

```
// C++/CSIM Model of M/M/1 Queue
//
// mm1.cpp (Based on Ex2cpp.cpp)
//
// Modified by Bill Bishop

#include "cpp.h"                          // CSIM class definitions
#include <stdio.h>

#define NARS   1000000                    // Number of arrivals
#define IAR_TM 2.0                        // Interarrival time
#define SRV_TM 2.0                        // Service time

event       done( "done" );               // Event named done
facility    f( "facility" );              // Facility named f
table       tbl( "resp tms" );            // Table of response times
qhistogram  qtbl( "num in sys", 101 );    // qhistogram of number in system
int         cnt;                          // Count of remaining customers
FILE        *fp;                          // Filehandle named fp

void customer( );
void theory( );

extern "C" void sim( int, char ** );

void sim( int argc, char *argv[] )
```

```
{
    max_processes( 1000000 );           // Set the process limit to 1,000,000

    fp = fopen( "csim.out", "w" );      // Create an output file
    set_output_file( fp );              // Instruct CSIM to use output file

    set_model_name( "M/M/1 Queue" );    // Name the simulation model
    create( "sim" );                    // Create a process named sim

    cnt = NARS;                         // Set the customer count
    for( int i = 1; i <= NARS; i++ )
    {
        hold( expntl( IAR_TM ) );       // Wait for interarrival interval
        customer( );                    // Generate next customer
    }
    done.wait( );                       // Wait for last customer to depart

    report( );                          // Generate a model report
    theory( );                          // Generate theoretical results
    mdlstat( );                         // Generate statistics on the model
}

void customer( )                        // Model an arriving customer
{
double t1;

    create( "cust" );                   // Create a process named cust

    t1 = clock;                         // Record start time
    qtbl.note_entry( );                 // Note arrival

    f.reserve( );                       // Reserve facility
    hold( expntl( SRV_TM ) );           // Service interval
    f.release( );                       // Release facility

    tbl.record( clock - t1 );           // Record response time
    qtbl.note_exit( );                  // Note departure

    if( --cnt == 0 )
    {
        done.set( );                    // If last customer, set done
    }
}

void theory( )                          // Print theoretical results
{
    double rho, nbar, rtime, tput;

    printf( "\n\n\n\t\t\tM/M/1 Theoretical Results\n" );

    tput  = 1.0 / IAR_TM;
```

```
    rho   = tput * SRV_TM;
    nbar  = rho / (1.0 - rho);
    rtime = SRV_TM / (1.0 - rho);

    printf( "\n\n ");
    printf( "\t\tInterarrival time  = %10.3f\n", IAR_TM );
    printf( "\t\tService time       = %10.3f\n", SRV_TM );
    printf( "\t\tUtilization        = %10.3f\n", rho );
    printf( "\t\tThroughput rate    = %10.3f\n", tput );
    printf( "\t\tMn nbr at queue    = %10.3f\n", nbar );
    printf( "\t\tMn queue length    = %10.3f\n", nbar-rho );
    printf( "\t\tResponse time      = %10.3f\n", rtime );
    printf( "\t\tTime in queue      = %10.3f\n", rtime - SRV_TM );
}
```