Managing a Reconfigurable Processor in a General Purpose Workstation Environment

Michael Winston Dales

UNIVERSITY
of
GLASGOW

Submitted for the degree of
Doctor of Philosophy
at the University of Glasgow

February 2003

Abstract

This dissertation considers the problems associated with using reconfigurable logic within a processor to accelerate applications from the context of a general purpose workstation, where this scarce resource will need to be shared by the operating system fairly and securely between a dynamic set of competing applications with no prior knowledge of their resource usage requirements.

A solution for these problems is proposed in the Proteus System, which describes a suitable set of operating system mechanisms, with appropriate hardware support, to allow the FPL resource to be virtualised and managed suitably without applications having to be aware of how the resource is allocated. We also describe a suitable programming model that would allow applications to be built with traditional techniques incorporating custom instructions.

We demonstrate the practicability of this system by simulating an ARM processor extended with a suitably structured FPL resource, upon which we run multiple applications that make use of one or more blocks of FPL, all of which are managed by a custom operating system kernel. We demonstrate that applications maintain a performance gain despite having to share the FPL resource between other applications.

This dissertation concludes that it is possible for an operating system to manage a reconfigurable processor within the context of a workstation environment, provided suitable hardware support, without negating the benefit of having the FPL resource, even during times of high load. It also concludes that the integration of custom hardware associated with applications into the system is manageable within existing programming techniques.

Acknowledgements

The following people must be acknowledged for their help and support:

- My supervisor team, which seems to have a higher turnover rate than expected (in alphabetical order): Dr. Richard Black, Prof. Gordon Brebner, Dr. Peter Dickman, Dr. Tommy Kelly, and Prof. Tom Melham. In addition I'd like to thank Prof. Ray Welland for helping sort out funding issues at the start, and Prof. John Gray for advice and encouragement, and John O'Leary of Intel for attending my 2nd year viva.
- All the people at Xilinx Edinburgh who have sponsored and nurtured me for the last n years, particularly Colin Carruthers (for keeping a watch and sorting things out), Gooby for being my internship boss, and to Scott Leishman for encouragement and advice.
- Those people who let me use compute cycles on their machines: the support folk, the Systems Research Group, Prof. Muffy Calder, Dr. Alice Miller, and Dr. Karen Renaud.
- My family, for accepting my antisocial behaviour and work hours in the last three years with grace.
- My fellow suffers, who reminded me I wasn't alone in all this and gave me a hand when need be: Adam Donlin, Huw Evans, Gary Gray, Jonathan Hogg, Sophie Huczynska, Gregory Hucznski, Rolf Neugebauer, Jonathan Paisley, Peter Saffery, Deborah Telfer, and Dave Watson. I'm sure my liver will recover at some point.
- Plus non–fellow suffers who pitched in all the same: Steffi Hahn and Neil Matheson.
- Coffee purveyors Little Italy and Heart Buchanan.
- And finally to Claire, for putting up with me during it all, despite the red hair.

Contents

1 Introduction **1**
- 1.1 Context . . . 2
- 1.2 Thesis Statement . . . 3
- 1.3 Contribution . . . 3
- 1.4 Outline . . . 4
- 1.5 Related Publications . . . 4

2 Background and Related Work **5**
- 2.1 Field Programmable Logic . . . 5
  - 2.1.1 FPGA Overview . . . 6
  - 2.1.2 Configuration . . . 7
- 2.2 OS Resource Management . . . 8
  - 2.2.1 Process Scheduling . . . 8
  - 2.2.2 Memory Management . . . 10
- 2.3 Research into Microprocessor/FPL Hyrbrids . . . 12
  - 2.3.1 Loosely Coupled FPGA and CPU Systems . . . 13
  - 2.3.2 Tightly Coupled FPGA and CPU Systems . . . 13
  - 2.3.3 CPU on FPGA Systems . . . 16
- 2.4 Existing Hybrid Systems . . . 17
  - 2.4.1 Xilinx Virtex II Pro . . . 18
  - 2.4.2 Other Commercial Hybrids . . . 20
- 2.5 FPL Management . . . 21
- 2.6 Security . . . 24
- 2.7 Architecture Evaluation . . . 25
  - 2.7.1 Overall Approach . . . 25
  - 2.7.2 FPL and Processor Integration Style . . . 25
- 2.8 Summary . . . 28

3 Requirements and High–Level Design **29**
- 3.1 General Approach . . . 29
- 3.2 Application-Level Requirements . . . 30
  - 3.2.1 Instruction Interface . . . 31
  - 3.2.2 Software Alternative . . . 31
  - 3.2.3 Namespace Management . . . 32
- 3.3 Operating System Requirements . . . 32
  - 3.3.1 Name Mapping . . . 33

3.3.2 Scheduling . . . 33
3.4 Hardware Level Requirements . . . 34
3.4.1 Dispatch Mechanism . . . 34
3.5 Device Programming . . . 35
3.6 Security . . . 35
3.7 Summary . . . 36

4 The Proteus Architecture 37
4.1 The Proteus Architecture . . . 38
4.1.1 General Approach . . . 38
4.1.2 The FPL Fabric . . . 39
4.1.3 PFU Interface . . . 41
4.1.4 The Dispatch Mechanism . . . 41
4.1.5 Software Dispatch . . . 44
4.1.6 Long Instruction Support . . . 47
4.1.7 Usage Information . . . 50
4.2 Initial Implementation . . . 51
4.2.1 The ARM7TDMI . . . 51
4.2.2 ProteanARM Overview . . . 53
4.2.3 PFU Interface . . . 54
4.2.4 Dispatch Mechanism . . . 56
4.2.5 ProteanARM Instruction Summary . . . 57
4.3 Architecture Simulation . . . 59
4.3.1 The SWARM Simulator . . . 59
4.3.2 Basic Performance . . . 60
4.3.3 Memory Interaction . . . 66
4.4 Summary . . . 68

5 Operating System Management 69
5.1 Custom Instruction Definition . . . 69
5.2 Operating System Support . . . 71
5.2.1 POrSCHE Overview . . . 72
5.2.2 System Call Interface . . . 72
5.2.3 Sharing Support . . . 74
5.2.4 The Custom Instruction Scheduler . . . 75
5.2.5 Scheduling Evaluation . . . 79
5.2.6 Run Time Statistics Monitoring . . . 83
5.3 Summary . . . 88

6 Application Development Support 90
6.1 Overiew Of Program Construction . . . 91
6.1.1 Compilation . . . 91
6.1.2 Static Linking . . . 92
6.1.3 Dynamic Linking . . . 92
6.1.4 The Run-Time Environment . . . 94
6.2 Requirements . . . 94
6.3 The Programmer's View . . . 95

6.4 Compilation 95
6.5 Static Linking 96
6.6 Dynamic Linking 97
6.6.1 Straight Link Approach 97
6.6.2 Demand Link Approach 98
6.6.3 Block Link Approach 99
6.7 Run-Time Environment 99
6.8 Summary 100

7 Further Work **102**
7.1 Architecture 102
7.1.1 Segmentation Based Architecture 102
7.1.2 Embedded Systems Targeting 102
7.1.3 Integration Into a Workstation Modern CPU Core 102
7.2 Operating System 103
7.2.1 Further Load Testing 103
7.2.2 Integration With Process Scheduler 103
7.2.3 Advanced Management Interface 103
7.3 Programming Model 104
7.3.1 Compiler Support 104
7.3.2 CID Namespace Management 104
7.4 Other Observations 104

8 Conclusion **105**

Bibliography **107**

List of Figures

2.1 Overview of Xilinx Virtex architecture and CLB layout 6
2.2 Process states 9
2.3 Two virtual memory spaces sharing a physical memory 11
2.4 Banded FPL allocation using control buses 17
2.5 Overview of PowerPC 405 core 18
2.6 CoreConnect architecture used on the Virtex-II Pro 19

4.1 Overview of the Proteus Architecture 39
4.2 Basic PFU Interface 42
4.3 The two-stage dispatch mechanism 44
4.4 Datapath for the ARM 7 52
4.5 Datapath of the ProteanARM core 53
4.6 Data formats for TLB entries 57
4.7 Instruction formats for ProteanARM ISA extensions 58
4.8 Twofish h function 63
4.9 Performance of core algorithms for test applications for both accelerated and unaccelerated implementations 65
4.10 Memory traces for the Twofish implementations 67
4.11 Memory trace for the protean Twofish implementation with zero keying 67

5.1 Layout of a custom instruction 70
5.2 Control flow in response to a custom instruction fault 78
5.3 Test run results for Alpha Blending and Twofish encryption 80
5.4 Test run results for audio echo test 81
5.5 Test run results for audio echo with modified scheduling policy 82
5.6 Software dispatch test results 83
5.7 Initial statistics output for varying sampling periods with 10 ms scheduling quanta . . 85
5.8 Initial statistics output for varying sampling periods with 1 ms scheduling quanta . . 86
5.9 Modified statistics output for 50 ms sampling periods 87

6.1 Overview of traditional compile and link flow 91
6.2 Shared libraries in a multiple address space operating system 93
6.3 Source building flow for new programming model 101

List of Tables

3.1 Summary of custom instruction names 36

4.1 Twofish performance with different key lengths and options 62
4.2 Twofish performance with different key lengths and options 64

Glossary

ABI	Application Binary Interface
APCS	ARM Procedure Call Standard
ASIC	Application Specific Integrated Circuit
bpp	bits per pixel
BRAM	Block RAM
CAM	Content Addressable Memory
CAU	Configurable Array Unit
CCCU	Configuration Control and Caching Unit
CCM	Constant Coefficient Multiplier
CID	Circuit ID
CIS	Custom Instruction Scheduler
CISC	Complex Instruction Set Computing
CLB	Configuration Logic Block
CSoC	Configurable System on a Chip
CSI	Configurable System Interconnect
CSL	Configurable System Logic
DCR	Device Control Register
DCU	Data Cache Unit
DES	Data Encryption Standard
DISC	Dynamic Instruction Set Computer
DMC	Digital Macro Cell
DSP	Digital Signal Processing
ECU	Execution Control Unit
ELF	Executable and Linking Format
FIFO	First In/First Out
FPGA	Field Programmable Gate Array
FPL	Field Programmable Logic
FU	Function Unit
GOT	Global Offset Table
HALT	Higher Abstraction Level Threat
HDL	Hardware Description Language
HLL	High-Level Language
IC	Integrated Circuit
ICT	Inverted Circuit Table
ICU	Instruction Cache Unit
IOB	I/O Block
IP	Intellectual Property
IPT	Inverted Page Table
ISA	Instruction Set Architecture

LFU Least Frequently Used
LRU Least Recently Used
LUT Look Up Table
MELT Malicious Electrical Level Threat
MMU Memory Management Unit
OCM On-Chip Memory
OoO Out of Order
OPB On-chip Peripheral Bus
PCB Process Control Block
PFU Programmable Function Unit
PIC Position Independent Code
PID Process ID
PLB Processor Local Bus
PLT Procedure Linkage Table
PRISC Programmable Instruction Set Computing
RIEU Reconfigurable Instruction Execution Unit
RISC Reduced Instruction Set Computing
RPU Reconfigurable Processing Unit
RTE Run-Time Environment
SALT Signal Alteration Logic Threat
SASOS Single Address Space Operating System
SIMD Single Instruction/Multiple Data
SLU Swappable Logic Unit
SoC System on Chip
SRAM Static RAM
SWI Software Interrupt
TCB Thread Control Block
TDM Time Division Multiplexing
TLB Translation Lookaside Buffer
URISC Micro RISC

Chapter 1

Introduction

There has been recent research and commercial interest in bringing together Field Programmable Logic (FPL) and microprocessors on a single device. FPL provides a flexible hardware system that can be tailored at run–time to implement custom circuits, essentially providing the designer with a form of changeable hardware. Although FPL provides a flexible hardware layer, for many applications it does not remove the need for software; software is better for defining larger algorithms and control flow that would otherwise not fit in the FPL. In addition it has been demonstrated that although a pure hardware solution is faster than software, optimised software which may run almost as fast can be produced in less time [Shand 1997]. It would seem then that a hybrid approach would allow for the best of both worlds: fast development of most of the application, with the core calculations being enhanced in hardware. Because of the attractiveness of the hybrid option, both the research community and industry have began investigating ways of combining FPL and one or more processors on a single chip, providing the system designer with the flexibility to divide their application between software and custom hardware as appropriate.

Despite the uptake in the embedded systems domain, the workstation and personal computer markets have not followed this trend, instead expanding processors with an increasing number of domain specific instruction set extensions (e.g., MMX instructions on Intel processors for speeding up multimedia applications), and more and bigger caches. However, with more specialised function units being added to CPUs we begin to see a lower utilisation of the silicon. For example, multimedia extensions will benefit CAD programs or games, but are not required for databases or spreadsheets. Furthermore, the new instructions must be solutions to a generalised version of a particular problem, and may not suit novel approaches to that problem that do not work in the way the hardware vendor anticipated [Abrash 1996]. Given these problems with domain specific function units, it would appear to make sense to provide a more flexible solution and allow application designers to provide their own custom instructions.

The lack of acceptance of the processor hybrids can in part be attributed to the lack of support in allowing the operating system to manage the FPL resource in current solutions. On a workstation the operating system is responsible for dynamically sharing resources between multiple competing application on behalf of multiple users. This is unlike an embedded system where typically there is a single process running and resource allocation is done either statically, or dynamically but with a known usage pattern.

The problem of managing the reconfigurable logic resource on a workstation can be compared to that of virtual memory management. In virtual memory management the operating system has to allocate a scarce physical resource between multiple processes with no prior knowledge of the

demands that these processes will make on the resource. The demands that processes make on the resource are not static: the requirements will change throughout the lifetime of the application, and the operating system needs to be able to expand the allocation during periods of high demand and may attempt to redistribute the resource during periods of low demand. The operating system needs to attempt to provide each process with enough of the physical resource to allow it to make progress, but not at the expense of other processes; in other words, it needs to provide a degree of fairness when sharing out the resource. Because the operating system has granted multiple processes access to the same shared resource, it needs to prevent either malicious or accidental accesses by one process to another process's allocation. This can only be achieved successfully on conventional systems by hardware support, but it is the job of the operating system to control that hardware and provide the correct level of security.

At the same time as the operating system controls the location of custom hardware, the applications running on the system should be agnostic of the management issues. It is envisaged that the programmer will have a simple interface to the new resource, whereby the program will register circuits with the system and then call upon them to be executed as needed, without having to be aware of whether the circuit is currently loaded and if so where it is loaded. At the application level, the system should provide all the mechanisms that programmers are familiar with to support a conventional programming model; for instance, applications should be able to share circuits, both internally between instances of a given application and between multiple applications.

1.1 Context

The aim of this work is to demonstrate that a reconfigurable processor can be used in a workstation device using a conventionally structured operating system, such as a UNIX implementation or Windows NT (and its descendants). This style of operating system supports the execution of numerous concurrent tasks for multiple users. The management of the FPL resource on the processor must integrate successfully with the existing operating system technologies used in such a system, like pre-emptive multitasking and virtual memory management. In a workstation environment the operating system is responsible for managing accesses to resources, and as such it should also be responsible for sharing access to the FPL resource on the processor. It should ensure fair, secure, and efficient usage of the inherently limited FPL resource.

In this system it is expected that a small set of running applications will utilise custom hardware to accelerate key parts of their functionality. The system will also need to support a set of legacy and new applications that are not accelerated, or indeed do not need specific acceleration. This means that support for accelerated applications should not have a detrimental impact on the performance of unaccelerated applications.

Applications designed to run in a workstation environment are built upon a rich set of well understood mechanisms. This includes compilation of languages to object code and static and dynamic linking techniques. Applications using the FPL resource should be built using these existing techniques (indeed, it should be feasible for unaccelerated applications to be developed with the same tools), allowing existing programming practices to be applied.

The aim of this work is to find a way in which the FPL resource can be integrated into the system without having to dramatically alter any part of the system, be it the hardware model, the operating system, or the programming model.

1.2 Thesis Statement

I assert that it is possible to integrate a suitably structured reconfigurable processor into a general purpose workstation environment, such that the reconfigurable logic resource on the processor can be managed by the operating system, and shared fairly and securely between multiple processes without a priori information of the processes' behaviour, without negating the performance benefit provided by the extra resource, or adversely affecting the performance of those processes which do not utilise the new resource.

I will demonstrate the validity of this assertion by providing: a suitable processor architecture, that meets the requirements to allow the operating system to manage the reconfigurable logic resource; an augmented operating system kernel that manages the additional resource; and a programming model that allows applications built in a traditional manner to utilise the new resource. I will demonstrate a small operating system running on a simulated processor design as an example of the approach.

1.3 Contribution

The contributions of this dissertation are:

- Demonstration of a basic operating system with Custom Instruction Scheduler (CIS) that allows applications to register custom instructions with the operating system, and have the operating system manage the the loading and execution of the custom instructions transparently to applications.
- Demonstration of a suitable programming model to support the use and management of custom instructions and their namespace in imperative languages, and support for such a system using traditional compilation and linkage techniques (including both static and dynamic linkage).
- Provision of a richer model of custom instructions than previously used in the literature, and demonstrate the benefit of the model from the perspective of managing custom instructions, in both the programming environment and the operating system.
- Demonstration of a suitable architecture layout to allow the FPL resource on a reconfigurable processor to be virtualised by the operating system, hiding absolute circuit location from the applications, and allowing the FPL resource to be managed both fairly and securely.
- Investigation of support for switching between hardware and software implementations of custom instructions in a way that is transparent to the invoking process. Theoretically such a feature can be used by the operating system to reduce circuit swapping when the FPL resource is overloaded.
- Initial investigation into the run time performance of an operating system managed processor hybrid. Comparison is made between applications with and without custom instructions, with and without operating system management, and with and without software alternatives to custom hardware.

The aim of the work described here is to attempt a first look at how such a system can be created. No claim is made that this is an optimal solution to the problems of managing a reconfigurable processor, rather that this is a sufficient solution such that the aims set out in the thesis statement are achieved.

1.4 Outline

The outline of the dissertation is as follows. Chapter 2 introduces the relevant technologies and related work that serves as a background to the work that follows. Based on the context and the existing body of work, a set of requirements for a system that supports a reconfigurable processor in a workstation environment is drawn up in Chapter 3.

Chapter 4 discusses a processor architecture that meets the requirements set out in the previous chapter. It goes on to describe an initial instantiation which is modelled in a simulator, and then demonstrates that the architecture to provide a basic performance increase for a bespoke set of applications.

Chapter 5 discusses the management by an operating system of the reconfigurable processor outlined in Chapter 4. It examines how applications register their custom hardware for management, and how the operating system shares the FPL resource on the processor. This chapter also demonstrates that the management techniques do not negate the performance benefit of the additional resource.

Chapter 6 considers the issues that arise when building an application with custom hardware, for underlying system proposed in the previous two chapters, using traditional compilation and linkage mechanisms. The chapter discusses how the use of custom hardware can be mapped from the programming language down to the calls used to register and invoke the custom hardware, which will work with both statically and dynamically linked software.

Finally, the dissertation concludes with Chapters 7 and 8, which discuss further research that could be done to follow up the work here and summarise the dissertation respectively.

1.5 Related Publications

The initial concept for this work was published in [Dales 1999]. Chapters 4 and 5 contain material on the basic architecture and performance which appeared in [Dales 2001] and [Dales 2003].

Chapter 2

Background and Related Work

This chapter introduces the relevant technology and background material to provide context for the work covered in the rest of the dissertation. The chapter is split into three broad categories: required knowledge, related work, and then analysis of the related work based on the context.

The first two sections provide a brief introduction to the fields of FPL and operating systems respectively, covering the basic concepts pertinent to the work that follows. Section 2.1 explains how FPL devices work, looking specifically at Field Programmable Gate Arrays (FPGAs) and the Xilinx Virtex range of FPGAs which will be used as the context for the FPL discussion in the remainder of the document. Section 2.2 provides a brief look at resource management in a workstation class operating system, focusing on processor and memory resources. Readers experienced with either topic should feel free to skip these sections.

Following the introductory sections is an overview of previous research and commercial activity in the field of reconfigurable processors and resource management issues as related to FPL. Section 2.3 covers research activities in combining microprocessors and FPL, and Section 2.4 covers commercial hybrid device offerings available today, while Sections 2.5 and 2.6 discuss the management of FPL real estate and security considerations respectively.

The chapter is concluded in Section 2.7 with an analysis and discussion of the various architecture and management options, and how they apply to the workstation environment context. This paves the way for the requirements analysis in the following chapter.

2.1 Field Programmable Logic

A Field Programmable Logic (FPL) device is a type of Integrated Circuit (IC) used to provide run-time or load-time tailorable hardware. At an abstract level, FPL devices contain a two dimensional array of cells that can be configured to behave like different types of logic gates that can be attached to a matrix of wires running through the device which connect the cells to form a circuit. The information used to configure these devices is held in RAM, allowing the circuit represented on the device to be changed simply by reprogramming the contents of the RAM. Such a device has numerous applications: it can be used as a prototype IC, used on a PCB before the final Application Specific Integrated Circuit (ASIC) design has been finalised; it can be used in applications where the designer may want the system to be upgraded at a later date whilst it is in the field; and it can provide the system with a flexible hardware resource that can be programmed to carry out different tasks at different times where multiple ASICs would otherwise have to be employed. Additionally, although FPL devices are typically more expensive than an ASIC for large volume applications, for smaller volumes FPL

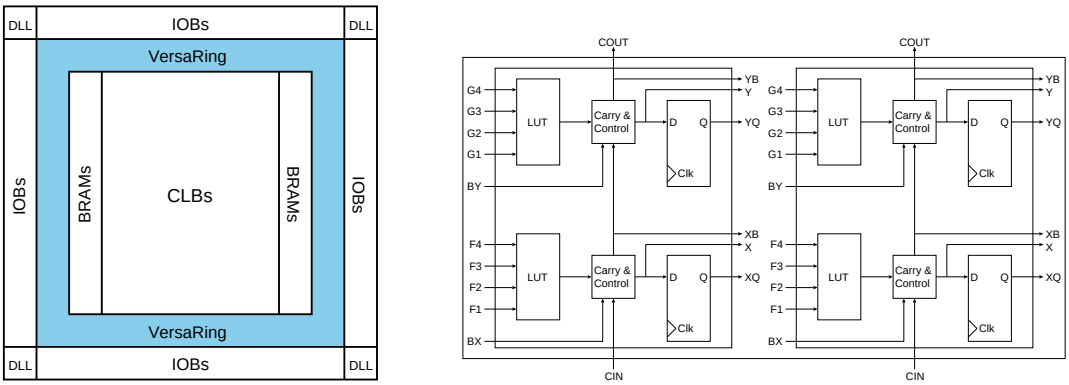

Figure 2.1: Overview of Xilinx Virtex architecture (left) and CLB layout (right)

devices make a suitable ASIC replacement as they do not have the high initial costs associated with IC fabrication.

The most popular FPL device today is a Field Programmable Gate Array (FPGA). Early FPGAs were limited to just a small array of cells and wiring resources, along with a set of special cells around the edge of the array which could be used to connect to device pins for I/O. Modern FPGAs expand on this, not only by providing many more cells and wiring resources, so that a modern FPGA can handle a circuit size of millions of gates, but also add extra features such as blocks of RAM, better clock control, dedicated multipliers, and so on. To help demonstrate what an modern FPGA looks like we will examine the Xilinx Virtex range of FPGAs [Xilinx 1999]. Though the Virtex has since been superseded by the Virtex-II, it is still a contemporary architecture in use today and is used throughout the rest of this document as the basis for FPL comparisons.

2.1.1 FPGA Overview

An overview of a Virtex device can be seen in the left part of Figure 2.1. Most of the FPGA is made up of the configurable cells, referred to as Configuration Logic Blocks (CLBs) in Xilinx nomeclature. These cells are the basic unit out of which circuits are constructed on an FPL device. The contents of the cells and how they simulate the basic logic will vary between manufacturer and architecture. The contents of a Virtex CLB is shown on the right of Figure 2.1. Each CLB consists of four Look Up Tables (LUTs), each with four inputs and one output, which are used to simulate combinatorial logic. The LUTs are programmed by loading values into Static RAM (SRAM) on the FPGA, and by changing the contents of the SRAM the LUT can represent different combinatorial functions. Note that a design does not need to use all the inputs to each LUT or all the LUTs in a given CLB, what is used simply depends on what inputs are considered active and connected to the wiring resources. As well as mimicking combinatorial logic, the LUTs in Virtex CLBs can be used as small memories or shift registers, providing one form of storage within the array. In addition to the LUTs, each Virtex CLB contains four registers which may be used to build sequential logic circuits. On modern FPGAs a high register density is important to allow designs to be efficiently pipelined for higher clock speeds. The carry and control unit is used for routing signals within the CLB and also handling special wires that connect CLBs within the same column to build fast carry chains. The Virtex range consists of nine devices, with varying CLB array sizes. The smallest device, the XCV50, consists of 384 CLBs whilst the largest device, the XCV1000, has up to 6144 CLBs.

Alongside the array of CLBs are the Block RAMs (BRAMs). These are large blocks of memory

designed to save the user from either having to use large numbers of CLBs for memory or having to go off chip for storage. Again, the amount of BRAM on a device varies, going from 4 Kbytes on the XCV50 to 32 Kbytes on the XCV1000. On the outer edge of the device are the special cells used to connect to the package pins on the device, referred to as I/O Blocks (IOBs). IOBs can be configure to operate as either input, output, or bidirectional. Each IOB is connected to a specific device pin, which can cause problems when routing (the term for working out how to wire up a set of CLBs) a circuit design, as it may prove difficult to get a wire to that specific location. To aid this, the Virtex uses an extra set of wiring resources called the VersaRing to make it possible to connect to an IOB that is not directly next to the CLB point at which the signal leaves the CLB array.

Not shown explicitly in diagram are the wiring resources used to connect between cells and other parts of the device. Typically each cell in an FPGA can connect to a hierarchy of wires, with wires that lead to the next cell in each direction, wires that span a local block of cells (e.g., a four by four block), and wires that will span the entire chip. All wires will run in either a horizontal or vertical direction across the chip. Cells will connect to wires using either tri–state drivers or multiplexors controlled by bits in SRAM. By altering the path of signals through the various wires using the connection points, cells can be connected together to form larger circuits. In addition to the wires used to connect cells together, there are a small number of global signals distributed to each cell on the array. These global wires are typically used to distribute clocking and reset signals. The Virtex uses a multiplexor based set of connections to connect to the various wire lengths on the chip.

2.1.2 Configuration

Programming a FPL device consists of loading the appropriate bits into the RAM that controls the behaviour of the various device parts. How and when this loading can take place again varies between manufacturer and architecture. The type of programming allowed has a large effect on how the device can be used: some devices can be configured only on power up, whereas some devices can have individual cells configured whilst the rest of the device continues to execute.

The most basic form of programming allowed is a *configurable device*, where the contents of the RAM are loaded when the chip is initialised. An example of such an FPL device is the Triscend A7, described in Section 2.4.2. To reprogram this device the entire chip has to be reset and reinitialised. This limits the application of such a device, especially if there are parts other than just an FPL array on the same IC.

The next stage up is a *reconfigurable device*, where the contents of the RAM can be loaded at run-time rather than just at startup. In addition such a chip may also allow the user to store the contents of the configuration RAM, which can be useful for extracting state from a circuit loaded on the array for debugging. An example of such a devices are the Xilinx XC4000 and Spartan ranges [Xilinx 1999]. In such a device the entire configuration RAM is reprogrammed at once, so when the device is reprogrammed any old configuration information is lost. Reprogramming an FPGA is a relatively slow operation, typically of the order of milliseconds: reprogramming the largest Spartan device at the highest speed (serialized data read in at 8 MHz) takes 41.3 ms. While the Xilinx Virtex supports a faster 8 bit at 60 MHz interface, this is still slow in comparison to modern memory bus speeds. One solution to the problem of slow reconfiguration speeds used by some devices is to have multiple planes of RAM on the device, each representing a different circuit configuration. The FIPSOC device discussed in Section 2.4.2 is such a device, which uses three configuration planes. Switching between configurations held in different RAM planes is a very low cost operation, typically taking only one or two clock cycles. In addition to the low switching costs, it is also possible to modify all but the currently active plane whilst a circuit is running.

One problem with such devices is that the entire array has to be programmed at once, even if the circuit being loaded only uses a subset of the available resources. The final device programming style is referred to as *partial reconfiguration* or *dynamic reconfiguration*, which allows much greater control. There are two approaches to partial reconfiguration: *fine grain* partial reconfiguration and *course grain* partial reconfiguration. Devices that support fine grain partial reconfiguration, such as the Xilinx XC6200 [Xilinx 1996], allow configuration at a CLB or wiring connection resolution. This means that only the components that are needed within a circuit are configured. This both reduces the amount of data that needs to be stored for a circuit configuration and the time it takes to reconfigure the device. Course grain devices can be programmed on a scale smaller than the entire device, but above the level of individual components. An example of this is the Virtex device, which can be configured on a column by column basis [Xilinx 2000]. The is less space and time efficient when it comes to configuration, but simplifies the programming interface on the device.

2.2 OS Resource Management

This section serves as a brief introduction the operating system concepts used throughout the rest of the dissertation. For a further understanding of the concepts discussed here the reader is referred to an operating systems text, such as [Silberschatz et al. 1998]. For this discussion, as in the rest of the dissertation, we assume a multiuser, multitasking style operating system using multiple private address spaces for processes, such as a UNIX implementation or Microsoft Windows NT and its descendants.

2.2.1 Process Scheduling

A modern workstation runs multiple processes for multiple users, all of which must share the processors available on that machine (typically one or two in a modern workstation). In a modern workstation there may be hundreds of processes running at once, so the processors must be shared using some form of Time Division Multiplexing (TDM). What happens is that periodically the operating system will perform a *context switch*, swapping the currently active process on a processor for another process. This way, with suitably frequent switches, all processes appear to be making progress despite only one being able to utilise a given processor at any one time.

The execution state of a thread of execution within a process is held by the operating system within a Process Control Block (PCB). For each thread a PCB contains a copy of the processor state (register contents, status flags, etc.) from when that process was last removed from the processor, such that the data can be placed back on the processor, or dispatched, and execution resumed as if the process had not been interrupted. The PCB also contains meta information used by the operating system to manage the process, such as a reference to its page table (discussed in Section 2.2.2), and a unique identifier for that process, the Process ID (PID). In multithreaded applications, each thread will have an associated Thread Control Block (TCB), which is similar to a PCB, in that it contains the active state for that thread, but contains less meta data. Global state, such as the page table for the process, is held in another globally used record. Depending on the operating system being used, either the operating system (kernel level threading) or the process itself (process level threading) will be responsible for managing the individual threads within the process. For the most part, this distinction is not relevant to this work, and we ignore the concept of multiple process threads, except where explicitly relevant.

The style of processor scheduler used depends on the style of workload the system is designed to run. For instance, if the user is running a set of interactive tasks then tasks will get frequent short bursts on the processor to provide suitable responsiveness. However, swapping between processes

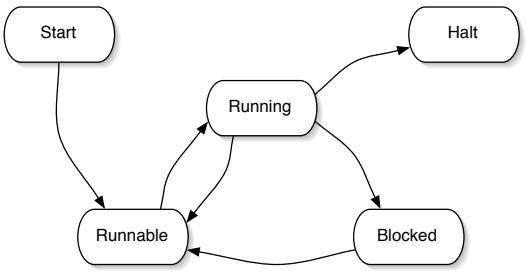

Figure 2.2: Process states

is not a zero cost operation: it takes time to context switch between processes, during which the system gets no useful work done on the processor in question. In addition it can causes caches to be flushed, which also causes a temporary performance degradation. Thus on a system running batch applications, where user responsiveness is less important, applications will get longer less frequent runs on the processor, reducing the amount of time lost on context switches. The amount of time a process gets on a processor is referred to as the *scheduling quantum*, and is typically fixed for all processes on a system, and is typically measured in milliseconds. The batch scheduler in Linux has a quantum period of 10 ms and NetBSD and Windows NT use 100 ms.

Processes are not always able to continue executing, for example they could be blocked waiting on I/O, during which there is no point in having this process be in control of the processor, or they could have requested to be made inactive for a period. The processes in an operating system are moved between a series of states that control where they are in relation to the processors, which is shown in Figure 2.2. Processes start in the start state when they are created and initialised, after which they are added to the set of runnable applications. Runnable processes are those that are able to make use of a processor, and it is from this set that the operating system will choose the next process to schedule onto the processor on a context switch. Running processes are those that currently occupy a processor, and there can never be more running processes than there are CPUs in the system (though there may be less). Processes stop running for three possible reasons. Firstly, the process runs for the entire quanta and the operating system hands control of the processor to another process, so the current process is put back in the runnable set ready to be run again at a later date. Secondly, the process terminates (either intentionally or through an unrecoverable error) in which case it is halted and destroyed. Finally, the process may either request an I/O interaction which causes is to block until the interaction is completed or request to be suspended temporarily. In either case the process is put in the blocked state until either the I/O interaction requested resolves or the suspend time period expires, after which it is made runnable again.

Selecting a process to run from the set of processes marked runnable is an important task and is handled by a *process scheduling algorithm*. Again, there are different types of scheduling algorithm that work better for different classes of workload. A scheduling algorithm needs to attempt to ensure fairness in the system; it has to ensure that all processes get some time on the processors so that they may make forward progress. The most basic algorithm is a *round robin* scheduler, where the set of runnable processes are held in a First In/First Out (FIFO) queue, ensuring that each process gets executed in turn. Round robin scheduling works well in a batch processing system, but in an interactive environment we may wish to make processes that were blocked on I/O a higher priority when rescheduling, so that they may respond more quickly to user requests. Such processes may be inserted into the head of the round robin queue, or a set of FIFO queues can be used, with each

representing a different priority and queues of higher priority being emptied first. However, these systems have potential problems such as starvation, where a low priority process never gets access to the processor as there are always other processes running at a higher priority.

2.2.2 Memory Management

When a system interleaves the execution of set of processes, it must ensure that they do not interfere with each others' data held in memory. If each application has hard coded into it a range of memory addresses to use, then without any form of protection or management applications are likely to attempt to utilise the same areas of memory, with most likely undesirable consequences. Trying to ensure that all possible applications never attempt to access the same piece of memory is unfeasible, so a technique is needed to allow multiple processes to exist in memory simultaneously.

An early technique was to give each application a contiguous stretch of memory as needed, with applications using relative offsets for calculating addresses or using a base address register in hardware to provide the correct offset for internal addresses. If there were more applications than could fit in memory, then *swapping* was used. Whilst some applications are held in main memory, those that do not fit are held on backing store. For processes to execute their memory image must be in physical memory, so there are essentially two sets of runnable processes under this scheme: those that can be ran from memory currently and those that need to be moved from backing store to physical memory first. The operating system will periodically swap processes between backing store and physical memory to ensure that all processes can make progress. The drawback with swapping is that it works at a large level of granularity, requiring entire processes' memory images to be moved on a regular basis, which became more expensive as address spaces grew.

The basic system used on most modern workstation operating systems is *virtual memory management*. Each process is provided with a private virtual memory space in which to work, to which no other process has direct access. The size of the virtual memory space is not necessarily bound by physical properties, but rather by the physical memory size in conjunction of the amount of backing store available. The actual addressable amount is typically much greater (e.g., 4 Gbytes for a 32 bit system); applications can address any part of the address range, so long as the total used amount does not exceed the amount of virtual memory available. Because the address space is private, no processes can access another process's memory. This virtual address space is referred to as a virtual memory, and addresses to locations within virtual memory are referred to as virtual memory addresses. To be useful, virtual memory needs to be backed up by physical memory so that processes may interact with it, but generally it is not possible for all virtual address spaces to be mapped into physical memory at once. To solve this each process's virtual address space is kept on disk[1] and parts are mapped to physical memory as they are needed. The operating system feeds the mapping from valid virtual address to physical address into address translation hardware held on the processor. When the current process issues a read or write request using a virtual address, the address translation hardware will either convert the address into a physical address before putting the address on the memory's address bus, or alert the operating system that it could not achieve the mapping, so that the operating system can load the part of the virtual address space needed into physical memory and update the mappings. If the physical memory is currently full, then the operating system will unload one or more virtual memory regions from physical memory and remove those addresses from the address translation hardware, freeing up room for the new mapping to be loaded. At any given point various regions from different

[1] Only the modified parts need actually be stored, the rest can be assumed to be undefined and as such needs no backing store until it is modified.

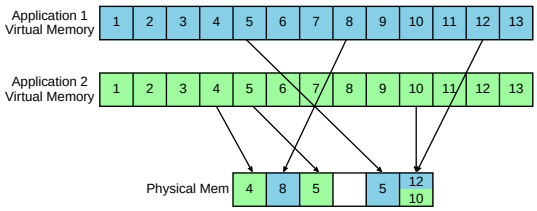

Figure 2.3: Two virtual memory spaces sharing a physical memory

processes' virtual memories may be loaded into physical memory, but because the address translation hardware controls what regions of physical memory a process can access, there is no danger of processes accessing each other's data.

If virtual memory was mapped on an individual location by location basis, then the number of access faults caused when a process tries to access an address that has not been mapped yet would be unacceptable. Most of a process's run time would be spent in the operating system resolving requests. Thus larger regions are mapped at once, reducing the number of faults that occur, with the trade off of possibly loading in data that will not be needed. There are two principle types of region used for loading sections of virtual memory into physical memory: paging, as used in most unices and Windows NT, and segmentation, as used in IBM's OS/2.

2.2.2.1 Paging

Paged virtual memory [Belady 1966] divides both the physical memory and virtual memory spaces into a series of fixed sized contiguous blocks called pages. When a process accesses a location within a virtual memory page, that entire page is transferred into a free physical memory page. An example of a pair of address spaces mapped into physical memory can be seen in Figure 2.3. The size of pages is always made to be a power of two (typically 4 Kbytes) to simplify the address translation hardware. This enables the physical address to be broken down into a page number (the most significant set of bits) and an offset within that page (the least significant bits), thus in the address translation hardware the most significant bits are simply replaced with the physical page number with the low order bits remaining unmodified. For each process the operating system must store a *page table*, which contains information about each page a process owns, noting access permissions, whether it is currently loaded, whether it has been modified, and so on. This table is created in a format understood by the processor, so that it can resolve page mappings when a process generates an address. However, accessing the page table is slow; the processor must read data from main memory each time an address translation is required. Instead a cache of recently used page address translations is kept on the processor in a Translation Lookaside Buffer (TLB). The TLB is a small lookup table that, given a page number, will return either a physical page number, or trap, causing the processor to examine the page table and load the correct entry into the TLB. Note that page table is specific to an individual application, so the processor needs to be pointed to the correct page table on each context switch. Obviously the TLB contents are also process specific, so must either be flushed on each context switch, or tagged with the PID of the process to which they belong. If tagging is used then look ups are done using a PID and virtual address tuple, which forms a system unique address space.

Given the number of processes running on a modern system, it is probable that more virtual pages will be required by the currently active set of tasks than there are physical pages. This means that the operating system will eventually have to evict some pages held in physical memory back to backing store in order to have space for new ones. There are numerous different algorithms used for

choosing which pages to select, referred to as *page replacement algorithms*. Classic page replacement algorithms include FIFO selection, random selection, Least Frequently Used (LFU), where the page that has been used the least number of times is evicted, Least Recently Used (LRU), where the page furthest away in history in evicted, second chance, where pages are examined in a cycle and pages unaccessed since the last time round are evicted, to name but a few. The operating system may choose to select any available page for eviction, or restrict itself to the set of pages belonging to the currently faulting process (to protect other processes from one with greedy memory access behaviour). To aid the operating system, the processor may maintain usage statistics (kept in the page table) to help the operating system implement history based algorithms like LRU and LFU. In a similar vein, the processor may also provide information on whether the page has been modified or not, as unmodified pages will not need saving to backing store before eviction.

2.2.2.2 Segmentation

Rather than using fixed sized blocks that bear no conceptual relevance to the process as is done in paging, segmentation uses logical blocks — segments — from within the program: procedures, data objects, and so on [Lister 1975]. When a particular segment is needed, the entire segment is moved into physical memory. Rather than using flat virtual addressing that is used in paging, a program generates addresses comprising segment ID and offset pairs. The address will then be translated in hardware by referring to a table that contains a list of valid segment IDs, their base address in physical memory, and a length field against which the offset value is compared to protect against running off the end of a segment. As before, when physical memory has insufficient space to accept new segments, the operating system has to find a way to free up physical memory, which it could do by moving segments in and out of memory similar to how pages are moved. However, this is a more complicated operation due to the variable length of segments; allocating arbitrary stretches of memory is a more complicated problem than managing the fixed size pages described earlier. For this reason, segmentation tends to be implemented on top of paging.

2.2.2.3 VMM and Swapping

Although by and large most modern operating systems rely on paging for virtual memory management, some systems implement a two level system that uses both paging and swapping (both BSD 4.4 [McKusick et al. 1996] and Windows 2000 [Solomon & Russinovich 2000] use this). By default the operating system will use paging to share physical memory between the processes on the system, but when the memory system is under heavy load, the operating system may spend a large amount of time moving pages in and out of memory. To reduce contention, the operating system will then decide to swap one or more processes out to backing store in order to reduce the level of contention for physical memory. After a certain time frame the operating system will then swap those processes back in and move others out so that all applications are able to make some progress.

2.3 Research into Microprocessor/FPL Hyrbrids

Having given the necessary background material needed to understand the work that follows, the next set of sections cover the existing body of work in reconfigurable processors and FPL management.

The idea of combining microprocessors and FPL is not new, and a body of research exists in this field already, which is reviewed here in terms of the general approach taken. This section simply

describes the existing work; discussion of the relative merits of the different approaches will be done in Section 2.7. The existing body of work can be described as belonging to three broad categories:

1. Loosely Coupled FPGA and CPU Systems
2. Tightly Coupled FPGA and CPU Systems
3. CPU on an FPGA Systems

2.3.1 Loosely Coupled FPGA and CPU Systems

The easiest way to take advantage of FPL from a microprocessor is to use a separate module connected over a bus (e.g., a PCI card) with an FPGA mounted on it. Numerous projects have used this approach to accelerate particular applications such as PostScript rendering [Singh et al. 1997, MacVicar et al. 1999] and graphics processing using PhotoShop filters [Ludwig et al. 1999]. Each of these projects used Xilinx FPGAs and memory mounted on PCI cards plugged in to a workstation to handle the respective processing algorithm. These cards come ready to use with device drivers, allowing programmers ready access to the new resource.

Whilst the simple nature of the interconnection makes this an easy option from a construction perspective, the solution does suffer from communication latency problems. Because the data being processed resides in main memory, and must be moved to and from the expansion card, any acceleration must amortise this communication overhead. In [Ludwig et al. 1999] some filters performed better in software than on the FPGA for this reason.

Currently, there is no access management controlled by the operating system, as this would require more fundamental support in the lower levels of the operating system. Under the proposed systems described above it is assumed that the card is dedicated to a single task over a long period, and no attempt is made to protect the contents of the FPGA. Given the communication problems described above, any attempt at fine grain context switching on the card would be likely to lead to unacceptable performance problems.

2.3.2 Tightly Coupled FPGA and CPU Systems

Moving on from having separate ICs for the processor and FPL, the next level of integration is to move the FPL onto the same chip as the processor. This reduces the communication overhead, bringing the reconfigurable device closer to the controlling processor and allows the FPL to take advantage of the traditional memory hierarchy.

2.3.2.1 PRISC

The Programmable Instruction Set Computing (PRISC) architecture was the first to propose linking a processor with FPL on a single chip [Razdan & Smith 1994]. The aim was to move away from application specific programmable logic architectures, and produce something that could be used to enhance different types of application. PRISC proposed augmenting a standard Reduced Instruction Set Computing (RISC) processor with multiple Programmable Function Units (PFUs), which connect to the processor's register file along with the conventional function units. The contents of these PFUs are generated by an extra compilation stage, which looks for code sequences that it can convert into PFU instructions, thus hiding the PFUs from the programmer. The amount of work placed in a PFU is quite small, limited to 15 levels of logic in order to meet the processor's timing requirements. The compiler typically generates up to 200 PFU configurations per application.

Each PFU consists of a LUT and wiring matrix, and does not use the IOBs or registers found in a conventional FPGA fabric. Because PRISC extends a RISC architecture, the aim was that all instructions should complete in a single cycle, so state in the array was not required. The lack of state in the PFUs removes the requirement for fabric state needing to be preserved on a process context switch. The SRAM which controls both the LUT contents and routing layout is memory mapped to the processor's address space, which means that no new instructions are required to load instructions into the PFU; conventional memory transfer instructions are used to load instructions. The only change the authors made to the instruction set was the addition of an execute PFU operation. This instruction simply takes a logical PFU identifier, and two input register indices and a result register index. In the initial PRISC test system there is only room for a single PFU, and with this is associated an ID register holding the logical identifier of the currently loaded instruction. If an execute PFU instruction occurs and the PFU identifier in the opcode does not match the contents of the ID register then a processor exception occurs. The operating system can then respond to this by finding the correct instruction and loading it into the PFU.

2.3.2.2 OneChip

The OneChip architecture was first proposed in 1996, since when there have been a number of variations. Here we describe the original implementation and the most recent implementation.

The original OneChip [Wittig & Chow 1996] design takes a more extensive approach to combining FPL with a processor than the PRISC authors. At an abstract layer, the OneChip authors propose a similar system to PRISC, with a RISC (in their case MIPS-II) processor core connected to a group of custom circuits loaded in reconfigurable logic. However, the actual implementation is substantially different. On a OneChip processor design the RISC core is surrounded by the FPL array, which can be used both for new data processing instructions and, using IOBs provided in the array, new I/O instructions (for example, the authors demonstrate using a UART on the FPL with two instructions as the interface). The FPL is not rigidly divided up into different instructions, rather a particular configuration of the FPL may divide up the FPL as it sees fit. In addition to including IOBs, the FPL array also includes registers, allowing for more complex, and therefore larger, instructions to be implemented (such as the aforementioned UART example). The FPL is a single context system, but multiple configurations can be stored in the on-chip SRAM bank for fast access.

One of the advantages of the first generation OneChip over PRISC for embedded systems is that the system can be used as a reconfigurable System on Chip (SoC) device, thanks to the IOBs. It is likely that such as system would benefit greatly from a partially reconfigurable FPL array, as it is unlikely that custom instructions that touch IOBs, thus interfacing with dedicated hardware in the outside world, will change between applications (other than freeing up space around the IOBs to make room for data processing instructions).

The third generation OneChip [Carrillo & Chow 2001] is a more complex design, featuring multiple FPGA link blocks as part of a superscalar RISC processor with out of order execution. As before the system can hold multiple configurations on chip, each being associated with a particular FPGA block. As there is only a finite amount of space for configurations on the processor, the architecture provides support for managing which configuration data is held on the processor, using a LRU algorithm to move configurations off the processor when there is no free space to load a new configuration. Unfortunately, the authors had not had time to measure the performance of this algorithm against other possible management algorithms. To enable management of the new instructions in the pipelined processor instructions must be specified with a fixed issue latency, describing the number of cycles the processor must wait before a result is ready.

Despite the more complex architecture, the new OneChip design is still designed with running just a single application in mind. Instructions are associated with a system global identifier which the application can then use to invoke their instructions as they are required. This means that should another process need to use the reconfigurable resource it must either be guaranteed to use a separate identifier space for registering instructions, or the operating system in charge must unload all the currently stored configurations on a context switch.

2.3.2.3 CoMPARE

The CoMPARE architecture also describes a RISC processor connected to a single array of FPL [Sawitzki et al. 1998]. Rather than containing just an ALU, it uses what the authors call a Reconfigurable Processing Unit (RPU). The RPU contains both a conventional ALU and an array of FPL, referred to as the Configurable Array Unit (CAU). Inside the RPU, the ALU and CAU may be connected to run either in parallel or in sequence, and unlike most function units, the RPU takes four operands and returns two results, as a way of increasing the data bandwidth of the system. Despite using a prototype with just an 8 bit data width, the authors demonstrate a two to four times performance increase over a 32 bit MIPS R2000. However, CoMPARE uses a single FPL array, which is dedicated to a particular circuit, unlike PRISC and OneChip, which support multiple instantiated circuits simultaneously. As with PRISC, the array is stateless, removing the need for its contents to be preserved over a context switch.

2.3.2.4 Garp

Another RISC based hybrid is the Garp architecture [Hauser & Wawrzynek 1997]. Garp ties a 32 bit MIPS-II core to a reconfigurable array. The array consists of a custom fabric which the authors claim is better suited to the type of operations used in a hybrid device than a conventional FPGA fabric would be. The array has been designed to provide access to the CPU's data bus, providing circuits with full access to the CPU's memory hierarchy. Additionally, rather than configuring the array to processes bits, the array works in terms of two bit pairs, reducing the amount of configuration data required by working on the assumption that most processor bound tasks work on data sizes greater than one bit in granularity. Custom instructions loaded into the reconfigurable array may run for a number of cycles, as determined by an instruction operand, and the array can run concurrently with the conventional ALU. The reconfigurable fabric on the Garp architecture does include a small amount of state (the authors kept the amount small to help reduce the context overheads), and uses configuration caching to help reduce circuit switching times.

2.3.2.5 Chimaera

Chimaera [Ye et al. 2000] is a pipeline optimised reconfigurable processor architecture. Chimaera adds a reconfigurable unit to the processor containing a large FPL array, which is allocated on a one dimensional basis between multiple circuits. The FPL array is associated with its own register file and is managed by an Execution Control Unit (ECU) and Configuration Control and Cache Unit (CCCU). Programs are built from software and a collection of custom hardware instructions. Each custom hardware instruction has an ID associated with it, which is then used to invoke the instruction. The ECU is responsible for decoding attempts to invoke instructions and interacts with the CCCU, which does the programming of the reconfigurable array. At any given time multiple custom instructions may be loaded into the FPL array, and the CCCU also stores a small number of recently used configurations for fast reload.

2.3.2.6 SHARC DSP Hybrid

[Graham & Nelson 1999] describes a hybrid architecture based on a Digital Signal Processing (DSP) processor, with the aim of producing a high-performance system for embedded applications. The authors have designed a processor that extends a SHARC DSP processor with a configurable array in a similar manner to the Garp architecture. As is common with embedded systems architectures, the SHARC contains on chip SRAM (512 Kbytes in this case) to which the FPL array has direct access, as well as providing the FPL access to the ALU's register file. Another similarity with the Garp architecture is the ability to run the conventional ALU in parallel with the reconfigurable array. No support is provided in this architecture for sharing the array or speeding up circuit switches, but given that the device is aimed at embedded systems this restriction is understandable.

2.3.2.7 Miscellaneous Tightly Coupled Hybrids

All the above architectures have been based on fairly simple conventional processor designs, which could be suited to a workstation environment. Other styles of architecture do exist, though these are not aimed at being used in a typical workstation; we include these examples here for completeness.

The Raw Architecture [Waingold et al. 1997] is a less conventional architecture, aimed at maximising parallelism inside the processor. A Raw processor consists of numerous parallel execution units, each with a simple ALU and a small area of FPL. This architecture is aimed at DSP/filtering applications where it is easy to map from the low-level parallelisation to the problem's requirements (typically systolic in nature).

Similarly, the Xputer class of devices represents a novel architecture that utilises FPL in its core [Hartenstein et al. 1989, Hartenstein et al. 1991]. The Xputer device attempts to provide the parallelism required for systolic applications (e.g., DSP applications) below the instruction level. Unlike traditional computers, the Xputer is not control driven, but data driven, thus they have no traditional instruction stream. The work on an Xputer is carried out in the reconfigurable ALU (r–ALU). Programs on an Xputer load large instructions into the r–ALU, thus remove much of the need for control flow. This architecture is again designed for applications which can take advantage of the benefits of systolic arrangement.

2.3.3 CPU on FPGA Systems

The extreme approach to removing communications problems between the processor and the FPL is to implement a processor on top of an FPGA. This work can take two approaches: implement a standard processor core next to the reconfigurable parts, or implement a completely reconfigurable processor where the core or the CPU itself will change.

The Dynamic Instruction Set Computer (DISC) is an example of a completely changeable processor [Wirthlin & Hutchings 1995]. Rather than implement an entire CPU, the DISC can actually be viewed as a formalised technique for scheduling circuits on a FPGA device which supports partial reconfiguration. The DISC borrows certain elements from a traditional CPU: it has a notion of instructions, and it uses a data, address, and control bus. However there are no inbuilt operations besides memory accessing and branching; everything else is loaded into the reconfigurable array. DISC allocates instructions on the array by allocating them space on a one dimensional basis. This can be seen in Figure 2.4, which shows two tasks allocated on an FPL array. The array has the address, data, and control buses running the entire length of the array along the axis of allocation, to which circuits can attach themselves as necessary. Each instruction has coded into it an opcode to which it should respond. The instruction dispatch unit on DISC places the opcode of the instruction it wishes to invoke

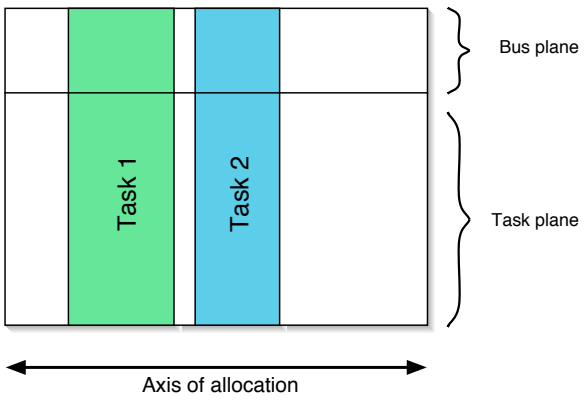

Figure 2.4: Banded FPL allocation using control buses

on the control bus, and circuits observe this at all times and respond as necessary. To overcome the limited space on the FPGA used to implement DISC, the authors designed DISC to allow it to swap circuits on an off the array. However, given the limited space, it is expected that circuits will actually reside on the array for a long enough period to amortise the reconfiguration costs.

A CPU based approach is described in [Gilson 1994] and [Gilson 1997]. These describe a RISC processor and Reconfigurable Instruction Execution Unit (RIEU) implemented entirely on an FPGA. It is implied that, similar to the CoMPARE architecture, only a single circuit can be present in the processor space at any one time. The system cannot operate on its own, as it relies on a host processor to carry out the reconfiguration of the FPGA. Again, this kind of arrangement does not lend itself well to the frequent reconfiguration we may encounter with heavy loads. The Nano processor [Wirthlin et al. 1994], which implements a RISC core on an FPGA, leaving the rest of the fabric for custom logic, is only designed to be configured once for a given application.

A different approach to constructing a flexible processor on an FPGA is proposed in [Donlin 1998]. The Flexible Micro RISC (URISC) architecture is based on a processor core with just a single move instruction built on a dynamically reconfigurable FPGA. To process data, custom processing elements are moved onto the datapath, so that when data moves through the processor it passes through the processing element before being returned to memory.

Although suited to many applications, this type of architecture does not lend itself well to a workstation style environment. Even with a reconfigurable function unit, most software will still make heavy use of normal CPU functions such as integer maths, address calculation and branching. There is no advantage of placing this functionality onto a reconfigurable array; it will just suffer space explosion and slow down.

2.4 Existing Hybrid Systems

In the last three years, the concept of a hybrid processor/FPL system has been taken up by numerous companies, notably by both the major FPGA vendors, Xilinx and Altera. This section outlines the current offerings in the field. In particular we provide a detailed case study of one, the Xilinx Virtex-II Pro, in order to highlight some of the detailed problems faced when implementing designs on such a system.

Each of the commercial hybrid offerings offer a tight coupling in terms of Section 2.3, either being

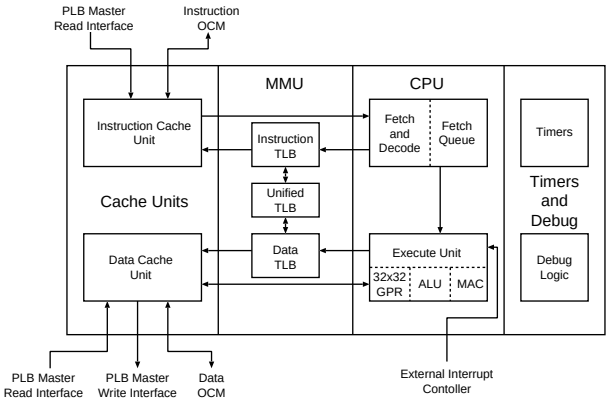

Figure 2.5: Overview of PowerPC 405 core

essentially part processor and part FPGA, or having the FPL as a functional unit of the processor. All of the offerings are aimed at embedded systems, using either microcontrollers or embedded versions of microprocessors. Additionally, not all the parts are reconfigurable; the Triscend A7 and Infineon 20xx are configurable, the custom logic part being fixed at design time.

2.4.1 Xilinx Virtex II Pro

The Virtex-II Pro architecture [Xilinx 2002] is Xilinx's hybrid FPGA and processor range, which was developed with IBM and released in early 2002. Virtex-II Pro devices are an extension of the Virtex-II FPGA architecture (which itself is an extension of the Virtex range described in Section 2.1.1) which contain fast I/O ports and between zero and four embedded processor blocks. These processor blocks sit within the CLB array, allowing custom cores to be instantiated in the CLB array and connected to the processor blocks.

2.4.1.1 The Processor Blocks

Each processor block in a Virtex-II Pro device contains a PowerPC 405 core. The 405 is a 32 bit implementation of the PowerPC architecture [Motorola 1997] developed by IBM, aimed at embedded systems, a general overview of which can be seen in Figure 2.5. PowerPC is a RISC based Instruction Set Architecture (ISA) which supports a set of different 32 and 64 bit implementations, designed to scale from embedded devices, like the 405, to large servers, like the IBM POWER4. The 405 cores embedded in the Virtex-II Pro can run at a clock speed of 300 MHz.

At the core of the 405 is a 32 bit integer ALU and a 32 entry general purpose register file, shown in the CPU section of Figure 2.5. In addition, there are eleven special purpose registers used for purposes like the program counter, link register (used to store the return address during a function call), status bits, and so on. The CPU section of the core also contains the fetch and decode logic, which is used to retrieve instructions from memory and work out how to process them. The MMU section contains a set of TLBs for use in virtual memory management (as discussed in Section 2.2.2). Once addresses generated by the CPU have passed through the MMU section, they are passed to the cache units, the Instruction Cache Unit (ICU) and the Data Cache Unit (DCU). The cache units are used to manage memory interactions between the 405 core and the rest of the system. There are two memory interfaces that the units access, which are mapped into different parts of the memory address space. The main access method is over the Processor Local Bus (PLB), to which other devices such

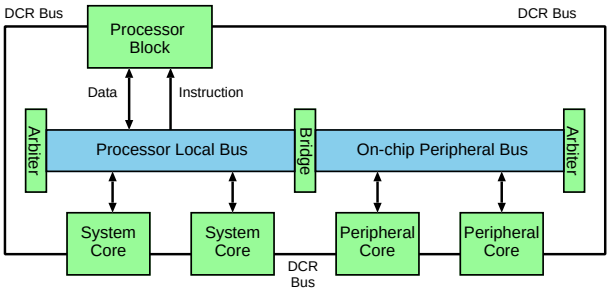

Figure 2.6: CoreConnect architecture used on the Virtex-II Pro

as memory controllers and peripherals are connected, or alternatively to small amounts of On-Chip Memory (OCM), which provide small high speed memories (the speed depends on the size of the OCM, but small enough memories can be accessed in a single clock cycle). The cache units divide the 4 Gbyte address space into 32 contiguous 128 Mbyte regions, over which the cache can be active or inactive. Reading regions that have the cache enabled results in the 405 doing a burst transfer of 256 bits to fill a cache line at a time, whilst unmapped regions are accessed one 32 bit word at a time. The caches themselves each consist of a 16 Kbyte 2-way set associative cache, and do not support any cache line locking abilities.

2.4.1.2 Processor Block Interface

The Virtex-II Pro uses the IBM CoreConnect architecture to connect custom IP cores to each processor block, as shown in Figure 2.6. The processor blocks are connected to a high speed bus called the Processor Local Bus (PLB). The PLB runs at processor speed and is used by the processor block and a small number of system cores. The PLB should not be used for peripherals, so as to reduce contention on this main system bus. Instead, peripherals and other devices, such as memory interfaces, should be connected to the On-chip Peripheral Bus (OPB), which is connected to the PLB over a bridge. The OPB is a 32 bit bus which runs at a slower clock speed than the PLB, which on the Virtex-II Pro is 100 MHz. Accesses to both of these buses are controlled by arbitrator devices which allow devices on the bus to be given a priority when contention arises. Cores attached to the PLB and OPB use a memory mapped interface to interact with the processor. Each core is designed to respond to a certain set of addresses, and when the core sees its addresses on the address bus it should respond by reading or writing data to or from the data bus and setting the appropriate bus control signals. Conceptually, the programmer writing software for processor blocks does not explicitly access devices, but instead reads and writes from memory.

Both the PLB and OPB are used for data transfers between parts of the system; control signalling is handled through a separate bus interface, the Device Control Register (DCR) bus. As shown in Figure 2.6, the DCR bus is driven directly by the processor using a special group of instructions, and can be attached to a register interface on each core. These registers are then read and written as appropriate to control the core. However on the Virtex-II Pro a separate DCR bus is typically used for controlling cores. An OPB to DCR bus bridge is attached to the OPB and then that interface is used for controlling cores. This DCR bus is then memory mapped rather than being controlled using the PowerPC's DCR instructions.

2.4.1.3 Interaction Between Software and Hardware

Although the memory mapped interface described above provides each device with an abstract interface similar to that of a memory device, a schism arises because peripherals have fundamentally different behaviour of which the programmer must be aware. The cache units in the processor blocks are designed to interact with devices that exhibit the properties of a memory system: reading and writing depends on order and not timeliness, burst transfers happen to sequential addresses, and so on. Unfortunately these assumptions do not hold true for many devices, and as such the programmer must code device interactions carefully to prevent the cache units causing undesirable behaviour.

Most modern processors will not necessarily issue memory interactions when they occur in the instruction stream; they may delay them until a more appropriate time, and may even rearrange the order in which the interactions occur. When communicating with a device rather than a memory this is typically an undesirable property. For example, a programmer sending a device on/off command will want it to be honoured immediately, but the memory system could delay these signals for a short time, so the programmer needs to explicitly force these memory interactions to occur. On the PowerPC this is easy to achieve using the *eieio* (Enforced In–order Execution of I/O) instruction, but it is down to the programmer to remember to include this instruction after device interactions.

To enable the processor core to use burst transfers to interact with a device, which may be necessary to obtain higher system throughput, the transfers to and from the device must go through the DCU. As described in Section 2.4.1.1, using the DCU allows the processor to transfer 256 bit cache lines in a single burst, rather than using individual 32 bit transfers. A mismatch occurs here because a device like a FIFO will have a single address mapped to the input or output on its interface, but the DCU expects to be accessing sequential memory addresses, starting from a cache line aligned address. Thus, when a burst transfer occurs, even though the start address may be correct, subsequent addresses will be incorrect, and could potentially cause other undesired interactions to occur. This means that the designer of the core must be aware of this and map any interface that wants to take advantage of burst transfers to an entire cache line worth of addresses.

From the software side, the programmer must ensure that they build up a cache line in the DCU atomically. To do this they must use multiple register instructions, which read or write multiple registers in an atomic instruction, which is the only way to guarantee that a cache line is filled or drained atomically. If the programmer does not do this, then there is no way to guarantee that the cache line will stay in the cache until the processor has done reading or writing it. A further complication is that when building a new cache line for writing, the DCU first reads the current contents of memory into the cache. It does this as the DCU does not know how much of the cache line will be modified by the software and needs to ensure that the entire cache line contains valid data for when it is eventually flushed. This can be undesirable when accessing a device, as unwanted reads form a device may have side effects. Thus the programmer must remember to explicitly blank the cache line, filling it with zeros, each time a new line is generated.

Overall, none of the above problems are insurmountable, but they all add to a series of subtle problems that system designers need to constantly remember when accessing devices.

2.4.2 Other Commercial Hybrids

The Triscend A7 [Triscend 2000] is an ARM based Configurable System on a Chip (CSoC) device[2]. The A7 consists of a 32 bit ARM7TDMI core (described in Section 4.2.1) running at 60 MHz connected over a Configurable System Interconnect (CSI) bus to an FPL array, referred to as the Config-

[2]Triscend also do a smaller 8 bit CSoC device, the E5, based on the 8051 Microcontroller.

urable System Logic (CSL) Matrix, which is equivalent to up to forty thousand gates. The A7 also contains typical embedded system processor parts: it has 16 Kbytes of on–chip SRAM, two UARTs, an interrupt controller, and a set of timer resources. The CSL Matrix has two interfaces to the rest of the system. Internally, registers in the CSL Matrix can be memory mapped to appear on the CSI bus. The processor core can then read and write these memory mapped registers to utilise the custom logic. The CSL Matrix also has an external interface to device pins, allowing it to be used as a form of glue logic between the processor and external devices. The A7 is a configurable device: the configuration bitstream for the CSL Matrix is read from memory on reset, so it is not possible to change the contents of the FPL at run-time.

The Altera Excalibur device [Altera 2001] is similar to the Triscend device, but on a larger scale. Excalibur uses an ARM922T core, running at 200 MHz, and has an array based on Altera's APEX 20KE FPGA range, capable of supporting designs of up to one million gates. Similar to the A7, the Excalibur has on-chip memory and embedded system peripherals. The processor, memory, and peripherals are placed on a stripe along one side of the chip, with the rest of the device being given over to the FPL. The FPL array has a internal bus interface, conceptually similar to the CoreConnect architecture used in the Virtex-II Pro, whereby custom logic is memory mapped into the processor's address space, and an external interface allowing custom logic access to device pins. Similar to the A7, the act of reconfiguring the FPL array also causes the processor to be put into reset state.

The mixed signal (i.e., both digital and analogue electronics) SoC part FIPSOC, provided by Sidsa, has a reconfigurable array connected to a microprocessor [Faura et al. 1997, Sidsa 2000]. The FIPSOC architecture consists of an 8 bit 8051 microcontroller, a single digital FPL array, a single analog cell array, some on-chip memory, and I/O peripherals. The digital FPL array is a LUT based architecture that uses both a multicontext fabric and supports partial reconfiguration. The array consists of Digital Macro Cells (DMCs), which contain several LUTs and registers and some internal wiring resources. The output of each DMC is memory mapped into the microcontroller's address space, enabling it to read any DMC output. The microcontroller's address and data buses can also be connected to the wire matrix in the FPL, enabling cores in the FPL to have memory mapped interfaces.

All the above approaches use a bus/memory map system to connect the processor to the FPL; an alternative approach to integrating the FPL can be seen in the Infineon 20xx [Infineon 2000]. In the Infineon 20xx is a DSP processor, which has an upgradable instruction set, similar to the PRISC architecture. In addition to the normal execution unit, the 20xx has space for 4 "PowerPlug" modules, which are custom instructions built at design time to suit a particular problem. Because the compiler needs to understand where the instructions are at all times, the contents fo each "PowerPlug" module is fixed at design time.

2.5 FPL Management

Despite the increasing size of FPGA devices, allowing greater amounts of logic to be realised on the array, there is still a need for a way to manage the FPGA's contents. There are two main reasons for this. The first reason is that a given problem may require more hardware than can fit onto the array at once, so the problem is partitioned into a group of circuits which are swapped on and off the array as demand dictates, a notion which is often referred to as *virtual hardware*. The second reason is that the FPGA may be shared between a set of tasks, each of which has its own circuit(s). The area of FPL management is now becoming a hot topic in the FPL resource community.

A way of managing an FPGA connected to a host computer is to allow the array to be virtually divided up between a number of circuits, so that multiple applications can reside on the FPGA simul-

taneously, typically relying on partial reconfiguration. One technique for this is using the Swappable Logic Unit (SLU) model described in [Brebner 1996]. Here the FPGA is treated as either a *sea of accelerators* whereby the resource is used to instantiate multiple independent SLUs, or as a *parallel harness* in which SLUs cooperate on a single problem and communicate with each other. Under the sea of accelerators model, applications register their SLUs with an operating system, make requests for SLUs to be instantiated, and requests for them to be executed. On registering an SLU the system becomes aware of the size and I/O requirements of an SLU and also the raw data required to later instantiate it on the array. When asked to instantiate an SLU, the system will attempt to find the minimal free space required for the SLU on the FPGA, removing existing SLUs using an LRU policy if necessary. Finally, on execution, the system maps the application's input(s) to the SLU's input register(s), and then similarly returns any output(s). A similar technique is proposed specifically for an XC6200 based expansion card in [Burns et al. 1997].

Configuration Caching is a technique used in processor/FPL hybrids when the FPL resource can contain multiple circuits of which only one can be active at a given time. The inactive instructions on the array are considered as cached, ready to be used. There are numerous policies that can be used, and the type of policy that can be used depends of the type of reconfiguration model the FPL supports. The configuration caching problem also differs from either processor cache management, or virtual memory management (which is more analogous), as the configuration bitstream size will vary from circuit to circuit, adding a further layer of complexity. A comparison of configuration caching techniques for different FPL models is given in [Hauck et al. 2000]. The authors divide the caching algorithms into three classes: run time algorithms, which use recent history information (e.g., LRU); complete prediction algorithms, which base the scheduling on the order of loads to get optimal scheduling; and general off-line algorithms, which use profiling information to estimate usage. To reduce the scheduling complexity, the authors apply a one dimensional allocation model. The authors propose suitable policies for each FPL model, which compared to not using configuration caching show a performance benefit. They also show that despite the additional hardware costs, both multicontext and partially reconfigurable devices out perform a single context device (based on estimated resources for the same die area).

Although the complete prediction and general off-line algorithms used in the above studies are useful for a constricted environment like an embedded system (and at that only a small subset thereof), in a more complex environment such as a workstation environment, where there is no way to gather the information required for such techniques, the only option is to use run time algorithms. [Sudhir et al. 2001] extends the work in [Hauck et al. 2000] by examining a variety of run time caching policies and applying them to the partially reconfigurable FPL models. Their most successful algorithm is a history-based algorithm which with suitable hardware support stores information about the order in which custom instructions occur. What makes this interesting is that it uses more detailing hardware gathering support than is used in the analogous virtual memory system, which uses very limited hardware support for usage statistics.

The above techniques have been developed under the assumption that there is a controlling entity, such as a microcontroller or microprocessor, attached the the FPL device. In such an arrangement the placement of circuits is left to the controlling entity. The alternative though is to place some logic on the FPL device itself to make it self controlling. Circuits could then be simply presented to the device and it would place them without any other form of intervention. In [Brebner & Diessel 2001] such an arrangement is explored using an FPGA. Dynamically fitting two dimensional cores of differing arbitrary sizes into a two dimensional array is a computational difficult task, demonstrated to be NP-complete, and to do so with minimal delay is NP-hard [Diessel 1998]. Because of this the work concentrates on one dimensional placement, which works well on architectures such as DISC and the

Xilinx Virtex, which are all configured in columns. The concept is to divide the FPGA into three regions perpendicular to the axis of allocation. The biggest area is used for SLUs to be loaded into for running, which is bordered by an area dedicated to buses used for communicating between SLUs. The final region is dedicated to logic used to manage the placement of SLUs as they arrive. In this work it is assumed that tasks will leave the array when completed, and tasks are run to completion, with other tasks being blocked until there is sufficient free space to support them. Thus the two main tasks that the management layer needs to handle are allocation of space for SLUs and compaction of the array. To achieve these tasks, the management layer keeps a one dimensional bitmap of allocated columns, which it uses to find free spaces in a first fit fashion for incoming SLUs. The management hardware generates a series of ones the same length as the number of columns needed for an arriving SLU and slides that alongside the bitmap until it finds enough free space. This operation can be carried out whilst other SLUs are running, so locating free space is a zero cost process from the point of view of existing tasks, but there is a latency while the match occurs, which is directly proportional to the width the of the device. Compacting the array is tricker; if the system is to avoid moving circuits off the array and reloading them then the FPGA architecture needs to support the ability to shift circuits along the array, which currently no architecture supports. However, with such a feature, SLUs could be compacted to make room for new SLUs as the array becomes fragmented during use.

In a virtual memory system, paging memory is pure overhead; the application whose pages are being loaded can make no progress whilst this operation is happening. Similarly in a processor hybrid system an application does no useful work whilst blocked on a custom circuit loading. The load time then becomes an important design consideration; if adding more functionality to a circuit will increase its load time, then the extra load time may negate the performance benefit of using the custom hardware. Although careful scheduling can attempt to reduce this problem, there will always be some startup cost where the circuit is brought in for the first time. One technique that could be used to hide this load is prefetching, a technique used in other fields of computing to ensure that data is loaded before it is needed, so the application does not need to wait for it to be loaded when it needs it. This technique assumes that it is possible to load the data either in parallel with other work or at a suitable idle period, but if successful can hide loading latency. Prefetching is not as simple as it may seem, as finding a suitable point to prefetch data, especially for large amounts of data, such that the data is not fetched unnecessarily, is difficult. If the data is prefetched ahead of branches then the prefetch may turn out to be unnecessary, or worse, remove data from memory that will be needed later. Thus, the algorithm used for prefetching needs to be carefully considered.

[Hauck 1998] describes prefetching for a single context reconfigurable array attached to a RISC processor running a single task. The basic idea is that in addition to instructions that execute custom instructions, there are instructions for prefetching instructions onto the array. These are inserted by the compiler at what it predicts to be a suitable point. Using the SPEC 2000 benchmarks, the authors first present an optimal prefetching run implemented by hand, which give an upper bound for the possible performance gains of an application when using prefetching. The upper bound does show significant benefits, with the applications reducing the time wasted waiting for circuits by on average 88.6%. Whilst the optimal upper bound is promising, real world examples are unlikely to achieve this. However, using the output of a static code analysis tool to insert the prefetch operations, essentially making the insertion of prefetch instructions part of the tool chain, they get an average reduction of delay of 51.9%, still a substantial saving.

2.6 Security

Security is an area of FPL design that has typically been neglected. Being flexible devices, FPGAs are susceptible to a number of attacks, ranging from the conceptual issues (loading a circuit that is functionally malicious) to physical issues (misconfiguring a device to cause electrical damage). Given the ubiquity of FPGAs today this issue will need to be considered seriously in future. For a reconfigurable processor hybrid in a workstation environment this issues is of great import. Unlike the embedded systems environment in which FPGAs traditional are found, reconfigurable processors will be more susceptible to attack from malicious programs like viruses.

[Hadžić et al. 1999] provides a detailed overview of the potential attacks that FPL based devices are susceptible to. The range of possible attacks can be considered as falling into three categories: Malicious Electrical Level Threat (MELT), Signal Alteration Logic Threat (SALT), and Higher Abstraction Level Threat (HALT). The first level, MELT, refers to attempts made to induce electrical conflicts either inside the device or at the external pins. In Section 2.1 we described how CLBs can be connected to wires using tri–state buffers. If a device was configured such that one CLB output a high signal on a wire at the same time another CLB drives the wire low, a short circuit can be created. Short circuits in an FPL device will draw a large current through that part of the device, possibly causing physical damage to the device if undetected. Misconfiguring IOBs, which typically use higher voltages, can similarly cause such damage.

The second and third classes of threat work at the functional level. SALT attacks reconfigure the device to produce meaningless output, which will either cause the rest of the system to produce incorrect results, or may induce unpredictable behaviour in other parts of the system. More maliciously, HALT attacks reconfigure the device to produce legal output that is designed to cause damage to other parts of the system.

A workstation with a reconfigurable processor must be prepared to deal with all three classes of threat. On a current day workstation the amount of trouble a virus or malicious user can cause ranges from simple annoyance to possible data loss. However, with a reconfigurable processor, a virus or "hacker" could potentially damage the processor with a MELT attack. Similarly, without proper security measures in place it is possible that processes may maliciously or accidentally reconfigure other processes' circuits, leading to undesirable consequences. A process may also attempt to access another process's configuration data to get access to circuit designs it should not have access to in a multiuser environment.

The authors of [Hadžić et al. 1999] offer some general courses of action that may be taken to prevent FPL devices from attacks. A form of configuration data analysis could be used to try and prevent MELT attacks. Before being loaded into the configuration RAM, a circuit could be analysed for short circuits and other potentially damaging conflicts, although this could potentially slow down configuration loading times considerably. Basic techniques such as using checksums to ensure bitstreams have not been modified could also be used. Another option is to not allow applications to submit circuits in such a low level format as configuration bitstreams. Instead, circuits would be provided using an higher level description which the system would then perform safety checks on before compiling down to a bitstream, equivalent to how a Java Virtual Machine performs safety checks on byte code before JIT compiling it.

On the device itself it is possible to provide some basic security against short circuits. Many FPGAs provide a current level register, the contents of which is compared with the current current being drawn through the system, which will then generate a signal that can be read by the rest of the system. This signal can then be used to either reset the device or shut it down before the device overheats and is damaged. To prevent short circuits within the device, it is possible to remove tri-state

drivers as a technique from connecting CLBs to wires, and instead use multiplexors. This makes designing the device more complex and potentially can reduce clock speeds, but makes it impossible for more than one CLB to drive a wire. The Xilinx Virtex uses such techniques, and some initial work at Xilinx has shown that the only way to damage a Virtex device by misprogramming it is by using IOBs.

2.7 Architecture Evaluation

Having laid out the relevant related work, this section looks at the applicability of the existing reconfigurable processor architectures to our given domain, the general purpose workstation environment. The aim is to see if any existing architecture makes a good candidate for being managed in such a system. If no ideal architecture can be found then the section should help describe which features do and do not work in the given domain in order to drive the specification of a new processor architecture that is suitable.

2.7.1 Overall Approach

Firstly, the simplest approach detailed was using a separate FPL device connected to the processor over a bus. This solution lacks cohesion, and has been demonstrated to suffer from considerable latency issues [Ludwig et al. 1999]. Another drawback of this architecture is that the programmer will need to consider synchronization and concurrency issues; it would be easier for programmers to manage application interactions with custom hardware in such a system were it more tightly integrated.

At the other extreme is building an entire CPU on an FPGA, but this solution is seen to be inefficient for a general purpose computing device. Although certain applications may benefit from a completely customizable datapath, most applications will not. It is unlikely that in the near future software will be built entirely of custom instructions, and indeed based on the limited amount of FPL that will be available for custom instructions, it is intended that smaller applications running on the machine should not use custom instructions. For example, on a UNIX system many tens of daemon processes run in the background. If every one of these required the use of custom logic then we do not believe the system could reasonably cope. Additionally, the need for traditional instructions to deal with operations such as integer math, boolean logic, address calculation, and branching will still exist in most applications. Putting the logic for these instructions onto the FPL will increase their space and time requirements needlessly.

The more balanced approach would seem to be combining the FPL and processor core on a single IC. The integrated solution significantly reduces the communication latency in the system. Compared to the option of building the entire system on the FPGA, this solution allows dedicated hardware to be used for the traditional part of the processor core and FPL to be used just for the custom hardware that applications need. Numerous projects have shown that, at least when dedicated to specific applications, this approach provides a workable solution that is not too hard to program for, and at the same time provides reasonable performance benefits.

2.7.2 FPL and Processor Integration Style

Most existing commercial solutions that combine FPL and processor on a single device use a memory mapped interface between the programmer and the FPL, where custom hardware cores are attached to a bus and then respond to memory reads and writes. This solution assumes a clear division between the hardware world and the software world. For software to talk to custom hardware it will need to move

off the processor and across one or more buses. This in itself adds latency to operations involving custom hardware. Even just using load/store instructions to interface with the logic is slower than a traditional invocation of an instruction in a traditional function unit. In addition to this, as discussed in Section 2.4.1.3, the processor's memory interface needs to be manipulated by the programmer to stop interference from the cache unit(s). In a workstation system where the programmer cannot halt interrupts then this interface is unmanageable if the device wants to make use of burst transfers.

There are additional design time problems with the memory mapped interface approach. It is the role of the custom hardware core to respond to the correct address range, and this must be built into the core at compile time. However it is important that no two cores have overlapping address ranges. In an embedded system this is fairly easy to ensure, assuming there is a central architect on the project who can manage that. On a general purpose workstation this is impossible to manage, equivalent to all applications existing in a single address space using position dependent code. A solution could be to have the operating system insert address ranges into the custom hardware each time it is loaded onto the FPL and adjust the application's virtual to physical memory mappings as appropriate. Although workable, this is not a easy solution, and we would prefer to have a simpler system that does not require modifying the custom cores at run time.

In addition, the FPGA style architectures would require a suitable custom hardware core placement algorithm to be devised. Dynamically fitting two dimensional cores of differing arbitrary sizes into a two dimensional array is a computational difficult task. If the operating system were required to place and route circuits on the array with each context switch then the system would be unusable. In addition, the operating system not only has to find space for the circuits, but also ensure that they are attached to the appropriate buses, which may impact other circuits and possibly be unrouteable without tearing up the array and replacing and rerouteing everything. It is possible to simplify this matter though by applying some notional ordering to the array, but at a obvious loss of flexibility. A system such as that described in [Brebner & Diessel 2001] could be used: the system buses run along the length of one side of the array and the array is then allocated in strips on the perpendicular axis. This removes the bus routeing problem and turns the two dimensional scheduling problem into a one dimensional problem, which reduces the complexity of the placement problem from being NP-complete to $O(n)$. This solution is made slighly more problematic on an architecture like the Virtex-II Pro, where the processor block is surrounded by the FPL array rather than being off to one side like it is in the Altera Excalibur range of devices.

The model used by PRISC, CoMPARE, GARP, and the Shark DSP is more suited to providing a custom instruction set, as they both integrate the FPL onto the processor's datapath. In these architectures the FPL is placed directly on the processor core's datapath. Using a piece of custom hardware is then similar to invoking any other instruction in the system. The interface between the processor and the FPL is fixed, accepting values off the internal buses and returning a result similarly. This interface is less flexible than the memory mapped interfaces. Under the memory mapped interfaces custom cores may use an arbitrary range of addresses (so long as the total range of addresses fit within the system's address space) and may use both single word transfers and multiple word burst transfers. However, the benefits of moving the logic into the datapath are reduced latency on instruction invocation and returning the result and not having to interfere with the memory management unit every time a custom instruction is used. A loss of bandwidth due to not having burst transfers could be countered in two ways. The first way is to simply allocate a wider register file for use with the reconfigurable unit, just as is done for SIMD units in many processors. The Motorola MPC7400 PowerPC processors use a 32 bit wide register file for the conventional integer unit, but use a 128 bit wide register file to supply their AltiVec SIMD unit [Motorola 1999]. The other alternative is to use more than the conventional two inputs for a function unit, as seen in the CoMPARE architecture, which can be

configured to use either two or four inputs into its configurable unit, the CAU.

The main concern with the approaches suggested in the research literature is that they make it hard to share the array between multiple applications. Work such as CoMPARE, GARP, and the SHARK DSP Hybrid only allow for a single custom instruction to be loaded onto the processor at once. This leads to both internal and external contention. By internal contention we mean that single application may wish to use multiple custom instructions within a loop which can not be combined in a single instruction (e.g. decrypting some audio data and then filtering it), so the application will be forced to keep swapping circuits on and off the array. If there are other applications that use custom instructions then we will get external contention. With a round robin process scheduler (ignoring processes blocking for data), it is guaranteed that the moment more than one process uses a custom instruction, all processes will need to reload circuits after a context switch. Given the size of modern FPGA devices, it seems reasonable that multiple pieces of custom logic should be allowed to reside on the array at once.

In work like OneChip, Chimaera, and DISC, a single array is used, but multiple circuits may be loaded onto the array at once. On OneChip however this is a static configuration, so it reduces the problem of internal contention (but does not remove it as it does not allow arbitrary arrangement of cores) but does not help with external contention. DISC uses a partially reconfigurable array to allow multiple instructions to be moved on and off the array whilst other instructions remain unmodified. DISC treats the array in a one dimensional fashion with address, data, and control buses running along the plane of allocation. Instructions are loaded into free spaces on the array, with other instructions being unloaded to make room if necessary. The RFU in Chimaera is similar in general layout to DISC, though with more complex management support (regarding configuration caching) and integration into a larger processor model. The setup used in both DISC and Chimaera solves the problem of internal contention, as an application may move multiple circuits on and off the array independently. However, like the circuits used on the commercial memory mapped architectures, custom hardware cores in both have a static ID associated with them. In DISC, on the first cycle of an instruction the control logic places the instruction opcode on the control bus and if a core sees its own opcode on the bus then it responds by setting an acknowledge signal on the control bus high. In Chimaera the ECU uses the ID to invoke circuits loaded on the FPL. Because it is unfeasible to assume that no application may use the same opcodes for custom hardware, the operating system would need to clear any instructions from the array on a context switch in these systems. The other issue about such an architecture is that of fragmentation. As circuits are moved on and off the array the array will most likely become fragmented, with strips of CLBs that are too small to be useful. Defragmenting an array without suitable hardware support would require taking circuits off the array and reloading them, which will be a time consuming operation. A possible hardware solution to this problem is discussed in [Brebner & Diessel 2001], where the SRAM behind the fabric supports shifting configuration data along the array, but no current fabric supports such a method.

The solution used by the PRISC architecture successfully eliminates internal contention. PRISC uses an array of multiple independent FPL blocks called PFUs, each of which can be reconfigured without affecting any other blocks. To tackle external contention the PRISC architecture associates a register with each PFU which contains the numeric ID of the instruction in question. Software instructions refer to this ID when attempting to execute a circuit in a PFU; if there is a match then the circuit executes appropriately, otherwise an exception occurs which allows the operating system to rectify the situation. Through careful use of these IDs it is possible to ensure that circuits from multiple applications can reside on the processor. It is unreasonable to assume that applications will have distinct ID ranges, so the operating system will need to clear the index registers on a context switch and then reload the indexes appropriate for that application as faults occur.

The PRISC design seems the most suited to working with multiple circuits for multiple applications, but it is not without its drawbacks. PFUs on the PRISC can only last for a single cycle, as the fabric does not contain any state. This drastically limits the complexity of circuits that can used and the depth of logic they can use, being limited to that which will fit in a single clock tick. The ID register arrangement forces a static linking between the assigned ID and the circuit. Dividing up the array statically into fixed sized blocks will lead to wastage of FPL as not all circuits will fill each PFU. The sum of the unused spaces in the PFUs could potentially be sufficient for another circuit, but can not be used due to the static boundaries. Similarly circuits cannot exceed the size of an PFU, although given the limits of not having state it would be possible to determine an upper bound for the size of circuit that could fit in a PFU given the time constraints.

2.8 Summary

This chapter has presented the background material necessary for understanding work in the rest of the dissertation, and provided an examination of the existing work in the field, looking at current state of the art in reconfigurable processors and FPL management issues.

From looking at the reconfigurable processor work, it can be seen that no existing architecture is ideal for management by an operating system in the unpredictable and dynamic workstation environment. This means that before examining the core issues of this dissertation, the operating system management and programming support, proper hardware support for conventional management techniques will need to be addressed.

In the FPL management literature, there has been no work so far to address the issues of managing an FPL system for a workstation environment. Although some work has looked at how FPL could be shared between applications, no work has examined in details the effect of frequent context switching with unpredictable workloads on the overall system performance. This is something that will be addressed in this work.

However, the existing body of work in the field provides a good basis on which to build. The next chapter will build on what has been discussed in this chapter and outline the requirements for a workstation system build on a reconfigurable processor.

Chapter 3

Requirements and High–Level Design

The previous chapter discussed the existing body of work in this field; this chapter moves onto outline what the requirements for managing a reconfigurable processor in a general purpose workstation environment. The main aim of this project is to investigate how an operating system can manage the reconfigurable resource on a processor, given the dynamic workload and the need to share out the limited resource in a fair and secure manner. At the same time the aim is to do so without requiring significant alterations to how either the operating system or applications are structured. As outlined in the previous chapter however, no existing architecture is perfectly suited to the type of environment specified, so before examining the operating system and programming language issues, this work will also consider what makes a suitable architecture for the target environment.

The aim of this chapter is to provide an initial set of requirement analysis and high-level design decisions, laying the ground for the detailed design discussion in the following three chapters. The structure of the discussion in this chapter will focus on how the custom hardware used by applications moves through the overall system from application development and the associated programming model, through the operating system management layers, and down to the actual hardware upon which it will execute.

3.1 General Approach

The motivation for this work was to allow applications to use application specific custom instructions as opposed to providing general purpose domain instructions, such as Intel's MMX and Motorola's AltiVec. As such, the model outlined by PRISC makes a suitable starting point for this work. The PRISC's fixed division of the FPL into identically sized blocks, PFUs, makes the FPL easier to allocate than a single large shared block of FPL allocated on either a 1D or 2D basis, and the ID registers allow applications to be decoupled from the location of their circuits. However, as outlined in the evaluation section of the previous chapter (Section 2.7), the PRISC architecture lacks certain interesting properties. The PRISC FPL blocks do not support stateful sequential logic, which would allow for more complex circuits, and lacks an ideal dispatch mechanism, as it does not support the notion of processes.

Another aspect of the approach taken from PRISC, GARP, Compare, and Chimaera is the lack of IOBs in the FPL fabric. In the context of a workstation system, applications will be expected to use the traditional memory hierarchy to move data about, so the FPL will only be connected to the conventional processor datapath, and will not directly use device pins. This aids significantly in making the system secure. It prevents IOBs being misconfigured to cause damage to either the

processor or connected devices.

The fixed divisions between the blocks of FPL will simplify management of the array; as this is an initial exploration of the management issues in the workstation context it is felt that this simplification makes a good starting point. However, it is hoped that the parts of the system that do not directly relate to the placement of logic in the FPL blocks, such as the programming model, can be designed as to support other arrangements.

An important point to address at this early stage is how many PFUs are anticipated. From an availability perspective, the more PFUs available the better, but there is only a finite amount of silicon available, so more PFUs means smaller PFUs, which means less complex circuits can be used. To address this, a small number of applications were selected for acceleration in order to determine what size of circuits they would require to instantiate (the test applications are described in Section 4.3.2). The core parts of their respective algorithms were replaced with bits custom hardware, the sizes of which were used to determine the size of the PFUs.

For this experiment, a model based on a simple ARM 7 core (described in detail in Section 4.2.1) connected to FPL blocks based on the Xilinx Virtex fabric was assumed. The applications used a mixture of single and multiple cycle custom instructions, which were built and synthesised for a Xilinx Virtex device. The custom instructions developed were between 100 and 500 CLBs in size, ranging from simple Single Instruction/Multiple Data (SIMD) arithmetic to encryption convolution and a complete graphics filter. Based on these example circuits, assuming that 600 CLBs represents a reasonable upper bound of custom instruction size, then the number of PFUs a processor might have can be estimated. Based on the die size for the largest Xilinx Virtex part, and subtracting the die area of an ARM 7 core, and then dividing the numbers of CLBs left by 600, a rough estimate for how many PFUs can be used is achieved. The estimate is very rough, but as this work is concerned with the management of the PFUs rather than the detailed design of such a processor, this is considered sufficient. By subtracting the die size of an ARM720T core, it is estimated there would be roughly 6000 CLBs to divide between PFUs, which provides ten PFUs on our hybrid. As already stated, this figure is only a guide, but it allows us to put the rest of this work in context knowing that there are only a relatively small number of PFUs available.

Of course, this value should increase over time as device density increases. The top end device of the current Xilinx flagship FPGA range, the Virtex-II, has just under quadruple the logic density of the top end Virtex device. Giving the option of either larger or more PFUs. Despite the possibility that in the near future devices may have several tens of PFUs, in relation to the number of applications an general purpose workstation does, this is unlikely to mean that the resource will not be saturated. What is likely to happen, as with other resources such as memory, disk space, and network bandwidth, applications will expand to utilise the resource to the point of saturation.

3.2 Application-Level Requirements

In order to aid application developers in making use of the FPL resource in such a system, the interface between applications and the custom hardware should be as simple as possible. The aim is for the use of custom hardware to be integrated into a modern software design flow without requiring any radical changes to that flow. Just as programmers today generally do not need to be concerned with the hardware details of Out of Order (OoO) and superscalar execution models on modern processors, the same should be true of mechanisms used to utilise custom instructions.

There are two main routes by which custom instructions can be utilised in a program. The first route is for the programmer to provide a custom instruction description and explicitly invoke that

hardware when needed. The compiler and linker then have to translate the inclusion of the custom instruction and the specific invocations into the correct operating system and hardware calls. The custom instruction description may have been provided in house as part of the application development, or might be from a third party library of custom instructions (c.f. cores provided by FPGA vendors). The other route is for the compiler to analyse the source code for a program and generate the hardware to go in the the PFUs. This work will focus on the first route, as this is the route that was used to develop the example applications in Section 4.3.2, and having the compiler generate instructions will require the same linking mechanisms just not the techniques for specifying custom instructions at the program language level. The actual generation of the hardware for custom instructions is considered outwith the range of this body of work.

3.2.1 Instruction Interface

To simplify the hardware model (see Section 3.4 below), custom instructions are syntactically similar to other traditional instructions in the system; they take one or two operands and return a single result, which is the interface used in PRISC and Chimaera. Although the actual interface at the hardware level will be more complicated, this is as complex as the software developer needs to understand. The advantage of the simple interface is that it does not require substantial changes to the conventional processor datapath, and will be simple to program; for example, it avoids the synchronisation issues with a memory mapped bus interface as discussed in Section 2.4.1.3. The obvious drawback to this solution is that the interface may potentially be a bottleneck in the system, with the units being limited by the amount of data they can receive. However, this approach has been used successfully in several existing architectures, and initial experiments will determine whether it is indeed is a bottleneck.

3.2.2 Software Alternative

Although it is hoped that circuit swaps will not degrade performance given sensible management techniques, the number of FPL blocks available on a processor in the short term future is unlikely to be great, if these blocks are to be sufficiently large to house circuits of the size anticipated. It is entirely possible that under heavy load the demands made of the FPL resource will significantly outstrip supply, and the system may end up thrashing, causing the system to spend a large amount of time loading and saving configuration bitstreams. Bitstreams have the potential to be several tens of thousands of bytes long, and may need to be brought in from disk to contiguous physical pages before the actual loading can take place to ensure a continuous configuration stream during configuration.

Thus, it may prove useful for applications to be able to fall back on a software based alternative to their custom instruction; although slower, issuing to this would relieve the contention on the FPL resource. The decision to use either the hardware implementation or the software alternative would be taken by the operating system at invocation time. If the FPL resource was considered to be overloaded then subsequent invocations of unloaded instructions that have software alternatives defined would be issued to software until such time as the operating system decided to promote the instruction in question to hardware. This could potentially increase system throughput by eliminating the overhead of circuit swapping. Having the alternative would not necessarily be a requirement of the system: for some applications, such as real-time applications, it may not make sense to fall back to a slower alternative, but this option may have benefits for less time critical applications, so is worth exploring. Real-time applications could, however, benefit from specifying a software function that was called if the operating system could not load the application's instruction immediately, so as to take corrective action for possible missed deadlines.

Because the operating system should not be involved in the invocation of each custom instruction, the hardware layer will need to support the dispatching to both hardware and software, and it must do so such that the operation is hidden from the application.

3.2.3 Namespace Management

An important part of constructing a software system is managing objects, where an object is defined to be anything to which a program may wish to hold a reference: functions, blocks of data, files, devices, etc. To be able to work with objects, applications use various names within a set of namespaces to be able to locate and use the objects [Saltzer 1978].

Existing work that extends software with custom hardware treats custom instructions as simply the bits representing the hardware configuration bitstream, which are moved around explicitly. However, in this work custom instructions are treated as named entities, similar to functions and variables, that contain the hardware definition, software definition, state, and meta data. Thus, a custom instruction becomes more than just a bitstream: it is now a package containing all the various parts that constitute a custom instruction.

Although it is not the intention of this work to investigate the application layer in detail, it is assumed that the compiler and language should support the notion of custom instructions; they should provide the facilities for generating a custom instruction and for associating names with them. These names can be used to invoke the instructions from within the language and for duplicating and sharing instructions. For instance, it should be possible to associate a single name with different instructions during the lifetime of an application — one might have a filter instruction which maps to different filters during execution depending on the effect the user wants at that time. This distinction between names and actual custom instructions is important, as it has a great impact on the linkage mechanisms used to resolve names to custom instructions.

In addition, the ability to use traditional static and dynamic linkage methods used to build applications in a workstation environment should be ensured. For example, not only will the system provide shared libraries of code, but it may also provide libraries of code and custom instructions or just libraries of custom instructions.

At the machine instruction level, custom instruction invocation will use an opcode for identification, rather than the symbolic name used at the programming language level. The linker will be responsible for translating symbolic names to these opcodes, which are referred to as Circuit IDs (CIDs). CIDs will be a unique name for each point of instruction usage with a process, and will need to be mapped to custom instructions when the application registers its custom instructions with the operating system. These CIDs will be used by the process to invoke custom instructions at run time, and it is the role of the lower layers to ensure that these invocations are mapped successfully.

3.3 Operating System Requirements

An important role of an operating system in a workstation environment is to allow multiple processes to share access to physical resources in the computer in a fair and secure manner. As described in Section 2.2, this is typically achieved by virtualising each resource; each process is told it has complete access to the resource, and then the operating system multiplexes access to the physical resource, with assistance from the hardware. The operating system creates a virtual machine for each process in which each process believes it has access to all the resources and is not aware of other processes. During execution, the operating system must map a dynamic set of virtual machines onto

the physical machine without prior knowledge of the demands each virtual machine will make; the best an operating system can do is attempt to make a guess as to upcoming requirements based on past performance.

The operating system will spatially and/or temporally divide each resource at a suitable granularity. The aim of the operating system is to multiplex access such that there is enough to share between the competing applications, yet ensure that each application gets a large enough share so that it can make a reasonable amount of progress (the definition of the term reasonable is both application and user specific). For example, the CPU scheduler will typically give each process a small time slice on the processor (of the order of ten to a hundred milliseconds), before removing it and putting the next one on. This slice is sufficiently small that many processes can appear to make progress over a short time frame, but long enough that each process can make a reasonable amount of progress before it must release the processor.

In the system proposed in this work, the operating system should be responsible for managing access by processes to the limited FPL resource, and just like other resources the FPL resource should be virtualised.

3.3.1 Name Mapping

Although typically not responsible for carrying out the translation, the operating system is responsible for maintaining a mapping applications' virtual names for resources to the underlying resource. For example, the operating system must maintain page tables for virtual memory that contain mappings of virtual memory addresses onto the physical memory address used to reference the memory, if the page is loaded. This information is used by the operating system to program the processor's address translation hardware.

In the system proposed here, the operating system needs to maintain a set of mappings for the CIDs used by each process, keeping a reference to the hardware configuration bitstream and software implementation for each instruction, noting also that circuits may be shared, so multiple CIDs may map onto a single instruction. It will also need to note to which implementation a CID should be dispatched to, and if it is dispatching to hardware what PFU it is currently loaded into. This information can then be used to program the hardware appropriately to carry out the translations at run time (see Section 3.4.1).

3.3.2 Scheduling

Given the prediction that in the near future processors will support only a relatively small number of PFUs, it is highly probable that there will be times when the total active demand for PFUs exceeds the number available. This will result in points where the operating system will either have to block a requesting process until PFUs become free, run the requested custom instruction in software, or swap circuits in and out of PFUs during execution, essentially using time division multiplexing on top of the space division multiplexing that the PFUs provide. This means that, if state is allowed in custom instructions, the processor must support a technique for storing and restoring that state as circuits are moved off and back onto the processor.

The problem with selecting circuits for moving on and off the processor is deciding which is the best circuit to evict. This is similar to the problem in virtual memory management when deciding which page should be evicted, and the configuration caching work described in Section 2.5. Ideally the hardware should provide useful information about the usage of the PFUs, to allow the operating

system to make an informed decision about which circuit to evict. The information provided should be sufficient to allow a range of different eviction policies to be used.

3.4 Hardware Level Requirements

This work is interested in the hardware aspects related to managing the FPL resource, rather than the low level details such as how the FPL fabric is constructed. As a result, questions such as what type of FPL fabric is best suited for the type of applications that will be used in the given environment are not considered. The work is limited to just the parts of the hardware construction which affect how the operating system and user applications interact with the FPL. This work focuses of the dispatching of custom instructions, how the instructions interact with the rest of processor, and how the software drives the FPL. However, the work cannot be completely agnostic to the fabric design, but where it does touch on fabric design is is only as it has a direct consequence on the management issues.

One of the constraints of the workstation environment is that the processor must be able to guarantee that the operating system can regain control of the hardware at any point within a short timeframe. Being able to guarantee control to the operating system is a requirement for supporting a pre-emptive multitasking environment, and having short interrupt latencies is a requirement for supporting interactive or soft real-time applications. The changes made to the processor to integrate reconfigurable logic should not have an impact on either of these attributes. Configuring an FPL array can take a relatively long time compared to the normal maximum delay for an interrupt (for example on the ARM the maximum delay is twenty CPU cycles, which is a lot less time than it takes to configure a FPL device), and must therefore be interruptible. Because custom instructions may run for many cycles, they must either have a maximum duration or themselves be interruptible.

Most modern workstation operating systems use multiple private virtual memory spaces for processes. This means that the bitstream for a custom instruction, when held in application memory, can not be guaranteed to be either paged in or, once paged in, held contiguously in memory. Either the hardware or the operating system will need to cope with this artefact, as one would typically assume a contiguous block of bytes for the bitstream.

3.4.1 Dispatch Mechanism

The dispatch mechanism is responsible for mapping user invocation requests for custom instructions to the actual instantiated instruction, similar to the ID registers in the PRISC system. In the system proposed, the dispatch mechanism needs to be able to translate each custom instruction invocation made by applications into one of the following: an invocation on hardware loaded into a PFU, a call to the software alternative specified by the applications, or a call to the operating system to request that it resolve the unbound request. Correspondingly, the hardware will need to handle the namespace mapping from the process-specific CID in the invoking instruction to the correct hardware loaded in a PFU, the software function that is being used as an alternative, or provide a trap to the operating system if no correct mapping information is present.

The dispatch system needs to be flexible enough to cope not only with one to one mappings, but also with many to one mappings. If a custom instruction does not use any state (for example, the SIMD arithmetic instructions developed during the initial demonstrators), then there is no reason why multiple system names for custom instructions cannot map onto a single instance.

3.5 Device Programming

As was described in Section 2.1.2, there are numerous ways of programming an FPL device. Obviously, because it is unacceptable to reset the processor between loading circuits, a reconfigurable array is desired, rather than the configurable array used in devices like the Triscend A7. A limited form of partial reconfiguration can be advantageous in this context. With a set of reconfigurable PFUs, if we wished to off-load a circuit without losing any state contained within, then we would have to store the bitstream for the entire circuit. However, the configuration bits used to set the routing and LUT contents will not change, only the state held in the registers needs to be preserved.

Thus, it makes sense to separate the configuration data into two parts, or two contexts, one for the static configuration and one for the stateful configuration. This means that only the stateful part of the configuration data needs to be saved and reloaded. This separation also makes it easier to share configuration bitstreams, by having a distinct section for the parts that are the same for all and for the parts that are specific to a particular application. This can save memory in an operating system that uses multiple private virtual address spaces, as it reduces the number of pages that need to be duplicated due to modification between address spaces.

3.6 Security

There are two angles from which security concerns need to be cosidered: FPL related security issues, and ensuring the new processor operations for FPL interaction have the correct privileges. The FPL security issues were outlined in Section 2.6; MELT attacks which attempt to cause physical damage pose an obvious threat in the context of a processor, where the processor can be damaged accidentally or maliciously. However, this is of less concern to this work, which is more about the management and control aspects.It is assumed that the fabric designers will work with this in mind, taking steps such as using multiplexor-based routing. The base fabric assumed for this work is the Xilinx Virtex that uses multiplexor based routing, which prevents such attacks on the fabric. Of more concern are the SALT and HALT attacks, which are semantic attacks, such as reconfiguring another application's instructions without it knowing or accessing data held in another PFU. Whilst there is little the system can do about such attacks on the functional output of custom hardware, there are control signals attached to custom hardware which it will use to indicate status information (such as completion) to the processor, and this could potentially cause problems. A circuit that never signals its completion could potentially lock up the processor, preventing the operating system from regaining control.

The other aspect to consider is deciding which operations can be carried out by user applications and which operations should be carried out only by the operating system. Microprocessors typically operate in at least two modes: user mode, in which applications run, and supervisor mode, in which the operating system runs. The majority of instructions can be executed in either mode: no harm can be done to the system with conventional data processing instructions or load/store instructions, for instance. However, operations like enabling and disabling interrupts, modifying virtual memory maps, and so on, should not be possible from user mode; only the operating system should be allowed to perform such tasks. Similar precautions must be made for instructions that interact with the FPL resource. User applications should be able to use their own custom instructions, but not be able to invoke or modify other applications' instructions. As the custom instruction software interface is designed, the privilege level of each new instruction, and the consequence of this, needs to be considered.

3.7 Summary

The concept of namespace management can be seen as a unifying theme used to tie together all the various stages. At each layer, the way in which custom instructions are handled changes, along with the style of name used to refer to them. However, the overall aim of the system as a whole can be seen as trying to map the programmer's name for a custom instruction at the point of usage in the source code down to an invocation of a specific custom instruction instance. A summary of the various names used can be seen in Table 3.1.

Layer	**Name**	**Referred Objects**
Programming language	Symbolic bitstream name	Circuit bitstream
	Symbolic function name	Software Implementation
Linked process image	CID	Circuit bitstream address
		Software function address
Operating system	PID/CID	Shared circuit bitstream address
		Software function address
Hardware translation	PID/CID	PFU index
		Software function address

Table 3.1: Summary of custom instruction names

Custom instructions are more than just a bitstream to load onto the PFU, although this is by and large the overall aim of the system. Custom instructions are entities consisting of a circuit bitstream, circuit state information, a software alternative, and relevant meta information. These appear as named entities to the programmer, associated with human readable symbolic names. During compilation and linkage, these symbolic names are converted into process unique machine readable CIDs. These CIDs are registed with the operating system in conjunction with a specific custom instruction, and can then be used as opcodes in instructions used by the process to invoke their custom instructions. However, this namespace is not system unique, so the operating system needs to combine the CID with the relevant PID to form a system unique name, which can be used at invocation time to map to the correct instantiation. The hardware layer will have been programmed by the operating system to dispatch this system-unique name to a PFU or a software alternative.

Although there is a lot more to the work that follows, this path, of converting from symbolic custom instruction names down to the invocation of a instruction instance, is a guide. The next three chapters will follow this path from the bottom up, starting at the hardware level and working up through the operating system to the compilation and linkage model.

Chapter 4

The Proteus Architecture

From the work reviewed in Chapter 2 and the requirements laid down in Chapter 3, we find that no single architecture supports all the facilities needed to sufficiently manage the FPL resource on a reconfigurable processor hybrid in a workstation environment. The aim of this chapter, then, is to describe a suitable architecture that provides all the facilities we feel are necessary for supporting management, before we go on to look at the actual management issues in the next chapter. Here we introduce the Proteus Architecture[1], which describes a possible way of integrating reconfigurable logic into a microprocessor core such that the new resource can be easily shared dynamically between multiple processes under the management of a suitable operating system. We do not aim to provide a fully detailed description of a reconfigurable processor architecture here, but to just concentrate on the issues relevant to managing the FPL resource in our given context.

This chapter is split into three main sections. The first part of the chapter introduces the general Proteus Architecture, which describes what we view as a suitable structure for a reconfigurable processor for use in a workstation environment. The Proteus Architecture itself is not a specific implementation, rather a set of mechanisms that could potentially be applied to many CPU architectures. The Proteus architecture defines the basic interface between the FPL and the rest of the processor, how instructions are moved on and off the processor, how the security issues are approached and so on, at a general level. The second part of the chapter describes an actual simulated implementation of the Proteus Architecture based on the ARM7 RISC processor from ARM Ltd. This section details the hardware changes required, such as how the FPL resource was integrated with the ARM datapath, and what new instructions were required to control the new resource and the associated management hardware, providing a concrete example of the principles of the Proteus Architecture. Although the ARM is a simple processor design, lacking many features found in modern high-end microprocessors (such as superscalar execution units, out of order execution, and so on), using it as the basis for an initial prototype simplifies the job of interfacing the processor core and the FPL, allowing the work to concentrate on the resource management aspects of the integration problem. In the final section of the chapter we explain the simulation environment used to model the ProteanARM and describe the results of getting a set of basic applications to run on the raw architecture without operating system support.

[1] Proteus: A sea god, the son of Oceanus and Tethys, fabled to assume various shapes. O.E.D. Second Edition.

4.1 The Proteus Architecture

As discussed in the previous two chapters, no existing reconfigurable processor architecture meets all the requirements we believe necessary for providing support for operating system management. The Proteus Architecture is an attempt to provide a view of an architecture suitable for use in the dynamic environment of a workstation managed by an operating system. The aim is not to provide a full architecture description, but to explore the parts of the reconfigurable processor concept where changes are both needed and preferable to make the processor suitable for our target domain.

In this section we give a high level description of the major features and general layout of a reconfigurable processor suitable for use in a workstation environment. In the next section we will provide an example implementation to provide a concrete example of the concepts discussed here.

4.1.1 General Approach

The concept behind the Proteus Architecture is providing applications with the facility to load custom instructions onto the processor, similar to the the PRISC architecture. Applications will have at their disposal the traditional set of instructions expected from a processor, such as integer and floating point instructions, branch instructions, and so on, but additionally will be able to extend the instruction set at run time to include custom instructions of their own specification. In essence, applications have a virtual instruction set. Instructions are loaded onto the processor as they are needed and later taken off again to make room for other instructions.

Modern microprocessors used in desktop and workstation machines are organised into sets of functional units for different types of work. For example, the IBM PowerPC 750 [IBM 1999] has six execution units: two integer units, a floating point unit, a branch processing unit, a load/store unit, and a system register unit. The data processing units, the integer units and the floating point unit are associated with a register file for each data type: a 32 entry 32 bit register file is linked to the integer units and a 32 entry 64 bit register file is linked to the floating point unit. In addition to execution units being a useful logical arrangement of the processor, such arrangement also helps the processor to be superscalar: the 750 can execute two instructions in different function units each cycle. Each unit in the processor can only execute a single instruction at the time, so many processors double up commonly used function units, as seen by the two integer units on the 750.

The Proteus architecture extends the processor with another execution unit containing the reconfigurable logic. This execution will contain a series of PFUs that can have their contents loaded at run time. It is feasible that an implementation of a Protean processor may use multiple execution units for the PFUs in order to have multiple PFUs executing at once, but we do not consider that here. Associated with the new execution is a register file, whose size and data width is not tied to any other register files. Having a separate register file makes logical sense, as the reconfigurable execution unit is not tied to any existing data type, and the size can be altered independently of other register files should the execution unit require, for example, more bandwidth. The alternative would be to link the reconfigurable execution unit to an existing register file. However, this leads to the question of which one; the reconfigurable execution unit is data type agnostic, and could potentially process all of the commonly found data types on the processor. The benefit of linking it to an existing register file would be that it could then be well placed to process the most commonly manipulated data type, which would typically be the integer register file. However, this would also require that the register file used be wide enough to support all the data types a user may wish to process, and in most workstation processors this is either the floating point unit or SIMD unit. It is thus simpler to simply allocate a new register file for this data-type agnostic unit.

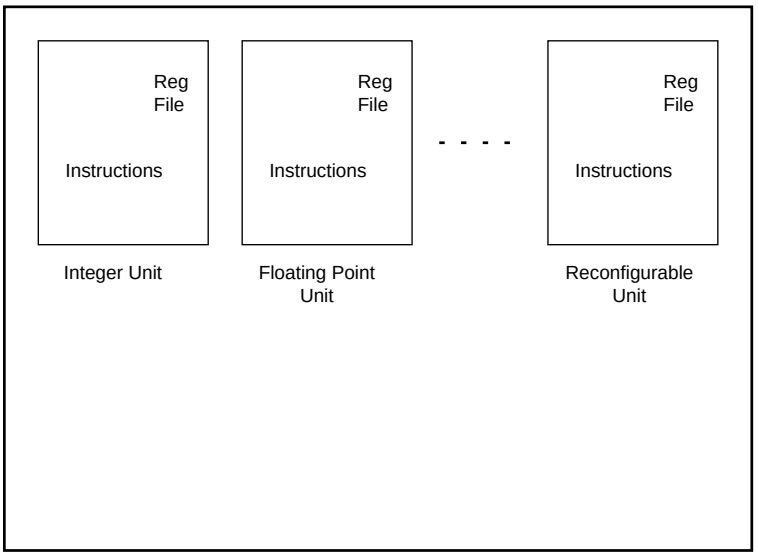

Figure 4.1: Overview of the Proteus Architecture

There is a draw back to hosting data in a new register file however. Because it is expected that applications will use a mixture of traditional and custom instructions within an algorithm, there is potential for data to need to be moved repeatedly between register files as it is processed by the two instruction types. Worse still, a programmer may attempt to avoid this by duplicating existing instructions in PFUs to avoid excessive data movement. In the initial sample applications, described in Section 4.3, this was not found to be a significant problem. Most of the new instructions developed worked on data units not understood by the microprocessor: the custom instruction used by the Alpha Blending example used pixel formatted data, the audio echo processing used SIMD techniques to process audio data four samples at a time. Indeed, this highlights a possible use for the reconfigurable execution unit to provide operations for non-native data types. The applications that did require data to move back and forth did so very rarely, and the process did not significantly add to the compute time for the algorithm.

To conclusively decide which option was best is beyond the man power of the project, and does not significantly impact on the management aspects of the system in which this work is interested. The only side effect of having an extra register file is that it will need to be preserved and restored across context switches, which is not seen as a significant overhead. Given how few registers the prototype platform (the ARM) has and the fact that there was no clear direction from the initial limited selection of test applications, we opted for a separate register file.

Given this arrangement of a new execution unit with its own register file, the general overview of a Protean processor can be seen in Figure 4.1. Very little of the processor datapath has been altered, with only a single execution unit being added.

4.1.2 The FPL Fabric

Although the focus of this work is the management of the reconfigurable resource in a hybrid processor architecture, the work cannot be agnostic to the low-level details of the reconfigurable fabric. The construction of the fabric has important consequences on the type of functionality a custom instruction is capable of, on the management costs of moving circuits about the system, and on system security.

Our starting position for this discussion is to consider the type of fabric used in an FPGA like that described in Section 2.1.1, and from there decide what features we do and do not require.

Each PFU will essentially be a small FPGA, but with a fixed I/O pattern where the data and control lines enter and exit the array. The FPL will be directly connected to the processor's datapath, reading and writing data to and from the normal operand buses for the execution unit. This means the I/O requirements on a PFU are much simpler than those of an FPGA. PFUs will have no need to interface with external pins of the processor since the reconfigurable processor's I/O interface is no different from that of a typical processor. Data to be processed by the PFUs will be moved over the existing data buses. This means that PFUs do not need the IOBs found in a traditional FPGA fabric. This significantly reduces the possibilities for the processor to be physically damaged through misconfiguration, as IOBs present a large opportunity to cause electrical damage to an FPL device. Having a fixed I/O pattern also removes the need for the additional layers of routing placed around the edge of the array on FPGAs (the VersaRing as it is referred to on Xilinx devices).

The other avenue for attacking a reconfigurable processor physically is to misconfigure the FPL array itself, connecting multiple drivers to a single wire. The way round this is to use multiplexor based routing, which prevents multiple drivers being connected to a line, but this increases the routing complexity at the silicon level and decreases the maximum speed at which a device can be clocked. However, security is such an important concern on a workstation system that this option must be taken.

As stated in the previous chapter, the FPL fabric should support stateful elements like registers, unlike the PRISC and CoMPARE architectures, whose' fabrics are stateless. Without registers applications will not be able to use sequential logic in circuits and combinatorial logic circuits will only be of a limited logic depth, as the number of CLBs that can be traversed in a single clock cycle on modern high speed devices is quite low. The drawback of having state in the array is that it needs to be preserved and restored when circuits are swapped in and out of PFUs, adding to the management overheads of using the FPL resource. Without any attempt at optimisation, this will mean that the operating system will need to store circuit bitstreams back out of PFUs before replacing them as the circuits may contain state which will need to be recorded so that the instruction can be correctly restored later. This will significantly increase the circuit switch overheads; for example, in the prototype architecture discussed in Section 4.2 the PFU bitstreams are 54 Kbytes long, a not insignificant amount.

The alternative to moving the entire bitstream off the processor is for the fabric to support partial storage of the array configuration, such that state can be extracted from the array without having to store the entire set of configuration data. There are two ways this could be implemented: as separate configuration planes, or as a partial configuration. As discussed in Section 2.1.2, some FPL devices support having multiple banks of memory to store alternative configurations. Using a similar abstraction, separate planes could be used to configure the stateless and stateful elements on the array. Each instruction would come as two bitstreams, one containing the stateful elements and one containing the dynamic elements. The operating system would then only need to save the stateful bitstream from the PFU in order to be able to restore the circuit correctly later; there is no reason for the processor to offer the facility to store the static part of the bitstream, as the operating system must already have access to this information in order to load the circuit. The second option is to support partial configurations. If the FPL supports either fine-grain or course-grain partial reconfiguration then only parts of the array that refer to state need to be stored and restored, reducing the amount of data that needs to be moved off and on the processor. This partial description could then either be patched into the original bitstream in memory, so that just one configuration pass needs to happen upon reloading, or reloading may become a two stage process, where the original bitstream is loaded and then the partial configuration applied. Either of these techniques is sufficient to support reducing the management

costs of moving circuits on and off the array. The separation of static and dynamic parts of configuration bitstreams is also a requirement of allowing flexible sharing of custom instructions, which will be discussed in Chapter 5.

Modern FPGAs support three types of state within the array: registers in CLBs, LUTs used as small RAMs and shift registers in CLBs, and larger BRAMs that sit alongside the CLBs. We obviously want to reduce the amount of state on the array as much as possible to reduce the circuit switching overheads as much as possible, but we need enough state that we can produce interesting and useful circuits. We also need to consider the overall usage model. The circuits in the array are meant to behave like custom instructions, accelerating software, rather than being autonomous hardware units. Under our model, application state should reside in either the processor register files or in main memory, and only pass through the reconfigurable logic when it is being manipulated. Given this, we feel it is practical to not use the large RAM blocks in our FPL. Using them would dramatically increase[a] the amount of state that needs to be transferred and storing application state in them breaks the model we expect applications to use. XXX

4.1.3 PFU Interface

The integration between the PFUs and the processor's datapath is conceptually the same as for any other execution unit. The instructions conceptually behave like any other other once they have been loaded with a configuration. For example, in a load/store RISC architecture the data supplied to the PFUs will come from the register file and the result will be written back to the register file. The PFUs will be connected to two input buses and a single output bus, the widths of which will be the same as the register file associated. The main difference between a PFU and any other instruction is that a PFU accepts a bitstream. However, we envisage that the configuration data will not move over the conventional data-processing buses inside the processor core, but rather move through a different channel between the memory interface and the FPL configuration SRAM (see discussion in Section 4.1.6.1).

Additional control signals are needed to manage the execution of the circuit loaded into a PFU. Firstly there will be the traditional clock and reset signals used to control the state of the circuit loaded into the PFU. Each PFU's clock will only be active when that PFU is being executed. The reset signal will reset all logic elements within the PFU, and can be used as a way for the operating system to reset any state in a circuit before letting another process use it. It is useful to provide an indicator to circuits of when the first cycle of an invocation is, to allow the circuit to carry out any initialisation it may have to do. A one bit input signal is provided for such purposes, going high on the first cycle of an instruction an low on subsequent cycles. On the other side, the circuit will be expected to drive a signal high to indicate completion, at which point the processor should record the output of the circuit on that cycle and store it as indicated in the invoking instruction. A summary of the PFU interface can be seen in Figure 4.2.

4.1.4 The Dispatch Mechanism

The dispatch mechanism is the part of the processor that is responsible for mapping an application's request for a circuit using the associated CID to the appropriate custom instruction, i.e., dispatching the custom hardware or the nominated software alternative, or if no suitable mapping occurs notifying the operating system. The PRISC mechanism using ID registers associated with each PFU is sufficient for basic operation, but needs reprogramming on every process switch, does not support mapping

[a] get figures

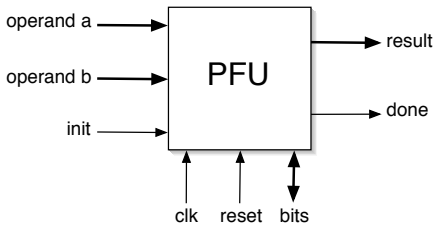

Figure 4.2: Basic PFU Interface

multiple opcodes to a single circuit, and does not support the software alternative mapping. Thus, for the Proteus Architecture we require a more complex dispatch mechanism. The dispatch mechanism in the Proteus Architecture should allow the operating system to virtualise the FPL resource, providing a level of indirection between applications and the PFUs they wish to use. This both decouples applications from the location in which their instructions are loaded and also allows the operating system to control which circuits the applications can access.

In our system, an application instruction will contain a CID as the opcode to the custom instruction it will want to use. Previously the application will have registered the custom instruction with the operating system using that CID, and we assume the operating system has a way of understanding when instructions are shared (see Section 5.2.3). The operating system then has all the information it needs to dispatch an application's attempt to invoke a custom instruction to the correct hardware or software routine. Theoretically, the application could use a system call to invoke the custom instruction without any hardware support, but this would increase the latency on an instruction dispatch significantly, and the aim of providing the reconfigurable execution unit is to speed up execution as much as possible. In this section we describe the hardware support used to allow the application to quickly execute custom instructions independently of their current load status and PFU location.

4.1.4.1 Namespace Issues

The first problem that needs to be addressed is that of external contention. CIDs are only application unique and a given CID may be used in different applications to refer to different custom instructions. In PRISC this problem was removed by flushing the mapping hardware on a context switch so it was not possible for an incorrect mapping to be accessed. As applications try to issue instructions after a context switch, the access attempts will fault and the operating system will reload the mappings as needed. This mechanism also gives potential for applications to share circuits externally; applications A and B may use the same circuit with different names, and the forced reprogramming of the ID registers is sufficient to allow this. The ID register mechanism does not directly support internal sharing of circuits, as it is not possible to hold two names for a single circuit in the ID registers. If an application does this then the operating system will have to fault between invocations using the different names to update the ID register. Although this mechanism works, we would rather avoid the additional faulting penalty that occurs after a context switch. Under this system, even if just one application is using custom instructions it will suffer access faults after a context switch.

The alternative we propose is to combine the application unique CID with the system unique PID to form a system-wide unique ID tuple for each custom instruction reference. Modern microprocessors already store the PID of the currently active process in a register, so the information needed to make a reference, the CID and the PID, will be available on the processor at the time the reference is being translated. It is an important distinction to make that an ID tuple in our system is not a unique *name*

for a custom instruction, but rather a unique *reference* to the custom instruction. As such there is no single global name for a custom instruction, but only a collection of references to a custom instruction.

4.1.4.2 General Behaviour

Given the namespace requirements, we now consider how the dispatch hardware will be used. Multiple references to a single custom instruction means that the dispatch hardware has to be more complicated than the one to one mapping used in the PRISC system. Theoretically it is possible for the entire ID tuple space to point to a single custom instruction, which the dispatch system will have to cope with. The ID tuple space is likely to be quite large; in our prototype system we have an 8 bit PID register on the processor and allow 7 bits for the CID in the instruction format, which gives a global ID tuple space of 32768 entries. However, it is unlikely that this space to be fully assigned at any given time: not all PIDs will be in use, and of those that are only a subset will make use of custom instructions, and those that do are unlikely to utilise the entire possible range of CIDs. Of those CIDs that are assigned in a process only a small number are likely to be active at any one time, similar to the way an application linked with a library of code only uses a small subset of the library typically. As such, it does not make sense to attempt to provide hardware large enough to cope with all possible mappings at once, rather we want to take advantage of statistical multiplexing. Given that there will not be enough space to hold all the possible mappings, below are enumerated the possible responses to the invocation of a custom instruction:

- The instruction is already loaded into hardware, and an ID tuple mapping exists in the dispatch hardware. Instruction is decoded to run in hardware.
- The instruction is to be run in software, and an ID tuple mapping exists in the dispatch hardware. Instruction is decoded to run in software.
- The instruction is loaded into hardware, but no ID tuple mapping exists in the dispatch hardware. Exception called to the operating system.
- The instruction is not loaded in hardware and no ID tuple mapping exists in the dispatch hardware. Exception called to the operating system.

Given that the dispatch hardware will need to hold two types of reference, the dispatch mechanism is split into two halves: dispatch to hardware and dispatch to software. The difference in size of the respective references (3 or 4 bits for a PFU reference and up to 64 bits for an address reference) means it makes sense to use two tables rather than one to save on wasting hardware when storing PFU references. It is assumed that the operating system will not place the same reference in both halves, but should such a case arise then preference is given to the hardware dispatch.

To provide the statistical multiplexing we propose a similar arrangement to a TLB, as discussed in Section 2.2.2 with respect to virtual memory management. In our system, we use two TLBs, one to translate ID tuples to PFU references and one to translate ID tuples to addresses for software dispatch. The size of each of these tables will be implementation defined. When an instruction enters the decode stages of the processor's pipeline, the ID tuple information will be fed into both of the TLBs in parallel. If there is a match in the first TLB, giving a PFU reference, then the instruction will be decoded to execute using that particular PFU. If there is no match in the first TLB, then the result of the second TLB will be used if there is one. If the second TLB returned an address then the processor will start a branch to the software routine at that address using a special branch mechanism,

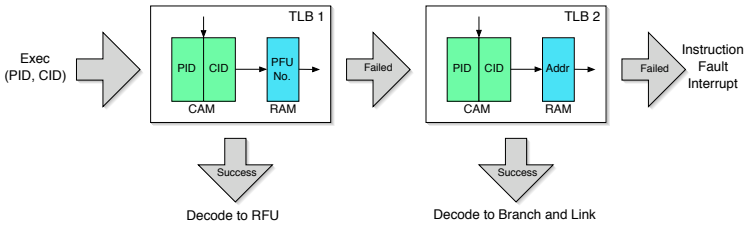

Figure 4.3: The two-stage dispatch mechanism

which is discussed in Section 4.1.5. If neither TLB returns a match then the instruction will decode as an exception, either using an existing exception type if one is appropriate or using a new exception specifically for this type of event; either way an exception will occur and alert the operating system to what has occurred. Figure 4.3 shows a logical summary of this mechanism (the actual implementation would run the TLBs in parallel).

This mechanism is clearly more extensive and flexible than that provided by the PRISC system. The combining of PIDs with CIDs to for a globally unique name removes the need for flushing the mapping hardware on a context switch. It also meets our requirement of allowing multiple names to be used for a specific circuit both internally and externally. In addition, no previous work has provided the facility to dynamically determine at run time whether a custom instruction gets invoked as a circuit in FPL or as a software routine, so in that respect this mechanism is unique.

4.1.5 Software Dispatch

The programmer's view of the software dispatch mechanism is that they have to provide a software function similar to a conventional procedure that has the same interface as the hardware instruction — that is, two word-sized inputs and a single word-sized output. This function is then called by the processor as an alternative to the custom hardware when a process attempts to execute a custom instruction. To provide this functionality, the system needs an appropriate branching mechanism in the processor control logic and the compiler needs to be able to understand the concept of a software alternative.

The overview of a software dispatch is as follows: the custom instruction invocation will cause the processor to branch to the start of the software alternative routine, which will then need to get the parameters of the original instruction, process the parameters, then store the result appropriately, before returning execution to the point after the original execution. From this overview we see two obvious problems: locating the parameters to the original invocation instruction, and storing suitable information so that the process can return to the correct point of execution after the routine has completed.

First, we consider the problem of locating the original instruction's arguments. The software function needs to be able to work out where the data to be processed is held and where to store the result of the function once processing has been completed. This information will be held in the calling instruction, but once we have branched this information is no longer readily available and must be regained. The instruction will have contained either actual data (immediate operands) or an indirection to the data to be used (either a register index on a RISC processor or a register index or memory location on a CISC processor). The simplest solution to this problem is to find the instruction that caused the processor to branch and then decode it manually. The address of the instruction following the custom instruction invocation must be stored in order for the function to

return successfully (see discussion on the branch mechanism below), so by subtracting the length of one instruction from this address we can get the instruction that we are interested in and decode it. This is the mechanism used for floating point emulation on systems such as the ARM. When an ARM binary runs on a processor without floating point hardware, and attempts to execute floating point instructions, an invalid instruction trap occurs. The operating system then finds the faulting instruction, works out what it was trying to do, and then decodes it and runs the floating point function in software. Code to do this can be found in ARM versions of Linux and NetBSD.

The drawback to this approach is that it is very slow. Locating the instruction, separating out the operands, understanding them, and then getting the data that they refer to will take tens to hundreds of cycles (depending on the complexity of the instruction format for that architecture and the type of operands it supports). Given the very nature of what we are trying to do — accelerate a piece of software with custom instructions — we are adding overhead to what is already a time sensitive operation. We would like to avoid this overhead as much as possible by making the processor do part of the work. At the point when the processor decodes the custom instruction to software it can store the operands to special purpose registers, and provide an access mechanism to allow the software alternative to read the operands. This will remove the need for the software to find the instruction and extract the operands. However, the software instruction will still need to interpret these values and turn them into data that can be used for processing. Instead, we want the processor to take care of all of the problems with getting the data. The processor should support a set of extra user-mode instructions, one for each operand that an instruction could use, which allow direct access to the data represented by those operands and to the storage location for the result.

This is best illustrated by an example. If we have a RISC load/store architecture where the instruction format to issue a custom instruction contains three operands: one being the index of the destination register in the register file associated with the PFUs, and the other two representing source operands which can be either be indexes into the reconfigurable unit's register file or immediate values. The reconfigurable unit in the processor will contain three special-purpose registers used to assist in recording these values for later recall. While the processor datapath is carrying out the branching operation, the special purpose registers will be filled with the operands that would have been used for the instruction. One will be filled with the index of the destination register, while the other two will be filled with with either the immediate value contained in the instruction or the value stored in the specified register. Once execution has begun in the software function, it can request the values that the operands referred to using an instruction similar to a normal register transfer from the reconfigurable execution unit's register file, only requesting an operand index rather than a register index. To write back the result, the function will use an instruction similar to that used when writing to a register in the reconfigurable unit's register file. The control logic of the processor will be responsible for ensuring that the correct type of data accesses occur when these instructions are issued, reading and writing either register files or main memory as necessary.

This mechanism then abstracts over the difficulty of using the original instruction operands in software. In addition to the user mode instructions for access the special purpose registers, other instructions will be required to allow the special purpose registers to be preserved over either a process or thread switch. It is important to note that the contents of these registers will be lost if the handler for a custom instruction itself calls a custom instruction which is dispatched to software (a likely occurrence given that one instruction has already faulted, indicating the system is overloaded). Although the first software fallback routine will most likely have already read its source operands, it is unlikely to have attempted to store its result, and will not be able to do so correctly after the second routine is invoked. Although it could be part of the Application Binary Interface (ABI) to preserve these registers guarding against such an occurrence, the practice of calling custom instructions from within

a software fallback should be seen as bad. If execution is within a fallback then the reconfigurable unit is currently overloaded and any subsequent attempt to use a custom instruction has a high risk of failing.

Next we consider the actual branch mechanism itself. On the surface it looks like a similar problem to a conventional procedure call, but it is subtly different due to the unpredictable nature of whether the hardware or software will be used.

First we need to understand how a conventional procedure call works. There are two parts to a procedure call: the operations performed in hardware to move execution to the new procedure and ensure that the subroutine can return, and the operations performed in software to ensure that the caller sets up parameters correctly and the callee does not overwrite data precious to the caller. At the hardware level, the process will issue a branch and link instruction which will be supplied with an address to which to jump. This instruction will move execution to the specified address and move the address after current address, i.e., the address of the instruction that should be executed upon completion of the subroutine, into a special register called the link register (on the ARM a general purpose register is nominated as the link register and on the PowerPC a separate register is used for the link register; either technique is fine so long as the special purpose branch instructions know where to find the link register). Once a subroutine is completed it then calls an instruction to move the contents of the link register to the program counter, allowing it to resume execution at the point after the initial call to the subroutine.

At the software level, compilers follow an ABI, which defines a set of conventions for register use, stack use, and so on during the execution of a program. Part of this definition concerns procedure calls, where there are two important points: how to pass parameters and receive results, and who is responsible for ensuring that register contents are preserved across a procedure call. Nothing actually enforces the ABI in hardware, it is merely a convention so that various parts of the software know where to find information they need. Parameter passing is typically done by assigning a range of registers to hold the parameters, assuming the number of parameters to a function is sufficiently small to fit in that range (say four values in the ARM procedure call standard); any additional parameters are placed on the stack. The same register range is used to store the return value of a procedure call. In addition to setting up the parameter list, when a procedure call occurs, the caller will have scratch values in registers that it will not want to lose, and either the caller or the callee will have to ensure that after the procedure call these values haven't been modified. One option is to have the caller back up and store all the registers around a procedure call, but this is inefficient as not all subroutines will overwrite all the registers. The other extreme is for the callee to preserve all the registers it will need, but again for small subroutines this will require them to create a stack frame which they otherwise would not need to if a few spare registers where available to them. Instead what happens in practice is that the caller will ensure that some of the registers are either preserved or no longer needed so that small subroutines will have enough space to work with without needing a stack frame, and the callee will then preserver registers above and beyond this that it might need. Exactly where the cut of point is is down to design decisions and profiling done by the ABI designers. However the caller *must* preserve the link register, as the contents of the link register will be overwritten during the branch to the subroutine.

The problem with the software dispatch system is that the callee did not expect to be making a procedure call (which is what a software dispatch is, in essence), so the conventions for making the procedure call will not have been observed. In particular the callee will not have preserved any registers, most importantly though it will not have preserved the link register, unless it has already done so in the act of calling a procedure already. This means that either the software dispatch routine cannot modify the link register, which would have been the obvious place to put the return address, or

the ABI needs modifying to preserve the link register when custom instructions are used. If we do not allow the software dispatch routine to modify the link register then we will need to define another link register specifically for the software dispatch routine and additional instructions for using it. In this case we feel it is simpler to extend the ABI to state that all procedures using custom instructions must pessimistically preserve the link register upon starting, and use that preserved value when finishing. To decide what is the best approach actually requires a large body of code to be tested under each option, but there is insufficient time for such testing to occur in this project, so instead we picked what we felt was the most efficient option. All procedure calls bar leaf procedures will preserve the link register upon entry already, so only those leaf procedures that use custom instructions will need to do the extra work of storing the link register.

4.1.6 Long Instruction Support

For a processor to be practical for a workstation environment, the processor needs to guarantee that the operating system can regain control of the overall system at any time and that interrupts can be serviced within a short timeframe of occurring, in order to preserve an interactive environment. To help maintain this, instructions on processors need to be designed to either be interruptible, else if they are atomic, run for an amount of time that will not cause an excessive interrupt latency. For example, the ARM Architecture Reference Manual specifies a maximum bound for instructions since all instructions are atomic. This means that operations involving PFUs must not violate this set of rules.

4.1.6.1 Reconfiguration

Reconfiguring the processor's PFUs will require a comparatively large amount of data to have to be moved onto the processor (when compared to other "large" transfers like cache lines). Halting the processor during the transfer of such a large amount of data would have a negative impact on interrupt latencies. For example, the longest instruction on an ARM core is a load multiple instruction that loads all sixteen general purpose registers, taking eighteen clock cycles and moving 64 bytes of data (which on an ARM750 will require a maximum of three cache lines to be read). Transferring several tens of kilobytes in a similar fashion is not acceptable; the processor has to be willing to respond to interrupts from sources like timers, input devices, and disk to provide the user with a suitably responsive system.

Another restriction on loading circuits in a workstation environment is that transferring bitstream data needs to work with the memory model of a workstation environment. The bitstream used by a process will be held in a virtual memory image that may not be currently in physical memory and when loaded into physical memory may not be held contiguously. This means that either the loading and storing mechanism needs to be able to cope with virtual memory arrangements, i.e., cope with noncontiguous physical pages and the possibility of page faults, or that the operating system must be responsible for ensuring that the bitstream is held entirely in contiguous physical memory. As discussed in Section 5.2.2, when an operating system receives a system call, it typically copies the parameters to the call into operating system memory. In addition, a proportion, if not all, of operating system memory is unpaged. Thus, providing the operating system set aside a large enough contiguous stretch of memory, all custom instruction bitstreams could be held in contiguous physical pages, should the hardware require that. The alternative is to allow the addresses for the address required for the bitstream to go through the normal memory translation hardware. If this technique is used, the operating system will either need to page in the entire bitstream array or the loading mechanism on the processor will need to be able to handle potential page faults.

There are two possible techniques that can be used for loading the bitstream onto the processor, and at the management level we are agnostic to which is used, just so long as the technique used is implemented in such a way that it can be interrupted and restarted. One technique is for the SRAM behind the FPGA to be memory mapped, which is how the Xilinx XC6200 was configured. Configuring a PFU is then simply a case of doing a memory copy, and no special instructions are required to carry out the configuration. Because memory copying will be done using existing parts of the processor's instruction set, this is inherently interruptible, and, if interrupts are handled correctly by the operating system, restartable. Carrying out a memory copy in this fashion will also work well with the virtual memory model of the operating system, as it is not doing anything with the memory system that could not already be done. Because instruction loading may be interrupted, there is a possibility that a partially configured PFU may be invoked. If the architecture has been securely designed then this should not cause any physical damage, but may cause bad results to enter an application. However, it should be the operating system's role to ensure that this can never happen by managing what is in the dispatch hardware. This virtualisation gives the operating system a way of preventing PFUs being accessed without its explicit permission.

The alternative technique used to configure FPL devices is to use a serialised data stream. The SRAM behind the FPL either acts like a large shift register, as in the Xilinx XC4000 range, or a series of smaller shift registers, as is the case for the Xilinx Virtex range. If the FPL exports such an interface then the processor needs to load data onto the processor in a stream from memory. This operation is logically similar to the memory copy instructions found on CISC ISAs such as the Intel IA-32 instruction set [Intel Corporation 1998]. Such mechanisms need to be interruptible to allow the operating system to both regain control of the processor within a short response time and handle page faults. The only difference is that instead of copying the data to a section of memory, the data is simply being moved onto the processor. The design of the circuit load instruction is such that when it is interrupted the program counter will not have been moved on, and the bitstream address value used is incremented during execution, so simply reissuing the instruction will cause the load to continue from the point of interruption.

A problem with simply streaming data from memory into a shifted FPL device is the difference in speed at which data can be transferred from main memory to the processor and at which a serially configured FPL device can accept data. The Xilinx Virtex device can handle a maximum configuration throughput of 8 bits at 8 MHz, where as processor bus speeds are much higher and much wider. This means that the a reconfigurable processor is unlikely to be able to process the incoming bitstream at the rate at which data is transferred from main memory. One option is for the load bitstream instruction to request data at a slower rate. This will potentially waste cycles however. The only reason other instruction cannot be issued during configuration is that the process ties up the memory buses. Thus, an alternative solution is to load the bitstream at full speed into a special bitstream cache, which can then be drained whilst the processor runs other instructions, so long as no attempt is made to either issue another load instruction or execute the PFU being reconfigured. This means that whilst we cannot speed up the actual loading process, other work may usefully be done by either the operating system or other processes whilst configuration completes. The load instruction would consume the processor core until such time as the entire bitstream was loaded into the bitstream cache, then it would appear to complete. However, the PFU will not have been configured yet — the PFU will still be draining the contents of the cache. It is unlikely that an attempt to access the PFU being loaded will occur, as the operating system will be aware that it is still being loaded and not attempt to load another instruction into it, and applications should not be able to invoke it as the operating system should not allow any TLB entries to point to it. Once the cache has been drained and the PFU is ready for invocation an interrupt will be raised to alert the operating system to the fact, allowing it to load

the dispatch hardware with the relevant entries and to unblock the application that had caused the load to occur in the first place.

The drawback of this technique is that it can lead to more complex interactions when many processes are trying to load multiple instructions at once, as shall be demonstrated in Section 5.2.4.

4.1.6.2 Execution

In both the PRISC and CoMPARE systems, executing custom instructions does not not pose any timing problems, as they are only allowed to execute for a single cycle. However, in the Proteus Architecture we require that instructions can run for multiple cycles. This poses a problem, given the PFU interface described in Section 4.1.3, whereby instructions can execute for an unbounded number of cycles. This opens up the possibility that instructions may either run for a significant amount of time, causing an increase in interrupt latency and damaging the responsiveness of the system, or they may not terminate at all, causing the system to lock up.

The Proteus Architecture must therefore either limit the execution length of custom instructions such that they do not impede the interrupt response time or they must be made interruptible and reissuable. Limiting the length of execution of circuits loaded into PFUs is the easiest option, but could potentially restrict the complexity of instructions that can be loaded into the PFUs. For instance, the Alpha Blending example described in Section 4.3.2.2 used as a test example on our ARM based Protean Architecture exceeds the ARM recommended limit for interrupts. As processor speeds increase, custom instructions will need to include more stages to be able to meet the timing requirements of the processor, which means that any limit set in one generation of processor may fail on the next.[b] Rather than be limited by a fixed duration, the approach taken should be to make instructions interruptible. XXX

If custom instructions are interruptible then we need to be able to reissue them successfully. Given that instructions may contain state in them when they are started, it is not sufficient just to restart the instruction with the same operands: we need to resume the operation from the point at which it was interrupted. Under the PFU interface this can achieved simply by restarting the instruction without setting the initialisation signal high on the first cycle. So only the first invocation of an instruction before completion should cause the initialisation signal to go high, and all other invocations (which will be a result of the instruction being reissued before completion after being interrupted) will simply start clocking the instruction again without raising the initialisation signal high. This requires a single bit of state to be stored with a PFU to specify whether it was completed or not at the point it was stopped, which can be done by latching the output of the completion signal out of the PFU. If the instruction had completed when it was stopped then the completion signal will have been latched high, so a subsequent issue of the PFU will receive a high initialisation signal; if the PFU had not completed when the instruction was stopped, then the PFU will not receive the initialisation signal on reissue, and execution will simply resume from the point it was halted.

At the end of a custom instruction invocation, once the instruction has indicated that it has completed, the results should be latched and the program counter updated. If this takes more than a single cycle to complete on a given implementation, then this needs to be made uninterruptible, as the processor is writing back state and restarting the instruction may have undesirable consequences. Once source of interrupts that could cause the processor to suspend an executing custom instruction is the end of scheduling period timer interrupt. This means that the process will now lose control of the processor, and as such the status information regarding whether the instruction is in flight will need to be preserved in the current process's PCB and replaced with that the next process, to ensure that each

[b] Engage brain at some point

process's custom instructions are issued correctly. If the reconfigurable execution unit only allows a single custom instruction to be issued at once then it is sufficient to have a single status bit for the entire unit. If, however, multiple PFUs can potentially be in flight at once, then a status bit per PFU is need.

TODO: what about on CISC architectures when a page fault occurs during an operand write? TODO: read about ix87 math coprocessors.

4.1.7 Usage Information

It is the aim of the operating system is to make the most efficient use of the PFUs as possible, which involves trying to reduce the number of times circuits are swapped in and out of the array. To help the processor in this job we would like it to maintain some information about the usage of the PFUs. The question is what kind of information should be kept. There are two places to look for a guide as to what information to keep: virtual memory management literature and FPL resource management research.

The virtual memory management hardware on a processor is provided with a pointer to the page table describing the virtual to physical translation mappings for the current process. The page table is walked any time a virtual address is generated by the processor and not found in the TLB. On processors such as the Compaq Alpha [Compaq 1999] and Intel Pentium range [Intel Corporation 2001*a*] the page tables provide space for the processor to record basic usage information about access to that particular page. Typically they provide a single bit to note that the page has been accessed and a single bit to note that the page has been modified. These are cleared by the operating system when the page in question is loaded, and set by the processor on address lookup. This usage information can then be used by the operating system to determine which pages have been accessed (or more importantly which have not been accessed, and therefore make likely candidates for eviction) and to locate which pages need to be saved back to disk before eviction and which can simply be overwritten. The aim being to reduce the amount of time the operating system spends moving pages in and out.

Most of the research into FPL management focuses on the complex problem of circuit location inside a large FPL array. However, there has been some work on load ordering and configuration caching, as described in Section 2.5. Configuration caching is concerned with which circuits to leave on inactive areas of a large FPL device or in other configuration contexts in multicontext FPL devices [Hauck et al. 2000, Sudhir et al. 2001]. Both these problems are analogous to deciding which circuits should remain in PFUs on the processor. These research papers use three different classes of algorithm: offline analysis, for which hardware support is not needed; list based history algorithms which just use the usage sequence and not detailed calling information; and statistic based algorithms borrowed from virtual memory management, such as LRU, which could make use of statistics gathered by the hardware on usage. We discuss these in more detail in Section 5.2.4, but here we are only interested in the last option, which can make use of hardware support.

From this, we can see that to aid the operating system we want to provide hardware support for virtual memory style page replacement algorithms, where we provide statistics for each PFU rather than for each page. The hardware layer needs to provide suitable information for the operating system to implement the commonly used page replacement algorithms found in operating system literature [Silberschatz et al. 1998], such as Most/Least Recently Used, Most/Least Frequently Used, and Second Chance. Appropriate algorithms for use in this context are covered in Section 5.2.4.3.

In virtual memory systems, only a single usage bit is kept per page. This is suitably adequate for paging, where there is a large set of candidates, all of which are unlikely to be used during a single scheduling period. This means that when an operating system notes the usage information at

the start and end of each context, it can build up a usage history from these single bits over many contexts. If every page was likely to be used during the scheduling period then the single usage bit would not be able to differentiate the usage of different pages. On a context switch, the TLB has to be flushed, which means that subsequently in a process's scheduling period all memory access to pages will require a look up through the page table, at which point the usage bit can be set. Subsequent uses of that page are not necessarily going to require a page table access, so long as the entry is in the TLB, so a single bit usage history is all that can be guaranteed to be accurate.

For the PFUs, however, the processor can do better. The reason that page tables are not accessed (and therefore updated) on every memory access is that the cost of accessing main memory is high. For a PFU, however, no information needs to be accessed from off the processor. This means that every single PFU access can have information recorded about it. For recent usage algorithms, such as the LRU class of algorithms, a register can be associated with each PFU into which the current value of a monotonically increasing counter (say either the processor's cycle counter or a logical counter incremented on each PFU invocation) will be copied when an invocation occurs on that PFU or a PFU use loaded (this later case is to prevent the just-loaded PFU being evicted based on its old count which caused it to be selected for eviction before).

For counting algorithms, like most and least frequently used, the processor can provide usage counters that keep a record of how often a PFU has been used since the last time the counter was reset. Note that because instructions may potentially be interrupted and restarted, to get a count of the times an instruction was used the counters must be updated on the last cycle of a custom instruction rather than at the start. In our test architecture we have implemented both of these solutions so we can compare the behaviour of the various scheduling policies (Section 5.2.4).

4.2 Initial Implementation

In order to demonstrate that the Proteus Architecture is sufficient to support a workstation class operating system and to test various management policies, we produced a detailed model of an actual Protean processor on which to test an operating system and applications that use custom instructions. Although a silicon or FPGA based implementation was outwith the manpower for this project, a suitably detailed software model will be enough for us to demonstrate our claims that the architecture is sufficient to support.

As our base architecture we have extended an ARM microprocessor core, described in Section 4.2.1. Although the concepts of the Proteus Architecture could be applied to a more modern and powerful processor (e.g., PowerPC, Alpha, or IA-32 architectures), a simple RISC processor like the ARM makes it much easier to implement a suitable simulation for experimenting upon.

This section describes the details of our implementation, hereafter referred to as the ProteanARM. In Section 4.2.1 we describe the ARM processor core on which the ProteanARM was based, then in Section 4.2.2 we give an overview of how the architecture fits together, looking at how the concepts discussed in the previous section were applied to the prototype architecture. Discussion of the simulator and how the architecture works with some basic test applications follows in Section 4.3.

4.2.1 The ARM7TDMI

The ARM architecture [Furber 1996] was created by the Acorn Computer Group in 1985 for its series of desktop computers. Later on, ARM Ltd. were separated from Acorn to solely develop the ARM architecture. Now it is one of the most widely used embedded processor cores on the market.

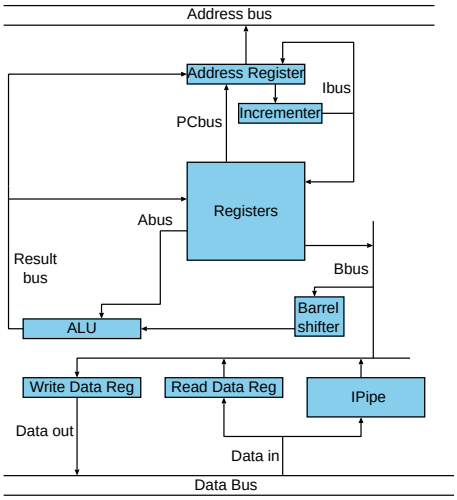

Figure 4.4: Datapath for the ARM 7

The ARM 7 architecture is a 32 bit RISC load/store architecture, an overview of which can be seen in Figure 4.4. The ARM 7 implements version three of the ARM ISA as defined in [ARM 2000], and uses a simple three stage fetch/decode/execute pipeline. The basic ARM core contains sixteen 32 bit general purpose registers, all of which are accessible from any operation, but the top register serves as the program counter. A small number of registers are mirrored in other modes, so that when a mode switch happens (such as a system call or interrupt occurs) not all registers need to be preserved. There is also a status register containing the status bits and current mode information. The register file is connected to a sixteen operation ALU, which contains the basic integer and boolean operations, a multiply unit, and a barrel shifter. All data processing is done only with register contents, with data needing to be moved to registers from memory before it can be processed. An unusual feature of the ARM instruction set is that all instructions are conditionally executed. Each instruction contains a qualifier that will be tested against the status bits: if true then the instruction is executed, otherwise it gets treated as a no-op (one of the qualifies is always true, to ensure that such instructions always get executed).

The ARM architecture uses a simple coprocessor interface to allow both on-chip and off-chip extensions to the core. An ARM system may have up to sixteen coprocessors, numbered from 0 to 15, although coprocessor 15 is reserved for the system management interface. Coprocessors speak to the rest of the system by attaching to the data bus and using a set of handshaking signals between the coprocessor and the main core. Coprocessors operate by observing the same instruction stream as the main core as it goes across the data bus and reacting appropriately when coprocessor instructions occur with the correct coprocessor number in them. When this instruction reaches the execute stage of the pipeline, the coprocessor uses the handshaking signals to let the processor know it is handling the request, and to let the main core know when it has finished. If no coprocessor acknowledges a coprocessor instruction then an undefined instruction interrupt is raised so the operating system may handle the request. The coprocessors all understand the same basic set of instructions, but how a co-processor reacts to instructions is entirely implementation specific. The basic operations are load/store data value from/to memory; transfer data value to/from main core register file, and data processing instructions. How the fields in these instructions are interpreted is up to the coprocessor. For example, a floating point coprocessor may use the spaces for coprocessor register names as an index into an

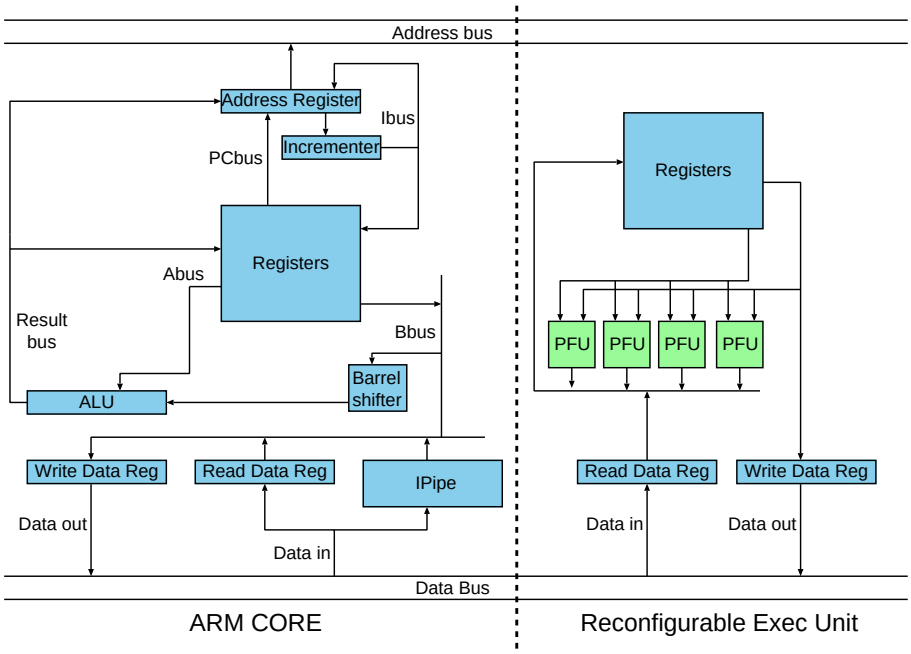

Figure 4.5: Datapath of the ProteanARM core, with a conventional ARM core (left) and a reconfigurable execution unit (right)

internal register file. However a management based coprocessor, like the system management interface in coprocessor 15, may use the same register name slots to pass the data onto the appropriate part of the processor (e.g., cache unit, MMU, etc.). Example uses for the coprocessor interface are the internal floating point unit on the ARM ARM7500FE [ARM 1996] and the Intel XScale [Intel Corporation 2000] for DSP extensions to the instruction set.

4.2.2 ProteanARM Overview

The ProteanARM is constructed by adding a reconfigurable execution unit to an ARM core as an internal coprocessor. The new coprocessor contains the PFUs, the associated register file and special purpose registers, and the dispatch hardware. The idea when designing the ProteanARM was to change as little of the ARM core as possible and to keep the reconfigurable unit as cohesive as possible. This was not completely possible, since some changes to the control logic in the ARM core were required to handle the software dispatch mechanism and the interrupt mechanisms. However, the changes were designed to be generic (such as the ability to allow a coprocessor to generate a branch address) and therefore useful to other coprocessors and not just the Protean coprocessor.

The general datapath for the ProteanARM is shown in Figure 4.5. The register file contains sixteen 32 bit entries (this is what the coprocessor interface allows for), and can have its contents loaded from memory or transferred from the main register file, and vice versa. Additionally, the entire contents of the register file can be transferred to and from main memory in a single atomic instruction, useful for storing and restoring process state during a context switch. As explained in Section 4.1.1, we estimate having ten PFUs available, each of which is connected to two 32 bit input buses and a single 32 bit output bus, which are in turn connected to the register file. Instructions in ARM coprocessors can only work with register contents, unlike instructions in the main core which may also use immediate

operands. Because the reconfigurable unit is in a separate coprocessor, long instructions on the reconfigurable unit could technically run in parallel with the main ARM core. However this ability is not exploited in our initial implementation.

For the reconfigurable logic part of the processor we assumed a fabric similar to that found in the Xilinx Virtex range of devices (see Section 2.1.1), with a suitable modification to allow the contents of the CLB registers to be saved as well as loaded (this is not possible on the Virtex, but can be done on other parts). One technical advantage of modelling the fabric on the Xilinx Virtex fabric is that it uses multiplexor-based routing, which means it is not possible to configure the array with a short circuit. This removes the physical security concerns regarding misconfiguring the ProteanARM.

Based on the figure of 500 CLBs per PFU, we can make an estimate of the bitstream size required to configure each PFU. The Virtex fabric bitstream uses a series of frames to configure the array [Xilinx 2000]. Each column of CLBs is configured with a series of 48 frames which may be configured independently. Each frame contains 18 bits per CLB, which amounts to 864 bits of configuration information per CLB. This gives a total of 432,000 bits per PFU, or 52.7 Kbytes per PFU. This gives us an upper bound on the amount of information needed to configure a PFU; smaller circuits may require less configuration information. Also of interest is the size of the bitstream that contains just the state information for each PFU; this gives the upper bound for the amount of data needed to be moved off the processor to store process-specific state. There are four registers per CLB and the data for each one is stored in a separate frame, meaning that for each CLB we need to store 72 bits, or 36000 bits per PFU (4.4 Kbytes). This gives an upper bound on the amount of information we need to store on a circuit switch to ensure that the circuit can be successfully reinstated. Although not an architectural requirement, the bitstreams used to configure the PFUs are divided into two sets of frames: those containing state and those without. This gives two separate configuration streams, which can be treated contiguously for loading purposes, and separately for storing purposes.

In addition to just a simple ARM core, we have modelled two on chip peripherals to support operating system management. The first is an interrupt controller for managing the two types of interrupt the ARM processor can handle; the second is a set of timers which the operating system can program to produce interrupts after specific periods. These peripherals are required to support a preemptive multitasking environment.

The ARM core we selected runs at 40 MHz. This is slow by modern processor standards, but our interest lies with the management issues, rather than the actual detailed implementation issues of making the system work at today's high clock speeds. Given that we assume a Xilinx Virtex device with a ARM core added, and the Virtex uses a much better silicon process than that associated with the ARM core we use, in all likelyhood the core could be clocked faster. However, the expertise required to make an accurate prediction of how the clock speed would increase was not available, so no attempt was made to persue this. The main aim of the architecture is to explore the basic interface; the detailed electronics are left for people better educated in such matters.

4.2.3 PFU Interface

The interface to each PFU follows the design laid out in Section 4.1.3. Each PFU has two 32 bit inputs being fed by the unit's register file, and a 32 bit result bus that connects back to the register file. The PFUs also have a start signal entering them that goes high on the first cycle of invocation and subsequently low, and an output signal which indicates completion.

On the ProteanARM, an extra cycle had to be inserted at the start of a custom instruction dispatch, in order to allow the status bit to be read and fed into the custom instruction. This cost may seem negligible, but several of our test applications use custom instructions which complete in a single cy-

cle. It could be beneficial to have two dispatch instructions: a single cycle dispatch where the custom instruction is guaranteed to complete in one cycle, and our existing multicycle dispatch instruction.

As stated in Section 4.1.3, the Proteus Architecture has a initialisation line ito a custom instruction which is high on the first cycle and low on subsequent cycles. On the output side, the circuit drives a done signal low during execution and high when it is complete. To allow instructions to be restarted after preemption by an interrupt, we record the done signal in the extra status bit, the value of which is fed into the initialisation line when the program issues a custom instruction. If this is a first issue of that instruction, then the status bit will be high from the last completed instruction's done signal. If the instruction had been preempted however, then the signal will be low, and the circuit will not receive the initialisation signal when reissued. Thus custom instructions can be interrupted at any point and reissued without restarting them.

4.2.3.1 Reconfiguring

In Section 4.1.6.1 we looked at two ways in which PFUs may be configured: memory mapping the SRAM behind the PFUs or by a sequential loading mechanism. Because the ProteanARM PFUs are modelled on the Xilinx Virtex FPL, we assume a similar loading mechanism as the Virtex, which uses a serialised data input, thus go for the latter option.

The top configuration speed of the Virtex device is lower than that of the ARM's memory bus, which is both wider and faster than the fastest Virtex configuration interface, so we model an instruction FIFO, into which the configuration data for a single PFU can be loaded at the full ARM memory bandwidth rate, which is then drained at a slower rate as the data is shifted into the configuration SRAM. The processor core is only active in the configuration process whilst the FIFO is being loaded; the draining of the data into the FPL can be done in parallel with other processor activity, so long as no attempt is made to either load another configuration bitstream or invoke the PFU being configured. Should either of these operations be attempted then the coprocessor will not acknowledge the instruction at run time, which will cause an undefined instruction trap to occur. This mechanism means that whilst we can not reduce the latency from the start of an instruction being loaded to it becoming available for use, other work may usefully be done for at least part that time.

When an application requires that an instruction is loaded, and the operating system will begin the loading operation, the requesting application will be suspended until the loading has completed. To be able to do this the operating system needs to be made aware of when an instruction has competed. This is achieved by the coprocessor causing an interrupt to occur through the on chip interrupt controller to alert the operating system that the loading has completed.

The core and coprocessor work together to carry out the loading of the FIFO. This involves adding a CISC style load instruction to the RISC processor ARM core. This instruction lasts as many cycles as it takes to transfer the data to or from the processor over the memory bus. During loading the instruction may be interrupted to allow the operating system to respond to other interrupts during loading. The load and store instructions modify the regsiter containing the bitstream address such that if the instruction is interrupted the instruction can simply be reissued and it will continue from the point at which it left off.

Storing the contents of an FPL device is more complicated, but handled similarly. Unlike loading, where a single instruction is sufficient to start the process, we need to fill the FIFO with data from a PFU and then do the storage routine, which requires two instructions, one to fill the FIFO and another to transfer the contents of the FIFO to memory, and an interrupt to indicate when the FIFO loading has completed.

4.2.4 Dispatch Mechanism

The ProteanARM uses two eight entry TLBs for implementing the conversion from the (PID, CID) ID tuples to either the PFU number or software pointer. Each ID tuple is 15 bits long: 8 bits come from the PID (the ARM system coprocessor only allows 8 bits for PIDs) and 7 bits for the CID (as much space as there is for opcodes in the execute instruction format). These are used at the decode stage of the pipeline when an instruction for executing a custom instruction is encountered. When an processor instruction requesting a custom instruction be executed is encountered, the CID from the instruction and the current PID, which is held in system management coprocessor, are fed into each TLB. Depending on the results the instruction is decoded in one of three ways: as an execute PFU instruction, a branch to software alternative, or a unloaded instruction trap.

The first case is very simple. The hardware TLB will return the corresponding PFU number for a given ID tuple which will be used to indicate which PFU should be used when this instruction is executed. Once translation has occurred the instruction appears just like a conventional coprocessor data processing instruction. The more interesting cases are what happens when a match is found in the software TLB and when no match is found.

As discussed in Section 4.1.5, dispatching to software is a more complex operation due to the need to preserve the instruction operands and the extra instructions required to retrieve them, and unlike most other parts of the ProteanARM, to solve this problem we required to modify the behaviour of the main ARM core slightly. When a match has been found in the software TLB, the instruction needs to cause the program counter to be stored safely, for the instruction parameters to be preserved for later use, and for execution to be jumped to the appropriate address. Storing the instruction operands on the ProteanARM is simplified thanks to the simple load/store nature of the ARM architecture, whereby all processing is done on data stored in registers, so the Protean coprocessor only needs to note the two source registers and the result register, which is only 12 bits and can be stored in a single register. This register can be read from and written to using a register transfer instruction so that it may be preserved across context switches.

Having preserved the operands in the coprocessor, the rest of the instruction resembles a traditional ARM branch and link only with the destination address coming from a coprocessor rather than the main core. Because the operand preserving is part of the coprocessor, the main core never needs to be aware of it. However the main core's control logic does need to be modified to handle the actual branching, as the coprocessor needs to modify the value of the the program counter held in the main core. The task is complicated slightly as the main core does not know at the decode stage whether the instruction has been decoded to run in hardware or software, so can not explicitly react to the instruction be preparing for an address to be generated from the coprocessor. Instead we augment the coprocessor interface with a signal that causes the main core to start a branch based on the value being put on the data bus by the coprocessor at that time.

The new control logic takes several cycles to complete once the core has been notified of the software dispatch. On the first cycle the address to be branched to is latched on the data output of the coprocessor. On the second cycle this will be moved into the main cores data in register, and on the third cycle it will reach the address register having moved through the ALU. Now the address has reached the address register it will take another two cycles to get the pipeline ready to execute the correct instruction. During these two cycles the old program counter value is moved to the link register and adjusted to point to the instruction that follows the instruction that caused the software dispatch. On the next cycle execution will continue from the software function.

As noted, this mechanism has the side effect or destroying the previous value of the link register unexpectedly; the ARM's ABI, the ARM Procedure Call Standard (APCS), only requires the link

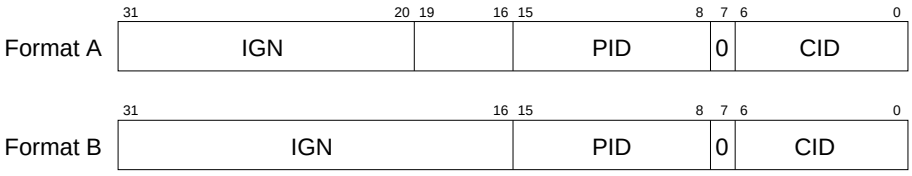

Figure 4.6: Data formats for TLB entries

register to be preserved if a procedure contains a procedure call, but here we are making an unexpected procedure call so the link register may not have been preserved. However for simplicity we simply augment the APCS to include the prevision that procedures that use custom instructions must preserve the link register on the stack too. Using the link register means that we do not need to add another register to the main core and that the return from the software alternative can occur in the same manner as the return from a standard procedure call. Once the software dispatch routine has been entered, the software can then make use of three normal register transfer instructions which have a special meaning in the Protean coprocessor. By setting the appropriate opcode in the core/coprocessor register transfer instructions the index for the target/destination registers in the coprocessor are taken from the register storing the operands of the faulting instruction rather than from the transfer instruction's operands.

When no match is found for a ID tuple in either TLB then the coprocessor handles the instruction as if it did not recognise the instruction, which causes an undefined instruction trap to occur on the processor. This causes the operating system to be entered, which can then locate the faulting instruction and obtain the CID the process was trying to invoke. It can then either rectify the issue and restart the process from the faulting instruction if the CID was valid, or terminate the process if the CID was invalid. Programming the dispatch TLBs to include the missing CID is done by writing to two control registers, one for the PFU dispatch TLB and one for the software dispatch TLB. Each register supports for opcodes, describing whether an access to the register is to add a TLB entry, remove TLB entry, lock a TLB entry, or flush the entire TLB. When describing a TLB entry, the operating system provides data in the formats shown in Figure 4.6: Format A is used when adding a new entry, and Format B is used for describing existing entries.

4.2.5 ProteanARM Instruction Summary

The new instructions defined for the ProteanARM can be seen in Figure 4.7. These fit into the standard instruction formats for interfacing with coprocessors on the ARM. This means that the main part of the core knows how to respond to each instruction essentially for free. However, to achieve this we did need to use two coprocessor numbers to cover the range of instructions. Given that this was required, it made sense to separate out the two coprocessors logically, and use one for user level instructions and the other for system level instructions used only by the operating system. Thus coprocessor 0 is used by applications when using the reconfigurable unit, and coprocessor 1 is used by the operating system. This can be seen in the instructions by bits 8 through 11. Only the first three instructions can be executed by user level processes. This provides another level of security in the system: only the trusted operating system can manipulate the configuration of the PFUs and the dispatch hardware, preventing applications accessing each others' resources without permission.

The execute instruction is simple enough, containing a 7 bit CID and the three operands for the custom instruction. Similarly the load/store instruction is simple to understand in comparison with the ARM guidelines, the only interesting part being the L bit, which specifies whether a single register or the entire contents of the register file is being transfered. The register transfer instruction is used to

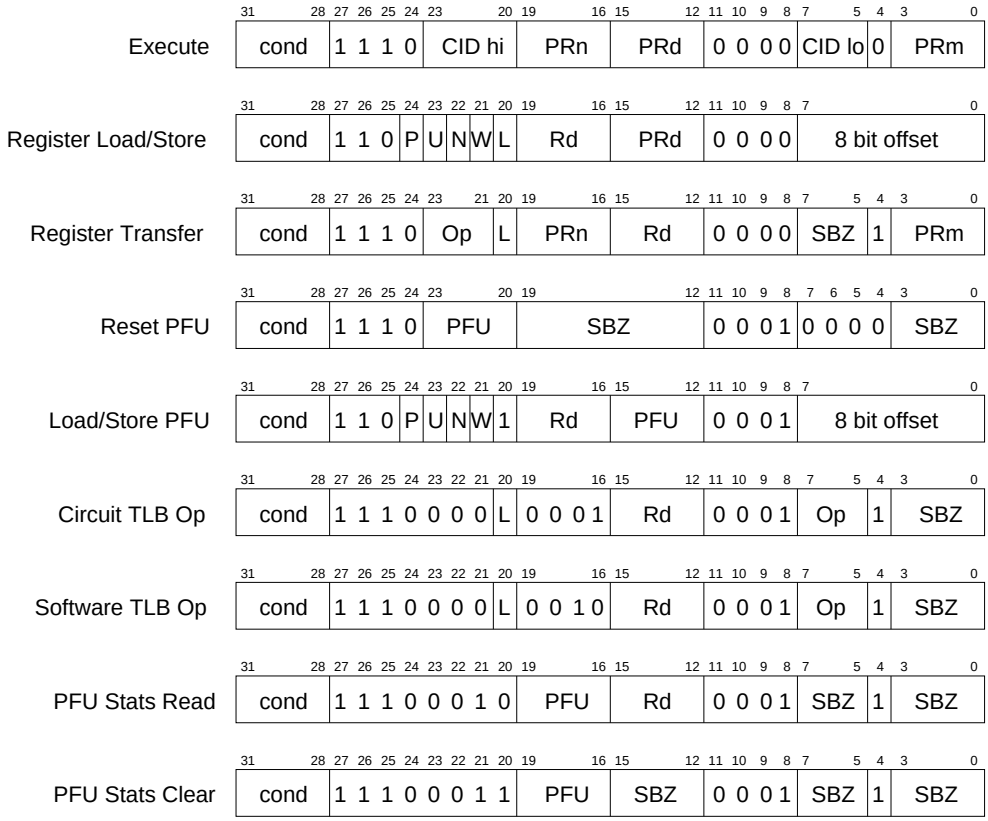

Figure 4.7: Instruction formats for ProteanARM ISA extensions

move data between the main register file and the register file in the reconfigurable unit, the direction being indicated by the L bit. The Op field specifies whether this transfer is between the register file in the unit, or between the implicit operand registers for use by a software alternative.

The remainder of the operations are used by the operating system for controlling the reconfigurable execution unit. The reset instruction resets a specified PFU, returning all registers to their default value as specified in the configuration (see Section 5.1 for a discussion of how this instruction may be used). Load/store PFU is used to transfer the bitstream for a circuit and its state onto the processor or the state off the processor at the specified memory address. The circuit and software TLB operation instructions are used to program the respective parts of the dispatch hardware. The final two instructions allow the operating system access to the usage statistics associated with each PFU.

4.3 Architecture Simulation

In order to develop and test the operating system and management levels in which we are interested we first developed a basic simulator of the ProteanARM architecture with which to experiment. In this section we describe the simulator we used to test the ProteanARM, some basic sample applications designed to run on the simulator, and the basic performance characteristics we observed.

4.3.1 The SWARM Simulator

To test the ProteanARM architecture and allow us to demonstrate the practicability of the operating system and programming model work, we needed a processor simulator for the ProteanARM architecture on which to build. The aim of the simulator is to allow us to demonstrate that logically the ProteanARM can be created, and that the model is sufficient to support operating system management. We will want to allow it to detail the changes made to the datapath and the way in which it is controlled. This it needs to be accurate down to the data flow level, to allow us to see how the various buses in the system will work, and what control logic changes are required.

When simulating a new processor architecture there are several options available. Firstly a hardware model can be build and then either simulated using hardware simulation software or synthesised and loaded onto an FPGA [Sawitzki et al. 2001]. However, building such a detailed model requires considerable effort, and is only of interest if we are trying to prove the specifics about how the FPL and processor work together, demonstrating that the two worlds can be physically connected together.

Instead a software model was used to model the processor. This technique has been used in other projects to provide an easy way to test architecture changes and seeing how they affect the higher layers. An example is SimOS [Rosenblum et al. 1995], which simulates a complete machine in software. SimOS provides a complete machine implementation of a MIPS based machine and a Alpha based machine, complete enough to run Silicon Graphics Inc.'s UNIX, IRIX on the MIPS implementation and Digital UNIX on the Alpha implementation. Because the system is build in software it is very easy for the authors to alter variables (e.g., cache size/types, TLB sizes, etc.), and very easy to monitor events within the system (e.g., disk accesses, cache misses, etc.). We want a similar arrangement, but aimed at the ProteanARM.

To provide a suitable model for our work we created a modular software mode of an ARM7 CPU code in C++, called SWARM (SoftWare ARM). SWARM is a cycle accurate model of the ARM 7 core, designed at the data-flow level, modelling the transfer of data around the CPU as it moves between registers, buses, and the various processing elements. SWARM is built in a modular fashion, allowing people to add new basic blocks to the system, such as different caches, peripheral devices,

and internal coprocessors. For the purposes of our work we are simply interested in a basic core with a new internal coprocessor attached to a block of memory, but other groups have added devices like UARTS and LCD controllers. SWARM is sufficiently complete a implementation of an ARM processor that microcontroller Linux was ported by another group to run on SWARM. For purposes of the initial testing no memory management unit has been implemented.

In a real ProteanARM PFUs contain an FPL fabric that is programmed by loading a bitstream which would have been produced by a hardware compiler tool. A detailed simulation of the PFUs is unnecessary for the purposes of studying the management of the PFUs themselves; all that we need to do is ensure that we can simulate the functional behaviour and loading times accurately. The PFUs are simulated functionally rather than attempting to provide an emulation of the actual fabric used. Programs destined to run on the simulated are linked with binary objects that are the same size as the estimates bitstream size for a PFU, but instead contain a software function compiled for the architecture that the SWARM simulator is compiled for and meta data describing the length of the software instruction and the amount of data needed to store any state used by the function. SWARM notes the bitstream being loaded onto the processor and stores the code segment for execution when a PFU invocation is simulated. Because this is only a functional simulation of the circuit loaded into the PFU, the actual circuit characteristics such as CLB count and timings have to be acquired by designing the circuit for a Virtex part using the Xilinx design tools.

Software can easily be developed for the simulator using the GNU c compiler, gcc, built to cross compile for the ARM in conjunction with a small custom C library and run time. Because we did not wish to spend time adding I/O devices and storage devices to the emulator, the simulator was designed to handle a special range of software interrupts (used on the ARM to implement system calls) as upcalls to the host system, which allows us to provide the facility to access the host file system for instance.

As with all simulation work, performance becomes a concern for anything beyond the basic tests. Despite investing time in optimising the performance of the simulator, on a 1 GHz Pentium III a second of simulated CPU activity can take several hours. This meant that for tests in the operating system experiments discussed in Chapter 5, which required 50 or more tests to generate a single graph, the run time became a limiting factor in what work could be practically tested.

4.3.2 Basic Performance

As part of the development of the general architecture we developed a small set of applications to run on the ProteanARM simulator. The purpose of these was to ensure that the general architecture worked and to allow us to get a feel for how it performed and developed, the results of which were used to tune the interface. Here we describe these basic applications and the estimated performance benefit from running on such an architecture.

In all the cases here the circuits used for custom instructions were developed in structural VHDL targeting the Xilinx Virtex part. The VHDL was then compiled using Xilinx Foundation software and then placed and routed using Xilinx Alliance to generate a timing figure and CLB count. The circuits were designed to run at 40 MHz, the maximum clock speed of the ARM core we selected.

Typically, to test the performance of a new processor design a benchmarking suite such as SPEC [SPEC 2000] is used. Such a benchmark is used to approximate behaviour over a wide range of user tasks. We did not follow this approach, instead using a limited number of bespoke examples. There are two reasons for this. Firstly, without implementing a compiler, taking large applications and accelerating them with custom instructions without a working knowledge of the algorithms used is a hard problem. To manually accelerate programs requires knowing where the program spends most of its time, and

then understanding how the algorithm works at an abstract level so that the best approach to acceleration can be selected. Thus implementing SPEC this early on in the project was deemed an unuseful application of manpower. The second reason is that, while the basic architecture should be demonstrably faster than an unaccelerated architecture, the focus of this work is on the management issues rather than the straight line performances. So while it is important to have a set of test applications, we are more interested in developing the operating system mechanisms rather than an exhaustive set of test applications.

4.3.2.1 Audio Processing

The first example is an echo filter algorithm taken from an Intel MMX technical note [Intel Corporation 2001*b*]. The origianl aim was to show that the ProteanARM could support MMX type instructions in its PFUs. But, thanks to the flexibility of reconfigurable logic, we were also able to perform optimisations not possible in MMX.

The algorithm processes a WAVE-format audio file, which uses 8 bits per sample at an 8 kHz sampling rate. The samples are stored as unsigned values, but must be normalised to be between -128 and 127 for processing. This is done by xoring each sample with 80_{16}. To add e echos of delay d to sample s we look back in the file's history and add attenuated values. The attenuation factor, G, is greater the further back in time we go. This is expressed as follows:

$$s'[n] = s[n] + \sum_{1 \leq i \leq e} (G^i \times s[n - i \times d])$$

The aim of the implementation is to use SIMD techniques to work on four octets per instruction. The key operations of the MMX example are a SIMD signed add, a variable SIMD arithmetic shift left, and an xor operation to normalise the samples. We improve on this by building the normalisation into the add and shift instructions, by partially evaluating the xor with constant operation into the circuits' inputs and outputs where necessary. This level of customisation was only possible with reconfigurable logic. Had this not been possible then we would have been forced to either use the xor instruction in the main ALU, which would have meant moving data back and forth between the main ARM core and the reconfigurable execution unit, or to use a PFU to provide a xor function in the reconfigurable execution unit, which would have been a waste of a PFU, as it would have only occupied a small fraction of the PFU, especially if we partially evaluated the constant into the circuit.

In the completed example, the shift instruction required 75 CLBs and the adder instruction 20 CLBs. Both take less than a single cycle to complete at 40 MHz, but could easily be pipelined to run faster given how little of our predicted PFU size they use. Using the new instructions reduces processing a single sample from 570 cycles (in optimised C code) to 98 cycles.

4.3.2.2 Alpha Blending

Alpha blending is the process of superimposing images that contain a level of transparency (their alpha value). Images are made up using 32 bits per pixel, an octet for each of the Red, Green, Blue, and Alpha channels (referred to as RGBA format). Without the alpha channel, summing images fits nicely into the SIMD add with saturation instructions found in MMX. But summing RGBA is more complex. The equation for summing the alpha channel is different to that for summing the colour channels, and summing the colour channels depends on the result of summing the alpha channel. The equations for the alpha and colour addition can be seen below, with f and b referring to the front and back images:

$$A_n = \frac{A_f \times A_b}{255} \qquad C_n = \frac{((255 - A_f) \times C_f) + (\frac{A_f \times C_b \times (255 - A_b)}{255})}{255 - A_n}$$

The aim of this example is to experiment with a circuit that adds two pixels in a single instruction, fitting nicely into the 32 bit datapath of the ProteanARM. The alpha blending circuit is both larger and more complicated than the circuits used in the previous example. It implements a number of Booth's multipliers and naive dividers, requiring the circuit to include state and take multiple cycles to complete.

The alpha blending instruction uses 436 CLBs and takes 27 cycles to generate a result at 40 MHz, substantially less than the corresponding 377 cycles required by software. This circuit only just made the timing requirements at 40 MHz, so would require pipelining to be implemented for a higher clock speed. We feel that this represents a reasonable level of circuit complexity for what would go into a PFU, unlike the audio echo instructions are very basic combinatorial instructions. As such we feel that the CLB count provided by this provides an indicator of how large we expect PFUs to be. However, this is obviously just an estimate, and many more applications would have to be developed to get a more accurate feel for the PFU size.

4.3.2.3 Twofish Encryption

The next example implements the core part of the Twofish encryption algorithm [Schneier et al. 1998], which is used in programs such as Secure Shell [Ylonen et al. 2000]. Twofish is a block based secret key encryption scheme. It works on 128 bit blocks of data and uses either a 128, 192, or 256 bit key. Encrypting and decrypting in Twofish uses a similar algorithm, simply with the order of the stages involved reversed. For the purposes of this discussion we refer to encryption throughout, but everything we discuss applies to both the encryption and decryption processes. There are two stages to using Twofish: key scheduling, which sets up a session for processing data, and encryption which is when the actual data processing occurs. Spending more time on the key scheduling means less time is spent at the encrypt stage and vice versa. The four standard implementations as outlined in [Schneier et al. 1998] were implemented in C and measure without acceleration first. The four implementations are referred to by the amount of key processing they do, with zero doing the minimal key processing and full essentially turning encryption into a series of LUT accesses. The following table, Table 4.1, shows the number of clock cycles for each stage for each of the key lengths.

Keying Option	Clocks to Key			Clocks to Encrypt		
	128	192	256	128	192	256
Full	46461	56736	67603	2330	2330	2330
Partial	38772	49047	59914	3868	3868	3868
Minimal	30578	40853	51720	4353	4353	4353
Zero	7757	9840	12003	5436	6652	7932

Table 4.1: Twofish performance with different key lengths and options

The encryption and decryption paths are very similar, and at their core have a manipulation function, referred to as the h function, that takes in 32 bits of the data to be encrypted, X, and modifies it to produce another 32 bit value, Z. A diagram of the h box can be seen in Figure 4.8. The data value enters the h function, and is processed through a series of q boxes, which are static 8 bit permutation

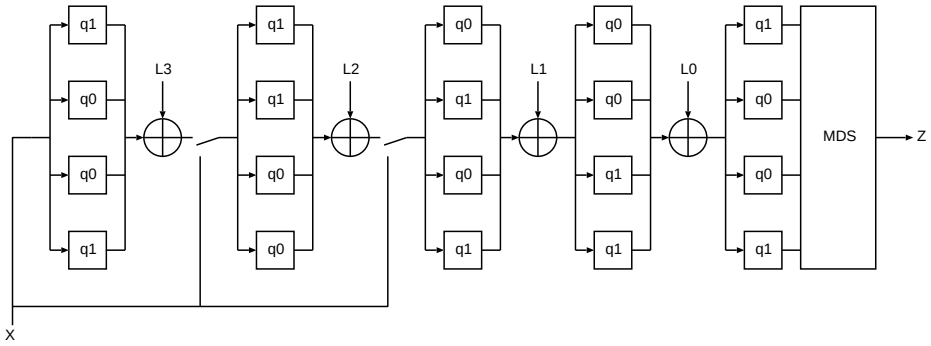

Figure 4.8: Twofish h function

boxes, and is XORed with L values, which are derived from the session key. The path through the h function is dependant of the key length, with a shorter path being taken for a shorter key (represented by the switches in the diagram). Because the q boxes are static, and the data values match the data width of the processor bus, the h function makes an obvious choice for a custom instruction, but for the key dependant L values, which total as the same number of bits as the session key. There is no way to fit this L value across the datapath for a single invocation of the h box. For instance if we have a 128 bit key, then we need to pass in 160 bits per invocation, but we only have 64 bits available on the ProteanARM. A possible solution would be to write a circuit that takes the L value in over consecutive cycles. This would require the circuit to be invoked three times (twice to read in the 128 bit L value, and once more for the 32 bit data value). The circuit would then only produce a meaningful result every third invocation. Whilst this solution is workable[2], we waste two cycles every invocation to get around the bandwidth limitation of the processor. Obviously we could expand the datapath width to 128 bits, but then other key sizes would still require several load cycles.

A possible solution for a certain class of applications like this one is to use *partial evaluation*. Partial evaluation is similar to constant folding in software compliation [Aho et al. 1986], whereby expressions whose value is known to be constant at compile time are collapsed and replaced with that constant, only applied to hardware. The flexibility of reconfigurable logic allows us to decide before loading a circuit that one of its inputs is constant and fold that into the circuit before loading it, the aim to be to improve the runtime performance of the circuit [Susanto & Melham 2000]. Although the authors in [Susanto & Melham 2000] found that it provided little benefit for runtime performance, partial evaluation can be applied to the PFU bandwidth problem. In the Twofish algorithm, the L value used in the h function is constant for a session, which means that for all PFU invocations for that session, of the 160 bits needed (assuming our 128 bit key size), only 32 bits (the data being encrypted) change between invocations and the other 128 bits (the key) remain constant. Thus, in the key setup stage, the L value can be partially evaluated into the circuit, reducing the input requirements to just the 32 bits of data to be processed, at the additional cost of having to write the L value into the bitstream before loading. Typically generating hardware bitstreams is a slow process due to the placement and routing process which can potentially take many hours, which would be an unacceptable cost for most applications (like secure shell). However, in this case only a fixed sized constant is being modified in the bitstream, so there is no need to modify the layout of the circuit. Indeed, because just a constant is being modified in the bitstream, this particular application would require very little modification to

[2]This solution would not work with the programming model developed in Chapter 5 however, as we do not allow state to persist between invocations (see Section 5.1).

the bitstream, and thus the process would be computationally inexpensive.

This technique of course requires that the processor vendor either makes public the bitstream format, or provides tools for generating new bitstreams in such a fashion. Some FPGA vendors do not like to reveal the exact bitstream format for their devices, and some hardware compilers may attempt partial evaluation on the base circuit when it is first compiled as part of other optimisations [James-Roxby & Guccione 2001]. Xilinx, who do not publish the bitstream format for their FPGA ranges do produce a suitable tool, developer for generating run time paramertizable cores, called JBits [Guccione & Levi 1998]. JBits takes a compiled hardware description and turns it into an intermeditate format, which can then be parameterized before finally being turned into a bitstream. Because JBits works over already placed and routed hardware descriptions this process is very quick compare to traditional hardware compilation techniques.

Assuming that the Twofish core is partially evaluated, then we can successfully implement the core function of Twofish in a two cycle PFU circuit (two cycles were required to meet the timing requirements) taking 143 CLBs.

Processor	Keying Option	Clocks to Key	Clocks to Encrypt
ARM7M	Full	46461	2330
ARM7M	Zero	7757	5436
ProteanARM	Zero	9265	816

Table 4.2: Twofish performance with different key lengths and options

The number of cycles taken to do the key setup is currently misleading, as it doesn't include the circuit configuration overhead, however, we do not envisage the partial evaluation process in this case to require more than a series of stores of the L value in the appropriate places in the bitstream, and thus should add significantly to the 9265 cycles quoted for the setup time. At worst it will require 128 individual words to be read, modified by a single bit, and written back to memory.

4.3.2.4 Intercal

In the examples discussed so far, all the applications use custom instructions to speed up a specific part of an algorithm, but this is not the only model for custom instruction usage. One alternative is for the programmer to use a domain specific language which has custom operators designed to suit that particular application. Those operators that are not natively supported by the processor could be implemented as custom instructions.

Intercal [Woods et al. 1996] is a language designed in 1972 for the purpose of being obfuscated and confusing to use. Despite it not being particularly useful, Intercal does make an easy to modify example of a language which uses non-native operators. Intercal uses five operators, none of which are supported on any real processor. In this example we modified the compiler to output code that would use custom instructions rather than function calls to software implementations of the operators.

The five Intercal operators are unary AND, unary OR, unary XOR, mingle, and select. The first three operators take in a single 32 bit or 16 bit value and generate each bit n in the result by applying the appropriate logical operation to bits n and $n+1$ in the operand. The fourth operator, mingle, takes two 16 bit values a and b and generates a 32 bit result with the two numbers interleaved, such that the result is $a0b0a1b1a2b2$.... Finally, select takes two 32 bit values a and b and returns a result that is the logical OR of the two values with all the ones moved to the least significant places. Although this is not the most obvious set of operators it is sufficient to carry out computation.

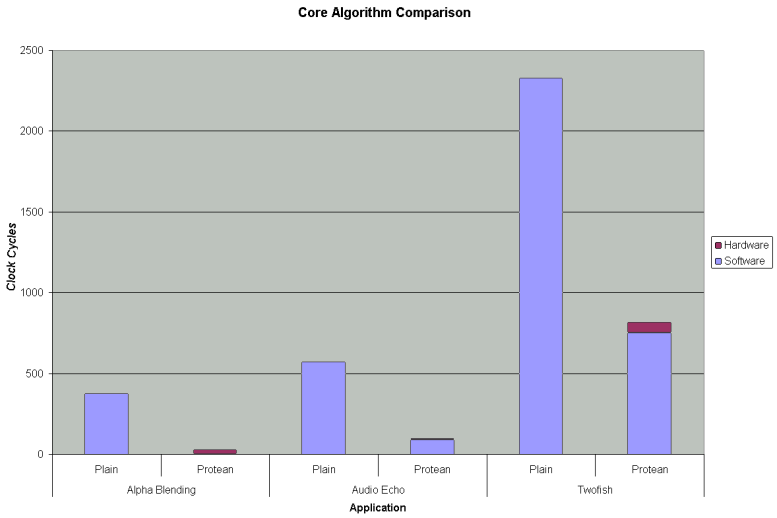

Figure 4.9: Performance of core algorithms for test applications for both accelerated and unaccelerated implementations

Using Intercal we were interested in the behaviour on the processor when using a compiled Intercal program. This example will have a much denser set of instruction invocations and rely more heavily on the reconfigurable execution unit. The five Intercal instructions could be implemented as four single cycle instructions and one sixteen cycles instructions (the circuits for this example where not implemented, just designed given the focus of this test was not raw performance monitoring). The Intercal compiler was then altered to emit custom instruction invocations in place of function calls to the software implementation of the library. Based on the experience of hand compiling a small intercal program, a register allocation policy was also decided.

The register allocation policy is interesting. In most the previous examples the register file associated with the reconfigurable execution unit was predictably under used; applications just pulled in the input to the registers as needed and saved them back out, in effect treating the register file as a small buffer before feeding data into the PFU(s) they used. However, the Intercal example uses the reconfigurable unit more heavily, but still did not begin to stretch the limits of the register file, typically using less than half the available registers. As a result the compiler was modified further to do two passes of the source code, working out the maximum register usage and the most frequently used constant operands in the first pass. It then use the most frequently used constants in the registers that would not be used.

Discuss performance estimate for prime numbers.

4.3.2.5 Discussion

All the basic applications shown above proved relatively easy to implement and provided a basic performance gain. A summary of the benefit for each algorithm can be seen in Figure 4.9 which shows the unaccelerated and accelerated cycle counts for processing one unit of data for the given algorithm. Furthermore, the accelerated value is broken down by the number of cycles spent in software and the number of cycles spent in hardware. In all cases, for our given test platform, we can see a significant reduction in the core algorithm speed for each of the test applications, with a large amount of work in each case being moved from software to hardware.

TODO: Discuss reconfiguration overheads.

4.3.3 Memory Interaction

A side benefit of moving part of an algorithm onto the processor is that it reduces the number of memory interactions required whilst the algorithm runs as less instructions are required and less temporary storage locations are required (obviously there is a high cost associated with moving custom instruction on and off the process, but here we consider just the execution costs). This is especially true of certain applications that have poor locality properties. The Twofish encryption algorithm described in Section 4.3.2.3 suffers noticeably from this problem. When developing the various software based implementations, in an attempt to increase the performance of the algorithm functions have been inlined and loops unrolled. This meant that by the time one block had been processed the code for the algorithm was no longer in the instruction cache. Secondly the algorithm uses large look up tables which it accesses in a hard to predict fashion (this is part of encryption after all), so these tables exhibit poor locality of data and do not remain in the cache for long.

We ran the four standard implementations of the Twofish code, each of which exhibits a different level of key pre-processing, from no pre-processing, to full pre-processing. The pre-processing involves taking the session key and pre-processing it into LUTs. To examine the memory behaviour of this application we tested each implementation at encoding and decoding ten blocks of data with a 128 bit key. We used the SWARM simulator with the default 8 KByte 4-way set associative cache. The results can be seen in Figure 4.10, with green dots representing a cache miss and red dots representing a cache hit.

In the graphs the ten encrypts and the ten decrypts can be clearly seen as the lower and upper set of slanting lines in the lower half of each graph. The line of addresses hit close to the x axis just before the first encrypt round is the key setup processes.

From this we can see that both the minimal and partial optimisations suffer from very poor cache performance, with the performance of the full keying implementation suffering too. The main cause of this is that operations that were previously done on word units mainly on the processor have been turned into many more byte wise memory accesses. In addition, in order to reduce the cycle count, the techniques of loop unrolling and function inlining are likely to have some contribution too.

The first point to take from this data is that concentrating on cycle counts alone is not a good metric for how well a algorithm performs; we also need to examine what type of cycles the algorithm is using. But more importantly, the use of LUTs to speed up algorithms is not necessarily a silver bullet to reduce the number of cycles an algorithm takes. Thus, for certain cases it is possible that using custom instructions to generate data could out perform a LUT based solution, despite the custom instruction having to do more calculating at data processing time. To show this the memory trace for a zero keying implementation using custom instructions can be seen in Figure 4.11. Although the key setup time is misleading (the cost for generating and loading the bitstream is not shown), the time for processing blocks is notably less than it is in all the LUT implementations, due to the better memory behaviour.

This characteristic obviously only applies to a limited class of applications; for example, the alpha blending and audio processing examples do not suffer this for of performance penalty. However, it is an interesting side effect of moving work onto the processor not noted in any of the previous research literature.

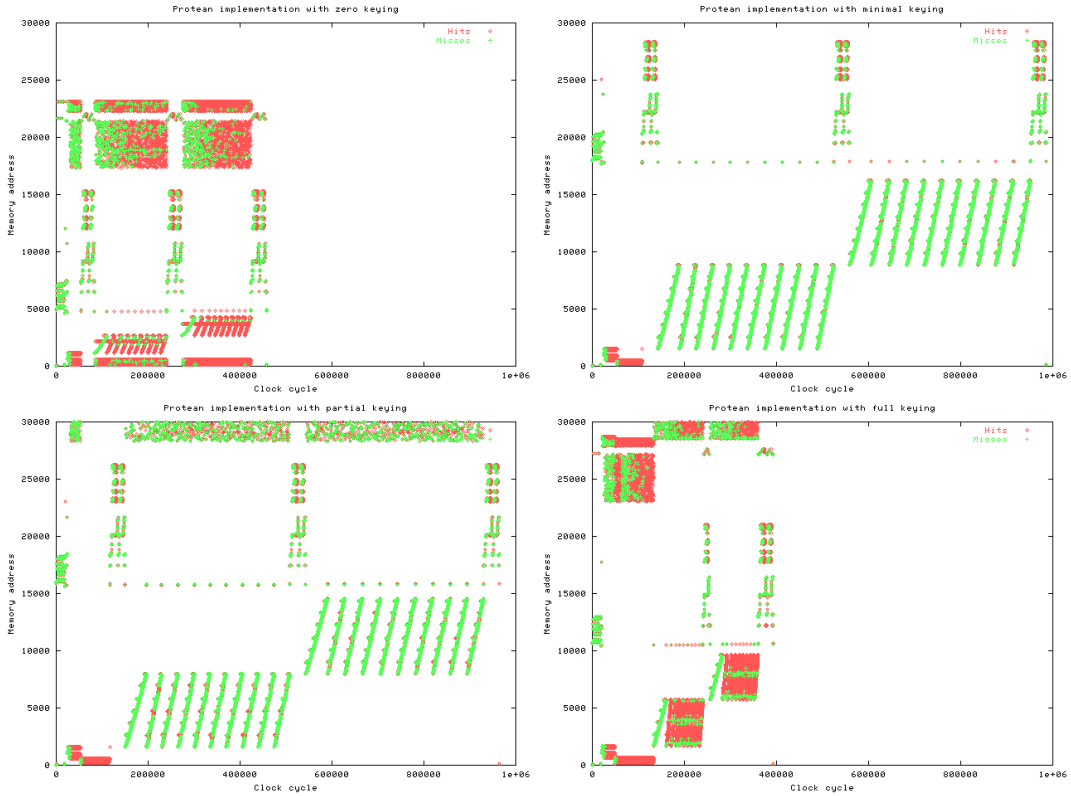

Figure 4.10: Memory traces for the Twofish implementations left to right, top to bottom, a) zero, b) minimal, c) partial, and d) full

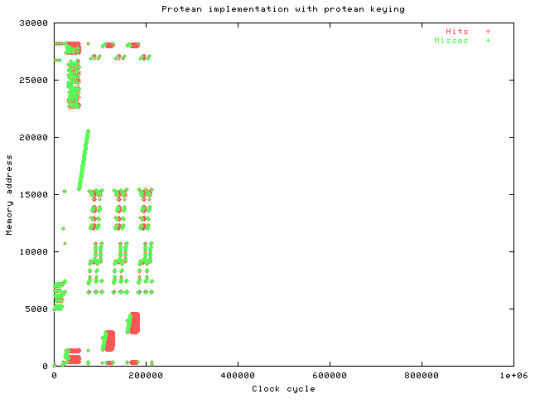

Figure 4.11: Memory trace for the protean Twofish implementation with zero keying

4.4 Summary

In this chapter we have layed out a basic architecture style that is practicable for supporting loading custom instructions for applications in a workstation style environment. It provides a machanism for loading multiple custom instructions on the processor at once, and to allow the operating system to virtualise the reconfigurable resource. The architecture also provides support for allowing instructions to be dynamically dispatched to either hardware of software without the application needing to be aware, and supports usage statistics for aid the operating system in selecting PFUs for eviction. Based on the architecture proposed we presented a basic implementation based on an ARM core, called the ProteanARM, which provides an incite into some of the partical problems in implementing such a system.

Having devised the basic architecture, we can now move on to examining the operating system issues in managing such a processor.

Chapter 5

Operating System Management

Having described the Proteus Architecture and a base implementation, the ProteanARM, in the previous chapter, this chapter and the next describe an operating system kernel and programming model that demonstrate that applications and application developers can easily take advantage of the reconfigurable resource.

Before examining the operating system aspects of managing a reconfigurable processor, this chapter starts with a look at what constitutes a custom instruction. Existing literature assumes a custom instruction to be just the series of bits that describe the configuration data for a block of FPL; we consider a custom instruction to be a much richer entity consisting of many parts, to which we apply principles from the field of programming languages. Having defined what we consider a custom instruction to be, we then look at how this entity is managed both in the operating system and in programming models.

The second section of this chapter examines the operating system interface and facilities used to manage the FPL resource on the processor. This focuses mainly on the Custom Instruction Scheduler (CIS) which is used to decide when and where to load custom instructions as they are used by user level processes. We introduce a small kernel that was developed containing a CIS, called POrSCHE (Proteus Operating System and Configurable Hardware Environment). POrSCHE is a simple preemptive multitasking kernel that has been extended to manage a reconfigurable execution unit using a variety of scheduling algorithms. Although the work here is discussed mainly in the context of the the ProteanARM, which has been designed to support operating system management, the basic principles should be applied to other architectures.

Throughout this chapter and the next we will have a need to describe data structures used both by user processes and the operating system. To simplify the description we have opted to use C structure syntax when doing so, using the C99 standard types [ISO/IEC 1999]. Unless specified explicitly, the size of types is as default for the platform of implementation (e.g., an integer would be 32 bits long on an ARM based implementation and 64 bits long on an Alpha based implementation).

5.1 Custom Instruction Definition

Previous work in building reconfigurable processors has focussed on the details of the hardware, with little or no attention being paid to the programming model that is used to construct applications to run upon them. The conventional wisdom is that applications will work in terms of configuration bitstreams, which are just large blocks of binary data, accessed though a pointer. As part of integration within the wider system, we consider a much richer approach to custom instructions.

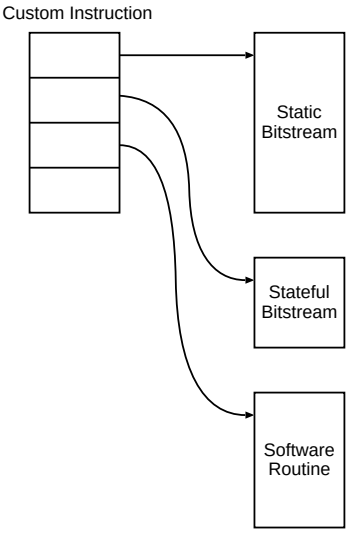

Figure 5.1: Layout of a custom instruction

Already we have detailed in the previous chapter how a process will need to manage two configuration bitstreams — the stateful and stateless bitstreams — and a software implementation for each custom instruction it uses. We conceptually treat a custom instruction to be a cohesive entity consisting of these parts, along with any additional meta data required by the programming model and the operating system (which we introduce as needed throughout this and the next chapter). This cohesive entity can then be named and used throughout the system without the need for all the constituent parts to be referenced individually.

When in memory this entity could be presented as a contiguous monolithic block of data, but this is not very flexible. For example, this does not allow a circuit to be have multiple instances within a process space without complete duplication, which would lead to needless duplication of the static parts of the custom instruction. Instead, we follow the notion of building a custom instruction as a record of references to the individual parts, turning custom instructions into objects similar to the closures described in [Saltzer 1978]. A diagram of this arrangement can be seen in Figure 5.1. A custom instruction will be used through a reference to a structure containing references to the constituent parts and any useful meta data. This structure will exist once for each instance of a custom instruction used. When there are multiple instances of an instruction within a process space each instance will use the same state bitstream and software instance, but will point to a private copy of the initial stateful bitstream data (see Section 6.7). The programmer can then use this single reference to utilise the custom instruction in the program code. Logically, the programmer will see a custom instructions as something that can be instantiated, supplied with data and generates a result based on that data. The actual mechanisms behind this the programmer does not care about. All the operations the programmer wishes to apply to a custom instruction need to be done as a whole, rather than as a collection of parts. This is discussed further in Chapter 6.

The structure of the custom instruction can be seen in Listing 5.1. The bitstreams are simply references to areas of memory containing the appropriate configuration bitstream, and the software implementation is handled by a function pointer. The meta data used is not specified yet, and will be explored in the remainder of the chapter. It is assumed that the programming tool chain will support a technique for generating custom instructions as a whole, and allowing the programmer to associate

```
typedef word_t soft_circuit_t(word_t a, word_t b);
typedef struct CITAG
{
  const addr_t     static_circuit_bitstream;
  addr_t           dynamic_circuit_bitstream;
  soft_circuit_t* software_implementation;
  // meta data goes here
} custom_instruction_t;
```

Listing 5.1: Custom Instruction definition

one or more symbolic names with it.

It is worth considering the role of state within a custom instruction at this point. Although how state behaves in a custom instruction could be considered part of the programming model, it has important consequences on how the operating system manages the custom instructions, so is worth discussing here.

The first consideration is the fact that we expect custom instructions to possibly move between hardware and software during the lifetime of an application. This behaviour limits the use of state within a custom instruction, because it is unlikely for there to be an easy way to transfer any state between the two implementations when the operating system decides to move between implementations. The means two things for the behaviour of the system. Firstly, a custom instruction can not be moved from hardware to software whilst the hardware implementation is in use (though the reverse is possible, as the software can complete regardless of the state of the hardware, as the software never becomes unavailable), and secondly custom instructions can not rely on state existing between invocations, as the operating system may move the instruction without the application being aware. Custom instructions should be designed to work like real instructions: they should no require any reloading between contiguous invocations, but they should not behave differently if a resetting does occur between invocations.

The next consideration is where the default or reset state comes from. There are two possible ways of specifying the initial state of a circuit: by setting the contents of the register in the bitstream as would be done for reloading the state after a context switch, or (as done in Xilinx devices) by an extra bit in the configuration information that specifies an extra input to the register that specifies its value on reset. The advantage of the specifying the register contents in the bitstream as opposed to by the default reset value is that designers can create a one shot initialisation that never occurs again. However, in our system, where each instruction invocation should be the same as the last, this is less useful. Using this technique would require reloading of state from memory to reset a circuit, a many cycle cost against the single cycle cost of simply applying the reset signal. Thus we specify that custom instruction designers may only use the initialise by reset value for custom instructions in order to reduce the amount of data that must be moved to reset a circuit.

5.2 Operating System Support

Having described the basic hardware layer that we believe to be sufficient to support a workstation style operating system and applications, we now look at the operating system. The role of the operating system is to manage the FPL resource and share it between the applications. As already stated, we want this resource to virtualised, so applications do not need to be concerned with when and where their custom instructions are loaded, only that after registering a custom instruction and a CID with the operating system it can subsequently use the CID as an opcode in an execute instruction until such

time as it either decides to unregister the instruction or the process terminates.

The work described here has been implemented by extending in a kernel created as a demonstration platform, called POrSCHE (Proteus Operating System and Configurable Hardware Environment). There is no reason that the work developed here could not be applied to an existing operating system, but generating a very simple clean kernel made it easier to concentrate on developing the new parts rather than worrying about how to integrate it with a more complex operating system. In this section we have assumed a monolithic style kernel, with the FPL management systems rolled into the kernel. We do not see any reason why the FPL management subsystem has to be in the kernel: it could just have well been implemented in a privileged server on a microkernel system. The only real requirement is that the management entity needs to be run in system mode so it can use the management instructions on the reconfigurable execution unit.

This section starts with an overview of the POrSCHE kernel in Section 5.2.1, then goes on to look at the system call interface provided to applications and how the operating system manages sharing of custom instructions in Sections 5.2.2 and 5.2.3 respectively. The issues around scheduling custom instructions are discussed in Section **??**, and then a demonstration of the effects of scheduling are provided in Section 5.2.5.

5.2.1 POrSCHE Overview

POrSCHE is a very simple operating system kernel implemented simply as a platform for testing on the ProteanARM. POrSCHE was first developed without support for the reconfigurable unit, and as such is a full kernel runnable on basic platforms. It has been successfully ran on an ARM Simulation model, a DEC StrongARM based network computer reference platform, the Triscend A7 development board, and a Xilinx Virtex-II Pro simulation model using an embedded systems reference design.

POrSCHE is a pre-emptive multitasking operating system, which uses a simple single level round robin scheduler, and supports all the process states described in Figure 2.2. There is just one priority level for all processes, with both the runnable process set and blocked process set being implemented as simple queues. The running set in our system can only ever hold a single member, as we only consider a single processor system. If there are no runnable processes then a system idle task is scheduled to run until such time as the runnable queue becomes non empty.

POrSCHE lacks any virtual memory management, with all processes running in a single shared address space, due to the processor simulator model used to develop the ProteanARM lacking any virtual memory management facilities. This is reflected in the use of address in a process's memory used by the operating system.

A minimal set of system calls have been provided for basic job control. For intraprocess job control yield, sleep, and halt are provided, and for interprocess control spawn and terminate. Each platform implementation of POrSCHE supports a basic set of output routines to allow processes to output progress messages. The total code size for POrSCHE, including the generic kernel, platform support code, basic C library (portions of which were taken from NetBSD's C library), test applications, and custom instruction related extensions, comes to 37.5 kilolines of code (include white space and comments). The core kernel functions only account for 2.5 kilolines of that.

5.2.2 System Call Interface

System calls represent the interface between user processes and the operating system. In a protean system, just as in other such systems providing user processes with reconfigurable logic, the interface

for managing the FPL resource is quite simple. The user processes will want to register and unregister circuits, and to execute their circuits. This interface is proposed in the RAGE system [Burns et al. 1997], an FPL middleware system, between applications and their virtual hardware management system (equivalent to our Custom Instruction Scheduler, discussed in Section 5.2.4). In a protean system however, applications are allowed to directly invoke their custom instructions without needed to interact with the operating system. This reduces the interaction between processes and the operating system to the registering and unregistering of custom instructions. The interface to these two system calls will look like:

```
int register_custom_instruction(int CID, custom_instruction_t* inst);
void unregister_custom_instruction(int CID);
```

When registering a new custom instruction, the operating system will first check whether that CID is already assigned or not, and if the CID is used it should fail and return an error. If the CID is available for use, then the operating system can register the custom instruction, updating the internal data structures used by the operating system for managing custom instructions. If the operating system was to perform any security checks on the configuration bitstream, as discussed in [Hadžić et al. 1999], then these would also be done now.

Typically, when a process calls a system call, the operating system will copy the parameters passed to it to operating system memory. This is so as to ensure that the data provided is always available to the operating system (should the effect of the system call last beyond the life time of the process) and to ensure that application does not change the parameters before the operating system has finished processing the data. This is trivial for basic parameter types, but we need to note that a custom instruction is a reference to a larger structure, and simply noting the reference does not prevent the process registering the circuit from modifying the custom instruction's data later. This is considered unfeasible for a protean system. For instance, the operating system may have carried out security checks on the bitstream, so later modification of the bitstream before invocation may circumvent those checks. Similarly, if an instruction is being shared between multiple applications modifications of a bitstream without the operating system's permission may lead to inconsistencies in operation. The behaviour when faced with data structures containing references depends on the operating system in question. Systems based on BSD 4.4 copy all referenced data into operating system memory [McKusick et al. 1996], whilst Windows 2000 does not [Solomon & Russinovich 2000], instead locking the memory in the user process for the duration of the call.

The simple solution to this is for the operating system to copy the sensitive parts of the custom instruction, the configuration bitstreams, into operating system memory. This is what is done in the RAGE system [Burns et al. 1997]. Once in operating system memory they will be safe from modification by the owning process (the user can attempt to modify their own instruction, but it will have no effect on the instruction once it had been registered) and this also enables the operating system to ensure that circuits are always in memory for loading and unloaded, protecting against page faults during custom instruction transfer. In the ProteanARM system we have assumed that the hardware can support page faults during configuration, but on other platforms that do not provide for such a circumstance, this option allows the operating system ensure that configuration is not interrupted. The drawback of this technique is that copying the bitstream into operating system memory adds significantly to the cost of registering a custom instruction. In the programming model developed in Section **??** the run time environment registers all instructions when an application starts up to ensure that CIDs are always ready. Under this model instructions that may not be used are registered, which if the process is linked with a library of circuits could lead to a potentially high startup cost.

The alternative is to mark the memory areas in which the bitstreams are used as read only. Most virtual memory systems allow memory to be marked read only on a page by page basis, which gives a 4 Kbyte granularity on most modern operating systems. The respective sizes for both the static and dynamic configuration bitstreams are larger than a page size, 48.2 Kbytes and 4.4 Kbytes, and could have their storage requirements rounded up to the nearest page, allowing the operating system to simply lock those pages when an instruction is registered to prevent unauthorised modification. The advantage of this technique is that it does not significantly add to the registering costs of an instruction, at the expense of slightly increasing the size of a custom instruction.

Because the simulation model of the ProteanARM does not include an Memory Management Unit (MMU), we have assumed the later model. It use useful to note however that the former model is useful for architectures that would not tolerate page faults occurring during reconfiguration.

5.2.3 Sharing Support

Given the desire to avoid frequent context switching of circuits due to the time lost in moving circuits on and off the processor, there is an obvious benefit if the operating system can identify multiple instances of the same custom instruction being used, and allow them to share a single loaded instance on the processor. For instance, it is possible that applications on a system hosted on a reconfigurable processor will use libraries of circuits. Although in an ideal world each circuit would be tuned to a particular application, some software vendors may not be able to afford to develop such circuits. Instead they may use prebuilt libraries of circuits[1] will be supplied for such application developers to use, such as a library of multimedia type instructions. Just as shared libraries of code are only loaded once into memory (see Section 6.1.3), we would like circuit libraries to only be loaded once onto the processor if possible.

Because circuits may be potentially shared, this implies that circuits registered with the operating system will not change during that time. For example, problems would occur if a process registers a circuit which is then shared with another process, and then either process tried to modify the custom instruction in some way (perhaps partially evaluating the bitstream such as was done in the Twofish example in Section 4.3.2.3 or changing the software alternative function). Either one process would end up with its instruction not being modified when subsequently invoked, or it would be modified when not expected. This leads to the requirement discussed in Section 5.2.2 that circuits registered with the operating system can not be modified by the application whilst registered. This means that if an applications wishes to modify its custom instruction (for example, partially evaluating constants into the circuit) then the applications must first unregister its instance before modifying it. When reloading the circuit it should then be detected as a different circuit and not shared as before.

A custom instruction consists of many parts, and we are not interested in sharing all of them. Indeed, it can be though of that we are not sharing custom instructions at all, but merely the hardware implementation of custom instructions. The aim of sharing is to optimise on use of circuits already in PFUs, so there is no need to share the software alternative in a custom instruction. It is entirely reasonable that two custom instructions use the same bitstream but different software alternatives (for example, on may be a real time application which will not want to run a software alternative but take corrective action, and one may be prepared to run a slower implementation and this have a software implementation that does the calculation), but for the purposes of sharing these two instructions should be considered equal. Circuits that are stateful obviously do not want the content of the stateful part of the bitstream sharing between instances, so the stateful bitstream is not share between applications;

[1] Current FPGA vendors go to a lot of effort to provide useful blocks of logic (cores) that users can simply plug together, ranging from basic maths functions to entire Ethernet devices.

this must still be swapped when another application goes to use the custom instruction. This still represents a significant reduction in the amount of data transfered around the system; on the ProteanARM this reduces the amount transferred by 91%.

The operating system needs a way of identifying custom instructions that share the same static configuration bitstream. At the point at which an application registers a custom instruction the operating system is simply presented with the custom instruction bundle and the CID the application wishes to associate with that custom instruction; from this the operating system then needs to decide if the new instruction matches any other loaded into the system. The simplest method is to do a brute force bitwise comparison of the static bitstream in the newly registered custom instruction with all the others currently registered with the system. However this will quickly become costly as circuits are registered with the operating system, although we imagine that most instructions will fail early on in the comparison. We would like a way to help the operating system identify likely matches, so as to reduce the number of direct comparisons required.

One way to do this is to associate some form of name with the configuration data as part of the custom instruction bundle. This can then be used as a first stage comparison: instructions with the same name may possibly be the same. Because we can not guarantee that the names associated with a bitstream in a custom instruction are unique or can be trusted, it still requires a bitwise comparison to occur once a match on the name has been based, but with a suitably sparse namespace, this will reduce the number of bitwise comparisons that are required. Although a human readable names for custom instruction bitstreams could be used, we propose using a digesting function, such as the MD5 message digesting algorithm [Rivest 1992]. This gives a short name (128 bits in the case of MD5) with a very small probability of name clashes occurring (the authors conjecture that the difficulty of coming up with two messages having the same message digest is on the order of 2^{64} operations). This digest can be computed at compile time, requiring no additional runtime overhead.

A possible further optimisation is to only do the comparison when the process first attempts to invoke a custom instruction. If large libraries of prebuilt custom instructions are being used, then the process may register many instruction that it has no intention of using (this is an artefact of the programming model suggested in Chapter 6). Thus it may be preferable to lazily test for duplicate circuits being used. This would avoid potentially expensive process initialisation costs and remove unnecessary circuit comparisons, but at the expense of potentially adding significantly to the latency of a custom instruction on its first issue.

5.2.4 The Custom Instruction Scheduler

Having developed POrSCHE as a basic kernel for a general purpose processor, we then extended it to support a Custom Instruction Scheduler (CIS) to manage the reconfigurable execution unit on the ProteanARM. In this section we look at how the basic CIS is structured.

The CIS is an individual component of the operating system (although in the POrSCHE implementation we have integrated it within a monolithic kernel). The CIS is responsible for maintaining all the information regarding what applications have registered custom instructions, what instructions are loaded, and so on. The only explicit interaction with the rest of the system is the requirement to tell the process ID of the current process, the ability to move the current process to the blocked set of applications, and at a later time request the moving of processes from being blocked to runnable.

```
typedef struct CRTAG
{
  struct CRTAG*          next;  // Used for building linked list
  struct CRTAG*          prev;

  uint32_t               cid;          // CID associated with
  custom_instruction_t* instruction; // pointer to custom instruction
                                     // information in process.

  bitstream_info_t*      shared_bitstream;
  struct CRTAG*          share_list; // Used to list other instances
                                     //using same bitstream

  struct PCBTAG*         owner;        // Reference back to owning proc
} ci_record_t;
```

Listing 5.2: Custom instruction record

5.2.4.1 CIS Data Structures

In POrSCHE, the CIS maintains a list of registered custom instructions for each processes, which is linked to the PCB for that processes. The list contains a node for each instruction registered, the structure of which is shown in Listing 5.2. The first two entries are there to form a doubly linked list, which is referenced from the PCB. The next two entries record the information passed in the register system call which created the node. The custom instruction record is actually duplicated in operating system memory to prevent the application from modifying it after registration has completed. The next two entries are used for circuit sharing. Finally a link is held to the owning processes PBC structure, which is useful when carrying out operations on the circuit.

The shared bitstream structure is used to record information about the individual configuration bitstreams for each unique configuration loaded in the system. The contents of this record shown in Listing 5.3. Each time an application registers a custom instruction, the operating system examine a global list of bitstream_info_t structures comparing the MD5 digest associated with each instruction with instructions in the list. If a match is found, and the subsequent bitwise comparison is a success then the new instruction record associated with the process will reference the existing bitstream record, otherwise a new bitstream record is created and added to the global list. Currently a sequential search is used to look for digest matches, but a faster approach, such as storing digests in a trie, could be used.

The bitstream_info_t structure is used to manage information about circuit bitstreams. Firstly it notes whether that circuit is currently loaded onto the processor, being loaded onto the processor, or is currently not on the processor; this information is used when deciding what action to take when an invocation faults. If the instruction is loaded then the RFU location is noted. This information is held here rather than in the custom instruction record above so that instances can be shared between multiple processes.

To allow the operating system to easily manage PFU selection and eviction, the CIS maintains an Inverted Circuit Table (ICT), similar to the Inverted Page Table (IPT) used in some virtual memory management systems [Silberschatz et al. 1998]. The ICT contains an entry for each physical PFU with a reference to the custom instruction record that relates to the circuit currently loaded into that PFU and the process that was last using it.

```
enum LOAD_STATE {LS_NOTLOADED, LS_LOADING, LS_LOADED};

typedef struct BITAG
{
  struct BITAG*          pNext; // used for building linked list
  struct BITAG*          pPrev;

  enum LOAD_STATE        load_state; // indicates status of bitstream
  int                    rfu; // where loaded on processor
  uint32_t               nSize;  // size of static bitstream

  addr_t                 bitstream;  // pointer to static configuration data
  uint32_t               digest[4]; // MD5 digest of config data

  int                    ref_count; // reference count
  struct CRTAG*          user_list; // list of using ci_record_ts
} bitstream_info_t;
```

Listing 5.3: Circuit bitstream record

5.2.4.2 CIS Operation

The initial CIS implementation uses a demand loading system for loading custom instructions. The first time a process attempts to invoke a custom instruction the dispatch hardware will cause an exception to be raised in the operating system, which will be handled by the CIS. Although the idea of pre-fetching custom instructions has been examined for this domain [Hauck 1998], we do not consider that here. This section outlines the operating of the Custom Instruction Scheduler (CIS), which has then been implemented in POrSCHE. Here we are concerned with the mechanism of how custom instructions are scheduled on the processor, rather than policy decisions which follow in Section 5.2.4.3.

Figure 5.2 outlines the possible responses taken by the CIS as the result of receiving a custom instruction fault. The first action taken by the CIS is to look for a matching CID in the process's list of custom instruction records. If no match is found then the process has attempted to execute an unknown custom instruction, and as a result is terminated (this error is similar to an illegal memory access). If a match is found then the CIS checks to see if the instruction is already loaded into a PFU, either for the current process or another process.

If the circuit is not loaded into a PFU then the CIS attempts to find an empty PFU into which it can load the custom instruction. If there is a free PFU on the processor then the custom instruction is load and the hardware dispatch TLB updated, after which the process can be reissued at the faulting instruction and execution can continue using the new custom instruction. If there is no free PFU, the CIS has to make a policy decision on how to handle the fault. The CIS may either choose to evict a circuit from a PFU based on an eviction policy, or it may defer the instruction to run in software. If an eviction policy is used, the policy will nominate a PFU for eviction. The CIS will then remove mappings to this PFU from the hardware dispatch TLB, and if necessary store any configuration state back to the last process to use that circuit if it was preempted whilst in execution. Once this is done the CIS can continue as if the PFU in question was free when it first looked. If a software alternative is used then the the CIS loads the appropriate mapping into the software dispatch hardware and reissues the process at the faulting instruction. Before reissuing the process, the CIS will add the process's custom instruction record to the end of a FIFO recording which custom instructions are currently running in software. When a PFU becomes available at a later time the CIS will consult this queue to see if it can promote any custom instructions from running in software to running in hardware.

If the circuit is loaded into a PFU, then the CIS will check what state it is in before allowing

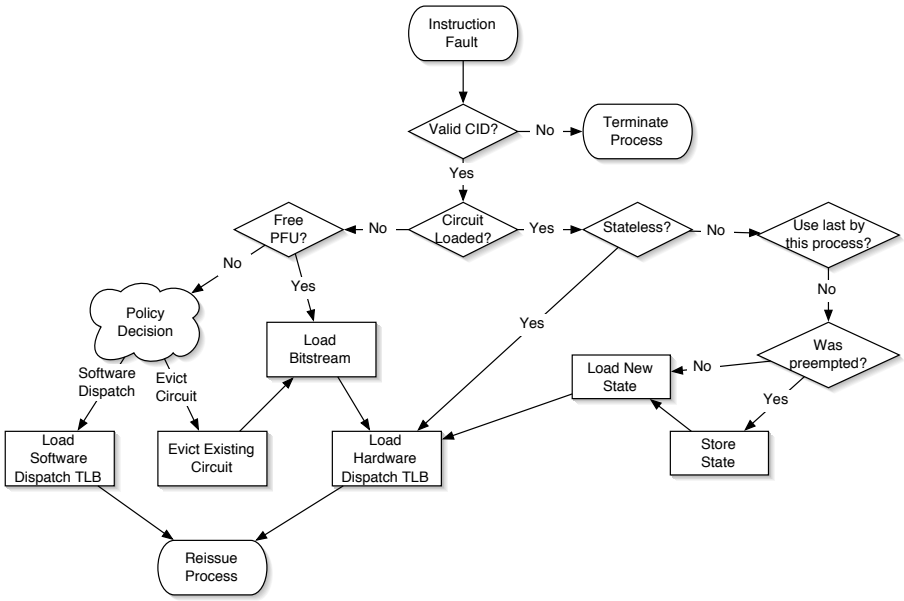

Figure 5.2: Control flow in response to a custom instruction fault

the current process to use it. If the instruction is marked as stateless, then the CIS can add a new TLB entry for the PFU and reissue the application from the point at which it faulted. If the circuit is stateful, then the CIS checks to see who was last to use it. If it was the current process, then the fault must be the result of the original hardware dispatch TLB entry being displaced, and this can simply be replaced and the process resumed. If not then the CIS checks with the PCB for the application last using it to see if the state needs preserved or not. If there was no active state in the circuit then the circuit is either reset or has the current process's state loaded over it, depending on whether this application was preempted whilst using this circuit last time. Once the circuit has the correct state in it, the CIS can update the hardware dispatch TLB and reissue the process. If the circuit was stateful then the CIS must unload the previous mapping for that PFU held in the dispatch TLB, as in future the PFU may hold incorrect state, and the CIS must prevent other instance users from accessing or modifying this state.

As discussed in Section 4.1.6.1 reconfiguration may either occur in a single long operation which occupies the entire system for its duration, or all or part of the configuration can be done independent of the main processor core, and the processor may do other useful work whilst configuration occurs. This later case requires the CIS to be more complicated and interact with the process scheduler, and this is what has been implemented on the ProteanARM and supported in POrSCHE. When the CIS finds a free PFU into which to load the custom instruction in question, it starts the loading process and then removes the current process from the set of runnable processes and adds it to the set of blocked applications. Other applications can then be scheduled to run, allowing the system to make progress whilst the PFU is being programmed. It is impossible for any application to attempt to invoke the PFU being loaded as the application that had requested it is blocked and the dispatch hardware will not have been programmed correctly yet, so other applications that may want it will be caught be the operating system and similarly blocked. Once the operating system receives the interrupt indicating that loading has completed all the processes that were blocked on that instruction loading are moved back into the set of runnable applications. If another application attempts to load a custom instruction

whilst the processor is still busy loading an instruction from the FIFO, then the process is also blocked until the completion interrupt is received by the operating system here.

The operating system may wish to attempt to ensure that blocked requests to load instructions are rescheduled in order in order to prevent starvation, which could potentially occur on some processes if there is a large number of circuit swaps occurring.

5.2.4.3 Circuit Eviction Policies

In Section 4.1.7 we discussed the hardware support that could be provided to assist the operating system in selecting circuits in PFUs for eviction. The existing research literature in managing FPL devices has focussed on three techniques for deciding which circuits to evict when space ran out: off line based algorithms, load sequence based algorithms, and history based techniques similar to page replacement policies used in virtual memory management. The first technique is not suited to the more dynamic workstation environment; it is simply not possible to do a static analysis of the load order of instructions in such a dynamic environment. The run time algorithms can both be applied to the workstation environment, but the load sequence analysis is unlike to prove suitable. Load sequence analysis looks at the load order of circuits onto the FPL and attempts to predict what circuits will be needed next based on recent history. For this to work, the ordering of circuit invocations needs to have some form of pattern to it; however in a workstation environment with many applications using circuits according to different algorithms, the ordering is unlikely to follow a set pattern. However, such an algorithm may be useful if the operating system observes a working set policy, whereby applications are allocated a set of PFUs and must reuse just these for their instructions.

Given the obvious parallels between PFU replacement and page replacement, using standard page replacement policies for managing circuits provides the most obvious route. However, as noted in [Hauck et al. 2000] there are important differences. The main difference is that page sizes are constant, where as bitstream sizes may vary substantially; this could mean that if one bitstream is 10 Kbytes and another 100 KBytes, it potentially makes sense to preserve the later bitstream in some situations, even though the former may be used more often. For our work, we have assumed that all bitstream sizes have to fully configure each PFU. We do this as we have no idea how much of a PFU would need configured in reality, and so assume a worse case scenario; if we can demonstrate a performance benefit in this case then smaller bitstreams would provide less management overhead, and the results would be even better.

A basic set of policies has been implemented in the CIS, based on recent history information. We have round robin and random eviction policies, which make no use of usage history provided by the processor, most recently used and least recently used, which use the logic counter support hardware, and most frequently used and least frequently used, which make use of the invocation counters. In addition to this we have an implementation that does no evictions at all, and instead uses software dispatch.

5.2.5 Scheduling Evaluation

The aim of this section is to get a feel for how the system performs when managed by an operating system. In Section 4.3 we saw how some basic applications benefitted from using custom instructions, and the aim in this section is to see if they still achieve that benefit despite facing competition for the FPL resource from other applications. This section also will provide us with a feel for what the overheads are like for various loads, and how the different custom instructions scheduling policies behave, with particular interest being paid to the software dispatch mechanism.

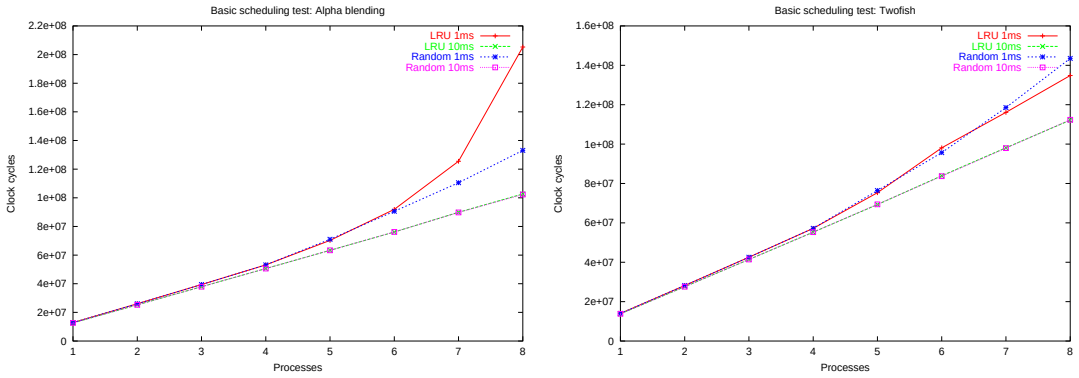

Figure 5.3: Test run results for Alpha Blending and Twofish encryption

Note that these tests are not attempts to demonstrate how the system would perform under a realistic workstation workload; orchestrating such a test is outwith the manpower of this project. The aim of these experiments is to observe the effect management overheads of sharing the FPL resource have on overall system performance.

5.2.5.1 Basic Management

The first test is to observer how performance of the system behaves as the FPL requirement exceeds demand. The aim of this test is to vary the FPL requirement from below the number of available PFUs to over the number, and see how the system performs. This test will then be repeated for a number of eviction policies and and with two different scheduling quanta values. The two values used are chosen in an attempt to approximate a batch scheduling load and an interactive scheduling load. For this we use a 10 ms scheduling quanta to represent batch scheduling (which is the value used by Linux for batch scheduling) and a shorter 1 ms value to represent a more interactive environment. This interactive value would obviously depend greatly on the actual load on the system, and as such the 1 ms value can only serve as a guide; real tests with real loads would need to be carried out in order to get a more detailed evaluation.

The first experiment tests the performance of the ProteanARM under load using just the reconfigurable execution unit without the software dispatch facility. For each test application eight runs were performed with increasing numbers of concurrent applications. The experiments were repeated with two different values of the process scheduling quantum and with two different circuit scheduling policies. The circuit scheduling policies used are LRU and random. As a performance metric we measure the number of cycles required for all processes in a run to complete.

Figure 5.3 shows the results of the two tests for applications using just one instruction. In all cases the the increase in completion time is linear with the number of concurrent processes until PFU contention occurs (that is greater than four instances for these two tests). Once contention occurs circuit switch overheads reduce the overall performance. At a 10 ms quantum value the extra overhead has only a small effect of completion times, however at the 1 ms quantum value the increased number of switches causes a larger performance reduction. In this case the LRU policy generally performs worse than the random placement policy as it interacts badly with the round robin scheduling policy, which means typically applications lose their circuits after a context switch. Despite this, both applications perform an order of magnitude better than the unaccelerated version. It is clear from the LRU 1 ms curve however that although all applications make suitable progress, the bad interaction between the

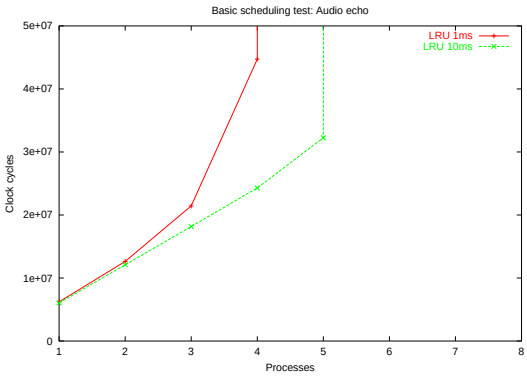

Figure 5.4: Test run results for audio echo test

process scheduler and the circuit scheduler can degrade overall performance significantly.

The results for the audio echo test however are much worse, as shown in Figure 5.4. In this case the graph only shows the results of LRU scheduling. Although initially the graph follows the same trend as the first two applications, only with contention starting after two instances due to each instance using two instructions, quickly the performance of the system collapses, with the results for the 5th run at 1 ms and 6th run at 10 ms going off the scale[2]. The bad performance for these runs is due to the scheduling interactions caused by the load FIFO. The audio echo algorithm uses two custom instructions in a tight loop, invoking them in sequence once per iteration. When a process reaches the first invocation on the first iteration, it is blocked whilst the instruction load happens. The other instances then attempt to run, but also block because they can not load other custom instructions. Eventually the load FIFO is drained and the applications blocked on the load are released back into the run queue. Assuming the idle task has not been activated, i.e., not all applications have blocked waiting to load, then whatever process is running is about to attempt to execute the custom instruction, so this too will start a new load and then block. When we eventually get back round to the first app to try loading an instruction, it will invoke it and then immediately block as it can not load the second instruction it needs. This is then repeated for the second instruction. Because there are more instructions needed than PFUs, each instruction gets displaced between invocations from the processor, so the end result is that for a large number of simultaneous processes using multiple instructions in a tight loop, each instruction only gets used once before having to be reloaded, so the system essentially stops making progress due to the number of circuit loads that are required. If we take this to the extreme, then it is possible to envisage a situation whereby applications never make any progress because their instructions are unloaded before the have even been used.

This behaviour was found to also, on a rare basis, for the other two sets of application, but is hard to reliably duplicate. Essentially the problem is due to a race condition and even a small change in the timing of the system can affect whether the problem occurs or not. This problem only occurs because other processes are allowed to potentially use the FPL resource before an invoking application has. If the system was blocked fully until a PFU was full programmed, then this situation would not occur. The delayed loading system has significantly increased the complexity of behaviour regarding the FPL resource.

We can approximate the situation where processes are not interrupted during loading while still letting other applications run during instruction loads by adjusting the scheduling policy slightly such

[2]The final value was not recorded due to the simulation taking too long to complete. The first four runs at 1 ms took roughly one hour, and the fifth run had not completed after six days.

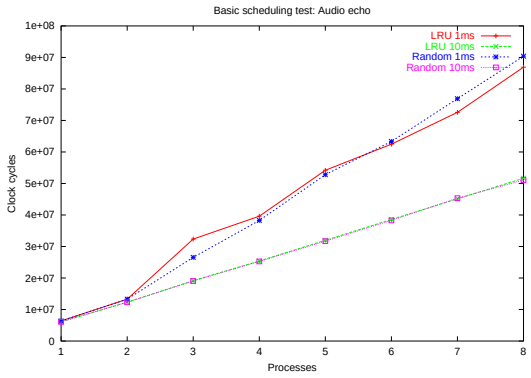

Figure 5.5: Test run results for audio echo with modified scheduling policy

that when an instruction load has completed (i.e., drained from the load FIFO), the task that requested the new instruction is immediately rescheduled. This means that in this case the application will invoke its instruction, then fault on the next load, be rescheduled once the second load has occurred, and then run for a complete The results for running this new policy with the audio echo application can be seen in Figure 5.5. Now each application will get two short periods on the processor whilst it loads its instructions, then an entire scheduling period to use these instructions, after which they will be swapped out as other processes try to use the PFUs, and the process will repeat itself.

This policy works well for the given test, and certainly prevents the resource starvation evident in the unmodified scheduler, but this basic policy leads to unfair scheduling for applications that are able to make loads, whilst other processes will tend to get less CPU time, specifically processes without custom instructions, which may be continuously deferred whilst the system is overloaded with a set of accelerated applications. What is needed during these times is an alternative way of handling overload of the reconfigurable execution unit.

5.2.5.2 Software Dispatch

The obvious alternative swapping circuits once all PFUs are full is to for applications to use software alternatives. This will reduce the number of circuit switches required, allowing the overall system to spend less time doing meta work, at the cost of slowing down the applications trying to use custom instructions. To discover how the software dispatch system compares with using circuit swapping we compared the round robing circuit scheduling runs from above with the use of software dispatch. As for the tests in the previous section the time taken for various sets of concurrent applications are taken for 1 ms and 10 ms scheduling quanta. The end results can be seen in Figure 5.6.

In these tests the runs all show contention being reached at the same point as before, after which we can see the divergence as the different ways of handling the resource contention are used. In both cases the software dispatch does not perform significantly worse that the hardware accelerated versions, and all tests complete significantly faster than the unaccelerated implementations as before. In both tests the circuit swapping versions run faster that the software dispatch tests, but a the shorted switching periods we see the alpha blending perform worse in hardware, as the cost of switching circuits does not provide an advantage. From these basic results we can infer that for batch systems which use long scheduling periods that circuit switching is better, but for times when the number of circuit switching increases (say in an interactive system) the software dispatch alternative is useful. Note that the results here reply on the fact that those processes that are dispatched to software eventually have their custom

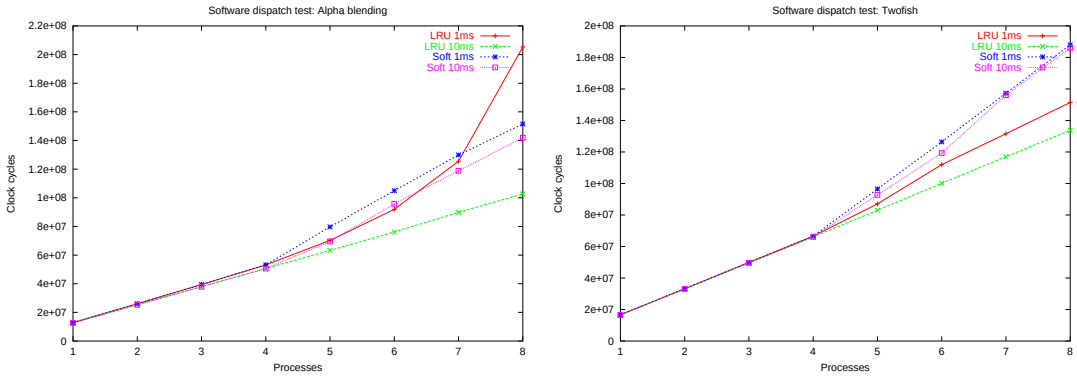

Figure 5.6: Software dispatch test results for a) Alpha Blending and b) Twofish encryption

instructions promoted to run in hardware. Otherwise the performance with be closer to that of the unaccelerated applications.

These results show that dispatching to software alternative implementations for custom instructions is a viable technique for handling contention, but does not perform as well as switching circuits at anything above the shortest of scheduling quanta periods. What is needed is a technique to allow the operating system to mix the two methods depending on the current load. We explore this in the next section.

5.2.6 Run Time Statistics Monitoring

Although we find that generally context switching circuits during a heavy load itself does not hamper system performance dramatically, certain usage patterns can lead to significant reduction in progress with the amount of usage a processes gets from a custom instruction being low before it is evicted and requires reloading. This is demonstrated in Section 5.2.5.1 where we see examples of bad process scheduler and circuit scheduler interactions, particularly the audio echo example. At such moments the operating system needs to step in and take corrective action otherwise applications using the reconfigurable execution unit will make no significant progress. There are two obvious causes of action open to the operating system in this set of circumstances: it may elect to force applications to use software alternatives for custom instructions, or it may invoke a swapper, to temporarily remove one or more processes that use custom instructions from the runnable queue to remove the contention. For this work we will examine the former option, and disable the priority scheduling mechanism used to enable the audio echo processes to not overload the system: the idea is that we use the software dispatch system instead to resolve the issue.

To be able to take corrective action the operating system needs to be aware of when the reconfigurable unit is overloaded. We do this in a similar fashion to how the operating system detects when the virtual memory system is overloaded, by using information about the recent history of operating system events. In 4.4 BSD the operating system gathers statistics on various system events, such as page and disk activity, context switches, etc. This information is used by the operating system for activities such as calculating a process's priority, invoking the swapper, etc. In order to attempt to detect when the reconfigurable unit is overloaded such that applications are not making progress, a statistics gathering engine was added to POrSCHE. The question is what statistics can be used to provide an indication of when the system is overloaded. It is also important to note that there is a trade off to be made between frequency of sampling, the overhead of the sampling, the accuracy, and responsiveness.

From observing the behaviour of the previous overload examples we noted that there is a dramatic increase in the number of context switches during contention, the number of loads go up, both successful attempts and failed attempts. The problem with detection is that simply noting a lot of loads does not mean that the system is overloaded. It is okay for an application to have to load its circuits on each run at the CPU, so long as it does get an entire quanta of usage.

To observe the behaviour of these values we ran a set of tests where we ran from one to six concurrent instances of the audio echo application, one to twelve concurrent instances of the alpha blending process (which gives an equal number of custom instructions in use at the top load) and one to six concurrent instances of an application without an application to give a set of control values. Each new process instance occurred at a 150 ms interval. The tests were run with a 10 ms scheduling quanta and a 5, 10, 20, and 50 ms statistics gathering period. The first two set of tests were then repeated using a 1 ms scheduling quanta. The results for the context switch and failed load counts, along with the number of processes running and how many PFUs are currently utilised can be seen in Figures 5.7 and 5.8. Note that the process count includes the supervisor task that was responsible for launching the test applications eacho 150 ms.

These graphs show several things. Firstly we can see how the number of context switches and failed loads does indeed increase as the FPL resource becomes more congested, particularly with the smaller scheduling quanta. From the control run it is also obvious that this is directly related to the PFU usage. We can also see that for the sorter scheduling periods the results are quite noisy, indicating that a longer stats period is more suitable.

Based on this, we demonstrate the ability to reduce contention on the FPL using failed loads as a metric using a statistics period of 50 ms. This is merely to server as a demonstration of the ability to respond to overload, and is not intended as a concrete rule as to an ideal cut off point. The workloads we present are artificial, and more extensive work with more diverse workloads is required for a working value to be selected. Obviously context switches is a poor indicator, as it merely reflects that applications are blocking on resource usage, rather than indicating to what they are related. Monitoring failed loads, as opposed to other possible indicators like the average number of processes blocked on a load is very cheap to implement. The number of failed loads is linked to the average scheduling period used, so the point at which the CIS alters the scheduling policy needs to vary inversely with the average scheduling period length.

For our tests we used a value of 75 over the scheduling period in ms as the cut off. When the number of failed loads in the previous statistics gathering period exceed this value, the CIS will start dispatching custom instruction loads to software. The results for this can be seen in Figure 5.9. Here the change in policy can be seen by the drop in failed loads as the cut off limit is reached, indicating that all custom instruction requests are now being satisfied. On the next sampling period the reduced number of failed loads will be noted and the CIS will return to the original circuit eviction based policy. This can be seen by how the failed load count increases again as new processes are spawned. As processes using hardware instructions unregister their custom instructions, those running in software will be promoted to run in hardware.

While this approach successfully prevents the system from suffering overload during periods of high contention, it has the drawback that for long lived applications it can lead to a situation where some processes will remain using hardware and some remain using software, leading to unfairness. Thus the CIS needs to use a third level of policy modification. After a suitable period, perhaps in the order of seconds, the CIS should attempt to rectify the unbalance. One possible solution would be to use another scheduling policy to swap circuits from hardware to software, say following a round robin queue of processes using custom instructions. Those currently using hardware would be put to the back of the queue, and those at the front of the queue promoted from software to hardware. Another

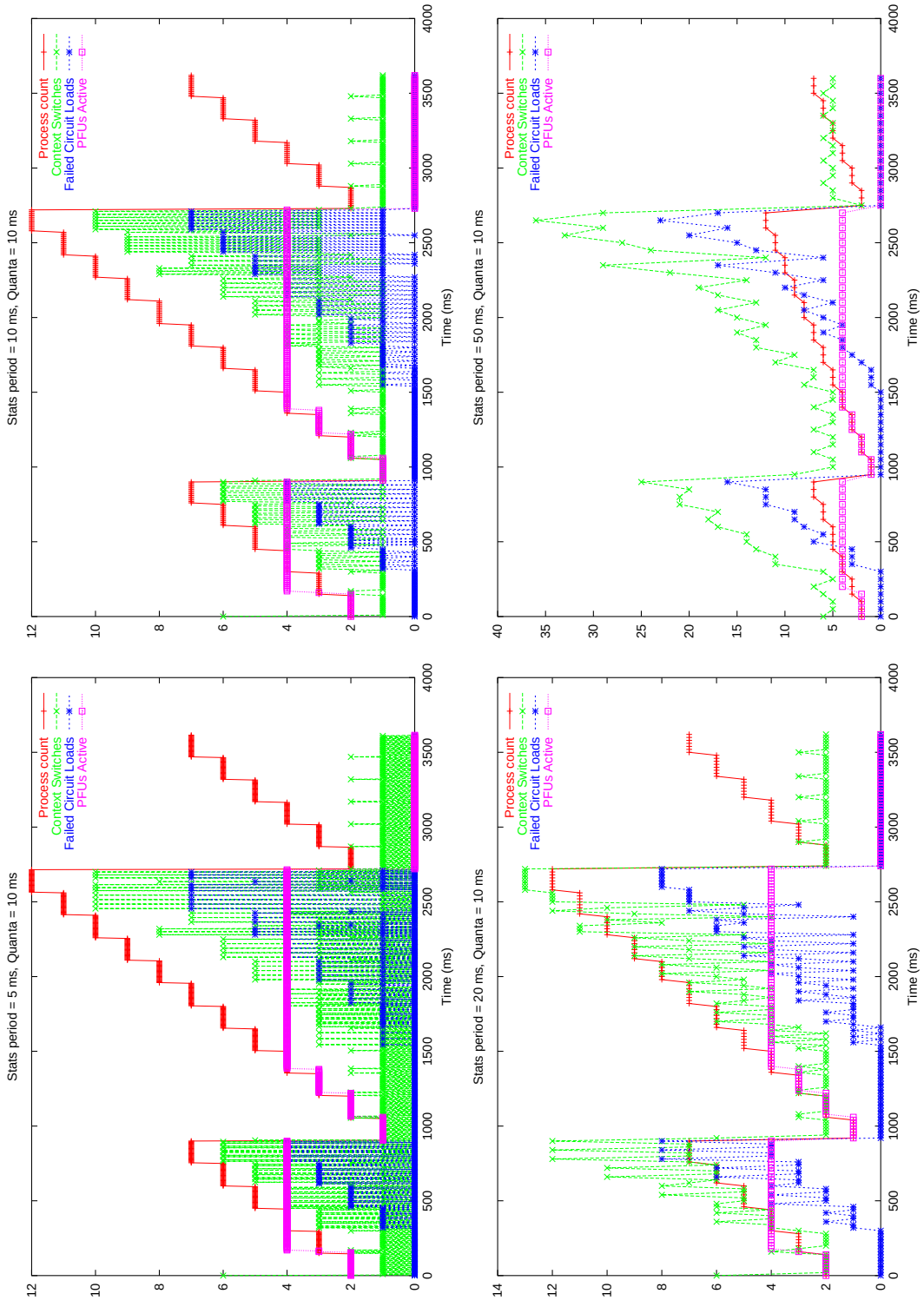

Figure 5.7: Initial statistics output for varying sampling periods with 10 ms scheduling quanta

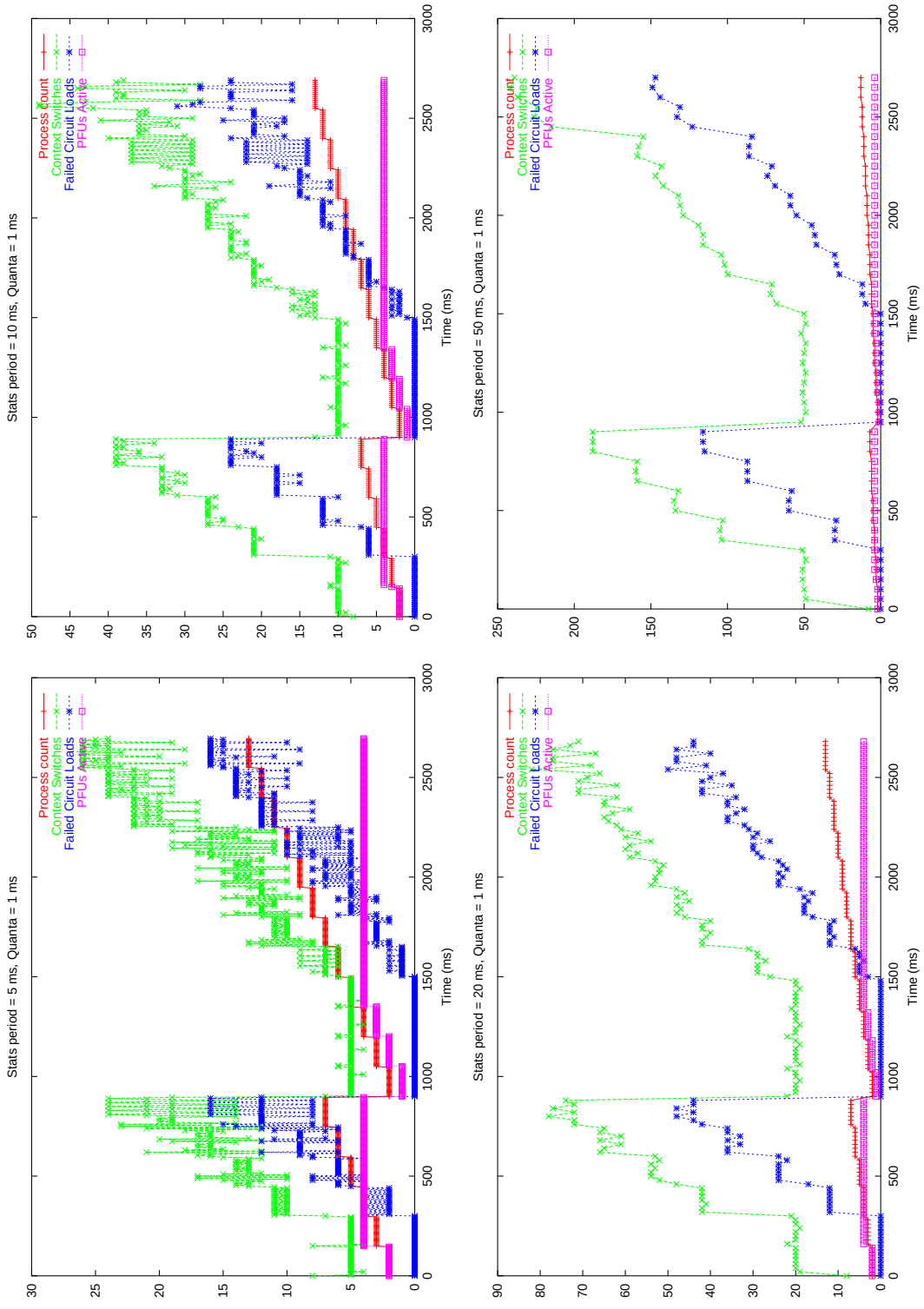

Figure 5.8: Initial statistics output for varying sampling periods with 1 ms scheduling quanta

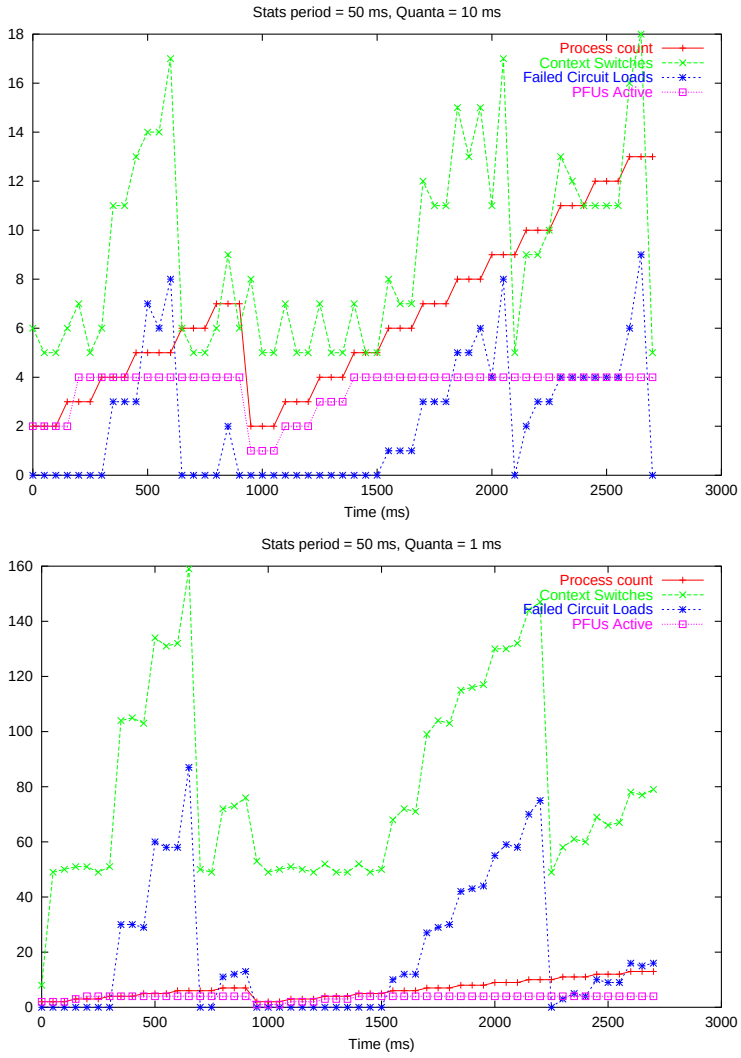

Figure 5.9: Modified statistics output for 50 ms sampling periods

solution would be to use a jubilee system [Black 1994]. Every jubilee all custom instructions would be unallocated and then processes would compete as before for access to the PFUs.

5.3 Summary

In this chapter we have presented that it is possible to utilise and use custom instructions on a reconfigurable processor within the confines of a workstation environment.

In the first section we outlined a rich custom instruction mechanism that goes beyond the traditional notion of a custom instruction as just a single configuration bitstream. We provide a richer data structure that supports the ability to treat stateless and stateful parts differently, include a software alternative implementation, and encapsulate any necessary metadata, while providing a single point of reference for the application developer. In the next chapter we will demonstrate how a programming model could support this model of a custom instruction.

In the remaining part of the chapter we outlined a model for how a traditionally structured operating system could manage a reconfigurable processor based on the Proteus Architecture. We have shown how the applications can register a custom instruction with a process specific CID and then use that CID in machine code instructions to invoke the custom instruction, without being aware of where the custom instruction resides, thanks to the operating systems management. This has been reinforced by practically demonstrating the system working in the POrCSHE system by running the test applications introduced in Section 4.3. When there are less active custom instructions the operating system safely allocates them to individual PFUs without the applications having to made aware that they are sharing the FPL resource, thanks to the virtualisation.

The initial experiments described in this chapter have provided an interesting insight into the behaviour of such a system during times of contention. For the general case, the cost of repeatedly loading circuits after a process has been switched out is not that prohibitive, even for short scheduling periods. Even using the software dispatch mechanism to handle contention still lead to results which competed well the unaccelerated applications. The software dispatch mechanism essentially unaccelerates applications while there are no PFUs available, so as long as some applications are running in hardware, we would expect the overall system performance to better. The software dispatch mechanism will however be slightly slower than had the routine been written in software: the code for entering the software routine will be less efficient than a traditional procedure call and prevents optimisations such as inlining to have occurred at compile time. However, the performance would be significantly worse did the processor not assist with decoding the the faulting instruction, saving several tens to hundreds of instructions per invocation. The instruction for the ProteanARM is quite simple, only allowing operands to come from the reconfigurable unit's register file, which would be trivial to decode, compared to an architecture that let operands come from multiple sources, increasing the costs where there no hardware support significantly.

However, the experiments also revealed problems with the system due to the introduction of the circuit load FIFO. Although this system allows the processor to run software during the relatively low configuration speed — which would be even more important on processors with faster clock speeds than the 40 MHz the ProteanARM is simulated at — it dramatically increased the complexity of the system's behaviour. This would lead to the situation whereby an application would lose the use of a PFU before it had managed to get significant use from the circuit, leading to a collapse in performance of the system. We outlined several solutions to this problem. The most obvious solution is not to switch to another process during a circuit load, but this essentially negates the use of the FIFO in the first place. Instead the operating system could guarantee that the process that runs afterwards

will immediately give up the processor once the load has done. Another option not covered above is to let processes that do not use custom instructions to run between the process faulting on a load and it regaining the processor. This would allow other processes to get a useful amount of the processor and still prevent other processes using the reconfigurable unit, which is the cause of the problem.

However, the above suggestions ignore other scheduling criteria the operating system may have to meet, so we also investigated a higher level custom instruction scheduler that watched for the reconfigurable unit becoming overloaded and then switch the policy used when a custom instruction can find no free room from one that evicts circuits to one that used the less efficient, but workable, software dispatch. To do this the operating system was extended to monitor statistics regarding the reconfigurable unit and periodically use this data to switch between the eviction path or the software dispatch path. We demonstrated this working for our prototype system using the number of failed loads in proportion to the average scheduling period.

The overall result is that we have successfully demonstrated a way for an operating system to manage a reconfigurable unit, fairly and securely, without applications having to be aware the underlying management.

Chapter 6

Application Development Support

Although we know from the previous chapter that it is possible to manage a reconfigurable processor with an operating system sharing the FPL resource between multiple competing processes, this is not the whole store. To demonstrate the overall system to be practicable, we need to be able to demonstrate that applications can be designed using traditional development techniques for the given domain. Although we are agnostic to how custom instructions are created, either by hand by the programmer, as part of a library of custom instructions, or generated by the compiler, all techniques require a programming model that sets rules for how programs are built ready for execution and details the sequence of interactions between the program and the rest of the system. The aim of this section is to describe such a model. We do not claim that this model is the best model, rather that it is sufficient to demonstrate that the system architecture we propose is complete. We feel that there are more advanced models that could be developed and this is in itself an avenue for further research. For the purposes of this discussion we shall assume that the programmer is responsible for utilising custom instructions; cases where the compiler generates the custom instructions are simply a subset of this case in terms of managing custom instructions.

The simplest way to consider the programming model is to again think about namespace management, considering how custom instructions will be treated as they go from the programming language level to the system calls to register the custom instruction and the actual instructions used to invoke them. At the highest level, the programmer should be able to refer to custom instructions using a symbolic name, just as they do for other objects in a program such as procedures and variables. By the time the application has been fully built these must be translated into process unique CIDs used in system calls and invocation instructions. In between these two levels are various intermediate stages as the program is turned from source to a final executable image in memory, which is the roll of the build system: the compiler, the linker, and the run-time environment. There are numerous possible policies for managing how the namespace is managed at the various levels, of which here we will discuss one, and outline other possibilities in Section 7.

Before discussing the programming model, this chapter starts with a short overview of the process of building a program text from source code and libraries. This overview is not meant to be comprehensive, but lays out the basic mechanisms that the proposed programming model will work with. Following the overview we lay out the requirements for the programming model, and then go on to discuss a suitable model to meet those demands.

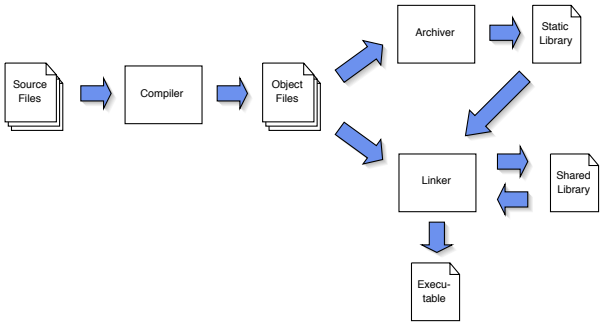

Figure 6.1: Overview of traditional compile and link flow

6.1 Overiew Of Program Construction

The process required to take source files into a program that can be loaded into the system is divided up into several stages. Figure 6.1 shows the stages typically used in a Unix based system, such as producing a.out format binaries for BSD [McKusick et al. 1996] or Executable and Linking Format (ELF) binaries for Linux. The two most important parts are *compilation*, which turn the source code files into binary *object files* containing machine code and data, and *linking* which join together the object files, along with any libraries, into an application image. Libraries can be build either by making a collection of object files (done by *archiving* the files on a Unix system) or by constructing a dynamic library (a task also carried out by the linker). This is the system we will assume for the purpose of this work. Although other systems, such as Windows, use slightly different techniques, the main flow is the same[1].

Although the main focus of compiling and linking is to turn the source code into a set of machine code instructions, these two tools are responsible for mapping the objects used by the programmer as they move from being symbolic references down to memory addresses. In the next four sections we examine the role of each stage, paying particular attention to the namespace management issues, as this is the issue that will need address with respect to custom instructions.

6.1.1 Compilation

At compile time a source file is converted from a textual representation into an object file. An object file contains machine code to carry out the operations described in the source file, data that was included statically in the source file, and meta data about the file that enables it to be linked with other object files at link time. When the source code is translated to machine code, the symbolic names for objects present in the source code will be lost. However, not all the objects referred to by symbols can be found in the one source file (e.g., references to library code), and similarly other source files may wish to reference objects defined in this source file by symbolic name. To solve these problems the compiler generates two sets of information in the meta data section of the object file. The first set of data contains a list of all the symbolic references within the source file that could not be resolved and where in the machine code section the references are needed. The second set lists the location of globally available symbols defined in the source file and their location within the object file. These

[1]There may be more stages involved depending on the language used as well. For example, C code is ran through a pre-processing stage first to expand macros. Such language specific stages are not discussed here, as they have no affect of our work.

two lists are then used by the linker to resolve references between object files when building the final program text.

6.1.2 Static Linking

Static linking is used to generate a stand alone executable image, i.e., one which contains all the code and internal data needed to execute successfully. The linker is presented with a list of object files and libraries that are used to generate the executable image. A statically linked library is notionally no different from a single large object file; it contains unlinked code and data and the meta data needed to link with other object files and libraries. Traditionally on Unix systems static libraries are simply a collection of individual object files in one large file, which is the task performed by the archiver in Figure 6.1.

At link time the linker will attempt to join together the set of object files into a single image and resolve any unbound references using the meta data stored in each file. The linker will first join the files together so that each object has an absolute address associated with it, after which the linker will compare the list of unresolved references with the list of globally exported symbols and attempt to finalise all the references within the program text. For each object file or static library it will look at the list of unresolved references and if it finds an appropriate symbol in another object file or library the linker will modify each instruction that relies on that reference so that it points to the correct address within the image. If not match is found then linking will fail with a suitable warning.

6.1.3 Dynamic Linking

One drawback to static linking is that very common libraries (e.g., the standard C library) will be duplicated in each process's executable image. This may lead to the same code being mapped into physical memory multiple times, wasting memory. One way to tackle this problem is to allow processes to share the read only sections of libraries, so only a single instance is required in memory. To enable this the libraries are no linked at build time, but at either load or run time. This allows the process to use operating system features to ensure that all processes' use the same copy of the library text. This late binding has additional advantages; for instance, it allows libraries to be fixed and all applications dynamically linked to it can take advantage of the new library without needing recompiled. The solutions used to implement shared libraries will depend very much on the structure of the operating system being used (for example how the memory system works) and the facilities it can provide to the application. The structure of the solution also depends on whether the dynamic libraries are loaded when the process first starts up or during the execution as needed. Here we will concentrate on how dynamic libraries work in Unix [Gingell et al. 1987], which uses a load time mechanism.

Dynamic linking is essentially the same task as static linking, only done at a later time. Applications and shared libraries are built with unresolved symbol references in them, which must be resolved at either load time or run time. The shared libraries a process uses need to be mapped into the process's address space and then unresolved symbols matched with global symbol names, and then the appropriate links made in the program to match up the references to the objects. If there are any unresolved links then the application will exit with an appropriate error message. However, there are more complications with dynamic linking than static linking when it comes to namespace management. Unlike static linking where object files and libraries will have a fixed place in the final application, shared libraries can make no such assumption. Libraries need to support being mapped into an arbitrary (from the library's point of view) locations of the address space as it can not know

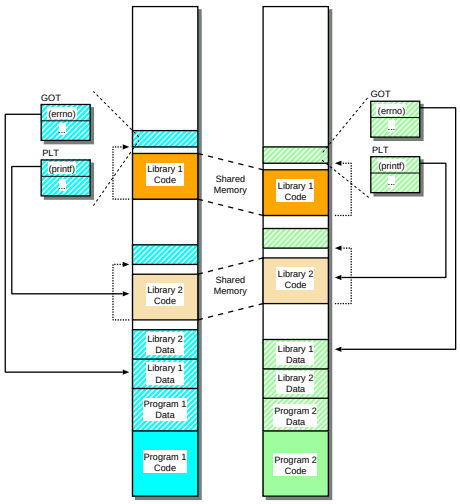

Figure 6.2: Shared libraries in a multiple address space operating system

where in an application it will fit in, as different applications are of different sizes and have different sets of libraries loaded.

The linking process is handled by the Run-Time Environment (RTE) when the process starts. Shared libraries are loaded into the process's address space using the operating system's ability to memory map file with a copy-on-write facility (this is also how shared libraries are handled in Windows 2000 [Solomon & Russinovich 2000]). This mechanism allows applications to shared the same copy of a file in memory, with only pages that have been modified having a private copy in a given process's address space. This means that the unmodified parts of the library (text and constant data) are shared between processes, and only parts that are modified (normal data and text modified during linking) need be duplicated for each process. Within a shared library all references that can be resolved at compile time use Position Independent Code (PIC). PIC means that references are done using relative rather than absolute addresses.

Once the shared libraries used by a give application have been loaded into memory, a link stage has to occur to bind the unresolved references both in the application and the libraries to the objects they refer to. Each shared library will contain the appropriate meta data to link symbolic names for objects with their offset within that module. Using this information then the library could be linked in the same way that static linking occurs, with the instructions referring symbols being patches to use the appropriate address. However, this could potentially lead to a lot of the shared libraries' pages being modified and therefore duplicated, negating one of the major benefits of shared libraries. The solution use for this is to send references to between modules through module specific indirection tables: a Global Offset Table (GOT) for global data, and a Procedure Linkage Table (PLT) for procedure calls, as shown in Figure 6.2. Each indirection table contains an address of the actual object needed for each external reference made in that module. This at link time only the GOT and PLT need to be modified and the majority of the process image is untouched. The GOT needs to be completely resolved at load time, as there is to easy way to trap data references, but the PLT is actually evaluated lazily, with procedure references only being resolved when, and more importantly *if*, they are needed. At load time the PLTs are set to refer to a procedure that will complete the linking for that symbol and the return execution to where it was before the look up was attempted.

The operating system never needs to get involved in dynamic linking, beyond supporting the

memory mapped file functionality (indeed, this was one of the original design goals for implementing shared libraries under Unix). The application code does not get involved with the linking, the load time linking is entirely handled by the process's RTE. This means that the programmer is agnostic to what linking will be used to generate the final program. Dynamic libraries are generated using the static linked, and differ mainly by not having a main entry point defined.

6.1.4 The Run-Time Environment

The RTE consists of the support code that gets linked with an application required to set up the environment that the process expects to use. For example, the C RTE, amongst other tasks, is responsible for setting up the heap ready to be used for malloc/free and opening FILE structures for the three standard file descriptors. The RTE is the first code that gets executed when a new process is created, and only once it has finished creating the environment does it call the what the program designated as the main entry point for their program (the function main in C). Similarly, at the other end of the program, the RTE is responsible for shutting down the process once the program finishes, releasing resources and setting the processes return value if necessary. The dynamic linkage mechanisms that are invoked during run time binding are also part of the RTE.

6.2 Requirements

The aim in this section is to examine how these existing techniques can be extended to include support for managing custom instructions. The most important issue that has to be dealt with is managing the namespace for custom instructions as they are translated from human readable symbolic names down to process specific CIDs. We want to decouple the programmer from the problem of providing specific CIDs, just as programmers do not allocated specific addresses to variables and procedures. Allocating CIDs manually will cause a problem with module integration (clashes in the namespace), will place an extra burden on the programmer, and is likely to lead to program maintenance problems in the long term. Instead, the job of assigning CIDs should be the job of the linker, which is the only time when the assertions about the entire CID namespace can be successfully made. The model needs to work with both static and dynamic linking.

In this system a new type of symbolic reference will be needed to allow programmers to name and reference custom instructions. The compiler will need to support this new type of naming, and be able to generate the appropriate meta data when generating object files. The linker will then be charged with resolving references with custom instructions held in other object files and libraries.

Registering custom instructions with the operating system should be handled by the language's RTE. Although technically the programmer could be made responsible for registering and unregistering instructions, this requires the programmer to be completely aware of what custom instructions they are using, which they may not be if custom instructions are used in a library. It will be the RTE's job to ensure the integrity of the CID namespace.

As part of the tool chain we assume the existence of a hardware compiler that will take a Hardware Description Language (HDL) implementation of the custom instruction hardware and output the resultant bitstream or pair of bitstreams (one static and one stateful) as an object file. This object file must export a global data symbol for each bitstream part, so that the linker can construct the custom instruction record successfully at the final link stage. It should also export an interface definition for use with the software source (e.g., a C header file) so that the user can define a custom instruction based on this. The compiler should also generate the hash used by the operating system for sharing

the custom instruction, either as a globally exportable symbol or as a constant in the interface definition file. If an instruction bitstream is instantiated as part of a custom instruction definition more than once in either a executable image or a shared library, then at some point before execution each instance will need to be provided its own copy of the stateful part of the bitstream.

6.3 The Programmer's View

We start by considering the programmer interface, specifically looking at how the user specifies and uses custom instructions. Because we are introducing a new type of symbolic reference, we assume that the language will need to somehow support this. Symbolic names for custom instruction have different semantics from the symbolic names used for procedures and variables so must be considered as an addition to the language. For the purposes of this discussion we demonstrate how assembler language could be changed to use the new concepts; how individual languages support the new semantics is left for further research.

As discussed in Section 5.1, from a usage point of view, a custom instruction is seen as an atomic unit containing the circuit bitstream, the software implementation, and any associated meta data. Generating these structures with a symbolic name needs be part of the language. The compiler will need to support both a way for the programmer to specify a custom instruction and understand the special semantics associated with it. In something as low as assembly it simply requires a new command with a parameter list consisting of the symbolic name for the custom instruction, the symbolic name(s) of the circuit bitstream data, a symbolic name for the software alternative (or a null reference) and then constants for any meta data. Just like other global symbols, a custom instruction should only be instantiated once with a given symbolic name. When this file gets compiled it will contain a custom instruction data structure, which should become an externally accessible symbol similar to procedures and variables, so that it can be referenced from other source files.

To use a custom instruction the programmer should only need to use a symbolic name (and ensure that the actual instantiation is given in one of the modules at link time). In assembly a programmer would write:

```
exec_rfu  alpha_blend_inst, cr0, cr1, cr2
```

The compiler and linker would eventually resolve the symbolic name to a registered CID by run time.

6.4 Compilation

Several changes need to be made to the compiler to enable users to work with custom instructions. The main change structurally will be the need to track two types of symbolic reference: those for memory addresses (as used by global variables and functin calls) and those referring to custom instructions. We consider these in order of reference.

Firstly, the compiler needs to be altered to understand (though an language construct) that a given function is intended to be used as a software alternative. This function will then be compiled with the modified ABI, as discussed in Section 4.1.5. The compiler can also check that the function's interface is correct for the given architecture.

The second change is for the compiler to support the creation of custom instructions, as outlined in the previous section. Where custom instructions are defined, the compiler will add a new custom

instruction structure in the data section of the object file with references to the various parts being resolved is held locally (only possible for the software alternative) or at link time. A second exported symbol table will be created in the object file's meta data part containing a list of the symbolic names for custom instructions and an associated pointer to the custom instruction structure within the data section. At this point instructions have not had CIDs associated with them, as it is only possible to know what CIDs are valid at the final link stage.

When custom instructions are used within a source file, the compiler does not have enough information to generate the actual instruction that is used to invoke the custom instruction, as there is no way of knowing what CID to use. A prototype instruction can be generated, which will contain all the correct operands except the CID to be used. The compiler will generate a second list of symbols needed to link this object file, similar to the list used for global variables and procedure calls referenced but not found in that source file. This second list will contain an entry for each instance of a custom instruction being used, noting its location with in the machine code and the symbolic name of the custom instruction. This information will be used at link time to insert the correct CID into the instruction before execution. Note that even if an custom instruction is defined and used in the same file the compiler can not resolve the invocation unlike it would do for memory references. This is because there is no analogous notion with CIDs for relative addressing.

6.5 Static Linking

The static linker in our system needs to be extended in two ways: firstly, once the object files for the program have been brought together, it will need to be able to assign each custom instruction used a CID, bind custom instruction invocations to absolute CIDs, and build the appropriate data structures to allow the RTE to register and unregister them; and secondly, it will be responsible for if necessary duplicating the stateful part of bitstreams for multiple instances.

This initial task of the static linker will be the same, in that it will coalesce the object files into a single binary image. The next stage, just as it will bring together the list of unresolved memory addresses and the globally exported locations, the linker will bring together the lists of custom instruction definitions and the unresolved invocations. When brining together the lists of unresolved symbols in each module, the resultant list is processed so that each unresolved custom instruction only appears once, to assist later processing. At this stage the linker must ensure that there are no symbolic name clashes (i.e., no custom instruction has been defined multiple times); if there are then linking should fail with a warning.

Once this has been done, the linker can then resolve memory references as before. During this process all the custom instruction records will have their bitstream and software alternative fields completed. The next stage is associate with each custom instruction a CID and resolve invocation references. The list of unresolved custom instruction references is traversed sequentially, and for each entry the symbolic name looked up in the list of custom instruction structures defined in the application image. If there is a match then this custom instruction is resolved otherwise linking fails with a warning. To resolve a custom instruction, a CID is first allocated, taking the next unused CID starting from zero and counting upwards. Once a CID has been allocated all instructions referencing this custom instruction are modified to use that CID, completing the linking of the application code. As each instruction is resolved an entry is added to a list inserted into the program's constant data section that will be used by the RTE to register and unregister the custom instructions. Each entry in the list contains the allocated CID and a pointer to the custom instruction structure, and may also include information like the symbolic name of the custom instruction, which could be useful for

debugging.

6.6 Dynamic Linking

The static linking mechanism needs expanding in order to support dynamic linking. The problems faced with dynamic linking in a Protean system are similar to those found in traditional dynamic linkage, as discussed in Section 6.1.3. When linking both an application using dynamic libraries and the dynamic libraries themselves there is not enough information known about the CID namespace to assign CIDs to custom instructions contained in dynamic libraries; this needs to be done at the final link stage at load time. Here we discuss three possible solutions to the problem. The usefulness of a given approach will depend greatly on the density of instructions that make use of custom instructions held in other modules. It is out with the scope of this work to investigate the likely densities of custom instructions, so we settle for presenting the ideas, and leave investigation as to the suitability of each technique as further research. To demonstrate the possible range, consider a library that contains an alpha blend instruction, which will typically be called in a loop for a few discrete locations within a program, compared to a library of MMX instructions, which will be used more liberally as building blocks for an algorithm, and thus be used more frequently.

In this work we make one simplifying assumption that differs from that made in traditional shared library work, such as [Gingell et al. 1987]. Traditional shared libraries use indirection tables for accessing procedures and global data in other modules in order to reduce the number of modified pages in a shared library, but here we do not do that: we assume that it is sufficient to modify the instructions invoking custom instructions in their location. The first concern is that adding an indirection table will involve at least two branches, which could potentially involve two pipeline flushes, depending on how the branch prediction logic works. In addition for the case where there was a high density of instructions used the indirection tables may become a significant proportion of the size of the library. Finally we assume the main benefit from shared libraries containing custom instructions not to be the memory saving but the benefit of sharing the custom instructions themselves, which will bring potential benefits from reduced contention on the reconfigurable execution unit.

6.6.1 Straight Link Approach

The obvious way to handle the dynamic linking is to simply extend the approach used for static linking. Both the application and dynamic libraries are built with the list of unresolved custom instructions and associated lists of module relative locations where the custom instructions are used, and the list of custom instructions they export with a symbolic name and relative offset into the module for the custom instruction structure. At load time once the dynamic libraries have been mapped into the process's address space by the run-time environment, the run-time environment will then carry out the linking that previously the linker would have done in a static system.

This is obviously a simple approach, but has one main drawback: it resolves all instructions that require fixing, rather than those that just will be used. One of the advantages of the lazy approached of resolving PLT entries is that only those procedure calls that get used are resolved. If an application links against a large library (e.g., a graphics library) that uses another library of custom instructions (say an MMX style library), but the application only uses a small part of the original library, then a lot of unnecessary linking may occur.

```
typedef struct LTTAG
{
  addr_t instruction_location;
  word_t prototype;
  cid_t  cid;
  char** custom_instruction_name;
} CI_LINK_TABLE_ENTRY;
```

Listing 6.1: Dynamic link structure

6.6.2 Demand Link Approach

The solution of the problem of unnecessary linking is to have a more lazy approach, whereby processor instructions using custom instructions are linked on their first invocation. This means that no unnecessary linking will occur, and no extra linking will happen at load time delaying the startup of the process. This technique does not do away with all load time processing though, as the technique still requires the RTE to assign CIDs (discussed below) and provide the mechanism with a list of custom instructions (by their symbolic name) and their location within the process's address space once all the shared libraries have been mapped. Once these tasks are done however, the process proceeds as if there were no unresolved custom instructions.

At link time it is useful to have a complete map of the custom instructions in a process, thus the RTE will build such a list at the start. The link stage that built each module will insert a structure containing the symbolic name and relative offsets within the module of each custom instruction structure contained within. At load time the run-time environment will collate these lists into one process specific list containing the symbolic names and an absolute address within the process's address space for each custom instruction. In addition it will assign each instance a CID that is unique to this process in a similar fashion to that used for static linking, and will set a boolean field to false to indicate that this instruction has not yet been registered with the operating system.

The linking is done by replacing the processor instructions that use custom instructions in the modules with branches to a module specific linking procedure. When an unresolved instruction is reached the process will branch to this link procedure which will then use a module specific table of meta data to resolve the link and restart execution in the appropriate place. Each entry in the table consists of the structure shown in Listing 6.1. The first value in the structure contains the module relative address of the instruction that needs to be resolved. Entries in the table are stored by the linker that built the module in order of this field. The link procedure called at run time then searches the table based on this list (the list is ordered so that the procedure may do a binary search rather than a linear search). The prototype field contains the processor instruction that will replace the branch instruction in the program text, minus the CID.

What the linker does once it has found the structure relating to this branch depends on the contents of the CID field. If this is an invalid CID then the link procedure will try to find a matching entry for the symbolic name pointed to by the custom instruction name field in the global custom instruction map. If no match is found then the process is terminated with a suitable error message, otherwise linking occurs. The link procedure first checks the field in the global map to see if this instruction has been registered before, if not then it will register the custom instruction with the appropriate CID value and then set the registered field in the map to true, and if the field was already true it will do nothing more in that respect. Once the link procedure holds a valid registered CID for the instruction it will insert the CID into the prototype field, which it will then copy into the process image at the place the fault occurred. At this point linking is complete and execution can be resumed, but first as

an optimisation the linker will walk through the module's table containing the instruction resolution information and insert the CID into each one which uses the same symbolic name, so that next time an instruction of this type is required in that module the link procedure does not need to search the global custom instruction map. This done the faulting instruction returns execution to the point at which the fault occurred.

Whilst this solution removes the problem of time wasted due to unnecessary linking, it significantly increases the cost per link: the cost of invoking the link procedure will incur several branches, a possible string comparison, and a cache flush when the branch instruction is modified.

6.6.3 Block Link Approach

A third solution then is to modify the demand link approach to work on a scale larger than an individual instruction but smaller than an entire module. The `CI_LINK_TABLE_ENTRY` structure is extended to include a next field, which is used to generate a circular list of entries referring to the same custom instruction within a given scope, such as a basic block or a procedure. When an instruction fault occurs the link procedure not only resolves the faulting instruction, but all the other instructions on the circular list, reducing the number of times an instruction fault occurs. The circular nature of the list is needed for block sizes greater than that of a basic block, as we can not typically make statements about the order in which a block will be processed.

6.7 Run-Time Environment

The role of the RTE, as stated already, is to ensure that the application code runs in a consistent environment. For the programming model we have defined this means that the RTE must ensure that the CID space is valid before the programmer's code is entered. Thus the RTE is responsible for ensuring that before a custom instruction is used that the instruction has been registered with the operating system. With static linking this involves traversing the list of custom instructions built by the linker and registering each one with the operating system, and for dynamic linking providing the run time binding mechanisms described above.

The RTE is also charged with ensuring that no two custom instruction definitions that refer to the same bitstream use the same stateful part of the bitstream. Each custom instruction definition that references a bitstream will need its own copy of the stateful bitstream data, but the only time at which sufficient information is present is at the final link stage. For a load time dynamic linking system, just before the RTE registers custom instructions, it will check an additional status bit stored with each bitstream, which is initially zero. If the status bit is zero then the linker will set the bit to one before registering the custom instruction with the operating system. If the status bit is already set to one, then this bitstream is already in use, so a separate copy of the stateful part of the bitstream is needed. The RTE will duplicate the stateful part of the bitstream within the process's memory image (which it is safe to do so as the custom instruction using this originally can not have yet been executed), patch the memory reference in the custom instruction structure, and then registers the instruction.

It should be noted that the above mechanism is insufficient for run time loading of libraries. It may be that by the time a library is loaded that a custom instruction may already have been used and the stateful part of the bitstream has already been modified. In this circumstance it is required that the stateful part is always duplicated to ensure that there is always an unmodified copy. This could potentially be expensive in terms of space however, as we expect the times when multiple instances of one bitstream are used (rather than just using the one copy of an instruction many time) to be

infrequent.

6.8 Summary

This section has described how custom instructions can be fitted into the traditional Unix style compiler and link process, supporting the traditional techniques for statically and dynamically building processes. A summary of the whole flow can be seen in Figure 6.3. Although we have extended the programming model to include the notion of a custom instruction and the suitable symbolic representation, very little needed to be changed to be supported on our architecture. In particular, no further modifications to the underlying hardware were needed, suggesting that the current CID based abstraction is a reasonable one.

Although the manpower was not available to implement a compiler to produce binaries following the above model, the dynamic linking methods were all tested by modifying the Intercal compiler discussed in Section 4.3.2.4 to use lazy linking rather than static linking. These successfully demonstrated the correctness of the lazy linkage methods.

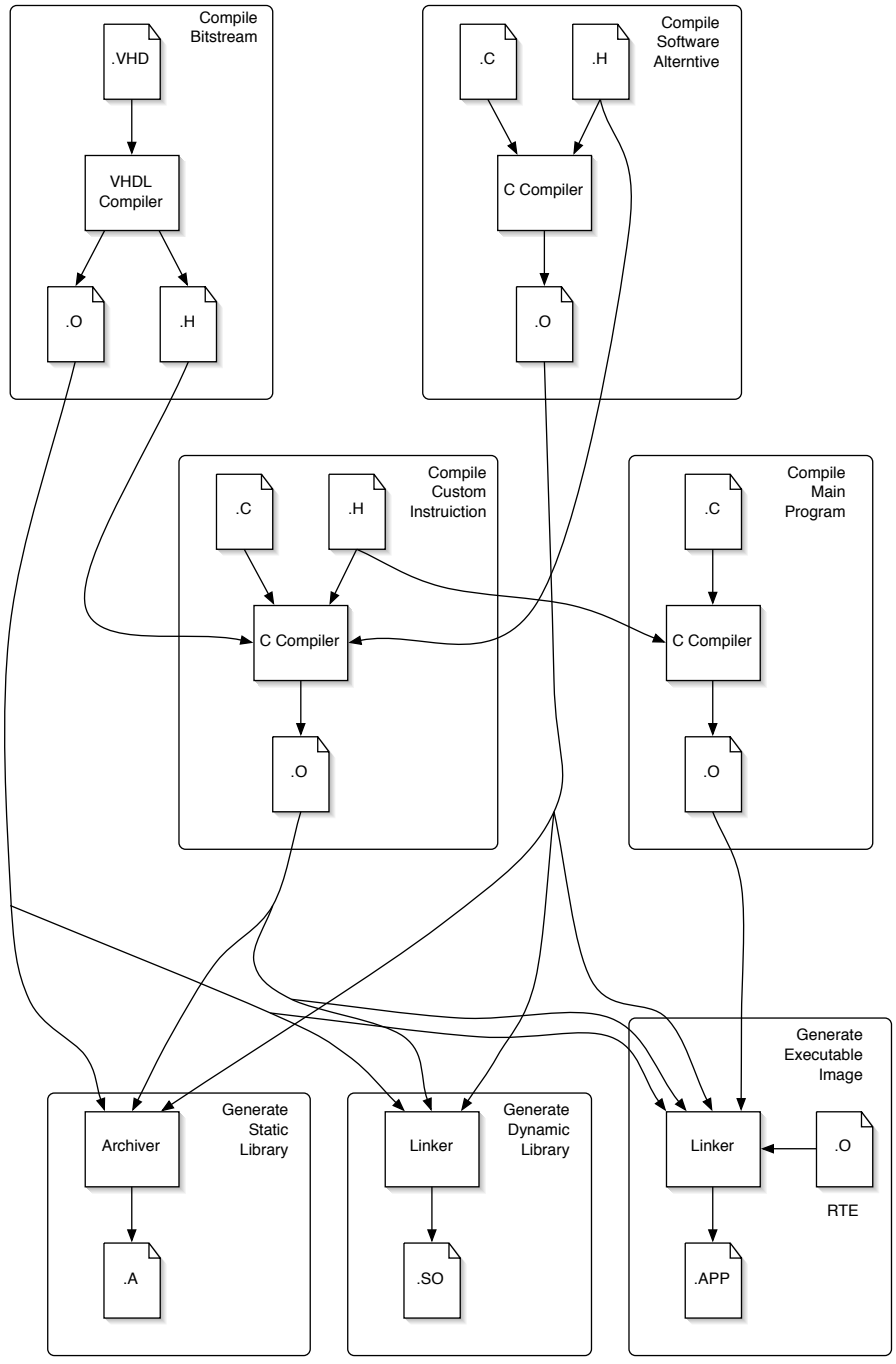

Figure 6.3: Source building flow for new programming model

Chapter 7

Further Work

The Proteus Architecture and Operating System is only intended as a prototype to initially explore feasibility of the use of FPL technology in a workstation environment.

7.1 Architecture

7.1.1 Segmentation Based Architecture

In order to simply the operating system's management problems the FPL resource was divided into a fixed sized set of PFUs. Although allocating circuits into a two dimensional FPL array on an arbitrary basis is a computationally hard problem, the problem becomes simpler if the two dimensional allocation problem is simplified to a one dimensional problem with the FPL being allocated on a single plane, as is done in DISC [Wirthlin & Hutchings 1995] for instance. The operating system would have a much more difficult problem when scheduling the circuits loaded onto the array as it would suffer from fragmentation, though solutions to ease this problem have been discussed [Brebner & Diessel 2001].

7.1.2 Embedded Systems Targeting

Although the aim of this system has been to develop a mechanism that can be used in a general purpose workstation, the a small version of the device may be useful for embedded systems work. One of the advantages of virtualising the FPL resource is that it decouples the software invocation calls and the hardware placement. Currently in embedded systems these two parts are tightly coupled, and the use of the dispatch mechanism with say only a couple of PFUs could prove a solution to the coupling problem for embedded systems. The question is whether the fixed partitioning would prove more problematic at the embedded systems level.

7.1.3 Integration Into a Workstation Modern CPU Core

In this work we have successfully integrated a reconfigurable execution unit into a simple microprocessor core, but modern microprocessor cores are much more complicated devices. It would be interesting to implement a simulation of the core in a larger superscalar, out-of-order device as done for the third generation of OneChip [Carrillo & Chow 2001]. Of particular interest is how the unknown and unbounded latency of custom instructions would complicate the dispatch and retirement

of instructions. In addition the much faster clock speed on a modern device would lead to higher overheads in terms of work that could otherwise be done when managing the FPL devices.

7.2 Operating System

7.2.1 Further Load Testing

The simulated load on our operating system is quite simple, and ignores interactions with other devices. Applications never block for I/O or paging for instance. This has two effects on the results. Firstly the simulated runs for completion time against number of processes must be considered a worst case scenario, because all the processes are constantly processing data and can always load their custom instructions immediately. If applications blocked occasionally waiting for data to process or to output already processed data, then the demand for PFUs over a microscopic period would be reduced. Conversely, the latency for loading a custom instruction is artificial, unless the operating system keeps all circuit descriptions held in physical memory. This is quite possible: BSD 4.4 keeps all such data in kernel memory whilst they may be used, and the RAGE FPL middleware system [Burns et al. 1997] kept circuit bitstreams in memory too. However, it would be interesting to see the affect of performance in circuit descriptions were held in paged memory.

7.2.2 Integration With Process Scheduler

Because the FPL resource on a process is scarce and can become a bottleneck when a large number of applications try and use it simultaneously (comparable to the early virtual memory systems that had little physical memory), the behaviour of the system relies on the Custom Instruction Scheduler and the process scheduler working in conjunction. In the prototype system we only implemented a simple single level pre-emptive scheduler, but a real operating system will use a more complicated algorithm, based on factors such as process priority, other resource usage statistics, etc. [McKusick et al. 1996]. Work needs to be done on examining how to best integrate the scheduling of applications with custom instructions with such a system.

Similarly, the process scheduler could be made more aware of how the FPL resource is being used. If a set of processes share custom instructions, then it would make sense to schedule such processes in sequence, though this could become quite complex if the applications share more than a single common set of custom instructions.

7.2.3 Advanced Management Interface

The basic register and use model of custom instruction interaction is quite limiting. If the process could tell when there were more PFUs available for use then it could attempt to parallelize the operating if suitable (e.g., the alpha blending example could then process multiple pixels in parallel). The drawback with such a system is that it is hard for the application to dynamically scale its usage in response to performance. The operating system on the other hand should have the right information to manage this.

One solution then would be to have the operating system perform the equivalent of a functional programming style map operator. The process could present a set of input buffers and an output buffer to the the operating system along with a custom instruction. The operating system would then attempt to run as many parallel instances as possible without causing PFU contention.

Another interface expansion would be to allow applications to express grouping of custom instructions. For example, the audio echo application would benefit if the operating system was aware that it needed to use both its custom instructions within a tight loop.

7.3 Programming Model

7.3.1 Compiler Support

There are many ways of generating custom instructions for use with applications, either by hand crafting the custom instructions using knowledge of how the application works, using prebuilt libraries of custom instructions, or using a compiler. Several projects, such as PRISC and CHIMERA, developed compilers.

7.3.2 CID Namespace Management

The programming model shown in Chapter 5 is very simple, whereby the CID namespace is constant throughout the life of an application. There is scope for research here looking at what other models could be used, and how suitable they are. For instance, it might be useful to support remapping of CIDs to other instructions at run time, to allow for polymorphic instructions. For example, the application could just used a "filter" instruction, which is mapped onto various different filter instructions throughout the lifetime of the application.

Another drawback is that if the application is linked with a large library of instructions it may run out of CIDs. For instance, SIMD instruction sets often go into the hundred plus instruction range (e.g., the Motorola AltiVec instruction set has 160 instructions), but on the ProteanARM there were only 128 possible CIDs. A more limiting CID static linking mechanism may be needed, or the ability to recycle CIDs at run time.

7.4 Other Observations

The practicality of this work has been based on the assumption that there is a low density of active custom instructions per process at any given time. If there are a large number of custom instructions being used then applications will fault back to software for the majority of the time, which may prove to be a hindrance as the procedure call mechanism for a software alternative is more time time consuming than a regular function call, which had it been designed as a software only function would possibly have been further optimised by the compiled through techniques such as inlining.

Chapter 8

Conclusion

In this dissertation we have examined the problems associated with using a reconfigurable processor as part of a system for use in a general purpose workstation environment, considering the support needed at the hardware, operating system, and application level. Perviously all work with reconfigurable processors has either focused on embedded systems or just the problems involved in connecting the FPL to the processor's datapath. The contribution of this dissertation is to make an initial exploration of the problems involved in using a reconfigurable processor in this complex environment. A solution to the problems has been proposed in the form of the Proteus Architecture, the Custom Instruction Scheduler defined for the POrSCHE kernel, and a new programming model to allow applications to be developed for this system.

Chapter 2 outlines the relevant background material and previous research in the fields associated. It starts with an introduction to Field Programmable Logic (FPL)

In Chapter 3 we lay out what we considered to be the requirements of using a reconfigurable processor in the environment of a general purpose workstation. Adding a new resource such that it can be easily utilised and managed requires consideration at the hardware, operating system, and application level. Drawing on the existing body of research and development in the field of reconfigurable processors and FPL management outlined in Chapter 2,

Chapter 4 introduced the Proteus Architecture, a proposal for a general architecture which supports being managed by an operating system. It also introduced the ProteanARM, and ARM based prototype implementation of the Proteus Architecture. The architecture model describes adding the FPL resource as a set of fixed sized blocks, referred to as Programmable Function Units (PFUs), in their own function unit within the processor. Instructions to invoke circuits loaded in the FPL do not invoke PFUs directly, but instead go through translation hardware, which allows the operating system to load circuits in to arbitrary PFUs without needing to alter the software using it. The dispatch mechanism also supports the use of software alternatives to circuits, allowing the operating system to move applications' custom instructions between hardware and software without their knowing.

This chapter demonstrated that the basic architecture model was sufficient for individual applications running on the processor to gain a performance benefit. It introduced a set of sample applications applications from different domains which utilised custom instructions to accelerate their core algorithms. Each applications was demonstrated to benefit from using custom instructions on the bare ProteanARM prototype without any other applications running.

In Chapter 5 we introduced the concept of a custom instruction being a rich data structure containing may parts, compared to the traditional view of a custom instruction being just a configuration bitstream. By making a custom instruction richer, we allow the programmer and the operating sys-

tem to use a single reference to refer to all information about a custom instruction: the stateful and stateless parts of the bitstream (separated to facilitate faster circuit switching and circuit sharing), the software implementation, and associated metadata.

After this we provide demonstration of an operating system managing a reconfigurable processor by taking a basic multitasking kernel and extending it to include management of custom instructions. The system has demonstrated that it is possible to manage the FPL resource between a set of competing processes without prior knowledge of their usage patterns such that applications still get an advantage from using the FPL resource, even at times of contention, and that all processes will be able to make progress. This is achieved through the use of an FPL scheduling system which is called as requests for FPL blocks are made and a periodic statistic gathering system which is used to direct allocation policy.

We then described a suitable programming model that would allow applications to be developed to run on a Protean system in Chapter 6. The aim of this is to demonstrate that programming software for a system using custom instructions does not require a radical new programming architecture, rather just an evolution of existing techniques. The model was devised to allow programmers to specify custom instructions by a symbolic name, just as they already do for functions and variables, and then for the compile and link stages to translate the symbolic name down to a CID used to invoke the custom instruction at the machine code level. This is done within the context of the traditional compile and link mechanism, including support for both static and shared libraries that both use and provide custom instructions.

Finally, in Chapter 7 we outlined possible interesting avenues for future work that could be pursued based on the work carried out in this research.

The result of this work has been to demonstrate that it is possible to build a FPL resource into a processor such that it can be managed by an operating system in order to allow multiple applications to utilise custom instructions without needing to be aware of each other's behaviour. We have also shown that this can be done in a secure fashion with suitable hardware support to prevent applications from interfering with each other or the overall stability of the system.

In addition to successful publication of the research, a measure of the success of this work is shown in the interest by engineers from ARM in the the dispatch mechanism, which they were interested in applying to a smaller ProteanARM like device for embedded systems.

Bibliography

Abrash, M. [1996], 'Ramblings In Real Time', *Dr. Dobbs Source Book* (November/December, 1996). Miller Freeman, Inc.

Aho, A. V., Sathi, R. & Ullman, J. D. [1986], *Compilers: Principles, Techniques, and Tools*, 1st edn, Addison–Wesley Publishing Company.

Altera [2001], *ARM Based Embedded Processor Device Overview*, Altera.

ARM [1996], *ARM 7500FE Data Sheet*, DDI 0077B edn.

ARM [2000], *ARM Architecture Reference Manual*, DDI 0100D edn.

Belady, L. A. [1966], 'A Study of Replacement Algorithms for a Virtual–Storage Computer', *I.B.M. Systems Journal* **5**(2), 78–101.

Black, R. J. [1994], Explicit Network Scheduling, PhD thesis, Churchill College, University of Cambridge.

Brebner, G. [1996], A virtual hardware operating system for the Xilinx XC6200, *in* R. W. Hartenstein & M. Glesner, eds, '6th International Workshop on Field Programmable Logic and Applications', Vol. 1142 of *Lecture Notes in Computing Science*, Springer–Verlag, pp. 327–336.

Brebner, G. & Diessel, O. [2001], Chip–based Reconfigurable Task Management, *in* G. Brebner & R. Woods, eds, '11th International Conference on Field Programmable Logic and Applications', Vol. 2147 of *Lecture Notes in Computing Science*, Springer–Verlag, pp. 182–191.

Burns, J., Donlin, A., Hogg, J., Singh, S. & de Wit, M. [1997], A dynamic reconfiguration run–time system, *in* J. Arnold & K. L. Pocek, eds, 'Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines', Napa, CA, pp. 66–75.

Carrillo, J. E. & Chow, P. [2001], The effect of reconfigurable units in superscalar processors, *in* 'FPGA', pp. 141–150.

Compaq [1999], *Alpha 21264 Microprocessor Hardware Reference Manual*, Compaq Computer Corporation.

Dales, M. [1999], The Proteus Processor — A Conventional CPU with Reconfigurable Functionality, *in* '9th International Workshop on Field Programmable Logic and Applications', pp. 431–437.

Dales, M. [2001], Initial Analysis of the Proteus Architecture, *in* '11th International Conference on Field Programmable Logic and Applications', pp. 623–627.

Dales, M. [2003], Managing a Reconfigurable Processor in a General Purpose Work station Environment, *in* 'Design, Automation, and Testing in Europe'. to be published.

Diessel, O. F. [1998], On Scheduling Dynamic FPGA Reconfigurations — A Partial Rearrangement Approach, PhD thesis, The Department of Computing Science and Software Engineering, The University of Newcastle.

Donlin, A. [1998], Self Modifying Circuitry — A Platform for Tractable Virtual Circuitry, *in* '8th International Workshop on Field Programmable Logic and Applications', Vol. 1482 of *Lecture Notes in Computing Science*, Springer–Verlag, pp. 199–208.

Faura, J., Moreno, J. M., Aguirre, M. A., van Duong, P. & Insenser, J. M. [1997], Multicontext Dynamic Reconfigurable and Real Time Probing on a Novel Mixed Signal Programmable Device With On–Chip Microprocessor, *in* W. Luk, P. Y. Cheung & M. Glesner, eds, '7th International Workshop on Field Programmable Logic and Applications', Vol. 1304 of *Lecture Notes in Computing Science*, Springer–Verlag.

Furber, S. [1996], *ARM Sytem Architecture*, Addison Wesley Longman.

Gilson, K. L. [1994], Integrated circuit computing device comprising a dynamically configurable gate array having a microprocessor and reconfigurable instruction execution means and method therefor. United States Patent 5,361,373.

Gilson, K. L. [1997], Integrated circuit computing device comprising a dynamically configurable gate array having a microprocessor and reconfigurable instruction execution means and method therefor. United States Patent 5,600,845.

Gingell, R. A., Lee, M., Dang, X. T. & Weeks, M. S. [1987], Shared Libraries in SunOS, *in* 'Proceedings of the USENIX 1987 Summer Conference', USENIX, pp. 131–145.

Graham, P. & Nelson, B. [1999], Reconfigurable Processors for High–Performance, Embedded Digital Signal Processing, *in* P. Lysaght, J. Irvine & R. Hartenstein, eds, '9th International Workshop on Field Programmable Logic and Applications', Vol. 1673 of *Lecture Notes in Computing Science*, Springer–Verlag, pp. 1–10.

Guccione, S. A. & Levi, D. [1998], XBI: A Java-based interface to FPGA hardware, *in* 'Proceedings of SPIE: Configurable Computing Technology and Applications'.

Hadžić, I., Udani, S. & Smith, J. M. [1999], FPGA Viruses, *in* '9th International Workshop on Field Programmable Logic and Applications', pp. 291–300.

Hartenstein, R. W., Hirschbiel, A. & Weber, M. [1989], Xputers — An Open Family of Non von Neumann Architectures, Technical Report 195/89, Universität Kaiserslautern, Fachbereich Informatik, Postfach 3049, D–6750 Kaiserslautern.

Hartenstein, R. W., Schmidt, K., Reinig, H. & Weber, M. [1991], A novel compilation technique for a machine paradigm based on field-programmable logic, *in* W. Moore & W. Luk, eds, 'FPGAs', Abingdon EE&CS Books, Abingdon, England, pp. 255–269.

Hauck, S. [1998], Configuration Prefetch for Single Context Reconfigurable Processors, *in* 'Proceedings of the sixth Internation Symposium on Field–Programmable Gate Arrays', ACM, pp. 65–74.

Hauck, S., Li, Z. & Compton, K. [2000], Configuration Caching Techniques for FPGA, *in* 'IEEE Workshop on FPGAs for Custom Computing Machines', IEEE.

Hauser, J. R. & Wawrzynek, J. [1997], GARP: A MIPS processor with a reconfigurable coprocessor, *in* 'Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines', pp. 12–21.

IBM [1999], *PowerPC 740/PowerPC 750 RISC Microprocessor User's Manual*, IBM.

Infineon [2000], 'Infineon introduces configurable CARMEL DSP Core for 3G wireless and broadband communication applications', Infineon Press Release.

Intel Corporation [1998], *Intel Architecture Software Developer's Manual*, Volume 2: Programmer's Reference Manual edn, Intel Corporation. Order Number: 243192.

Intel Corporation [2000], *Intel XScale Core Developer's Manual*, Intel Corporation. Order Number: 273473–001.

Intel Corporation [2001*a*], *IA–32 Intel Architecture Software Developer's Manual*, Volume 3: System Programming Guide edn, Intel Corporation. Order Number: 245472.

Intel Corporation [2001*b*], Using MMX Instructions to Implement Audio Echo Sound Effects. Unpublished Technical Report.

ISO/IEC [1999], *International Standard ISO/IEC 9899*, International Standards Organization.

James-Roxby, P. & Guccione, S. A. [2001], Automated Extraction of Run–Time Parametrisable Cores from Programmable Device Configurations, *in* 'Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines', IEEE, pp. 153–161.

Lister, A. M. [1975], *Fundementals of Operating Systems*, The Macmillan Press.

Ludwig, S., Slous, R. & Singh, S. [1999], Implementing Photoshop™ Filters in Virtex™, *in* P. Lysaght, J. Irvine & R. Hartenstein, eds, '9th International Workshop on Field Programmable Logic and Applications', Vol. 1673 of *Lecture Notes in Computing Science*, Springer–Verlag, pp. 233–242.

MacVicar, D., Patterson, J. & Singh, S. [1999], Rendering PostScript™ Fonts on FPGAs, *in* P. Lysaght, J. Irvine & R. Hartenstein, eds, '9th International Workshop on Field Programmable Logic and Applications', Vol. 1673 of *Lecture Notes in Computing Science*, Springer–Verlag, pp. 223–232.

McKusick, M. K., Bostic, K., Karels, M. J. & Quaterman, J. S. [1996], *The Design and Implementation of the 4.4 BSD Operating System*, Addison–Wesley Publishing Company.

Motorola [1997], *PowerPC Microprocessor Family: The Programming Environment*, rev 1 edn, Motorola Semiconductor Products Sector.

Motorola [1999], *MPC7400 RISC Microprocessor Technical Summary*, rev 0 edn, Motorola Semiconductor Products Sector.

Razdan, R. & Smith, M. D. [1994], High–Performance Microarchitectures with Hardware–Programmable Functional Units, *in* 'Proc. 27th Annual IEEE/ACM International Symposium on Microarchitecture', pp. 172–180.

Rivest, R. [1992], The MD5 Message-Digest Algorithm, RFC 1321, MIT Laboratory for Computer Science and RSA Data Security, Inc.

Rosenblum, M., Bugnion, E., Herrod, S. A., Witchel, E. & Gupta, A. [1995], The Impact of Architectural Trends on Operating System Performance, *in* 'Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review', pp. 285–298.

Saltzer, J. H. [1978], Naming and Binding of Objects, *in* 'Number 60 in Lecture Notes in Computing Science', Springer–Verlag, pp. 99–208.

Sawitzki, S., Gratz, A. & Spallek, R. G. [1998], CoMPARE: A Simple Reconfigurable Processor Architecture Exploiting Instruction Level Parallelism, *in* 'Proceedings of the 5th Australasian Conference on Parallel and Real-Time Systems'.

Sawitzki, S., Köhler, S. & Spallek, R. G. [2001], Prototyping Framework for Reconfigurable Processors, *in* G. Brebner & R. Woods, eds, '11th International Conference on Field Programmable Logic and Applications', Vol. 2147 of *Lecture Notes in Computing Science*, Springer–Verlag, pp. 6–16.

Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C. & Ferguson, N. [1998], *Twofish: A 128–Bit Block Cipher*. AES Proposal.

Shand, M. [1997], A Case Study of Algoritm Implementation in Reconfigurable Hardware and Software, *in* W. Luk, P. Y. Cheung & M. Glesner, eds, '7th International Workshop on Field Programmable Logic and Applications', Vol. 1304 of *Lecture Notes in Computing Science*, Springer–Verlag, pp. 333–343.

Sidsa [2000], *FIPSOC Mixed Signal System–on–Chip*, SIDSA Semiconductor Design Solutions.

Silberschatz, A., Galvin, P. B. & Peteron, J. [1998], *Operating System Concepts*, 5th edn, Addison–Wesley.

Singh, S., Patterson, J., Burns, J. & Dales, M. [1997], PostScript rendering with virtual hardware, *in* W. Luk, P. Y. Cheung & M. Glesner, eds, '7th International Workshop on Field Programmable Logic and Applications', Vol. 1304 of *Lecture Notes in Computing Science*, Springer–Verlag, pp. 428–437.

Solomon, D. A. & Russinovich, M. E. [2000], *Inside Microsoft Windows 2000*, 3rd edn, Microsoft Press.

SPEC [2000], Systems Performance Cooperative Benchmarks.
*http://www.spec.org/

Sudhir, S., Nath, S. & Goldstein, S. C. [2001], Configuration Caching and Swapping, *in* '11th International Conference on Field Programmable Logic and Applications', pp. 192–202.

Susanto, K. W. & Melham, T. [2000], 'Formally Analysed Dynamic Synthesis of Hardware', *The Journal of Supercomputing* .

Triscend [2000], *Triscend A7 Configurable System–on–Chip Family*, Triscend Corporation.

Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S. & Agarwal, A. [1997], 'Baring It All to Software: Raw Machines', *Computer* .

Wirthlin, M. J. & Hutchings, B. L. [1995], A dynamic instruction set computer, *in* P. Athanas & K. L. Pocek, eds, 'Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines', Napa, CA, pp. 99–107.

Wirthlin, M. J., Hutchings, B. L. & Gilson, K. L. [1994], The Nano Processor: A low resource reconfigurable processor, *in* D. A. Buell & K. L. Pocek, eds, 'Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines', Napa, CA, pp. 23–30.

Wittig, R. D. & Chow, P. [1996], OneChip: An FPGA Processor With Reconfigurable Logic, *in* 'IEEE Workshop on FPGAs for Custom Computing Machines', IEEE, pp. 126–135.

Woods, D. R., Lyon, J. M., Howell, L. & Raymond, E. S. [1996], *The Intercal Programming Language Revised Reference Manual*.

Xilinx [1996], *XC6200 Field Programmable Gate Arrays*, Xilinx.

Xilinx [1999], *The Programmable Logic Data Book 1999*, Xilinx.

Xilinx [2000], *Virtex Series Configuration Architecture User Guide*, Xilinx, Inc. Application Note: Virtex Series, XAPP151 (v1.5).

Xilinx [2002], *Virtex–II Pro Platform FPGA Handbook*, Xilinx.

Ye, Z. A., Moshovos, A., Hauck, S. & Banerjee, P. [2000], CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit, *in* 'Proceedings of the 27th Annual International Symposium on Computer Architecture', pp. 225–235.

Ylonen, T., Kivinen, T., Saarinen, M., Rinne, T. & Lehtinen, S. [2000], *SSH Transport Layer Protocol*, internet draft edn, Internet Engineering Task Force.