BORPH: An Operating System for FPGA-Based Reconfigurable Computers

Hayden Kwok-Hay So
Robert W. Brodersen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-92

http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-92.html

July 20, 2007

Copyright © 2007, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

BORPH: An Operating System for FPGA-Based Reconfigurable Computers

by

Hayden Kwok-Hay So

B.S. (University of California, Berkeley) 1998
M.S. (University of California, Berkeley) 2000

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Robert Brodersen, Chair
Professor John Wawrzynek
Professor Dorit S. Hochbaum

Fall 2007

The dissertation of Hayden Kwok-Hay So is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2007

BORPH: An Operating System for FPGA-Based Reconfigurable Computers

Copyright 2007

by

Hayden Kwok-Hay So

Abstract

BORPH: An Operating System for FPGA-Based Reconfigurable Computers

by

Hayden Kwok-Hay So

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Robert Brodersen, Chair

Reconfigurable computing is a promising technology to meet future computational demand by leveraging flexibilities and the high degree of parallelism found in reconfigurable hardware fabrics, such as field programmable gate arrays (FPGAs). However, despite their promising performance researchers have demonstrated, reconfigurable computers are yet to be widely adopted. One reason is the lack of a common and intuitive operating system for these platforms.

This dissertation work explores the design and implementation trade-offs of an operating system for FPGA-based reconfigurable computers, BORPH, the Berkeley Operating system for ReProgrammable Hardware. The goal of this work is to explore and demonstrate the feasibility of providing a systematic and easy to understand view into reconfigurable computers through OS support without incurring significant performance penalties.

BORPH provides kernel support for FPGA applications by extending a standard Linux operating system. It establishes the notion of hardware process for executing user FPGA applications. Users therefore compile and execute hardware designs on FPGA resources the same way they run software programs on conventional processor-based systems. BORPH offers run-time general file system support to hardware processes as if they were software. The unified file interface allows hardware and software processes to communicate via standard UNIX file pipes. Furthermore, a virtual file system is built to allow access to memories and registers defined in the FPGA, providing communication links between hardware and software.

The functions of BORPH are demonstrated on a BEE2 compute module. Performances of BORPH are measured to identify bottlenecks of our system. The clean OS kernel/user separation of BORPH has allowed us to improve overall system performance without affecting existing user designs. Furthermore, BORPH's unified run-time environment has enabled designers to make fair and end-to-end comparisons among software/hardware implementations of the same application. Most importantly, since the introduction of BORPH to our FPGA-based platform, we have observed increased productivity among high-level application developers who have little experience in FPGA application design.

Professor Robert Brodersen
Dissertation Committee Chair

to my family,

to Carmen, and

to God be the Glory.

Table of Contents

List of Figures v

List of Tables viii

1 Introduction 1
1.1 Related Work 4
1.2 Chapter Overview 6

2 Reconfigurable Computers 8
2.1 Reconfigurable Computer Architectures 9
2.2 Reconfigurable Fabrics 12
2.2.1 Granularity of Configuration 13
2.2.2 Method of Configuration 14
2.2.3 Field Programmable Gate Array 20
2.3 A Continuum of Computing 23
2.3.1 Spatial and Temporal Computing 23
2.3.2 Stored-Program Processor 25
2.3.3 Fully Spatial Computation 26
2.3.4 Between Spatial and Temporal Computing 27
2.3.5 Hardware/Software Terminology 29
2.3.6 Physical and Virtual Machine 30
2.4 BEE2 31
2.5 Summary 32

3 BORPH: The Operating System 34
3.1 Architectural Assumptions 36
3.2 Machine Abstraction Model 36
3.2.1 Reconfigurable Fabrics as Coprocessors 37
3.2.2 Reconfigurable Fabrics as Computational Resources 39
3.3 Hardware Process 40

3.3.1 Process Creation 41
3.3.2 The Lifecycle of a Hardware Process 42
3.4 Kernel Interface 44
3.4.1 Hybrid Message Passing System Call Interface 47
3.5 File I/O 49
3.5.1 Differences between Software and Hardware Design Patterns . 50
3.5.2 High Speed Streaming Data I/O 54
3.5.3 Runtime Streaming Mode Switching Support 57
3.5.4 File Streaming I/O Library 58
3.6 The ioreg Virtual File System 60
3.6.1 Basic Operation 61
3.6.2 Organization of ioreg Virtual Files 61
3.6.3 Example 63
3.6.4 Beyond Simple Register 65
3.6.5 Language Independence 66
3.6.6 Operating Mode 67
3.7 Summary 67

4 Implementation and Performance 70
4.1 Overview 70
4.2 Software Kernel Architecture 71
4.2.1 BOF file support 72
4.2.2 Reconfigurable Hardware Region (hwr) Support 72
4.2.3 FPGA Configuration and Resource Allocation 73
4.2.4 Software Fringe 73
4.2.5 Packet Communication Network 74
4.2.6 Process Scheduler and Signal Handler 74
4.3 Base Architecture 75
4.3.1 The SelectMap Bus Controller 77
4.3.2 Design of uK 77
4.4 Performance of Base Architecture 78
4.4.1 Hardware Process Creation 78
4.4.2 Reading/Writing ioreg Files 79
4.4.3 General File I/O from Hardware Processes 80
4.5 Advanced On-Chip Architecture 86
4.5.1 Elimination of PLB-to-OPB Bridge 86
4.5.2 DMA Enabled Control FPGA 89
4.6 Porting BORPH 94
4.7 Summary 96

5 BORPH Application Developments **98**
5.1 Conventional FPGA Design Flow . . . 98
5.1.1 Run Time Support . . . 100
5.2 Simulink-based Design Flow . . . 101
5.3 Sample Applications . . . 104
5.3.1 Example 1: A Real-Time Wireless Signal Processing System . 104
5.3.2 Example 2: Low-Density Parity-Check Decoders Emulation . . 106
5.3.3 Example 3: FPGA Video Processing with Commodity Software 107
5.4 Summary . . . 109

6 Conclusions and Future Directions **110**
6.1 Future Directions . . . 112
6.2 Closing Remarks . . . 113

Bibliography **115**

List of Figures

2.1 6 classes of reconfigurable computer architectures. 10
2.2 Simplified version of dedicated DSP blocks on a Virtex-4 FPGA. It is an example of coarse grain reconfigurable unit as it can perform only one of the several predefined mathematical functions. 13
2.3 Gate arrays are programmed by metal and contact layer specialization. 15
2.4 A PLA maps Boolean equations in sum-of-product forms directly with programmable connections in the AND-plane and OR-plane. 16
2.5 Making a connection between two wires using pass transistors controlled by 1 bit of configuration memory 17
2.6 Mapping the truth table of 2-input XOR function directly using memory with 4 locations of 1-bit wide data. 18
2.7 All 16 possible 2-input truth tables 19
2.8 A generic FPGA 21
2.9 Components of a generic FPGA 22
2.10 A spectrum of computers 24
2.11 Block diagram of a BEE2 compute module 31
2.12 BORPH utilizes a PowerPC based reconfigurable system that is built using a BEE2 compute module. 32

3.1 Two ways of organizing reconfigurable fabrics in a reconfigurable computer. 37
3.2 Simplified BOF file format 42
3.3 Executing a BOF file containing a free running counter. FPGA hardware is configured at prompt 1 and is unconfigured at prompt 3. . . . 43
3.4 BORPH's hardware kernel interface is constructed from one or more I/O connection points (`ioc`) 48
3.5 Left: BORPH's hardware system call (`hsc`) interface. Right: Format of packet transmitted through `hsc`. 48
3.6 BORPH's message format for file system read/write resembles its corresponding UNIX system call prototype. 53

3.7 A simplified flow diagram for `iock` packet handling. A streaming data connection is setup iff both source and sink are streamable and are ready to stream data. 58
3.8 The `bfsio` library contains two blocks. The read (bfsio_r) block generates read requests on behalf of the user and relays only data payload and EOF information to user. The write (bfsio_w) translates a stream of data from user into individual write request packets to the kernel. . 59
3.9 The `/proc` directory tree in of BORPH 62
3.10 BORPH's IOREG interface allows interacting with hardware processes via virtual files. The register `cntval` contains current value of the free running counter in `counter.bof`. This counter can be disabled by writing to `cnten` register. 64
3.11 A simple packet format for message exchanges between BORPH main kernel MK and its distributed kernel UK. 65

4.1 The two logical components of BORPH: MK and UK. MK is the main controlling software kernel. Each reconfigurable hardware region is individually managed by a low-level kernel called a UK. 71
4.2 Block diagram of BORPH system on a BEE2 compute module with OPB-based SelectMap controller in control FPGA (OPBSM). 75
4.3 Performance of reading/writing on-chip memory on a user FPGA using ioreg interface. 81
4.4 Hardware process file I/O performance. 82
4.5 The effect of data transfer size s on total time required by a hardware process to sink a regular file. Time is measured from control FPGA. . 84
4.6 Comparing software piped process chain with a mixed hardware/software chain. 85
4.7 PLB-based SelectMap controller (PLBSM). 87
4.8 Performance comparison between OPBSM and PLBSM. 88
4.9 Direct memory access (DMA) is enabled in the control FPGA for communication between PPC and the PLB-based SelectMap controller. On a user FPGA, a hardware arbiter is implemented to allow direct access to SelectMap bus from a user gateware design for file I/Os. . . 90
4.10 Hardware process creation time in 3 different on-chip architectures compared with the theoretical minimum time and software process creation time. 91
4.11 Normalized time for hardware processes to finish streaming files of various sizes. `stdloop.bof` uses PPC on user FPGA for file operations, while direct file I/O through `hsc` is used in `stdloop_hsc.bof`. 93
4.12 An example of porting BORPH to a single FPGA where user hardware processes are executed on a partially reconfigurable region. 94

5.1 An automatic hardware design flow that compiles high-level Simulink designs into executable BOF files. 102
5.2 Block diagram of a user FPGA. Compiled user Simulink designs are combined with a predefined EDK template of uK to generate user FPGA configurations. 103
5.3 Cognitive radio testbed system 105

List of Tables

3.1 Types of ioreg 65

Acknowledgments

This work could not be completed without the help, support and prayers of my colleagues, friends and family.

I have to thank my advisor, Bob, for his continuous support throughout my time at Berkeley. It was his adventurous mindset that inspired the creation of BORPH. It was his well-rounded and open mind that nurtures such highly cross disciplinary work of FPGA operating system. His passion for people, life, the nature and ideas from any research fields is going to continue be my aspiration even after leaving Berkeley.

I have to thank Professor John Wawrzynek for introducing me into this exiting field of reconfigurable computing and for his invaluable recommendations on the design of BORPH. I also want to thank members of my dissertation and qualifying exam committee, Professor Dorit Hochbaum, Professor Pravin Varaiya, and Professor Eric Brewer, for their time, effort and recommendations.

Throughout my time at Berkeley, I have been supported financially by the DARPA, C2S2.

The success of BORPH is due largely to the availability of the fine FPGA systems, design flow, and their users at the Berkeley Wireless Research Center. I want to thank Greg Wright for envisioning the original BEE (at that time the Biggascale Emulation Engine); to Chen Chang and Pierre Droz for building the BEE and BEE2 machines; to Andrew Shultz, Pierre Droz and Henry Chen for fine tuning and promoting the Simulink FPGA design flow.

I want to thank all the users of BORPH whose feedbacks have been invaluable

for the development of BORPH. It was your comments such as "BORPH is my life-saver" and "It (BORPH) has been instrumental in my research" that have kept the development of BORPH motivated. I must thank Artem Tkachenko in particular for serving as "guinea pigs" stepping on the never ending list of bugs during BORPH's early development.

To my family, I want to express my deepest gratitude for your endless support, enabling me to pursue this fine path of research at Berkeley, away from home, for many years.

To Pastor Samuel Wong, thank you for your support during my lowest point in my Ph.D. career.

To Carmen, thank you for your patience, support, encouragement, understanding, and prayer all these years.

Finally, and most importantly, I must thank my Lord for providing all the people and things that I am grateful for all these years; for carrying me through even the darkest valley and the brightest avenue; and for giving me the much needed patience and wisdom for research and for life.

I will lift up mine eyes unto the hills, from whence cometh my help.
My help cometh from the LORD, which made heaven and earth.

– Psalms 121:1-2

Chapter 1

Introduction

Reconfigurable computing has a long history. In 1960, Gerald Estrin proposed a "Fixed+Variable" machine that some believe to be the first reconfigurable computer[18]. This machine featured a fixed host processor and a section of reconfigurable hardware for application acceleration. Most reconfigurable computers (RCs) developed since then followed a similar machine organization.

However, the idea of computing using reconfigurable hardware has not been given as much attention as other simpler machine organizations. Until very recently, single processor systems have dominated the mass personal computer market. For high performance computing, researchers turn to various kinds of parallel processor architectures such as cluster of symmetric multi-processors (SMP). Over the years, the technology trend in computer architecture has been determined by two major factors: (1) Physical hardware implementation technology and (2) Ease of use. Implementation technology imposes a number of physical constraints on a computing system. For example, bus performance limits the scalability of bus-based shared memory SMP[17], while power dissipation limits performances of modern sophisticated super-scalar processors.

Ease of use of a computer system, on the other hand, is a subjective measurement that takes on multiple meanings depending on context. Nevertheless, it is the ease of understanding and reason about that the original *sequential stored program* computer concept developed in the 1940s still prevails among modern computers. It was the desire to have a smooth application development environment that motivated the development of the first operating system on the EDVAC machine[36]. It was the observation that easy to use user interfaces greatly improve computer users' productivity that fueled the study of human computer interface.

Technology-wise, reconfigurable hardware has improved dramatically since the 1960s. Today, reconfigurable computers are capable of delivering orders of magnitude higher performance than software solutions on conventional processor systems[43]. Their high performances have made them viable computing platforms for a wide range of application domains, such as high speed digital signal processing[31], multi-media processing, speech recognition[27, 33], bioinformatics[16], and radio astronomy. However, despite their promising performances, researchers, particular those traditionally not accustomed to hardware/software design environments of RCs, remain reluctant to adopt such technology. The main reason for this remains that *reconfigurable computers are difficult to use.*

There are two classes of difficulties. The first class of difficulty relates to design methodologies for RCs, i.e. the process of translating a conceptual design into a machine understandable program or configuration. This class of difficulty arises during *compile time* of an application. In the context of a conventional processor system,

such translation process involves two main steps. First, a conceptual design must be expressed in certain design language, such as C. Then, this user description must be translated by a compiler into a sequence of processor specific instructions according to the processor's predefined instruction set architecture (ISA).

Unfortunately, due to the flexible hardware architecture of RCs, this conversion process is not as straight forward as software compilation. Successful RC design methodology requires design languages that are high-level enough for them be easy to use yet powerful enough to fully express the parallel nature of an application. Furthermore, a smart compiler is needed to further extract implicit parallelism and sometimes even derive the correct computation architecture for each application. Finally, the tools must be able to efficiently perform placement and routing of the compiled application on to the physical reconfigurable hardware fabric of the target system to meet performance and resource constraints. In fact, the design methodology for RC is so important that it has been the focus for a majority of research in the reconfigurable computing field in the past.

The second class of difficulty involves how easy an RC application may interact with the system, user, and other applications during *run time*. It involves questions such as: "On which reconfigurable fabric, and for how long, should an application be executed?" "How does an application communicate with the user?" "How does an application perform general I/O operations?" Most existing RC systems have their own ad-hoc mechanisms to address the above system integration requirements. However, few research efforts have been put into addressing these common problems

in a standardized and systematic way. The focus of this dissertation work, BORPH, approaches this class of problems systematically from an operating system design perspective.

The goal of BORPH is to improve usability of reconfigurable computer systems through operating system support. Instead of abstracting reconfigurable hardware of the system as accelerators for software programs, BORPH allows users to implement applications directly using these reconfigurable hardware fabrics. User hardware designs therefore run as normal UNIX process in the system like a software program. BORPH OS kernel provides standard UNIX services, such as file system access, to these hardware processes. Productivity of novel RC users are observed to have improved as a combined result of such familiar OS interface, a hardware design flow with libraries that integrates with BORPH, and the availability of commodity software to coexist with hardware designs within the same system.

1.1 Related Work

A number of research projects have approached the task of designing operating systems for FPGA-based reconfigurable computers[10, 48, 46, 30, 21]. All of them are devoted to the problem of dynamic FPGA resource allocation, memory sharing or virtualization between software and hardware tasks on FPGA-based systems. We are not aware of any prior work that systematically offer runtime support directly to hardware processes as BORPH does. Furthermore, instead of relying on abstract "task" models commonly found in other reconfigurable computer operating systems,

the use of UNIX process semantics to model running FPGA designs is unique to BORPH.

On the other hand, most commercial FPGA-based reconfigurable computers[9, 51, 15, 6] are managed by off-the-shelf operating systems such as Linux and VxWorks. FPGAs on these systems are used mainly as software accelerators. Software and FPGAs communicate through the conventional device driver layer while FPGA designs must utilize vendor specific libraries with custom APIs. As a result, even if machines from different vendors are constructed using identical FPGAs, the inconsistent system interface prevent designs targeting one machine to be easily ported to another. Such inconsistent system interface greatly hinder collaborations among FPGA researchers.

Much consideration has been put into the design of BORPH's kernel/user interface, making it as close to conventional UNIX system as possible. It is designed to lower the barrier-to-entry for novel FPGA users and to increase portability of reconfigurable application designs. The importance of an intuitive and unified interface is well acknowledged by a number of works[45, 37]. The work of UltraSONIC[47] shares a similar design philosophy as BORPH in providing a unifying coarse-grain hardware software component interface. Furthermore, the choice of POSIX conforming pthread interface in the work of hThread[2] echoes well with BORPH's design philosophy.

The main contribution of BORPH is that by leveraging conventional UNIX semantics to FPGA-based reconfigurable computing, it provides a unique, unified environment for both FPGA and software application designers. The UNIX semantics is familiar to developers across many research domains, thus lowering the barrier-

to-entry into FPGA-based reconfigurable computing. Furthermore, since BORPH is implemented as an extended Linux kernel, a BORPH managed system may leverage all commodity Linux software applications for developing, testing, benchmarking, and deploying FPGA applications.

1.2 Chapter Overview

Chapter 2 provides brief background information about reconfigurable computing. An overview of various reconfigurable computer architectures will be given. We will introduce one of the most common reconfigurable hardware fabrics in contemporary RC, the field programmable gate array (FPGA) and illustrate how one can configure such hardware to perform any digital logic. Finally, we will describe the FPGA-based reconfigurable machine, BEE2, on which BORPH currently runs.

Chapter 3 goes into details of BORPH design and implementation. The concept of hardware process will first be introduced. Based on this notion of hardware process, we will describe various BORPH features: the BORPH hardware kernel/user interface design; the hardware file I/O subsystem; and finally the IOREG virtual file system.

Chapter 4 evaluates the performance of BORPH. Three iterations of hardware system design will be described. The multiple implementations not only provide a concrete performance benchmark of various BORPH subsystems, they also illustrate the benefit of kernel/user separation in hardware. Finally, these benchmarks demonstrate how end-to-end performance comparison between software and hardware can be achieved through BORPH's unified hardware/software run-time environment.

Chapter 5 explores the interaction between FPGA design methodologies and the run time operating system. Design methodologies for FPGA and other reconfigurable computers will first be briefly described. Then, the Simulink-based design flow developed at the Berkeley Wireless Research Center (BWRC) will be described as an illustration on how a design methodology integrates with BORPH. Finally, three actual FPGA applications are presented to illustrate how various features of BORPH work together to ease their development processes.

We will conclude this dissertation and provides insights into future research inspired by BORPH in Chapter 6.

Chapter 2

Reconfigurable Computers

This chapter provides a brief introduction into the field of reconfigurable computing. Broadly speaking, reconfigurable computing refers to machines that compute with dynamically reconfigurable data and control path. This is in contrast to a simple processor system in which only a single fixed data and control path is presented. As such, reconfigurable computing generally refers to a large spectrum of computers with drastically different designs and implementations. A number of excellent surveys are available in the literature[43, 8, 22, 42] and their results are not repeated here. Only a brief introduction is given here to make this dissertation self-contained.

In general, reconfigurable computers consist of zero or more physically presented sequential processors coupled with one or more reconfigurable fabrics. Such definition points to two important design aspects of reconfigurable computers: (1) The coupling of processor and reconfigurable hardware, i.e. their system architectures; and (2) The implementation of reconfigurable hardware.

2.1 Reconfigurable Computer Architectures

Todman *et al.* [43] extended the work of Compton and Hauck[8] and provided a 5-class classification of RC architectures as shown in Figure 2.1(a) to (e). The first four classes of systems are characterized by the physical presence of a single controlling processor. They differ in the way that the processor communicates with the reconfigurable fabric (RF) of the system.

In Figure 2.1(a), reconfigurable fabrics are connected to the processor through its system I/O bus. Although it provides the least data bandwidth between the processor and the RF, it is easiest to implement. A conventional processor-based system can be extended into a RC system by simply inserting an add-on card with reconfigurable fabrics to its peripheral bus. Because of this simplicity, it is by far the most common RC architecture found in commercial systems.

Figure 2.1(b) and 2.1(c) depict systems that incorporate RF into two different locations within the processor's memory subsystem. Data bandwidth between the processor and the RF is usually the performance bottleneck of a system. By connecting RFs directly to the processor's memory subsystem, these architectures provide the much needed bandwidth.

Figure 2.1(d) shows a system that integrates RF directly into the data path of the controlling processor as functional units. It allows the RF to have access to all local information about the running processor, such as the register file. Such tight integration ensures maximum integration between software and hardware. However, it is also this tight integration that limits RF speedup due to the lack of instruction

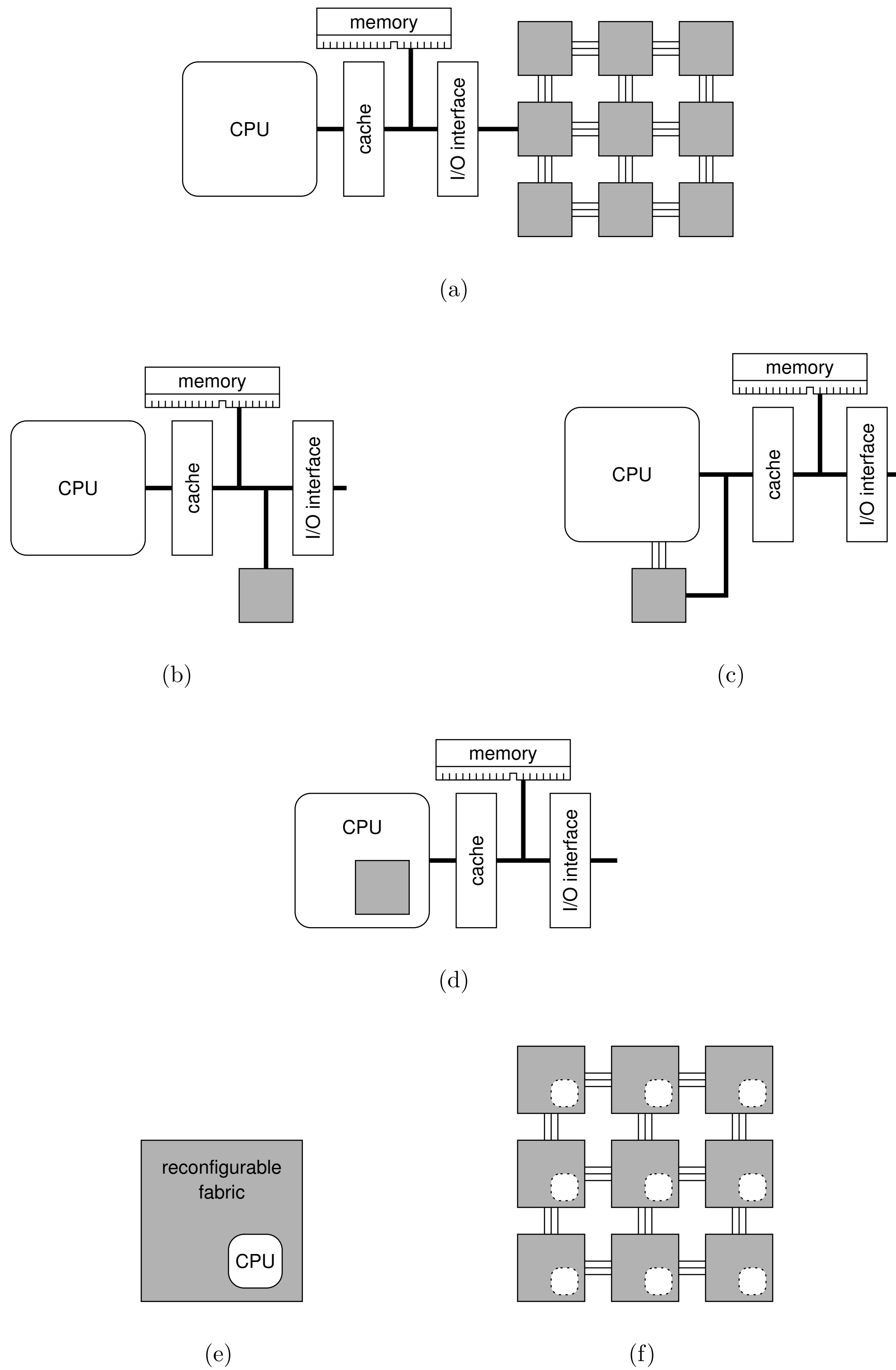


Figure 2.1: 6 classes of reconfigurable computer architectures.

level parallelism.

Figure 2.1(e) represents a new class of RC that is made only possible with advances in reconfigurable hardware technologies. Instead of connecting reconfigurable fabrics to a processor system, these machines embed processors within reconfigurable fabrics. These embedded processors can either be implemented physically or as *soft processors*. Soft processors are processors that are built as needed by an application using reconfigurable hardware. Examples are the MicroBlaze and PicoBlaze processors by Xilinx, as well as the Nios and Nios-II processors by Altera. This class of RC system has the benefit of allowing a user to determine the type and number of processors needed in the system, especially by using soft processor, thereby increasing system performance and efficiency. Most importantly, this class of system breaks away from the processor-centric compute model in the previous 4 classes of systems. By shifting away from the sequential compute model of the controlling processor, this class of system has much higher performance potential than the previous four.

Figure 2.1(f) illustrates the sixth class of system that consists of two or more machines in the previous 5 classes connected through a direct network. On a system level, these systems share similar properties such as system topology and routing strategies with conventional multi-processor systems. However, because of the proximity of computational fabric to the network, RC systems provide much higher potential performance benefit over multi-processor systems. For example, sending a word of data from a hardware application on one FPGA to another directly connected FPGA takes only a few clock cycles for synchronization. If the system is fully synchronized,

latency can potentially be further reduced to zero cycle, virtually doubling the size of the FPGA. Even for large complicated RC systems that must utilize complex packets for communication, the fact that the very same computational reconfigurable hardware fabrics are used to manage these communication packets significantly reduces such communication overhead. In contrast, because of layers of software overhead, communication latency on multi-processor systems, particularly cluster based systems, are order of magnitude higher[29]. Moreover, some systems employ a run time reconfigurable direct network, forming a hierarchical reconfigurable system as a result. For example, the RAW machine consists of an array of processing elements connected through a user programmable network[41]. The routing of the network is controlled as part of the user program. At the same time, each RAW processing element is itself a simple reconfigurable computer consists of a processor coupled with a custom made FPGA.

2.2 Reconfigurable Fabrics

One of the most important structures that differentiate a reconfigurable computer from a conventional processor-based system is its reconfigurable fabric. Many different types of reconfigurable fabrics have been proposed in the literature. In general, they can be characterized by their granularities of configuration and their methods of configuration. This section explores the design space of reconfigurable hardware and concludes with an introduction to currently most widely used fabric: the field programmable gate array (FPGA).

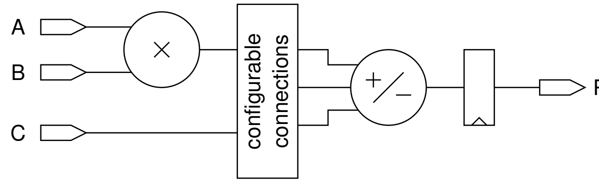


Figure 2.2: Simplified version of dedicated DSP blocks on a Virtex-4 FPGA. It is an example of coarse grain reconfigurable unit as it can perform only one of the several predefined mathematical functions.

2.2.1 Granularity of Configuration

The configuration granularity of a reconfigurable hardware fabric affects its flexibility in implementing different logic functions. There is always a trade-off between flexibility and efficiency of the fabric. Fine grain reconfigurable fabrics are very flexible. They can be used to implement any sequential and combinational Boolean logic function, but are slower and physically bigger in general. On the other hand, coarse grain reconfigurable fabrics are faster, occupy smaller areas, but are limited to implementing only one of the predefined functions.

Some reconfigurable fabrics, such as modern FPGAs, contain a mix of both fine grain and coarse grain reconfigurable units. For example, Xilinx Virtex-4 FPGAs[50] contain dedicated blocks similar to that in Figure 2.2 for digital signal processing (DSP) applications. This block can be programmed by the user to perform a combination of multiplication, addition or subtraction. Although the same functions could have been implemented using general fine-grain programmable fabrics on the FPGA, having such dedicated blocks result in designs that are smaller, faster, and consume less energy.

Unfortunately, the correct mix of coarse grain and fine grain reconfigurable units

is highly application specific. As a reconfigurable computing platform, designers must adjust the mix according to the area, power and performance requirements for the target application domain.

2.2.2 Method of Configuration

In the 1970s, integrated circuit designers were looking for timely and cost effective ways to manufacture application specific integrated circuits (ASICs). The result was the development of integrated circuits that can be programmed after they are fabricated. These post-fabrication programmable integrated circuits subsequently evolve into modern day programmable logic devices.

Non-Volatile, Single Program

Tracing back to their ASIC roots, early programmable logic devices are designed to be programmed only once: i.e., to program an IC to implement a specific application. One way to achieve such specialization is by pre-fabricating a *gate array* on silicon wafers as seen in Figure 2.3. Designing an ASIC is then reduced to a task of connecting these pre-fabricated transistor gates using upper metal layers. Manufacturing cost, as well as time-to-market, is therefore reduced as the number of fabrication mask is reduced. Strictly speaking, this method is not a post-fabrication logic programming mechanism. Nevertheless, this metal layer specialization is the predecessor for many subsequent programmable logic devices (PLDs).

Another non-volatile, single program method of configuration for PLDs is by using

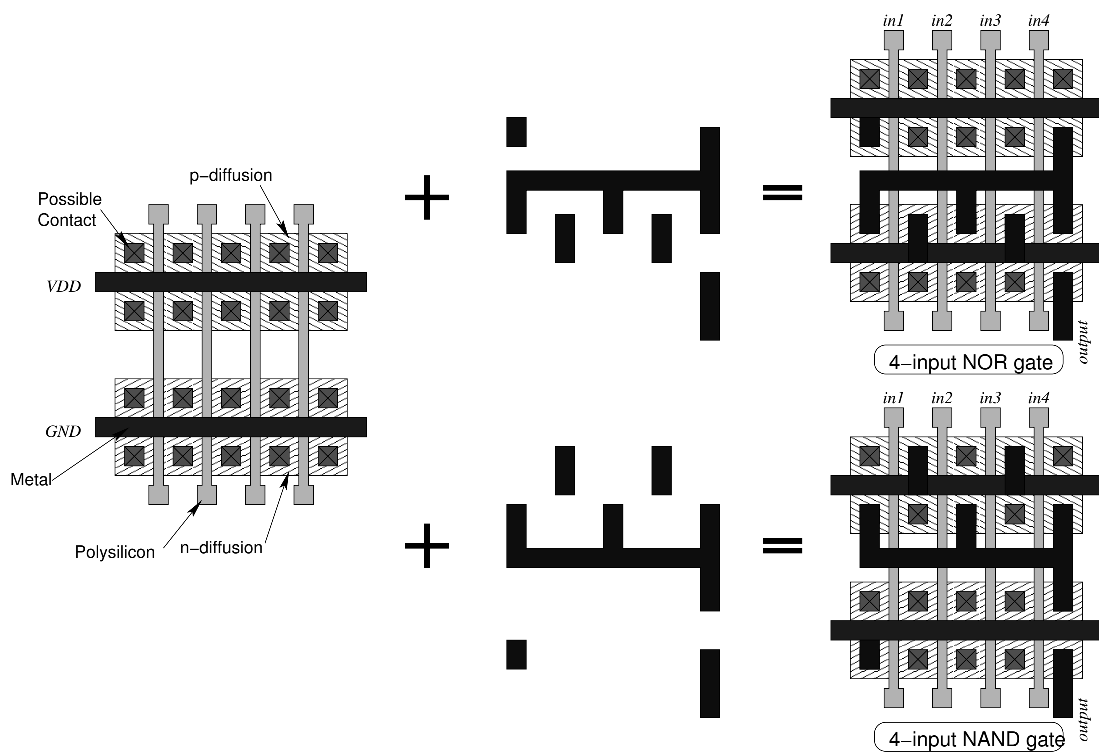
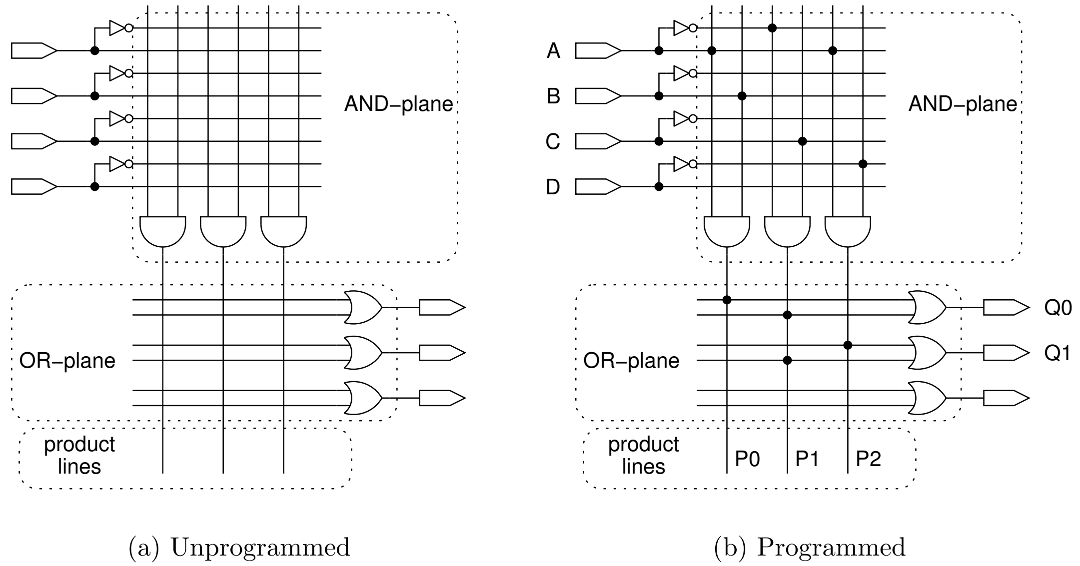


Figure 2.3: Gate arrays are programmed by metal and contact layer specialization.



(a) Unprogrammed (b) Programmed

Figure 2.4: A PLA maps Boolean equations in sum-of-product forms directly with programmable connections in the AND-plane and OR-plane.

fuses or anti-fuses to form metal connections. They are usually used in dense PLDs such as programmable logic arrays (PLAs). As oppose to gate arrays, PLAs are truly post-fabrication programmable logic devices. Logically, a PLA is programmed by directly mapping user Boolean functions in sum-of-product forms to the device.

Figure 2.4(a) shows a simplified block diagram for a PLA. A typical PLA contains a programmable AND-plane connected to a programmable OR-plane. Similar to the gate array concept, a PLA is configured by programming wire connections that connect the two planes and I/O pins. These connections are either made of anti-fuses or programmable switches. An anti-fuse behaves in exact opposite ways of a normal electrical fuse: i.e., the two terminals of an anti-fuse are disconnected by default until it is “broken” by strong current.

The first step to implementing user logic is to generate all *product terms* by making

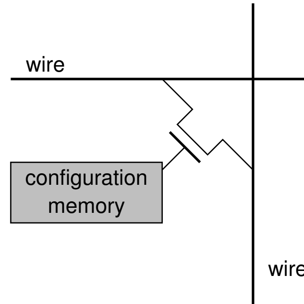


Figure 2.5: Making a connection between two wires using pass transistors controlled by 1 bit of configuration memory

appropriate connections to the AND-plane input. The results, carried in the set of vertical *product lines*, are then connected accordingly to the OR-plane to compute the final sum-of-products. For example, the PLA in Figure 2.4(b) is programmed to compute two equations: $Q0 = AB + \overline{A}C$ and $Q1 = A\overline{D} + \overline{A}C$. Product lines $P0$, $P1$ and $P2$ carry the intermediate results AB, $\overline{A}C$ and $A\overline{D}$ respectively.

Non-Volatile, Multiple Program

Some manufacturers use non-volatile, but programmable memory technology such as erasable programmable read only memory (EPROM) and electrical erasable programmable read only memory (EEPROM) to implement programmable logic devices. For instance, instead of using anti-fuses, PLAs described above can be programmed using pass transistors that are controlled by PROMs (Figure 2.5).

Another way to build programmable logic devices is to implement Boolean logic directly using non-volatile memories. Performing logic operations using memory is best illustrated by an example.

Figure 2.6 shows the truth table of an *exclusive or* (XOR) Boolean operation

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

(a) Truth table for XOR

address	data
0	0
1	1
2	1
3	0

(b) Mapped to 4 memory locations

Figure 2.6: Mapping the truth table of 2-input XOR function directly using memory with 4 locations of 1-bit wide data.

where output (Q) is 1 only if exactly one of either A or B is 1. One way to implement an XOR operation in hardware is by mapping its truth table directly using a read only memory (ROM). Using input A and B as memory address, this ROM reads out the expected value of Q. In this particular example, a 0 is stored at locations 0 and 3 while a 1 is stored at locations 1 and 2. When a ROM is used for looking up values, it is referred as a lookup table (LUT).

Figure 2.7 shows *all* 16 possible truth tables with two inputs, A and B, and one output, Q. For instance, the 7th column from the left shows the same truth table as Figure 2.6 for an XOR operation. Figure 2.7 illustrates the benefit of implementing a simple 2-input XOR function using a LUT: configurability. By implementing lookup tables using programmable memory, the logic function of that a LUT implementation can be changed simply by changing data values in various memory locations. For instance, to implement an AND operation, one can simply write 0 to locations 0,1,2 and write a 1 to location 3.

A	B	zero	AB	$A\overline{B}$	A	$\overline{A}B$	B	$A \oplus B$	$A+B$	$\overline{A+B}$	$\overline{A \oplus B}$	$\overline{B}$	$A+\overline{B}$	$\overline{A}$	$\overline{A}+B$	$\overline{AB}$	one
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Figure 2.7: All 16 possible 2-input truth tables

In general, any Boolean logic function with n variables can be implemented by using a lookup table with n inputs. A LUT with n inputs is usually referred as an n-LUT. Each n-LUT can be configured to implement one of the 2^{2^n} possible functions. However, large LUTs are expensive to build in terms of silicon area, speed, power and dollar cost. Therefore, lookup tables of moderate sizes from 3 to 6 inputs are usually used.

Volatile Configuration

Most modern commercially available programmable devices utilize volatile memories, such as static random access memory (SRAM), or even simple on-chip registers, to store device configurations. Since the device configuration is volatile, these devices must be programmed every time a system is powered up. The programming is usually accomplished by reading configurations from an external PROM or through standardized in-system testing protocol such as that from the Joint Test Action Group (JTAG). Architecturally, these modern devices can be identical to devices described above. However, since programming a device is performed through writing to SRAM or hardware registers, the programming speed is order of magnitude faster than pre-

vious non-volatile configuration solutions. Furthermore, modern devices support dynamic partial configuration, which is the ability to reconfigure parts of the device while the rest of the device continues to operate without interruption.

Such versatility in programming has created opportunities for using even off-the-shelf PLDs in reconfigurable computing. A number of commercial products[9, 51, 15] as well as numerous research projects[26] are based on off-the-shelf PLDs.

Custom, Multi-Context Configuration

Over the years, reconfigurable computing researchers have proposed a number of custom reconfigurable fabrics specific for their systems. Some of them are developed to integrate with existing processor datapath[38]. Others developed the notion of multi-context reconfigurable hardware fabrics[28, 25, 44, 11]. The idea of multi-context reconfiguration evolves from the need to reprogram a device in extreme speed. It is particularly useful for RC systems in which reconfigurable fabrics must switch between different functions quickly. As a result, these devices store multiple configurations in the device in parallel that can potentially be swapped in a single hardware cycle.

2.2.3 Field Programmable Gate Array

Field Programmable Gate Array (FPGA) is one of the most readily available commercial programmable logic devices. Starting as ASIC replacements similar to other PLDs, FPGAs have slowly evolved into complex embedded system platforms that are flexible and are able to deliver performances comparable to ASICs. The

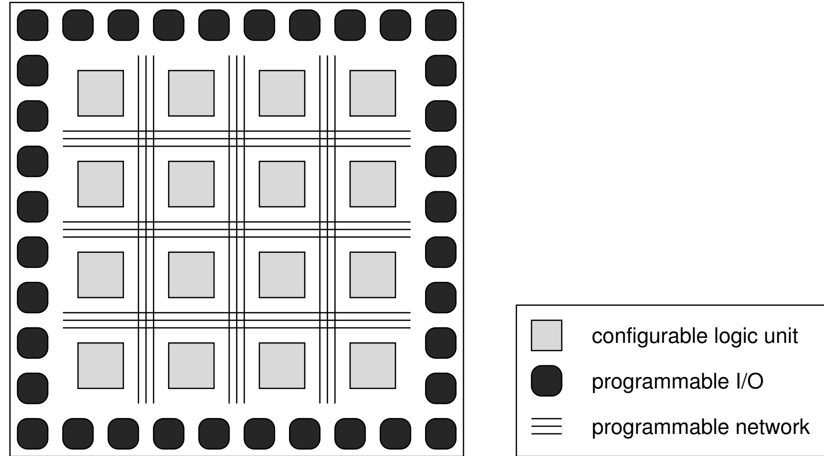


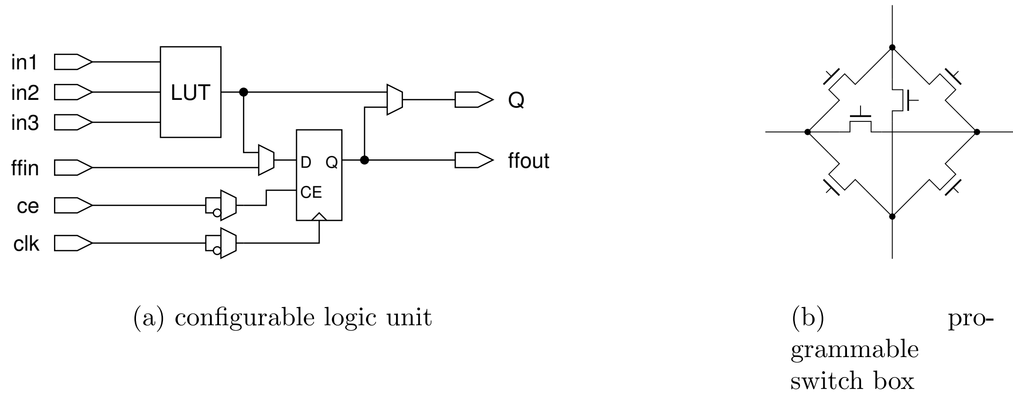
Figure 2.8: A generic FPGA

basic architecture of an FPGA is shown in Figure 2.8. FPGAs are generally made up of an array of configurable logic units connected with a programmable routing network. Programmable I/O blocks are used to control off-chip interfaces, such as signal directions and impedances of I/O pins.

Configurable Logic Unit

The basic building blocks of an FPGA are configurable logic units (CLUs) that are programmable according to user logic functions. Different FPGA manufacturers have slightly different implementations, such as "slices" for Xilinx FPGAs or "adaptive logic modules" for Altera FPGAs. Despite their different implementations, they share a similar basic design. Each configurable logic unit consists of a fine grain programmable combinational logic block that is optionally connected to one or more flip-flops (Figure 2.9(a)).

The programmable combinational logic block is usually implemented using lookup



(a) configurable logic unit

(b) programmable switch box

Figure 2.9: Components of a generic FPGA

tables. A number of multiplexers (MUXs) are used to fine tune the behaviors of a CLU. They are used to select output from either the LUT or flip-flop, or both. They also control polarities of clock and other control signals. By carefully connecting the flip-flops and LUTs of different configurable logic units using the on-chip programmable routing network, *any* synchronous digital logic design can be implemented.

Routing Network

The routing network of an FPGA links different configurable logic units together to perform user defined logic. Many different routing networks have been proposed to balance performance with routability of an FPGA. FPGA routing networks are generally regular, consisting of groups of short local wires combined with long global wires. They are designed for maximum flexibilities without introducing long wire delays. Wire to wire connections are made through programmable switch boxes similar to that in Figure 2.9(b). Such programmable switch boxes allow flexible connections

such as a T-joint.

2.3 A Continuum of Computing

On the surface, comparing the architecture of a single cycle processor with an ASIC design process seems remotely relevant. However, as it will be illustrated later, single cycle processors and ASIC designs can abstractly be viewed as extreme points in the broad spectrum of computing methods. This section gives an abstract understanding of reconfigurable computing with respect to other forms of computing in the hope to illustrate various benefits and trade-offs of reconfigurable computers.

2.3.1 Spatial and Temporal Computing

The fundamental function of a computer is to compute. Therefore, any machine that can perform the necessary computation to accomplish a user's conceptual design can be treated as a computer. In practice, there are many ways to perform computations. Example includes, but not limited to, sequential stored-program computing, communicating sequential processor (CSP), synchronous data flow, etc. Figure 2.10 organizes some of these computers in a hypothetical spectrum that corresponds roughly to how often the computer is reconfigured.

In general, machines on the left hand side of Figure 2.10 carries out computation sequentially, usually according to one or more instruction streams. Computations on these machines are serialized and spread over time. Consequently, they are in

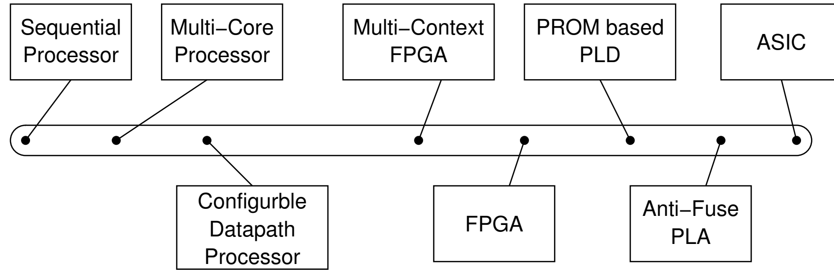


Figure 2.10: A spectrum of computers

theory relatively slower than those machines on the right hand side of the spectrum. On the other hand, those machines on the right of the spectrum tend to perform computations in parallel. Since more computations are accomplished spatially at the same time, these machines are in theory faster.

In practice, however, parallel computation has its limitation. First, one must be able to express high level designs in parallel to take advantage of the underlying spatial computation paradigm. Unfortunately, it is a non-trivial problem. To effectively express a parallel high level design, a designer must possess fluent knowledge about the choice of design languages, software compilers or computer aided design (CAD) tools, as well as the underlying computer architecture. Second, parallel computations require multiple operations to take place at the same time. As a result, more time and efforts are needed to setup the machine and to route the appropriate data to perform these operations. Therefore, parallel machines are relatively less flexible than sequential machines during run-time.

One of the benefits of reconfigurable computers is therefore their theoretical abilities to deliver performance of highly parallel computers while offering flexibilities of traditional stored-program processors. Essentially, reconfigurable computers pop-

ulate the middle section of Figure 2.10 by providing different mixes of spatial and temporal computing capabilities.

2.3.2 Stored-Program Processor

On the far left of Figure 2.10 is a single cycle stored-program processor. Such processor carries out its operations according to a stored program. This stored program computing forms the fundamental computation architecture for almost all modern processors, which is sometimes referred as the "von Neumann" architecture. The notion of stored program machine can be dated back to the 19th century where mechanical looms used punch cards for controlling cloth patterns. The same punch card concept was used in the design of Charles Babbage's Analytical Machine in 1837. Punch cards are used in most early computers in the 1950s.

The use of punch cards in looms helps to clarify an important observation to understand reconfigurable computing: *A processor instruction by itself is a simple machine configuration*. This observation is especially apparent for a reduced instruction set computer (RISC) or a micro-coded machine where each machine instruction performs one simple task. For example, an `add` instruction configures the arithmetic and logic unit (ALU) of a processor to carry out an addition operation. It configures the ALU to take input from the correct register from the register file. It also configures the machine to write the result of the ALU to the correct output register. Given the sequential nature of stored-program computers, a "program" can be defined as *a sequence of machine configurations*. A sequential processor therefore "reconfigures"

itself continuously according to the stored program.

Based on this observation, a stored program processor is therefore a fully temporal reconfigurable computer. At any given time, it performs only one operation. However, as observed by DeHon[12], it reconfigures itself after each operation, thereby spreading all operations over time.

2.3.3 Fully Spatial Computation

On the other hand of the spectrum of computing is an application specific integrated circuit (ASIC). More precisely, it is the silicon wafer that is used to build an ASIC that is "configured" during the manufacturing process such that it will carry out certain computing operations. It is a form of fully spatial computation because at any given time, many operations are performed in parallel over space. Furthermore, the same set of operation is performed every time. In the case of a synchronous ASIC, the unit of time is a clock cycle. An ASIC never "reconfigures" itself. Once an ASIC is fabricated, its function is set.

Anti-fuse based PLAs are programmable after they are fabricated. However, they can only be programmed once. PROM based PLDs are truly *re*-configurable devices. However, because of the need of external configuring devices and their long configuration time, they are usually configured only once for a particular application and are rarely reconfigured over the life-time of a product.

On the other hand, FPGAs can be programmed in-system relatively quickly in the other of milliseconds. Because their configurations are stored in volatile memory, they

can be reprogrammed quickly. As a result, despite being developed originally as replacements for ASICs, FPGAs are natural devices to perform fine grain reconfigurable computing.

2.3.4 Between Spatial and Temporal Computing

Occupying the spectrum between a processor and an FPGA is a spectrum of reconfigurable computers that exploit the design space of temporal and spatial computing.

To add a temporal dimension to a fully spatial FPGA, multi-context devices are developed[28, 25, 44, 11]. Multiple configurations of the reconfigurable fabrics are stored, or cached, inside the device to allow rapid device reconfiguration. A temporal compute model is therefore established when a device is reconfigured (rapidly) as part of the native compute model. Furthermore, the temporal reconfiguration does not necessarily apply to the entire device. Modern FPGAs allow dynamic partial reconfiguration that reconfigures part of the device while the rest of the device continues to run.

On the other hand, to add a spatial dimension to a fully temporal processor, RCs such as PipeRench[38] introduced reconfigurable datapath to a simple sequential processor. Furthermore, RCs such as Garp[23] utilized reconfigurable fabrics as dynamically executed co-processors.

Trade-offs between Performance and Flexibility

The architectural design of a reconfigurable computer is often the result of a series of trade-offs between performance and flexibility. Depending on the context, the performance of a RC may be measured by its speed, power consumption or silicon area consumed. On the other hand, flexibility refers to how easy a machine can be programmed to perform different tasks.

In general, architectures towards the right hand side of Figure 2.10 are programmed with longer instructions. Architectures with a longer instruction are more demanding on their reconfiguration subsystems than architectures with a shorter instruction. It is because transferring streams of long instructions from their backing stores takes longer than transferring streams of short instructions. Consequently, the architecture must either reconfigures less frequently or devotes more silicon areas for temporary storages such as caches, which was termed instruction density by DeHon in [11].

Fortunately, a “long” instruction may embed more information and may therefore be useful over a longer period of time. For instance, once an FPGA is programmed with its intended configuration, it may be left running until it is powered down. All the necessary actions needed to be performed by this FPGA are contained in this one very long instruction. In fact, when an FPGA is configured to perform data intensive signal processing, it becomes an extreme case in single instruction stream multiple data (SIMD) machine where only one single instruction is used. Nonetheless, more compile time effort must be spent to fully utilize such long instructions.

On the other hand, a stored-program processor fetches and executes a new instruction every cycle, making it very flexible. Each instruction by itself cannot accomplish much. Nonetheless, when executed as a collection, they become useful software programs. The difficulties associated with compiling for such model therefore lies with the temporal aspects of the instruction stream.

Architectures with multiple configuration contexts attempt to balance the benefits of spatial computers with the flexibilities of stored-program processors by providing quick context switching mechanisms. However, they are also faced with both compile time and run time implementation difficulties with architectures in both ends of the spectrum.

2.3.5 Hardware/Software Terminology

In conventional computer engineering terminologies, the term “hardware” is primarily used to refer to the physical machinery of the system while the word “software” is used to refer to the programs that determine the functions of the machine. In the context of a conventional stored-program computer, the distinction between the two is apparent: “Hardware” refers to the central processing unit (CPU) and its supporting platform, such as memory and I/O devices, while “software” refers to the sequence of instructions that control the function of the “hardware” platform to accomplish certain task.

On the other hand, in the context of a reconfigurable computer, the use of the word “hardware” sometimes becomes fuzzy and ambiguous. This is particularly con-

fusing with respect to fine-grain reconfigurable fabrics such as FPGAs because an FPGA performs in ways identical to an ASIC once configured. However, as argued in Section 2.3.2, configurations of reconfigurable fabrics of a RC are equivalence of instructions for a processor. Therefore, it is more appropriate to refer to these FPGA configurations as mere “programs” or even “software”. Unfortunately, the word “software” is easily confused with the sequence of processor instructions commonly known in the context of stored-program computing.

To aide the distinction, Hartenstein used the word “configware” as a counter part to “software” in the context of reconfigurable computer[4]. Similarly, the word “gateware” was introduced by Adam Megacz and is used by the RAMP project[24] to refer to pieces of FPGA logical designs that can be shared among researchers. In this dissertation, the word “gateware” will be used whenever appropriate to avoid confusion.

2.3.6 Physical and Virtual Machine

Reconfigurable computers are sometimes difficult to be classified because their machines can be reconfigured to compute in many different modes. This is particularly true for fine-grain reconfigurable hardware such as FPGAs. For instance, one design may choose to configure an FPGA into a shared memory multi-processor system using soft processors, while another design requires the same FPGA to be configured as a fully synchronous data flow machine.

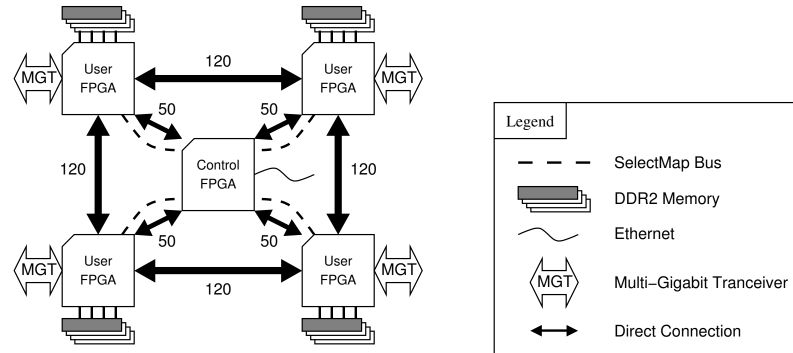


Figure 2.11: Block diagram of a BEE2 compute module

2.4 BEE2

BEE2 is an FPGA-based reconfigurable computer built at the Berkeley Wireless Research Center (BWRC) around year 2003[7]. Figure 2.11 shows a high level block diagram of a BEE2. Each BEE2 compute module contains 5 Xilinx Virtex-2pro (xc2vp70) FPGAs. The center FPGA, usually referred to as the *control FPGA*, is connected to each of the remaining 4 *user FPGAs* through an 8-bit SelectMap bus and a 50-bit direct connection. The control FPGA is usually responsible for system functions such as to configure the 4 user FPGAs and network communications. The four user FPGAs are connected in a ring topology. They are primarily used for executing user gateware.

Currently, BORPH is implemented on a BEE2 compute module. A BEE2 compute module exhibit the architecture depicted in Figure 2.1(f) in which each FPGA itself can be treated independently as a RC. For a BORPH system, an embedded PowerPC processor system is built using the control FPGA as a controlling processor system. The four user FPGAs are then used as reconfigurable fabrics to execute user

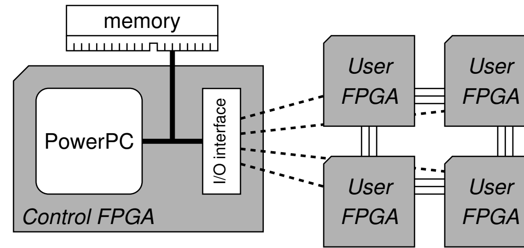


Figure 2.12: BORPH utilizes a PowerPC based reconfigurable system that is built using a BEE2 compute module.

gatewares. Figure 2.12 shows the resulting logical construction of a BORPH system and their partition over control and user FPGAs. Note that the resulting virtual system becomes one of those depicted in Figure 2.1(a).

2.5 Summary

This chapter explored the broad design space of reconfigurable computers. Reconfigurable computers can broadly be defined as computers that compute with dynamically reconfigurable control and data path. They are usually constructed from zero or more physically presented sequential processors coupled with one or more reconfigurable fabrics.

From a physical construction point of view, reconfigurable computers can roughly be classified according to both the way processors are coupled with reconfigurable fabrics in the system and the designs of the systems' reconfigurable fabrics. Reconfigurable fabrics can either be fine-grain or coarse-grain reconfigurable. The former allows maximum reconfiguration flexibilities while the latter optimizes system performance for certain task. Furthermore, the flexibilities of fine-grain PLDs allow them

to be programmed as virtual machines with application dependent compute model.

From an abstract computational model point of view, reconfigurable computers complete the continuum of compute model from fully sequential stored-program processor to fully spatial computation as in the case of an ASIC. It is this flexibility that allows RCs to deliver orders of magnitude higher performances than conventional machines.

BORPH is currently implemented on a PowerPC based system built using FPGAs of a BEE2 compute module.

Chapter 3

BORPH: The Operating System

The Berkeley Operating system for ReProgrammable Hardware (BORPH) is an operating system designed specifically for FPGA-based reconfigurable computers. The design goal of BORPH is to improve usability of FPGA-based RCs through operating system support[39]. In particular, BORPH approaches this goal by extending the familiar UNIX semantics to reconfigurable computers[40]. BORPH provides kernel support for FPGA applications similar to the way conventional OS provides support for software programs. On one hand, BORPH manages reconfigurable fabrics of a RC just like other processor resources such as CPU time and memory, allocating them to user applications as needed. On the other hand, BORPH isolates user designs from many low-level details about the system so that a user may focus on the actual application development.

The choice of UNIX semantics is both technical and ideological. Ideologically, the UNIX semantics is well understood and well studied by both software and hardware engineers. Such familiarity helps lower the barrier-to-entry for engineers with software or hardware background into the field of hardware/software codesign in FPGA-based

reconfigurable computer. Technically, some UNIX semantics, such as the UNIX file stream and file pipe concepts, resemble closely the semantics of digital signal processing (DSP) applications, which is one of the most important class of problems that can benefit from RCs. Finally, the Linux operating system, an open source UNIX implementation, has been widely ported to FPGA platforms. It allows BORPH to be ported to a number of different RCs with relative ease.

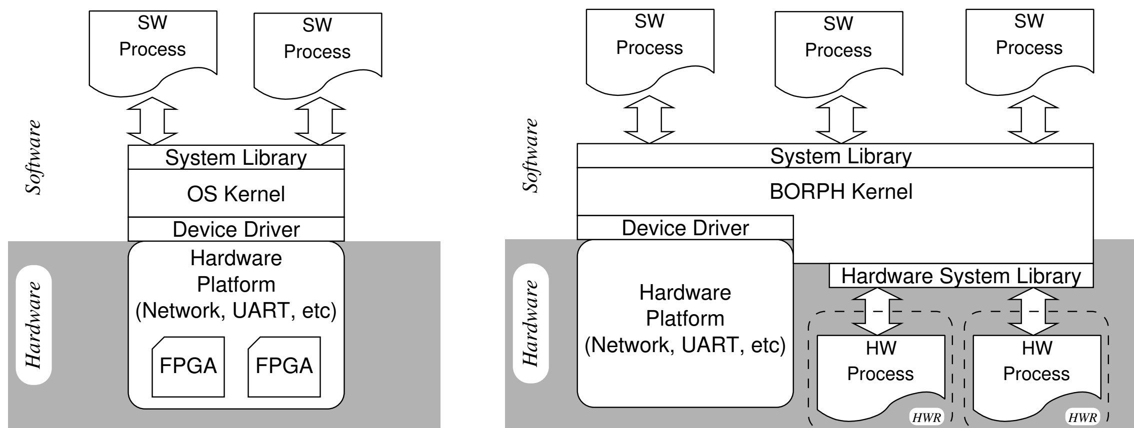
In a nutshell, BORPH addresses the run-time system integration issues with reconfigurable computer applications as outlined in Chapter 2. It addresses the FPGA applications I/O problems systematically through UNIX I/O services. Reconfigurable resource allocation and concurrent multi-user support are embedded in BORPH's kernel design. Furthermore, it provides a unified environment where meaningful end-to-end performance comparison among software, hardware, or hardware/software solution to certain problem be performed. Finally, BORPH provides an OS framework for future RC researches. For instance, without the BORPH framework, dynamic reconfiguration of FPGA is limited as a research topic that exists only on paper. The BORPH framework provides it with a context for implementation and comparison. It enables meaningful discussions such as whether dynamic reconfiguration should be performed at a system level during process startup or as dynamically linked library loading, or simply be used as part of the underlying executing model of the application.

3.1 Architectural Assumptions

Because of the wide range of reconfigurable computer architectures, it is useful to limit the discussion of BORPH to a subset of the architectures that BORPH is designed for. The design of BORPH makes very few assumptions about the underlying hardware platform. It is designed to run on RC systems that consist of a main controlling processor coupled with a one or more reconfigurable fabrics such as those depicted in Figure 2.1(a) to (c). A generic machine architecture for the current BORPH implementation was shown in Figure 2.12. Note that the specific architectural implementation subsequently affects only the performance of the system rather than its correctness. For example, performance of our current implementation has been improved multiple times through architectural changes as described in Chapter 4.

3.2 Machine Abstraction Model

Fundamental to the design of BORPH is its abstraction model of the underlying machine and reconfigurable fabrics. Although the underlying physical machine architecture is similar to most conventional RCs, BORPH is unique in its way to abstract reconfigurable resources of the machine.



(a) Reconfigurable fabrics as part of hardware platform.

(b) BORPH: Reconfigurable fabrics as computing resources

Figure 3.1: Two ways of organizing reconfigurable fabrics in a reconfigurable computer.

3.2.1 Reconfigurable Fabrics as Coprocessors

Figure 3.1(a) illustrates a conventional method of organizing reconfigurable resources on a reconfigurable computer. In such systems, reconfigurable fabrics are managed as part of the underlying hardware support platform, similar to the way a network interface card or a hard disk is being managed. Under such an abstraction, using reconfigurable fabrics for computation involves two steps. First, a user design, in the form of a configuration file, must be loaded to the underlying reconfigurable fabrics. Once a reconfigurable fabric is configured to perform certain function, it has essentially becomes part of the underlying hardware platform. Consequently, similar to the way a software program communicates with any other hardware devices, it communicates with the user design through a layer of device driver. There are a number shortcomings associated with such abstraction.

First, the controlling processor inherently forms a *master-slave* relationship with reconfigurable fabrics in the system when they are treated as part of the underlying platform. Consequently, communications with a user design configured in these reconfigurable fabrics are initiated from the processor most of the time. For instance, a software program typically initializes “hardware acceleration” by sending a block of data to the accelerator implemented on the system’s reconfigurable fabrics. It would then enter a polling loop until data is returned from the hardware accelerator. The polling may be replaced by interrupts if the software is running in kernel mode, such as in the case of a device driver. Such a software-centric methodology is sufficient when reconfigurable fabrics are used to accelerate software applications. However, it becomes cumbersome or even awkward in cases when the reconfigurable hardware design is the application itself. For example, when FPGAs are used for performing high-speed digital signal processing in hardware, there is little need for an extra software program that a user must program and manage.

Complicating the situation is the lack of standardized semantics for software programs to communicate with designs running on reconfigurable fabrics. In a typical computer system, different device drivers are written for different hardware devices that are connected. Although there exist standardized device driver interfaces, the task of developing device drivers are themselves non-trivial for most high-level application developers. Moreover, the fact that each application has its unique communication requirement inevitably implies a new device driver must be written for each application.

The lack of standardized I/O semantics is not limited to hardware/software communication within a system. Different RC systems developed by different researchers or companies often define their own interfaces and semantics. The lack of common semantics makes the already difficult task of porting a design from one RC to another even more daunting. This in turn hinders wide scale mainstream developments of reconfigurable computers.

Finally, modeling reconfigurable fabrics as part of the underlying machine platform obscures the distinctions between a machine's physical hardware platform and the hardware "configurations" that correspond to user applications. In other word, the notion of gateware may easily be lost when reconfigurable fabrics of a system are treated by the OS the same way as the underlying support platform.

3.2.2 Reconfigurable Fabrics as Computational Resources

BORPH logically separates reconfigurable fabrics of a reconfigurable computer from its underlying hardware support platform. The smallest unit of reconfigurable fabrics managed by BORPH is denoted a *reconfigurable hardware region (*HWR*)*. Figure 3.1(b) illustrates this concept. For example, the current implementation of BORPH on BEE2 defines each user FPGA as a HWR. HWRs are mere extensions to the controlling sequential processor as computational resources of the system. Their sole purpose in the system is to perform computation for user applications. In particular, they are used primarily for executing user gateware designs.

As an independent executing entity of the system, a gateware design has a *peer-*

to-peer relationship with other software or gateware designs. In other word, the controlling processor no longer forms a master-slave relationship with reconfigurable fabrics. Communication may therefore be initiated from gateware designs independent of any controlling software. Such *active communication* allows a gateware design to have tight control on data movement and potentially increase over system performance. File I/O described in Section 3.5 is an example of gateware-centric communication because file I/O operations are initiated by gateware. Traditional software-centric communications, such as to copy a block of memory, are handled by BORPH's IOREG virtual file system described in Section 3.6. As a result, high-level application developers are freed from the task of developing ad-hoc device drivers for each gateware application. All communications are handled by familiar and standard UNIX file I/O semantics.

3.3 Hardware Process

In conventional OS terminologies, a *process* is usually defined as *an executing instance of a program*. It means that the software program represented by an executable file becomes a running process when it is executed. Each process is allocated its own unique process ID together with its executing environment. A process forms a parent-child relationship with its spawning process.

BORPH extends this idea to reconfigurable computers, defining a *hardware process* as *an executing instance of a gateware program*. In other word, a hardware process is similar to a conventional software process except it *may* be executing on

reconfigurable fabrics of the system instead of the main processor. The notion of execution domain of a process is therefore extended to include spatial information, such as the reconfigurable fabrics that this process is executing.

The concept of hardware process allows a gateware design to transform from a passive entity under the control of some software program to an independent executing entity. Such transformation is essential to create a logical one-to-one mapping between user application and its corresponding executing instances in the system. Without such hardware process concept, application designer must develop both a gateware design executing on reconfigurable fabrics, as well as a software program that manage that gateware design. The concept of hardware process allows that only one gateware program be developed for each user application. This gateware design will then play an active role in interacting with the rest of the system during run time similar to the way software programs do.

3.3.1 Process Creation

A hardware process is created when a BORPH Object File (BOF) is `exec`-ed. As shown in Figure 3.2, a BOF file is a binary file format that encapsulates, among other information, configuration for reconfigurable fabrics. In conventional UNIX systems, a process is created with two system calls: `fork` and `exec`. When a hardware process is created, the same `fork-exec` sequence is employed. During the `exec` system call, the actual HWR is setup according to the configuration in the corresponding BOF file. Since hardware process creations are handled by the kernel, a hardware design can

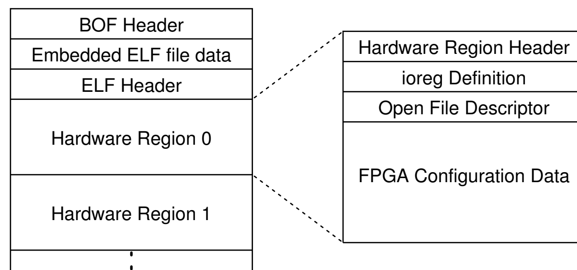


Figure 3.2: Simplified BOF file format

be started by any software program that is able to create normal UNIX process, such as the command shell, a C or Java program, etc.

Hardware process creations conform to the standard UNIX process creation semantics by maintaining all necessary parent-child and process group information. The created hardware process has its own memory space and execution domain. As a result, there is no shared memory between hardware and other processes in the system by default. Currently, memory attached to hardware processes must be exported by the IOREG interface. Each hardware process also has access to its execution environment, including its executing command line arguments. Similar to a conventional UNIX process, a hardware process occupies its own logical virtual memory space. In other word, there is no implicit memory sharing between a hardware process with any other processes in the system.

3.3.2 The Lifecycle of a Hardware Process

Figure 3.3 shows a transcript of executing a BOF file from a `bash` shell. The file `counter.bof` contains a free running counter design to run on a FPGA. When this

```
1:bash$ ./counter.bof &
[1] 1399
2:bash$ ps j
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 1386  1387  1387  1387 pts/3     1400 Ss    1000   0:00 -bash
 1387  1399  1399  1387 pts/3     1400 S     1000   0:00 ./counter.bof
 1387  1400  1400  1387 pts/3     1400 R+    1000   0:00 ps j
3:bash$ kill -9 1399
[1]+  Killed       counter.bof
4:bash$
```

Figure 3.3: Executing a BOF file containing a free running counter. FPGA hardware is configured at prompt 1 and is unconfigured at prompt 3.

file is executed at prompt 1, based on the file's header, the BORPH kernel select and configure an unused FPGA of the system for this file.

Once a BOF file is spawned, its status can checked by standard command like `ps` as shown in prompt 2. The output of the command `ps` shows the parent-child relationship between the starting `bash` shell and the hardware process `counter.bof` as well as its process group information. Of interest is the `STAT` column in the output, which shows `counter.bof` is at an "interruptible sleep" state. In our current implementation, to avoid a hardware process being put on the processor's run queue, it is marked with a Linux process state of `TASK_INTERRUPTIBLE`. In the future, a new process state will be introduced to indicate to the rest of the system that a process is being run on a HWR.

Similar to a software process, a hardware process can be terminated either by external UNIX signals (`SIGTERM`, `SIGKILL`), or by executing a gateware equivalence of `exit` system call.

3.4 Kernel Interface

This section explores the design of BORPH's kernel-user interface. The parallel and high performance nature of gateware designs impose a number of requirements on this interface that are not presented in conventional single processor systems.

In a conventional single processor system, both OS kernel and user processes are software executing on the same processor. However, in a FPGA-based reconfigurable computer, some of the user processes, specifically hardware processes, may be executing on reconfigurable fabrics instead of the processor that the software OS kernel executes. Furthermore, gateware designs run at a much higher speed and potentially generate requests to the OS kernel at much higher rate than a processor can handle. As a result, to provide low level support for these user gateware designs, some of the OS kernel functionalities must be implemented in gateware as well.

The part of BORPH OS kernel that executes in a processor, performing conventional OS functionality and interacting with normal software processes is called MK. The part of BORPH OS kernel that must be implemented in gateware to interact with hardware processes is called UK. The two are logically part of the same operating system kernel. However, depending on the system (e.g. our current implementation), they may be physically separated. This section focuses on the interface between BORPH's hardware kernel (UK) with hardware processes.

Typical software systems have the operating system kernel manages all system resources such as CPU, memory and I/O. The OS then provides user programs with a set of services, such as reading a file, through a set of predefined *system calls*.

System calls play the role of application program interface (API) into the OS kernel. Since crossing the user/kernel boundary involves low-level functions such as modifying memory protection and CPU privilege mode, the actual system call calling mechanism is machine dependent and is defined in a system's application binary interface (ABI).

The calling user program is usually suspended (blocked) while the OS services a system call. The kernel has complete control over how to proceed with the system call. Having such kernel-user separation not only shields a user program from all the low-level system details, but also allows the kernel to perform any management work, such as enforcing security and resource allocation as needed.

To serve similar management functions to hardware processes, BORPH defines a similar hardware system call interface. Similar to software OS ABI, the hardware system call interface must be kept at the lowest physical level such that it is design language independent. However, the parallel computation model of gateware applications post a number of requirements not commonly found in a software system. Specifically, gateware programs in general require both an asynchronous and a parallel interface into the BORPH kernel.

Asynchronous System Calls

System calls are conventionally executed synchronously to the executing user program. The user thread that requests system services is usually blocked while the kernel services the request[1]. Gateware applications, however, exhibit a parallel computation model in which every part of the design is performing certain function concurrently.

[1]Some new OS's are beginning to have provision for asynchronous I/O system calls.

As a result, there is no single thread of control that the OS may block while it services a system call. In other word, if one part of a user hardware design issues a file read request that blocks due to lack of data, it is unwise to block the entire user FPGA design because other part might, by design, still be operational. This is similar to a multi-threaded software program running on a single processor. If one thread must be blocked due to OS services, other threads must still be allowed to proceed. Without making any assumption about running user hardware designs, hardware system calls must therefore always be executed *asynchronously* to the user program. The handling of such asynchronous OS communication is left as a responsibility of user space designs.

Concurrent System Calls

Since gateware designs compute in parallel, it is possible that multiple portions of the same design demand OS services concurrently. In a single processor system, the OS kernel may service "concurrent" system calls without affecting overall system performance because such concurrency is limited only to the logical level. At the lowest level, the system processor must serialize the computation regardless of the software concurrency. On a hardware system, however, it is possible such concurrent system calls are issued on the exact same hardware cycle. For example, a dataflow hardware application may open 2 files for input and 1 file for output. For performance reasons, it is desirable to service the output file write and 2 input file read on each and every hardware cycle.

High Data Bandwidth Requirement

To minimize performance impact on a user gateware process, the BORPH kernel must ideally be able to provide OS services at hardware speed. It is particularly important for performance critical system services such as file I/O. On the other hand, some system functions, such as `sys_gettimeofday` that gets a system's time, has very low latency and bandwidth requirements. As a result, the design of BORPH's kernel interface must be flexible and generic on one hand, while extremely high performance on the other hand.

3.4.1 Hybrid Message Passing System Call Interface

The current BORPH design employs a hybrid message passing interface as its system call interface. It is "hybrid" because by means of high level handshaking protocol, the interface can be changed from operating in a message passing mode into a cycle accurate data streaming mode.

As shown in Figure 3.4, the BORPH kernel connects to user gateware programs with a predefined number of *I/O connection points* (`ioc`). Each `ioc` primarily consists of 2 unidirectional connections in each direction. All communications between the uK and the user are performed over this generic interface, usually in the form of request/acknowledge packets. With respect to file I/O, they can be thought as a physical realization of the logical "file descriptors" in software.

The left hand side of Figure 3.5 shows the physical interface between an `ioc` and a user application. Being essentially two 8-bit unidirectional connections guarded with

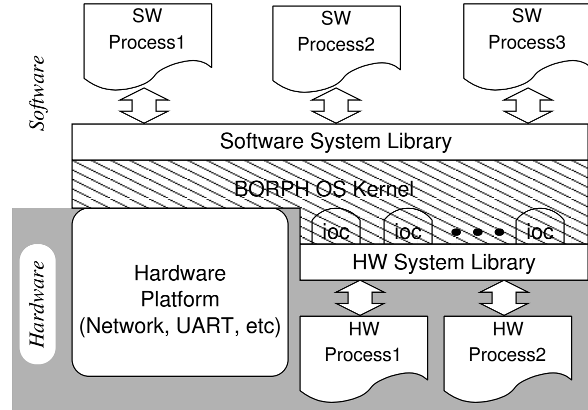


Figure 3.4: BORPH's hardware kernel interface is constructed from one or more I/O connection points (`ioc`)

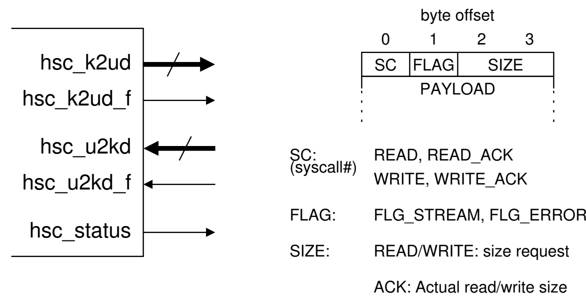


Figure 3.5: Left: BORPH's hardware system call (`hsc`) interface. Right: Format of packet transmitted through `hsc`.

a 1-bit valid (framing) signal, this physical interface is simple and generic by design. For example, keeping the data connection as byte-wide both simplifies protocol design and saves physical resources so that more `ioc`s can be instantiated for more parallel filesystem access if needed.

All kernel-user communications are carried out by a high-level packet based message passing protocol through `ioc`s. As shown in right hand side of Figure 3.5, the packet format is again simple and generic by design. In general, user applications request hardware system calls by sending messages to the kernel. The protocol man-

dates a request-acknowledge sequence between user and kernel for most system calls. Acknowledgement messages are used by the kernel to return any requested data or error status to the user.

As mentioned earlier, each `ioc` may independently switch from a message passing mode into streaming data mode. The discussion on the motivation and specific switching protocol is deferred to the next section as it is used solely for streaming file I/O.

3.5 File I/O

One unique service that BORPH provides to gateware designs is general UNIX file system access. This section describes the design consideration for BORPH's filesystem layer specific to gateware designs.

Unlike most other RCs, file system access provides FPGA designs an *active* mechanism for data I/O. In other word, it is a gateware design that logically and physically initiates such I/O communications. It is different from most conventional FPGA-based RCs, where FPGA designs only receive passive communications that are initiated by certain software programs running on the controlling processor. With respect to usability of the system, the presence of active I/O services is intuitive because it logically corresponds to the active, self-contained nature of FPGA designs running as gateware programs. Technically, allowing gateware designs to control their own I/O patterns allows much higher performance potential. A gateware design may, for example, initiate a long read request when its input buffer is running low on data.

Without an active reading mechanism like file read, the software supplying data to the gateware process must perform a second duty of monitoring the status of a gateware design's input FIFO status. Such double duty is both counter-intuitive and cumbersome, lowering over system performance.

In theory, providing file system access to FPGA designs can be as simple as providing a proxy between the FPGA and the processor. In practice, however, because of the unique nature of gateware processes, there are some subtleties that must be taken into account when designing BORPH's file system layer.

3.5.1 Differences between Software and Hardware Design Patterns

Designing for reconfigurable hardware inherit most of the design patterns from traditional synchronous digital hardware designs. All physical connections are expected to carry valid data on every sampling clock edge. No erroneous situation is ever assumed on the physical connection level because any error in the physical layer always results in system-wide failure. For example, synchronous data flow (SDF) designs often assume valid data is streamed into the design on every cycle. There is no notion of erroneous data. If the input data is invalid, a SDF design simply produces erroneous data.

On the other hand, file system access through the operating system kernel imposes a number of semantics not commonly presented in hardware designs. First, file system access is a logical layer service. Both read and write functions involve a pair of

request-response handshaking between a user design and the kernel. On the other hand, gateware designs usually assume a physical data connection that supplies data on every cycle without the need of any “request.”

Secondly, conventional UNIX file system semantics does not guarantee that the exact amount of reading/writing data requested by a user process will be return by the kernel. For instance, in the case of a file read, the kernel may choose to return less data than the amount requested by a user design due to the lack of data from the actual file or simply because the read system call is interrupted by a signal. Similarly, in the case of a file write, the kernel may choose to commit less data than requested by a user design due to lack of hard disk space. In both cases, partial reads/writes are considered normal file system behavior. However, for sake of simplicity, most hardware design assumes any data movement request will be honored exactly.

Finally, common to most system calls, both file read and write system calls have provisions for error situations and other “out-of-band” situation such as the reach of end-of-file. On the other hand, gateware designs tend to separate control and data into different channels and rarely handle situation such as end-of-file.

Filesystem Support in BORPH

Providing filesystem support to gateware designs implies that all the above mentioned file system semantics must be appropriately handled. As mentioned in Section 3.4, all gateware system calls are handled by BORPH’s hybrid message passing kernel interface. The design of messages specific to file system read and write are

such that the above software-centric I/O semantics are retained. This is to keep the interface generic enough that gateware designs with smart state machine or embedded system may choose to handle these semantics manually. For other gateware designs that choose to ignore these software semantics, a layer of user space hardware system library may be employed.

Figure 3.6 shows BORPH's message definition for file system access, together with their corresponding UNIX system call prototypes. For `read` and `write` system calls, the actual amount of data written or read is returned in the SIZE field of the corresponding ACK message. Following UNIX convention, this size is not necessarily the same as the amount in the original request packet. Furthermore, end-of-file status and any error condition are flagged in the FLAG field, with the error code embedded in the SIZE field.

Note that there is no corresponding file descriptor number (`fd`) in the packet definition. Each physical instance of `ioc` is associated with the corresponding file descriptor number when the file is `open`-ed. As a result, even in lieu of user specification, any system call initiated by this `ioc` is automatically associated with that file descriptor number. Furthermore, there is no information about *offset* into the file that a user application is accessing. Similar to normal UNIX kernel, the current file position is maintained by the kernel inside `ioc`. This file position is updated with each successful file read/write. This is designed both as a way to simplify the packet definition for user, and to allow some degree of hardware kernel management.

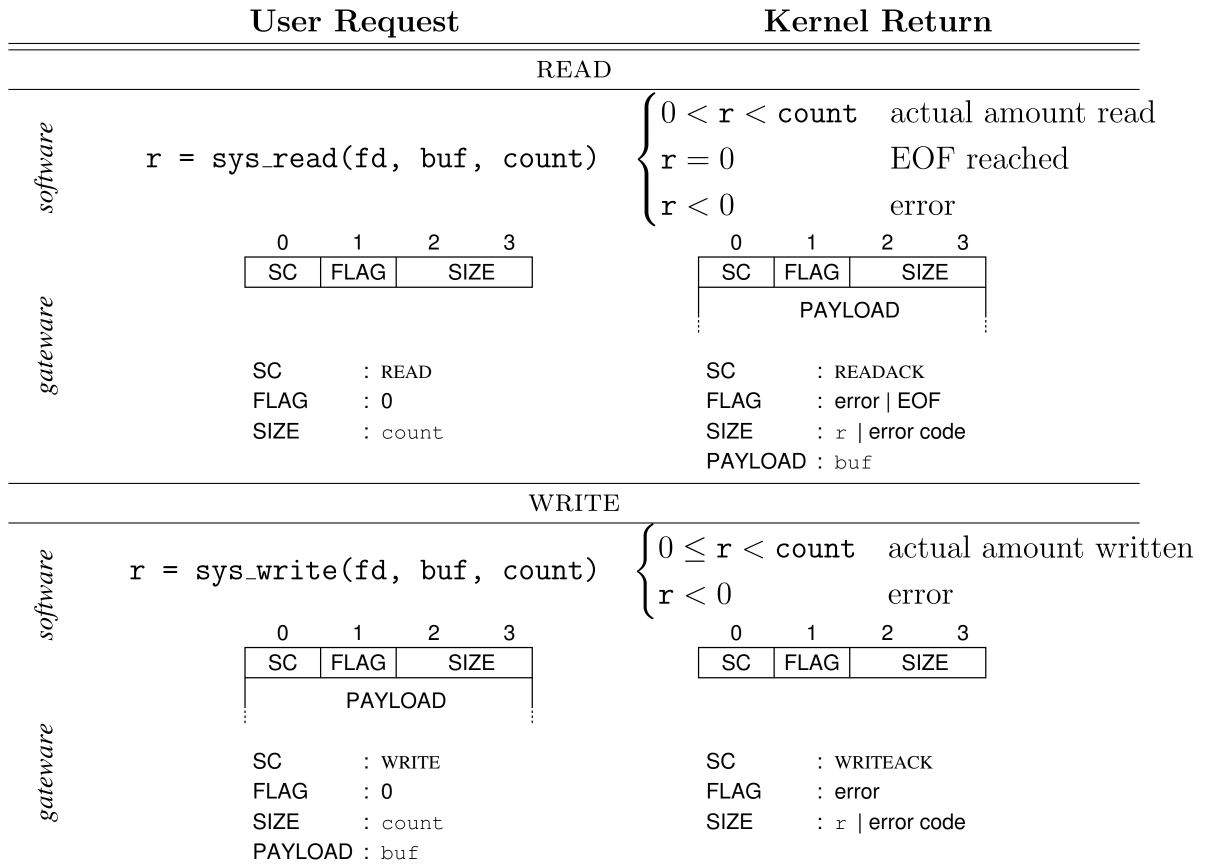


Figure 3.6: BORPH's message format for file system read/write resembles its corresponding UNIX system call prototype.

3.5.2 High Speed Streaming Data I/O

As observed by a number of researchers[13, 20], streaming data I/O presents one of the most intuitive abstraction as well as implementation for gateware applications. At the same time, conventional UNIX systems have the concept of software pipelines, or *pipes*, that represent unidirectional streams of data from one process to another. Based on these two observations, BORPH extends the conventional notion of UNIX file pipes to represent native streaming data for gateware designs.

Unix Pipe

In a UNIX system, a file pipe, denoted by a "|" character, is used to chain a series of programs in the shell command to collectively perform certain complex functions. Standard output of one program is fed as standard input to the next program in the chain. The OS kernel buffers data on behalf of the pipe, blocking programs on either side of the pipe as needed so not to overflow or underflow this internal buffer. Pipes can be created either from the command line or from within a program. Without lost of generality, we will restrict the discussion to shell command. For example, the following command finds all the lines in the file `student.db` containing the word `freshman`, sort the resulting lines and save the output to `freshman.db`:

```
bash$ cat student.db | tr 'a-z' 'A-Z' | grep 'FRESHMAN' | sort -o freshman.db
```

Furthermore, since "pipes" are simply special forms of "files" in UNIX, user programs are not required to differentiate between the two. During run time, using I/O redirection, a program that is designed to read from standard input may read from

either a regular data file or a UNIX pipe. For example, in the above command, instead of reading standard output from the program `cat` through a pipe, the program `tr` may read directly from the file `student.db` by the following command:

```
bash$ tr 'a-z' 'A-Z' < student.db | grep 'FRESHMAN' | sort > freshman.db
```

Streaming Data I/O Using UNIX Pipes

BORPH's virtual file system layer extends the concept of UNIX file pipe to hardware processes such that the end points of a pipe may either be a software or a hardware process. There are two cases to consider: (a) hardware/software pipe and (b) hardware/hardware pipe.

Providing hardware/software pipe support in BORPH is relatively straight forward because of the way UNIX systems represent most passive system resources using a unified virtual "file" abstraction. For example, regular data files, sockets, device driver handles, pipes, etc. are all represented as files. All these virtual files are managed by the kernel's virtual file system layer using identical interfaces. As a result, user programs in general do not need to differentiate the specific realization of an opened file. Such flexibility provides hardware processes the ability to communicate with standard software processes via file pipes using the identical mechanisms that are used to access regular data files. Specifically, the very same READ and WRITE messages described in the previous section may be used to read/write *any* UNIX virtual files, including hardware/software pipes.

On the other hand, supporting hardware/hardware pipes requires careful design

considerations to balance performance with kernel I/O manageability. When two hardware processes stream data to each other, it is desirable that the data stream be uninterrupted by the kernel for performance sake. In fact, a fully synchronous data flow (SDF) application requires data streaming at system clock rate. However such direct data streaming violates the usual request-acknowledge system call semantics for regular file accesses.

In general, to efficiently support hardware/hardware pipes, three requirements must be met:

1. To maintain performance, a hardware only management schemes must be presented to connect hardware processes directly without software intervention.

2. BORPH's hardware system call interface (`hsc`) must be generic enough to accommodate the two different I/O semantics.

3. A combination of both `hsc` and hardware library must allow switching between the two modes of operation during run time in response to I/O redirections.

Once all three requirements are met, a compiled user gateware design may then be able to switch between three mode of operations with simple I/O redirection: To read/write to regular data files, to stream data to standard software processes, and to stream data in high speed to another hardware process. For instance, a video filter design implemented in FPGA may read from a known video data file during development and debugging stage. Once it is stable, the same design will then be used to connect to a hardware video source for real-time processing.

3.5.3 Runtime Streaming Mode Switching Support

The BORPH hardware system call (`hsc`) interface is designed to work with both the normal packet based message passing mode (*packet mode*) and the high speed hardware streaming data mode (*streaming mode*) with user. The same physical interface is used in both modes. Support for different communication mode is handled by a high-level hand shaking protocol between the kernel and user.

On the kernel side of `hsc` is `iock`. While operating in the initial packet mode, it relays any read/write requests from the user to mK. It also buffers any data returned by the software kernel before sending back to the user. Once switched into streaming mode, it disables most of the packet marshaling and data buffering features. During streaming mode, its main responsibility is to route data streams from user gateware programs to BORPH's streaming data network. It monitors the connections and when instructed by mK, terminates or suspends the connection.

During run time, a user application may request a streaming data connection by setting the top bit in the FLAG field of a READ (WRITE) request packet. A streaming connection must be established during run time by `iock` because the connection peer of an `ioc` is only known when the corresponding file is opened. The two sides of a pipe can be thought as a connection peer. A connection point is *streamable* only if it is able to stream data through BORPH's streaming network. If both sides of a connection are streamable and both of them request streaming connections, then `iock` engages in a sequence of handshaking messages with the BORPH kernel to setup a connection with its connection peer (Figure 3.7). Once the streaming connection

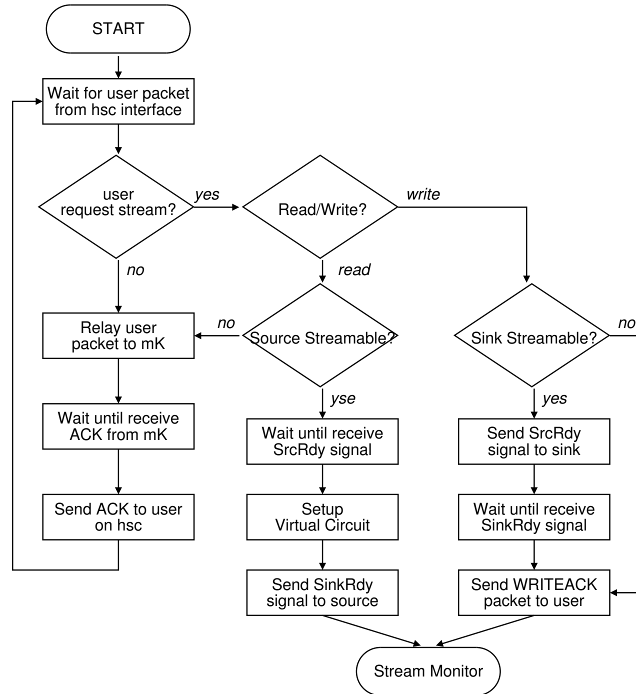


Figure 3.7: A simplified flow diagram for `iock` packet handling. A streaming data connection is setup iff both source and sink are streamable and are ready to stream data.

is setup, the source will be allowed to stream data by sending a WRITE packet with infinite payload. As a result, the sink will receive an infinitely size READACK packet containing the streaming data.

3.5.4 File Streaming I/O Library

While the introduction of a complex communication protocol provides an extremely flexible and efficient kernel-user interface, it substantially increases the burden on user design effort. Recall that one of the primary motivations for BORPH is to improve usability of reconfigurable computers. Therefore, it is essential that the burden of kernel-user communication be relieved from high-level application designers.

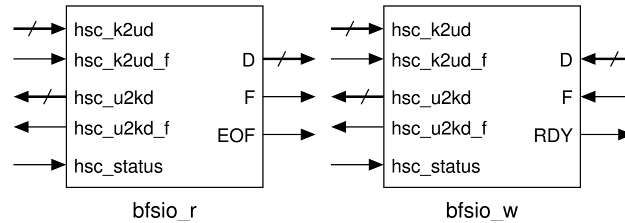


Figure 3.8: The `bfsio` library contains two blocks. The read (bfsio_r) block generates read requests on behalf of the user and relays only data payload and EOF information to user. The write (bfsio_w) translates a stream of data from user into individual write request packets to the kernel.

This is achieved through a layer of hardware system library.

This layer of hardware system library serves the same purpose as standard UNIX system library such as `libc`. The purpose of this layer is to hide all low level kernel communication complexities from the user. Furthermore, by means of an OS-neutral interface to the user, this layer of system library provides added portability for application developers.

As an illustration, a library of file streaming I/O library, `bfsio`, has been developed. As shown in Figure 3.8, this library contains two blocks; a streaming file read block (`bfsio_r`) and a streaming file write block (`bfsio_w`).

For file reads, `bfsio_r` initially attempts to switch into streaming mode by sending a streaming read request into `hsc`. If the streaming read request succeeded, the returned streaming data together with its flow control information is passed unaltered to the user. The user therefore sees a continuous stream of data from the data port (`D`) of the block. If the streaming read request failed, meaning the device that it is streaming is not streamable, such as the case of a regular data file, then `bfsio_r` resolves to repeatedly requesting packet mode read of a predetermined block size.

The data valid port `F` is asserted only when the kernel has returned valid data that can be passed back to the user. Its operation ends when an EOF is signaled by the kernel, at which time the port `EOF` is asserted.

For file writes, `bfsio_w` initially attempts to switch into streaming mode by sending a streaming write request into `hsc`. If the streaming write request succeeds, it signals `RDY` to the user. Any data sent by the user is subsequently passed directly through `hsc` as payload of an infinitely sized WRITE packet. If the streaming write request fails, then `bfsio_w` falls back to packet mode communication and resumes user data buffering. A WRITE packet is sent when the number of user data exceeds certain threshold or when the user asserts the `flush` signal.

By using this library, FPGA application developers only see a simple byte wide streaming data port with simple flow control for both read and write. With all the complexities of data buffering and kernel handshaking handled by such hardware library, users are left with the sole job of application development.

3.6 The ioreg Virtual File System

While standard UNIX file system access provides an active communication mechanism for hardware processes, BORPH's IOREG virtual file system offers conventional passive, software centric communication services.

In a processor-centric system, attached hardware devices are typically controlled by a set of special hardware registers that are memory mapped by software device drivers. BORPH encapsulates this common design practice by supporting it sys-

tematically via its IOREG interface. Instead of requiring each gateware designer to implement a new device driver for each new gateware application, the IOREG interface allows gateware design to expose predefined logical constructs as virtual files in the UNIX file system similar to that presented in [14].

3.6.1 Basic Operation

The basic idea of BORPH's IOREG virtual file system is to allow user to communicate with a running gateware design through simple UNIX file accesses. The actions of BORPH's IOREG virtual file system layer are initiated when a user reads or writes to an IOREG virtual file. The read/write file operation causes the kernel to perform machine dependent communication with the corresponding hardware construct. In our current implementation, the BORPH software kernel MK communicates with hardware processes via a BORPH specific message passing network[2]. As a result, a read file operation will causes the kernel to read live data from the executing gateware process, while a write operation causes the kernel to send the written data to the hardware process. No caching is performed by the kernel. Therefore, every read/write operations are translated into their corresponding communicating messages.

3.6.2 Organization of ioreg Virtual Files

All IOREG related virtual files are located under the running process's own `/proc` directory. Modern Linux systems populate the system's `/proc` directory with virtual

[2]Note that it is not the same message passing interface as `hsc`.

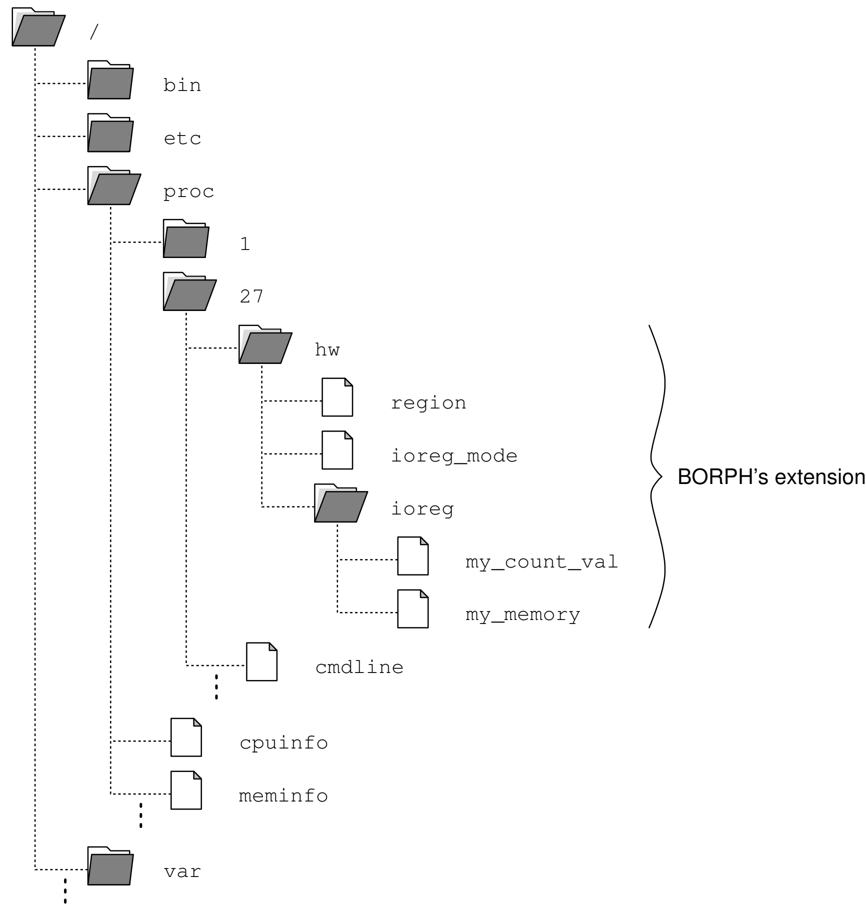


Figure 3.9: BORPH extends a standard Linux system's `/proc` directory with hardware process specific information.

files that contain information about the running system (Figure 3.9). For example, reading the file `/proc/cpuinfo` always return information about the CPU of the system. Furthermore, for each executing process with process id `<pid>`, a subdirectory `/proc/<pid>/` is created and is populated with process specific information such as its executing command line (`cmdline`).

BORPH extends this idea by introducing a `/proc/<pid>/hw` subdirectory that is populated with virtual files specific to a hardware process. It currently contains two

files and one subdirectory. Reading the file `region` returns status about the HWR that this hardware process is executing on. The file `ioreg_mode` is a read/write file that controls IOREG virtual files to operate in either binary mode or ASCII mode. The subdirectory `ioreg` is where all virtual files corresponding to the hardware process are located. Each virtual file in this directory corresponds to one hardware construct embedded in the user gateware design. For example, the gateware design with process ID 27 in Figure 3.9 contains two software accessible hardware constructs: `my_counter_val` and `my_memory`.

All IOREG virtual files are created when a BOF file is executed. Name, unique identifier, size and access mode of a virtual file are obtained from the executing BOF file header.

3.6.3 Example

The design `counter.bof` in Figure 3.10 contains a free running counter that stores its output in an ioreg register named `cntval`. The operation of this counter is controlled by an enable register called `cnten`.

In Figure 3.10, the file `counter.bof` is initially executed at prompt 1. Based on the returned PID of 2458, a user may then list the content of its IOREG directory at `/proc/2458/hw/ioreg`. The directory listing indicates that it contains 2 files. The file `cnten` has a write only file permission because it is a write only register. On the other hand `cntval` has a read/write permission. Both of them have a file size of 4 bytes because all registers accessible through the IOREG interface are currently

```
1:bash$ ./counter.bof &
[1] 2458
2:bash$ ls -l /proc/2458/hw/ioreg
total 0
--w--w--w-  1 user user 4 May 27 14:31 cnten
-rw-rw-rw-  1 user user 4 May 27 14:31 cntval
3:bash$ cat /proc/2458/hw/ioreg/cntval
A3B498E0
4:bash$ cat /proc/2458/hw/ioreg/cntval
B289E906
5:bash$ echo 0 > /proc/2458/hw/ioreg/cnten
6:bash$ cat /proc/2458/hw/ioreg/cntval
C103D024
7:bash$ cat /proc/2458/hw/ioreg/cntval
C103D024
8:bash$ kill -9 2458
[1]+  Killed       counter.bof
9:bash$
```

Figure 3.10: BORPH's IOREG interface allows interacting with hardware processes via virtual files. The register `cntval` contains current value of the free running counter in `counter.bof`. This counter can be disabled by writing to `cnten` register.

restricted to a size of 32 bits.

At prompt 3, the value of `cntval` is read by the UNIX command `cat`, returning a 32-bit hexadecimal value of `A3B498E0`. When the same command is issued at prompt 4, a different value is returned because the hardware counter is running freely in FPGA. The free running counter's action is disabled at prompt 5 by writing a value of 0 to the file `cnten` using `echo` command. Subsequently, the two file readings at prompt 6 and 7 return the same value.

Finally, the gateware program is killed at prompt 8, releasing the corresponding FPGA resources.

Type	R/W	Seekable	Size
Register	rw	no	4 bytes
On Chip Memory	rw	yes	any
Off Chip Memory	rw	yes	any
FIFO (from user)	r/o	no	width×depth
FIFO (to user)	w/o	no	width×depth

Table 3.1: Types of ioreg

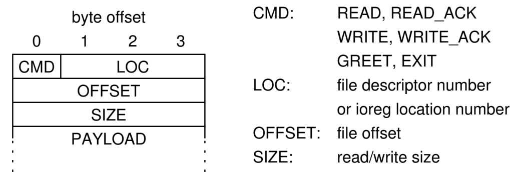


Figure 3.11: A simple packet format for message exchanges between BORPH main kernel mK and its distributed kernel uK.

3.6.4 Beyond Simple Register

Despite what the name might suggest, the ioreg interface supports not only simple single word registers. It also provides access to on-chip FIFOs, on-chip memories, as well as off-chip memories that a hardware process has access to. Table 3.1 shows the supported gateware construct by this interface and their differences when exported as virtual files in BORPH.

BORPH requires that the underlying machine dependent communication mechanism used by the ioreg layer be generic enough to handle all kinds of gateware constructs. It must be able to handle any requests with arbitrary *size* and *offset* values. For example, Figure 3.11 shows the packet format used by the current implementation of BORPH on BEE2.

With a generic underlying communication service, the differences among different

hardware constructs are enforced through top level file system permission settings. For instance, since logically an on-chip FIFO must always be read from the head, its corresponding virtual file is marked as non-seekable in the UNIX virtual file system layer, causing any file seek attempt to fail. As a result, all requests to a FIFO will always have an offset value of 0. On the other hand, an on-chip memory may be accessed randomly and therefore does not impose such restriction. Consequently, the kernel might initiate a request with non-zero offset to a user design.

3.6.5 Language Independence

Servicing IOREG communication in kernel space through an intuitive file I/O interface guarantees the language independence of this interface. For example, retrieving the contents of a 2048 bytes on-chip memory `Shared_Memory` can be accomplished by a simple shell `cp` command:

```
bash$ cp /proc/123/hw/ioreg/Shared_Memory ~/
```

or similarly, in a C program:

```
memfile = fopen("/proc/123/hw/ioreg/Shared_Memory", "r");
fread(buf, 2048, 1, memfile);
```

Being language independent allows a user to interact easily with gateware designs. In fact, a hardware process, utilizing its file I/O capability, may choose to interact with another hardware process via the very same IOREG interface. The flexibility of the IOREG layer allows complex tasks such as remote machine health monitoring to be implemented with even a simple UNIX shell script.

3.6.6 Operating Mode

The IOREG virtual file system layer can be operated in either a *binary* or *ASCII* mode. This operating mode is controlled on a per-process basis using the virtual file `/proc/<pid>/hw/ioreg_mode`. The examples in Figure 3.10 illustrates the operation in ASCII mode. In this mode, the value returned by a register is translated by the kernel into a hexadecimal string before returning to the user. For example, if the value read from a register is the value 58, the string “0000003A” is returned. Since the return values are human readable text strings, this mode is useful for simple command shell interactions with gateware designs.

For more complex operations, such as reading a large block of data from an on-chip memory, a user should set IOREG into its *binary* operating mode. In this mode, no translation is performed by the kernel. Raw data is returned to the user. For example, if the same value of 58 is read as above when IOREG is operating in binary mode, then a colon (:) character, which has an ASCII value of 58, will be displayed on screen.

3.7 Summary

This chapter has presented the design of BORPH. BORPH extends the familiar UNIX semantics to reconfigurable computers. Under BORPH, FPGA gateware designs execute as hardware processes, which are in most aspects identical to conventional UNIX processes. A generic, hybrid message passing system call interface is

defined through which user FPGA gateware designs communicate with the BORPH kernel. BORPH provides two primary I/O services to gateware designs.

First, BORPH allows gateware processes to access general UNIX file system. It provides hardware processes an active mechanism for both accessing regular data files and to communicate with other processes in the system through UNIX's pipe construct. Performances for pipes that connect two gateware processes are optimized by switching the message based connection into a streaming data channel, bypassing most of the kernel interventions.

Second, the IOREG virtual file system allows passive communication from the controlling processor to gateware designs. Gateware constructs within a gateware design, such as simple single-word registers, FIFOs, and memories, are abstracted as virtual files residing in the process specific `/proc/<pid>` directory tree. Reading or writing to these virtual files from any user program initiate direct communication with the running FPGA via the BORPH kernel. As a result, users may monitor and control running gateware designs asynchronously through simple file system operations. Because of the kernel's involvement, access to FPGA resources may be initiated by any UNIX programs: from simple shell scripts to complex compiled programs.

BORPH presents a new approach to using reconfigurable computers. By modeling reconfigurable fabrics of a reconfigurable computer systematically as native computing resources, BORPH establishes a solid foundation for reconfigurable computing research in a complete, self-contained framework. At the same time, the familiar UNIX semantics lower the barrier-to-entry for reconfigurable computers, allowing

novel users across different research domains to fully exploit potentials of reconfigurable computers. Such wide acceptance of reconfigurable computers is essential to foster future computing researches.

Chapter 4

Implementation and Performance

This chapter describes the current BORPH implementation and its performance on a BEE2 compute module. First, a high level overview of a BORPH system will be presented. Then, the design of BORPH's software kernel will be discussed. As mentioned in Chapter 2, BORPH runs on an embedded PowerPC processor system that is built using BEE2's FPGA fabrics. A base architecture of this processor system will be described as an illustration of basic BORPH implementation. Multiple architectural improvements are subsequently implemented to improve overall system performance. Performances of process creation, gateware file I/O and the IOREG virtual file system will be presented.

4.1 Overview

Figure 4.1 shows a high level block diagram of a BORPH system. BORPH has two main logical components: the MK and the UK. BORPH's main kernel, MK, is the main software operating system kernel that controls the entire system. It is similar in concept to conventional operating system and is currently implemented as

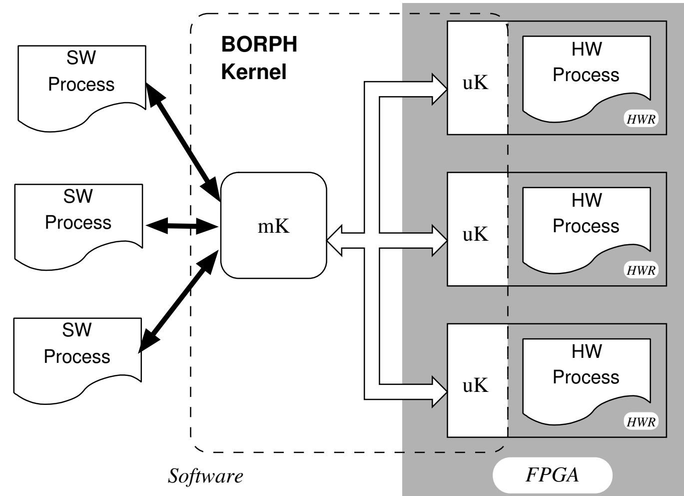


Figure 4.1: The two logical components of BORPH: MK and UK. MK is the main controlling software kernel. Each reconfigurable hardware region is individually managed by a low-level kernel called a UK.

an extended Linux kernel. The smallest unit of reconfigurable fabrics managed by BORPH is defined as a reconfigurable hardware region (HWR). Each HWR is managed by a hardware (or gateware) part of the BORPH kernel called UK. Each UK acts on behalf of MK for low level management of hardware processes, such as buffering file I/O data for gateware designs. As will be shown later, the communication performance between MK and UK has significant impact on file I/O and IOREG virtual file system's performance.

4.2 Software Kernel Architecture

BORPH main kernel, MK, is a modified version of a Linux 2.4.30 kernel running on the PowerPC 405 core in the control FPGA. A standard Debian PowerPC root file system is mounted over network file system (NFS), which provides familiar Linux

applications to the user. To provide kernel support for FPGA fabrics, a number of modifications have been made to the standard Linux kernel:

4.2.1 BOF file support

The standard Linux kernel contains an extensible interface to support user defined binary file formats. Making use of this interface, a binary file format kernel module called `binfmt_bof` has been developed to handle execution of BORPH Object Files (BOF). When a BOF file is executed, the `exec` system call initiates the `load_binary` function defined in `binfmt_bof`. It in turn triggers a series of BORPH specific functions such as allocating and configuring the necessary reconfigurable resources.

4.2.2 Reconfigurable Hardware Region (hwr) Support

A reconfigurable hardware region (HWR) is the smallest unit of reconfigurable fabrics in the system that the BORPH kernel uses for computational purposes. Since BORPH is designed to be portable to different reconfigurable computers, machine dependent definitions of a HWR are implemented as loadable kernel modules. A set of kernel APIs is defined for such HWR subsystem similar to the way device drivers are implemented in standard a Linux kernel. For instance, the HWR module must define a `configure` function that handles the details of configuring a particular type of HWR. The rest of the kernel, in particular the HWR execution daemon (`bkexecd`), calls this `configure` function without being burdened by the machine dependent details.

This abstract HWR definition and the extensible kernel API allows BORPH to

be ported to different RCs relatively easily. In our first implementation, we have defined a HWR type `hwr_b2fpga`, which corresponds to a user FPGA on BEE2. Porting BORPH to another reconfigurable computer will therefore primarily involve the implementation of a new HWR type as illustrated in Section 4.6.

4.2.3 FPGA Configuration and Resource Allocation

A kernel thread `bkexecd` is created to handle all HWR allocation and configuration routines. When a BOF file is executed by the user, `bkexecd` picks an unused HWR and configures that HWR accordingly. Users may override the behavior of `bkexecd` by locking a BOF file to specific HWR in the system. In our current implementation, this feature is used to lock a gateware design to be programmed on a particular user FPGA. This is useful for connecting user designs that require special external hardware devices that are available only to certain specific user FPGAs. In case a requested FPGA is not available, or no free FPGA is available for executing a relocatable BOF file, a device busy error (`-EBUSY`) is returned to the user.

4.2.4 Software Fringe

A hardware process may be blocked while accessing the general file system if the file is not ready. To handle this potential blocking, each opened file by a hardware process is managed by a software *fringe* running on the processor. The fringe performs file I/O operations on behalf of the hardware process and sends data back to the hardware process when it is ready.

4.2.5 Packet Communication Network

BORPH's current implementation utilizes a message passing network for communication between mK and uKs running on user FPGAs. A kernel thread (`mkd`) is responsible for handling all messages from hardware processes. It acts as the main message delivery hub where messages from all hardware processes are processed. Non-blockable actions, such as terminating a hardware process, are handled directly by `mkd`. For file I/O, the requests are relayed to the corresponding software fringe. The same packet communication network is used by the IOREG virtual file system layer. User read/write to IOREG virtual files are translated into packet messages that communicate with hardware processes.

4.2.6 Process Scheduler and Signal Handler

Since hardware processes do not run on the processor that BORPH runs on, the kernel must take special care during process scheduling. Hardware processes are handled differently such that they are never put on the software run queue.

Furthermore, since hardware processes may access terminal TTY's, they must respond correctly to "stopping" and "continuing" signals. For instance, a hardware process will receive `SIGSTOP` signal when a user press CTRL-Z on the running terminal, or `SIGTSTP` when it tries to read from a terminal while it is running in the background. Also, a hardware process receives `SIGCONT` when a user put the process in background, or back to foreground. The BORPH kernel is modified to handle these cases for hardware processes so that they conforms to standard UNIX semantics and thus be

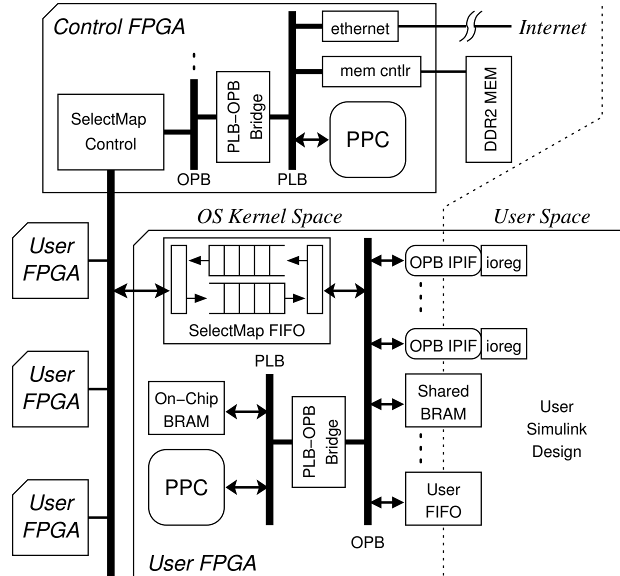


Figure 4.2: Block diagram of BORPH system on a BEE2 compute module with OPB-based SelectMap controller in control FPGA (OPBSM).

able to coexist coherently with other software processes in the system.

4.3 Base Architecture

BORPH's software kernel currently runs on the embedded PowerPC 405 core of the control FPGA on a BEE2. Besides standard system support hardware, such as an Ethernet device and memory controllers, special hardware must be designed to support BORPH's communication network between MK and UKs. Figure 4.2 shows the first proof-of-concept platform built on a BEE2 compute platform. Its simplicity serves as an illustration of basic BORPH function as well as a baseline for subsequent performance comparisons.

The part of the BEE2 compute module that makes up BORPH's infrastructure is

labeled “kernel space” in Figure 4.2. It includes the control FPGA and the part of the user FPGA (uK) that is responsible for communicating with the control FPGA. The part of the system that is responsible for executing user hardware designs is labeled “user space” in the diagram (HWR).

Implemented in the control FPGA is a standard on-chip processor system. Devices with high communication bandwidth requirements such as memory controller and Ethernet controller are connected to the processor via the high bandwidth peripheral logic bus (PLB). Devices with relatively lower communication bandwidth requirements, such as the RS232 UART, are connected to a simpler On-chip Peripheral Bus (OPB), which in turn is connected to the processor through a PLB-to-OPB bridge.

The part of the user FPGAs that belongs to the system infrastructure implements the functions of uK. It handles distributed hardware process management on behalf of the main kernel. For example, it controls the starting and stopping of hardware process on that user FPGA once it is configured. It responses to IOREG request packets sent from the BORPH kernel by reading or writing the corresponding values to a user design. It also keeps records of opened files by a hardware process and handles read/write requests by a user gateware design.

As mentioned in Chapter 5.2, the processor system on the user FPGA that makes up uK is inserted by our design flow automatically. As part of the system infrastructure, uK should never be unconfigured when a user hardware process terminates. However, it must currently be embedded within a user BOF file because the entire

FPGA is reconfigured for each hardware process creation.

4.3.1 The SelectMap Bus Controller

Physically connecting a user FPGA with a control FPGA is the SelectMap bus. It is a simple point-to-point, bi-directional, 8-bit bus connection running at 50 MHz. The bus is named after the fact that it reuses the same set of I/O pins to configure a user FPGA in SelectMAP mode.

A SelectMap bus master controller is implemented in the control FPGA. It serves the dual role of configuring a user FPGA and communicating with uK after a user FPGA is programmed. As shown in Figure 4.2, it is connected to the processor system OPB bus in this base architecture.

4.3.2 Design of uK

In this base architecture, uK is implemented using a simplified processor system based on the embedded PowerPC 405 core on a user FPGA. Embedded software running on the processor is responsible for all communications with mK. It is responsible for handling all packet marshaling. In response to ioreg virtual file system requests, it reads/writes to the corresponding hardware constructs in the user designs which are all connected through a multi-level OPB bus. Furthermore, it emulates all file I/O requests on behalf of the user design. The naïve design of this base architecture is simple to understand, but is also the culprit of suboptimal performance as shown in the following sections.

4.4 Performance of Base Architecture

In this section, performance of BORPH running on the base architecture (OPBSM) will be presented. In particular, the performances of process creation, gateware file I/O and the IOREG virtual file system will be benchmarked and analyzed.

4.4.1 Hardware Process Creation

A hardware process is created when an `exec` system call is received by the BORPH kernel on a BOF file. The request is then passed on to a kernel thread, `bkexecd`, for the actual configuration. Based on the BOF file header, one or more suitable FPGAs are chosen and configured accordingly using the SelectMap bus.

Creating a hardware process is similar to creating a normal UNIX process, in which a number of data structures, such as the kernel `task_struct`, must be updated and created for book keeping sake. In addition, the FPGA involved must be configured based on the FPGA configuration embedded in the BOF file.

Based on OPBSM, creating a hardware process takes about 900 ms, while creating a normal software process takes about 40 ms on the same processor. Since the theoretical minimum time to configure a user FPGA on BEE2 is 65ms, the minimum time to create a hardware process is $40 + 65 = 105$ ms. In other word, creating a hardware process in OPBSM takes about 9 times longer than the theoretical minimum time.

A close examination has shown that this process creation time is limited by data transfer speed from control FPGA to the user FPGA. There are three contributing bottlenecks.

1. The SelectMap controller on the control FPGA is controlled using simple programmed I/O. The processor is responsible for copying every byte of the 3MB configuration file to the SelectMap controller.

2. The SelectMap controller is connected to the processor through a PLB-to-OPB bridge, adding significant latency.

3. Despite running at a 50 MHz data clock, the SelectMap bus implementation takes 4 cycles to transfer one byte of configuration data to the user FPGA, limiting the maximum physical configuration speed.

As will be shown later, by eliminating bottleneck (1) and (2) in subsequent implementations, process creation time is reduced by more than 70%. Process creation time can be further improved if bottleneck (3) is also eliminated.

Note that modern operating systems like Linux employs demand paging techniques such as copy-on-write that significantly reduce software process start up time. Although demand paging of hardware configurations have been studied by other researchers in specialized partially reconfigured systems, such a technique is not being used by BORPH. BORPH focuses on the hardware process abstraction and its integration with the rest of the software system.

4.4.2 Reading/Writing ioreg Files

When a user reads or writes to a virtual IOREG file, the request is translated by the kernel into a message that is sent to the corresponding FPGA. The unique

identification number of the corresponding hardware construct is sent in the LOC field of the message. Each IOREG read (write) request is answered by the user FPGA by a read (write) acknowledge message, indicating the number of bytes read (written), or a negative value that indicates error condition. Adhering to the standard UNIX semantics for file read (write), the return value is passed directly back to the user process that initiated the request.

Figure 4.3 shows the performance of reading/writing an on-chip memory that is exported as an IOREG file using different read/write sizes, *s*. The transfer time remains low until *s* increases beyond about 64 bytes. Since there is no buffering in the file system level, the time needed for the operation is determined solely by data movement time, which includes memory copy time and hardware data transfer time. The effect of a small data cache (16k bytes), combined with a small 128 bytes SelectMap FIFO and the lack of DMA transfer are contributing factors for the slowing down. For large enough *s*, the speed levels at about 1.38 MB/s for both read and write[1].

4.4.3 General File I/O from Hardware Processes

Hardware processes initiate file I/O by sending messages to the BORPH kernel using the format described in Figure 3.11, with the LOC field denoting the Linux opened file descriptor number. All messages are received by `mkd`. Each opened file is managed by a *fringe*, which is implemented as a kernel thread on the control FPGA.

[1] Note: 1 MB = 2^{20} bytes; 1 kB = 2^{10} bytes

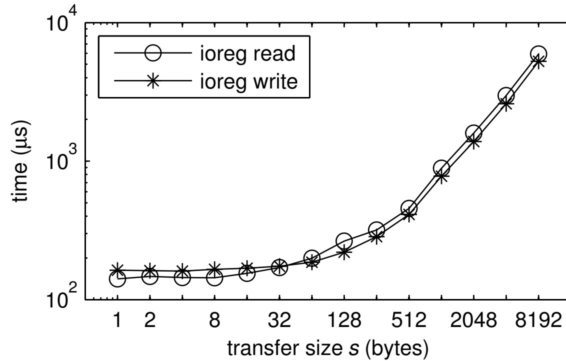


Figure 4.3: Performance of reading/writing on-chip memory on a user FPGA using ioreg interface.

A fringe is woken up by `mkd` as needed.

For the purpose of benchmarking hardware file I/O performance, a gateware design, `stdloop.bof`, and an equivalent software C program, `pipetok`, are created. Both designs repeatedly read s bytes of data from its `stdin`, and write the data back to `stdout`, until the end of file is reached.

Regular File I/O

First, to determine the performance of regular file I/O from a user FPGA hardware process, the two programs are run as follow:

```
bash$ stdloop.bof < datafile > outfile
bash$ pipetok < datafile > outfile
```

Figure 4.4 shows the file I/O performance of both processes with various file transfer sizes s. In the case of `stdloop.bof`, the time for each file I/O operation is measured directly on the user FPGA.

A hardware process file read is analogous to the C function call:

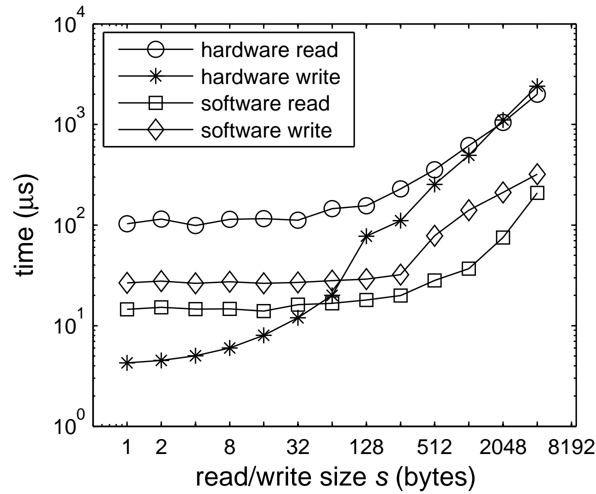


Figure 4.4: Hardware process file I/O performance.

```
read(fd, buffer, s);
```

For each read, a read request packet is sent from the user FPGA to the control FPGA. Recall that `mkd` is first woken up to handle this packet, which subsequently wakes up the corresponding fringe. The fringe then carries out the file read on behalf of the hardware process, blocking as needed. Once the requested data is ready, the fringe sends all data back to the user FPGA in one READ_ACK packet. Comparing a hardware file read to a software file read, a hardware file read incurs the overhead of interrupt handling and 2 extra context switches to the fringe. This overhead is reflected when value of s is small. Moreover, there is the overhead of data movement for large values of s. On the other hand, software reads require almost the same amount of time as fringe reads. The only difference between the two is the extra kernel boundary crossing time for software processes as they run in user mode.

Similarly, a hardware process file write is analogous to the C function call:

```
write(fd, buffer, s);
```

For each write, the data to be written is sent in the payload of a write request packet to the control FPGA. Upon receiving the packet, `mkd` is woken up. Since most file writes complete successfully without blocking, as an optimization, `mkd` writes to the file on behalf of the hardware process directly without involving a fringe, eliminating one context switch for each file write. Therefore, for small writes that can fit into the on-chip FIFO, the time to write to a file is the same as the time needed to write the packet into the FIFO, resulting in the very fast small hardware file writes. For write operations with larger payloads, however, the hardware process is delayed further by both the limited size of on-chip FIFO and the maximum packet size limit imposed by BORPH. Consequently, large hardware file writes approach the performance of hardware file reads.

In general, files are allowed to be larger than the size of a single read message. Therefore, multiple READ messages are required to process the entire content of a file, incurring multiple packet communication overheads. Figure 4.5 plots the time required for a hardware process to read files of various sizes using different values s for each read request. The values are measured from the control FPGA using the standard `time` command:

```
bash$ time sink.bof -s $SIZE < $INPUT_FILE
```

where `sink.bof` is a hardware design that reads the entire content of a file from `stdin` and then exit. This graph gives an overall system performance benchmark as

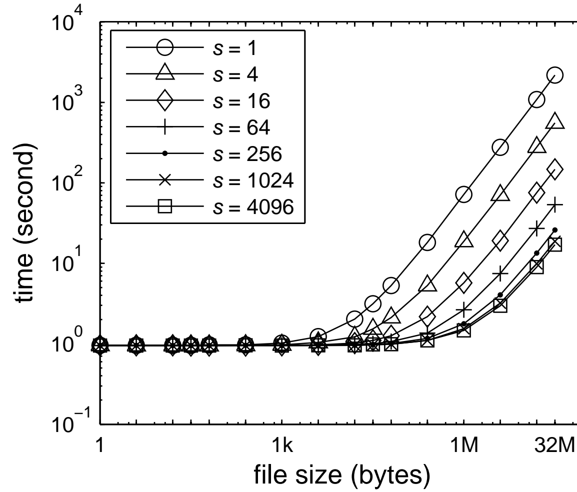


Figure 4.5: The effect of data transfer size s on total time required by a hardware process to sink a regular file. Time is measured from control FPGA.

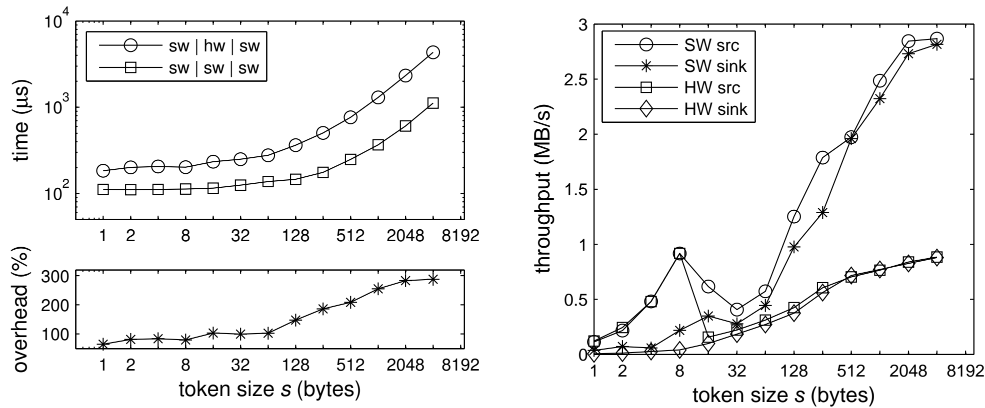
it includes system time such as the time for process creation and destruction.

From Figure 4.5, it can be seen that large transfer size s is essential for large files transfer to amortize the high overhead involved for each hardware file system call. For $s = 4096$, streaming a 32 megabytes file to a hardware process completes in 16.68s, resulting in a transfer speed of 1.91 MB/s. On the other hand, for $s = 1$, the same transfer takes 1871.73s.

Piped Process through Standard I/O

Having access to system standard I/O allows hardware and software processes to communicate with standard UNIX pipes. To evaluate the performance of such pipe, a second benchmark is performed. Instead of reading regular files, the two programs from previous section are run in a pipelined fashion as follow:

```
bash$ sendtok | stdloop.bof | recvtok
bash$ sendtok | pipetok | recvtok
```



(a) Latency between when `sendtok` sends a token and when `recvtok` receives the entire token

(b) Throughput of both hw-sw and sw-sw pipes

Figure 4.6: Comparing software piped process chain with a mixed hardware/software chain.

where `sendtok` and `recvtok` are software programs that repeatedly send and receive token of s bytes from their `stdout` and `stdin` respectively. The time for `recvtok` to receive an entire s bytes token from `sendtok` in both cases is shown in the top half of Figure 4.6(a). Communication overhead, which is the extra time needed for the mixed HW/SW pipe over the software pipe, is plotted at the bottom half of the diagram. For $s < 128$, the mixed HW/SW pipe's latency is about 60% more than the pure software pipe. The gap increases as s increases to almost 300% for $s \geq 128$ as the data transfer latency starts to dominate.

In the above piped organization, throughput of the pipe is also an important metric. Figure 4.6(b) shows the throughput of HW/SW pipe and SW/SW pipe for different token sizes s. Throughput is measured by `sendtok` at the source, and `recvtok` at the sink. The difference in sending and receiving rates are results of

buffering within the system. At the receiving end, throughput increases as s increases. For $s = 4096$, the mixed HW/SW pipe has a throughput of 858.56 kB/s, while the SW/SW pipe has a throughput of 2.8 MB/s

4.5 Advanced On-Chip Architecture

While the base architecture OPBSM is simple as a proof-of-concept, its performance is suboptimal. Two advanced versions of on-chip architecture on BEE2 were subsequently implemented. Apart from improving the much needed system performance, the two iterations of architectural changes also demonstrate the benefit of hardware kernel/user separation. Throughout the process of implementing these two advance architectures, backward compatibilities of user BOF files have been maintained. As a result of the consistent BORPH interface, BOF files created for OPBSM can be executed without recompilation in the two new architectures.

4.5.1 Elimination of PLB-to-OPB Bridge

The first architectural improvement is developed based on the observation that the PLB-to-OPB bridge in the control FPGA increases latency between the PPC and the SelectMap controller. A new version of the control FPGA system, PLBSM, was created with the SelectMap controller connected directly to the high bandwidth, low latency PLB bus. The resulting control FPGA architecture is shown in Figure 4.7.

Figure 4.8 shows the result of repeating the experiments from the previous section

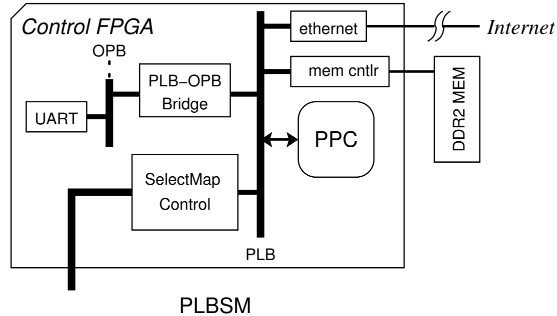
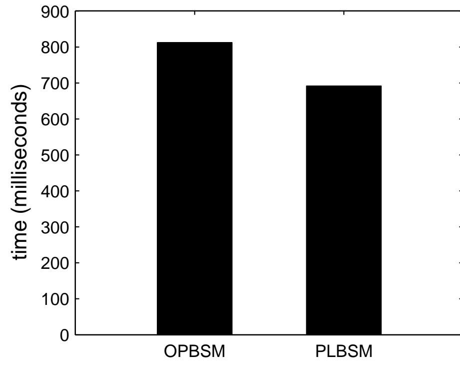


Figure 4.7: PLB-based SelectMap controller (PLBSM).

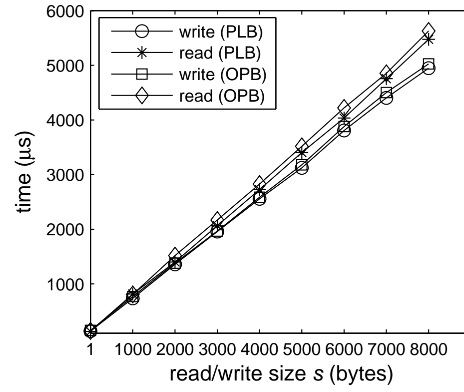
in PLBSM. Each sub-figure plots the result for both OPBSM and PLBSM. To highlight the performance difference between the two hardware architectures, only subsets of all data points are shown in the diagrams.

Figure 4.8(a) shows that process creation benefits most from the new architecture, with a 28.9% decrease in process creation time. To create a hardware process, no complex interrupt handling or message exchange is needed between control FPGA and user FPGA. Hardware process creation is dominated purely by the transfer of configuration data to the user FPGA. Therefore, all the performance gain in bypassing a PLB-to-OPB bridge is reflected in the process creation benchmark.

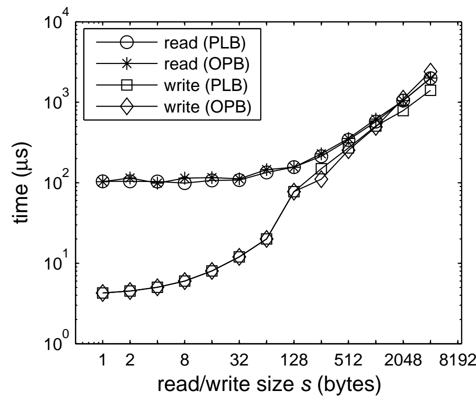
On the other hand, Figure 4.8(c) shows that there is only marginal improvement in hardware file I/O as measured from the user FPGA. This is as expected because the performance of hardware file I/O is largely limited by the speed of the processor system on the user FPGA, not the control FPGA. On a user FPGA, the processor communicates with the SelectMap bus FIFO through an OPB-connected controller. Here, the OPB-to-PLB bridge is again the bottleneck, undermining any improvement in control FPGA speed. Figure 4.8(d) shows that for file reads with large enough



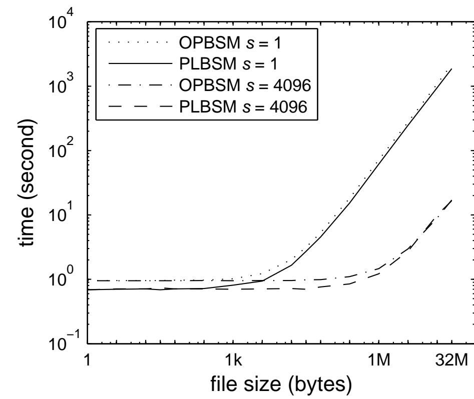
(a) HW process creation



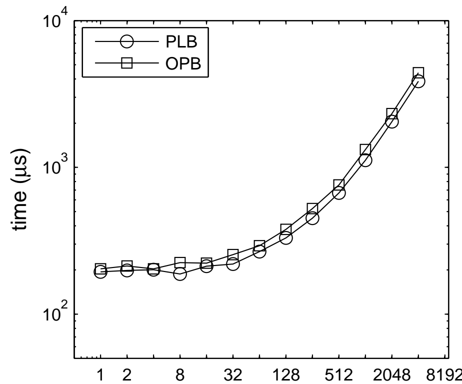
(b) ioreg read/write



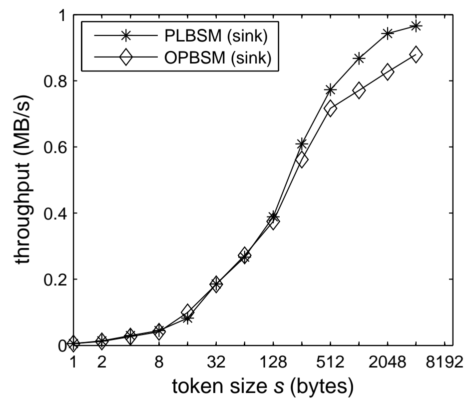
(c) HW file I/O



(d) File streaming from SW to HW



(e) HW/SW pipe (latency)



(f) HW/SW pipe (throughput)

Figure 4.8: Performance comparison between OPBSM and PLBSM.

size, PLBSM and OPBSM have similar performance. For small files transfers, the performance advantages of PLBSM are direct results of faster process creation times.

Performance of the IOREG interface also sees limited performance improvement: an average of 4% increase for large reads and 2% increase for large writes. The performance of the ioreg interface is largely limited by pure hardware data transfer speed. As in the case of hardware file I/O, since the user FPGA is limiting the data transfer speed, very little improvement is expected.

The chained hardware/software pipe does benefit from the architectural change. An average of 12% performance increase in latency (Figure 4.8(e)) is observed. With a transfer size $s = 2048$, the overall throughput increases 14% from 807 kB/s to 920 kB/s (Figure 4.8(f)). The pipelined process operation involves complex HW/SW interaction such as context switching on the processor, interrupt and message passing, etc. Therefore, a faster data transfer speed on the control FPGA allows the processor to perform these tasks more efficiently, resulting in a higher overall system performance.

4.5.2 DMA Enabled Control FPGA

Based on the performance benchmarks of PLBSM, a third on-chip architecture, PLBDMA, is implemented to further enhance performance. A performance comparison of OPBSM and PLBSM highlights two performance bottlenecks in the BORPH system. First, while eliminating the PLB-to-OPB bridge improves raw data transfer performance as expected, the increased performance in the control FPGA must be

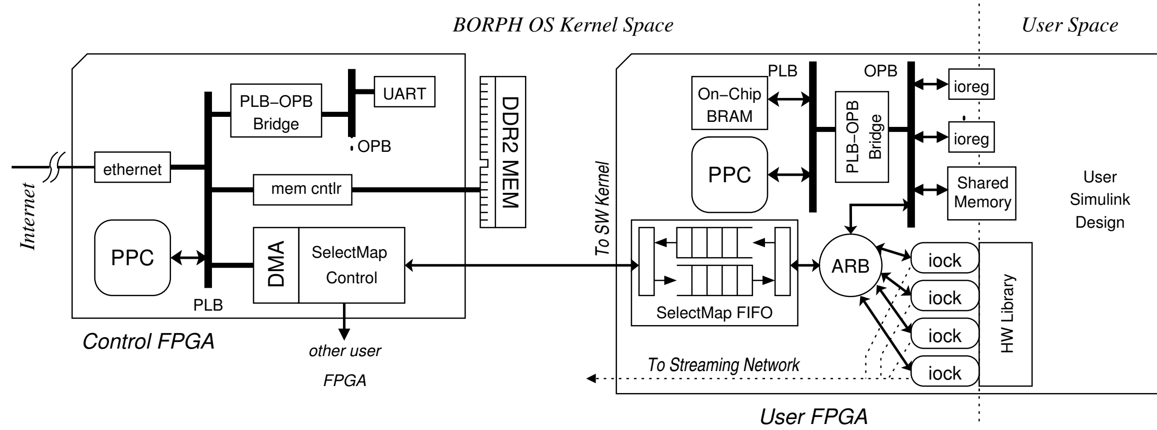


Figure 4.9: Direct memory access (DMA) is enabled in the control FPGA for communication between PPC and the PLB-based SelectMap controller. On a user FPGA, a hardware arbiter is implemented to allow direct access to SelectMap bus from a user gateware design for file I/Os.

matched by the user FPGA to have major impact in overall system performance. Secondly, in heavily loaded system, the processor currently spends a large portion of the processing time in data transfer, hindering the over system performance.

Based on the above two observations, the third on-chip architecture, PLBDMA, was developed as shown in Figure 4.9. Architectural changes are introduced to both the control and user FPGAs to address overall system bottlenecks.

On the control FPGA, direct memory access (DMA) is enabled for communications between the control processor and the SelectMap controller. This arrangement increases the raw data transfer speed and at the same time relieves the processor from intensive programmed I/O data transfer. Furthermore, instead of relying on the PPC on the user FPGA to emulate file I/O operations on behalf of the user design, a new UK architecture is implemented with fully functional `hsc` interface for gateware file I/O. An arbiter is implemented to allow direct access to the SelectMap bus inter-

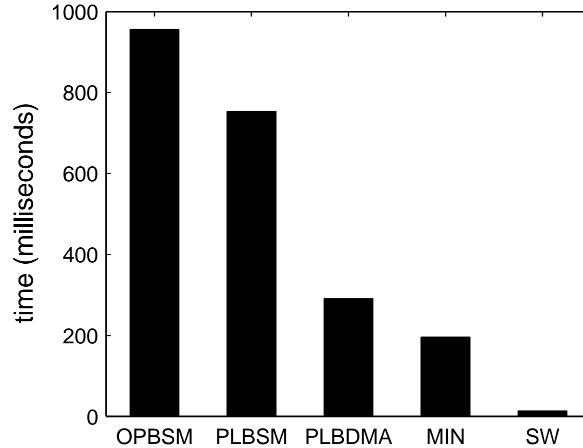


Figure 4.10: Hardware process creation time in 3 different on-chip architectures compared with the theoretical minimum time and software process creation time.

face from user gateware designs. All remaining system administrative functions and handling of IOREG messages remains duties of the PPC.

Process Creation

Process creation time is significantly reduced as a result of the performance improvement from DMA-enabled SelectMap controller. Process creation time is reduced by 70% when compared with the original OPBSM and is decreased by 61% when compared with PLBSM. As a result of DMA transfer, process creation time is currently mostly limited by the transfer speed of the SelectMap bus which runs only at 25% the speed of the theoretical maximum configuration clock of 50 MHz. Figure 4.10 shows the relative speed of hardware process creation in all three architectures compared with the theoretical minimum as well as normal software process creation time.

File I/O Performance

In PLBDMA, file I/O operations from a gateware design is performed directly through BORPH's hsc interface. BORPH's ioc controller, iock, has direct access to the SelectMap interface through a newly implemented arbiter, eliminating the need for PPC core operations. The result is improved hardware I/O performance.

To determine the performance advantages of this new architecture, a gateware design stdloop_hsc is implemented. It performs the same operations as stdloop.bof used in previous section except stdloop_hsc performs file I/O operations using BORPH's native hsc interface. The three programs, pipetok, stdloop.bof and stdloop_hsc.bof, read 1024 bytes from its standard input and write the result to standard output. The time needed to complete streaming files of various sizes is timed with standard UNIX time command as follow:

```
bash$ time stdloop_hsc.bof < datafile > outfile
bash$ time stdloop.bof < datafile > outfile
bash$ time pipetok < datafile > outfile
```

The normalized execution time of stdloop.bof and stdloop_hsc.bof with respect to pipetok are shown in Figure 4.11.

The time for hardware processes to process small files is completely dominated by the process creation time. As file sizes increase, more time is needed to transfer data from the medium where the file is physically located, as well as for in memory buffer copying. The gap in processing time between gateware and software processes is therefore reduced. With files of significant sizes, the communication overhead of hardware processes becomes the bottleneck. For stdloop.bof, the processor overhead

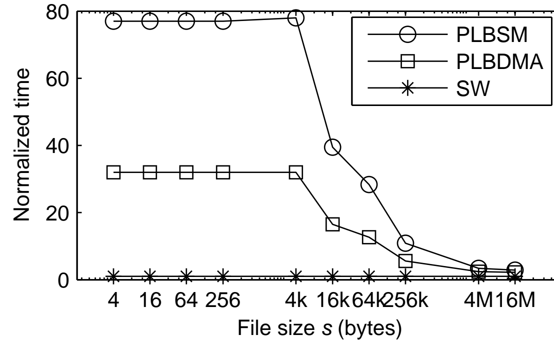


Figure 4.11: Normalized time for hardware processes to finish streaming files of various sizes. `stdloop.bof` uses PPC on user FPGA for file operations, while direct file I/O through `hsc` is used in `stdloop_hsc.bof`.

on the user FPGA is limiting its performance to 2.9 times slower than software. For `stdloop_hsc.bof`, no PPC operation is involved. Only a few cycles are needed to stream data through a gateware design on user FPGA. As a result, its performance is limited solely by the communication bandwidth on the SelectMap bus, as well as the processing overhead on the control FPGA. It is currently 2.2 times slower than software.

Note that no DMA transfer is used on the control FPGA to handle hardware file I/O. DMA is currently only used for user FPGA configurations. Performance of hardware file I/O, as well as the IOREG virtual file system layer is expected to increase significantly once DMA is implemented on control FPGA for all message passing operations.

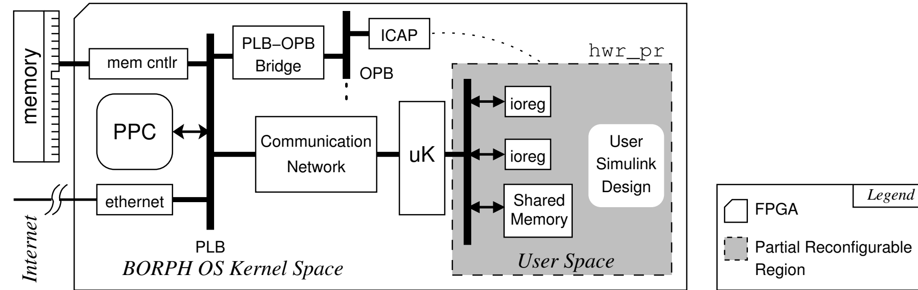


Figure 4.12: An example of porting BORPH to a single FPGA where user hardware processes are executed on a partially reconfigurable region.

4.6 Porting BORPH

Although currently implemented on a BEE2 compute module, BORPH is a general operating system design that can be applied to other FPGA-based systems. This section provides a high-level description on porting BORPH.

To make the discussion concrete, we will illustrates the process using an example of porting BORPH to an FPGA that supports dynamic partial reconfiguration. The top-level block diagram of the resulting system is shown in Figure 4.12.

Porting BORPH to the platform in Figure 4.12 requires changes to both hardware and software from the original BEE2 implementation. Logically, all hardware and software changes involved can be classified as part of the process of implementing a new reconfigurable hardware region (HWR). Denote this new HWR as `hwr_pr`. As shown in Figure 4.12, we assume only one instance of `hwr_pr` will be presented.

The first hardware change required is to implement logic that configures and unconfigures the newly defined partial reconfiguration region. Modern FPGAs provide native support for this purpose, such as through the Internal Configuration Access

Port (ICAP) in Xilinx FPGAs. For other platforms, different configuration logic will need to be implemented.

The second hardware change required is to reimplement the message passing network between MK and UK. The original BORPH implementation on BEE2 utilized the SelectMap bus for communication between control and user FPGA. Porting BORPH to a new platform requires a new communication mechanism between MK and UK. In the case of `hwr_pr`, since MK and UK are physically residing within the same FPGA, this communication network may be reduced to a simple memory mapped device on the processor system.

Most of the BORPH software kernel can be ported to any FPGA-based reconfigurable system without modification because they are written as high-level machine independent code. Two parts of the BORPH kernel are machine dependent. First, some part of signal handling code is processor dependent. Therefore, porting BORPH to a FPGA-based reconfigurable system with a different processor will require small modifications to its signal handling code. Secondly, the BORPH software kernel must be updated to handle any newly defined HWR type by registering a new corresponding HWR module.

As described in Section 4.2, BORPH's kernel design has a dedicated subsystem devoted to supporting different kinds of HWR types. This HWR subsystem works in ways similar to the standard Linux device driver subsystem. A set of virtual function calls must be implemented by each HWR type kernel module to handle hardware specific operations. For example, in the case of `hwr_pr`, the `configure` function must

be implemented to configure the correct reconfigurable region on the FPGA using the built in ICAP. It must also make use of the newly implemented hardware for communication between MK and UK. Work is currently underway to standardize this interface for MK-UK communication.

Supporting any additional system calls for hardware process can be accomplished by defining a new message that corresponds to each supported system call. For example, to add support of `gettimeofday` function to FPGA designs, a new message with unique CMD field corresponding to `gettimeofday` can be defined. Upon receiving such message, the main message handling thread, `mkd`, may then serve as a proxy that returns the current time of the day to the issuing hardware process.

4.7 Summary

This chapter has described BORPH's software kernel, as well as three different on-chip architectures of MK and UK as implemented on a BEE2 module. Performances of various BORPH features as implemented in the three different on-chip architectures are presented.

BORPH's software kernel is an extended version of Linux 2.4.30 kernel designed to manage gateware designs natively. To make BORPH portable, machine dependent information about reconfigurable fabrics are handled by a HWR abstraction layer, implemented as loadable kernel module with a pre-defined kernel interface. A new binary format, `binfmt_bof` is registered to handle BORPH's native BOF files. A kernel thread, `mkd`, is created to handle all message communications between MK

and uK. To handle hardware file I/O operations, each opened file is managed by a kernel thread called a fringe. A fringe acts as a proxy for FPGA designs to gain access to the general file system. BORPH's IOREG virtual file system layer is implemented by extending Linux's native `proc` filesystem. All IOREG related virtual files are created as subtree under a hardware processes `/proc/<pid>/hw` directory.

Three on-chip architectures, OPBSM, PLBSM, and PLBDMA with successive higher performance are presented. All three of them include a processor system based on an on-chip PowerPC 405 core in the control FPGA of a BEE2 compute module. The BORPH kernel running on this processor system communicates with uKs configured in user FPGAs through a message passing network built on top of the SelectMap bus. The same bus serves a dual role of user FPGA configuration bus. The three architectures differ in the way they communicate and configure user FPGAs.

Overall hardware/software system performance is limited by communication performance between control and user FPGAs. In OPBSM and PLBSM, both message passing and user FPGA configuration are performed using programmed I/O, which has shown to be a significant performance bottleneck. Direct memory access (DMA) is used in PLBDMA for user FPGA configuration which results in a 70% decrease in hardware process startup time. Eliminating the need of PowerPC on user FPGA is also shown to improve hardware file I/O performance by 24.1%.

CHAPTER 5

BORPH Application Developments

From the perspective of an application designer, the available design methodology and the ease of run time system interaction have significant impact on the usability of a reconfigurable computer system. The former determines how easy a conceptual design may be realized as an RC application at compile time, while the latter determines how easy an application can be executed, debugged and perform useful computation during run time.

This chapter will first describe briefly the conventional HDL-based FPGA development methodology. Then, our in-house Simulink-based high-level design flow and its integration with BORPH will be described. Finally, three FPGA applications are described as examples to illustrate how various features of BORPH work together during run time to improve overall user experiences with RC application development.

5.1 Conventional FPGA Design Flow

Because of its close ties to application specific integrated circuit (ASIC) development, FPGA application development usually follows industry standard hardware

design flows. The use of hardware description languages (HDLs) such as VHDL and Verilog are common among FPGA application designers. Once designs are described in HDL, they are simulated using a software simulator. The simulated designs are subsequently passed to vendor-specific tools that translate these HDL designs into configuration files specific to a particular FPGA through multiple stages of synthesis, translation, mapping, placing and routing. The resulting configuration files are then used to program FPGA devices using one of the many device specific configuration mechanisms offered by the vendor. Despite being described as cumbersome, HDL-based design flow remains invaluable for designers for controlling low-level resource utilization and execution on a FPGA.

However, as the sizes of modern FPGAs and the complexities of their applications increase, the needs for high-level design methodologies that shields users from low-level hardware details become apparent. Similar to the way high-level programming languages were developed to enable complex software programming, a number of high-level description languages have been proposed for FPGAs and other reconfigurable fabrics. Examples include Handel-C[34], Stream-C[20], StreamIT[1], JHDL[5] etc. As their names have suggested, many of them are developed with syntax similar to familiar software programming languages such as C, C++ and Java in the hope to ease the transition from software to gateware developments.

At the same time, because of the hardware/software nature of FPGA applications, some advocate language environments that are capable of expressing the entire hardware/software system using one unified environment. Examples include POLIS[3],

SystemC[35] and SystemVerilog[19].

5.1.1 Run Time Support

While design language environments allow designers to express and debug their conceptual designs during compile time, run time supports are essential for applications to perform useful computation with actual data. There are two common ways to provide run time support to applications. Each method integrates into its design language environment differently.

The first method, commonly found in embedded systems, is to generate the entire run time system *statically* based on actual user application requirements during compile time. Parameters of the system, such as the number of task running at any one time or I/O device memory maps are generated on a per-application basis. Because of the static nature of this methodology, RC vendors may choose to incorporate custom run time operating system kernel, or choose not to include a full OS kernel at all in order to optimize for system performance, memory consumption, etc. Furthermore, because of such flexibility on the generated kernel, the application interfaces into the run time kernel are often language dependent and vary greatly across different RC systems.

The second method, commonly found in large scale high performance reconfigurable computers, is to manage an RC with fully functional, online operating systems such as Linux or VxWorks. In this case, run time support is provided by the kernel. Applications must communicate with the kernel using the kernel defined interface.

Because of this kernel/user separation, the interface is design language independent. It is therefore, possible to develop applications in any design language suitable for the applications. Furthermore, the same interface may be applied to different reconfigurable computers across different vendors.

BORPH is one example of such operating system that extends standard Linux kernel interface to gateware applications. FPGA application developments in a BORPH managed system are therefore design language independent. Similar to the case of developing software programs, system libraries must be developed for each supported language. System libraries serve as bridges between the run time kernel and user applications. They isolate users from most of the complexities involved with run time kernel communications. They may come in the form of an actual included library in a text based language, such as VHDL, or in the form of a library block as in the case of our graphical design flow described next.

5.2 Simulink-based Design Flow

This section describes the Simulink-based hardware design flow currently employed at the Berkeley Wireless Research Center (BWRC) for FPGA application development. It serves as an illustration on how a FPGA design flow may integrate with the BORPH run time kernel. Figure 5.1 shows the major stages of our integrated hardware design flow.

Using this design flow, users describe their designs in Simulink using blocks provided by Xilinx System Generator[49]. The vendor provided blockset includes blocks

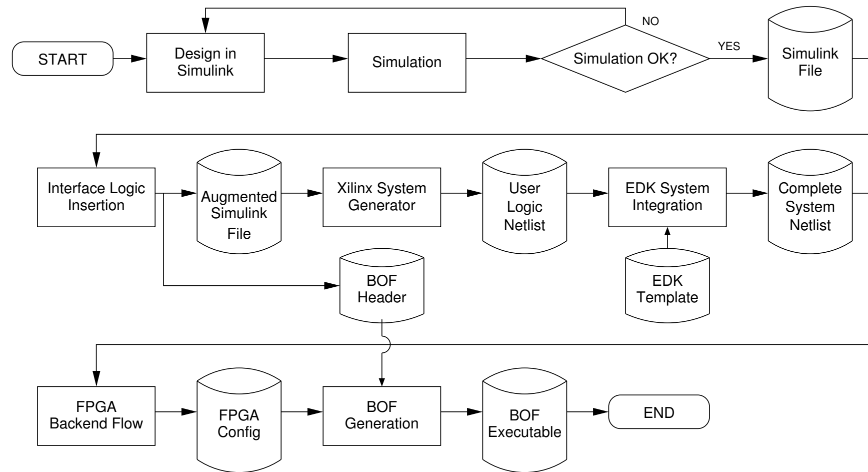


Figure 5.1: An automatic hardware design flow that compiles high-level Simulink designs into executable BOF files.

ranging from low-level single bit flip-flops, to complex hardware constructs such as adder, multiplier and finite impulse response (FIR) filter. To interact with the rest of the BORPH systems, users make use of data I/O blocks from our in-house library. It includes, for instance, custom library blocks for hardware constructs that are exported and accessible through the IOREG virtual file system such as register, shared memory, and FIFO. Once the design is created in Simulink, the user may optionally simulate the design within the Simulink environment before proceeding to hardware generation.

The hardware generation process is where the BORPH specific steps are involved. First, the user Simulink design is parsed to identify all instances of BORPH specific library blocks. These blocks serve as the boundary between the actual generated hardware and native Simulink blocks that are only for simulation purposes. Xilinx System Generator is then called to generate the necessary netlist from the user design,

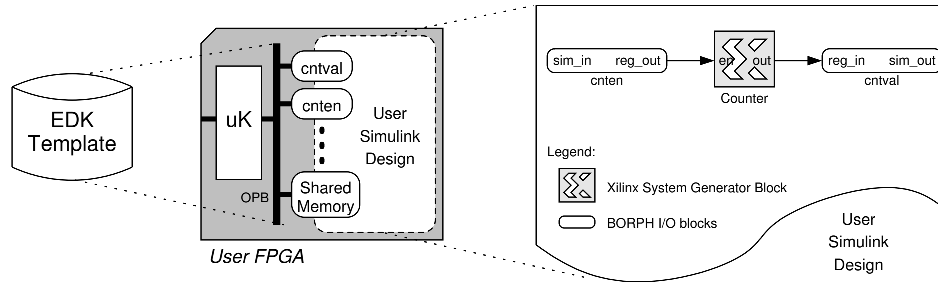


Figure 5.2: Block diagram of a user FPGA. Compiled user Simulink designs are combined with a predefined EDK template of uK to generate user FPGA configurations.

instantiating native library blocks accordingly. Clock and reset insertion, as well as data sample rate resolution, are handled by System Generator. The resulting low level netlist is then prepared as a block for use with Xilinx Embedded Development Kit (EDK) in our next step.

Next, a processor system (uK) is inserted as shown in Figure 5.2. All ioreg related blocks from a user design are connected to a multi-level On-chip Peripheral Bus (OPB) that is accessible from this processor system. All runtime communications with the central BORPH software kernel (mK) are handled by this processor system. A detail block diagram of uK is shown in Figure 4.2. The combined system is subsequently passed to vendor provided backend tools for synthesis, map, place and route.

Finally, from the top level Simulink design, a symbol file is generated that lists information such as address and size of all BORPH specific blocks. This symbol file is combined with the FPGA configuration file generated by the vendor tools to create the final BOF executable file.

5.3 Sample Applications

In this section, three FPGA applications are briefly described to illustrate how various features of BORPH work together to ease their development processes. The first two are developed using the Simulink-based design flow described earlier, while the last one is developed using VHDL only.

5.3.1 Example 1: A Real-Time Wireless Signal Processing System

The first application is a real-time wireless signal processing system for our cognitive radio project[31]. This system makes extensive use of the IOREG virtual file system for hardware parametrization and HW/SW communication. Furthermore, full backward compatibilities with existing Linux system allows software team to perform HW/SW system testing remotely over Internet.

Overview

Cognitive radios are smart radios that take advantage of under-utilized licensed spectrum for opportunistic tranceiving. In order to prevent interference to licensed primary users of the spectrum, a variety of techniques have been proposed for reliable sensing and non-interfering use of the spectrum. Our system is designed to validate those techniques. Figure 5.3 depicts our overall system design that involves multiple cooperative cognitive radios.

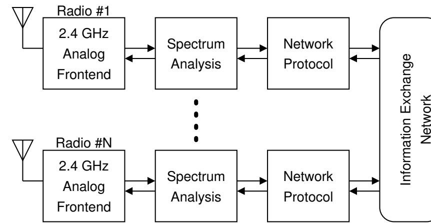


Figure 5.3: Cognitive radio testbed system

Each radio is logically separated into two parts that are developed by two physically separated design teams. The partitioning is done loosely based on the standard ISO network stack, where the physical layer is implemented in hardware and the higher layers are implemented in software.

At the physical layer, real-time spectrum analysis is performed by FPGA hardware that is connected to external RF frontend. The FPGA hardware is designed using our Simulink-based design flow described in Section 5.2. All high-level network protocols are independently developed in software.

Using BORPH for Communication and Synchronization

All communications between the spectrum sensing hardware and the software protocol stacks are done via BORPH's IOREG interface. Two 8192 bytes shared memories are exported as IOREG virtual files for data communication with software. In addition, more than 20 single word registers are defined. Most of them serve the function of controlling hardware parameters such as RF channel, amplifier gain, etc. Some IOREG registers, however, are used solely for synchronization purpose. For example, each shared memory is guarded by a pair of `enable` and `ready` registers.

The `enable` register is used by software to notify hardware its intention to read memory. When the data in the shared memory is ready, the hardware asserts the corresponding `ready` register. This two-way handshaking mechanism forms the basis of simple synchronization between software and hardware processes.

The file I/O capability of a hardware process is used to implement a low level debugging shell. It provides an additional way to debug the running FPGA and to display hardware status.

Remote System Testing

Our software design is developed off-site. BORPH provides a remote testing environment for our protocol group who doesn't have physical access to the hardware. With BORPH, our software team independently develops the protocol stack without the presence of the hardware by emulating it with software processes. As development progress, they then remote log onto the physical hardware for mixed HW/SW testing with a simple swap of hardware process in place of the emulating software process. Since BORPH runs with a fully functional Debian root file system, all necessary software development tools, such as `gdb` are available for debugging.

5.3.2 Example 2: Low-Density Parity-Check Decoders Emulation

In this project, FPGAs are used to study quantization effects on the performance of low-density parity-check (LDPC) codes[52, 53]. Because of the intense compu-

tational need for empirical study of LDPC code for even moderate bit error rates (BER), hardware emulations using FPGAs are employed.

Similar to the previous wireless signal processing example, this application relies heavily on BORPH's IOREG virtual file interface to dynamically customize design parameters to explore implementation choices. Furthermore, IOREG virtual files are used to export collected data for post-processing.

This application is unique that it does not require externally attached hardware devices for execution. As a result, during the design exploration phase, the same relocatable BOF file can be executed concurrently on multiple FPGAs. The fact that each BORPH system is networked allows each instance be conveniently started and parametrized through IOREG virtual files remotely. Having more than 10 instances of the same design emulating concurrently have significantly improved the productivity of the designers.

5.3.3 Example 3: FPGA Video Processing with Commodity Software

The standard conforming file I/O capabilities for hardware processes allow FPGA designs to communicate with commodity software via standard pipes. To illustrate this, we have implemented a simple Sobel edge detection program in FPGA (`yuvedgdet.bof`) that works with the MJPEG Tools[32]. The MJPEG Tools is a set of Linux programs that collectively perform complex video editing functions. Most of the programs in the tool set communicate with each other through piped standard

input and output, using a predefined raw video format (YUV4MPEG2).

Using BORPH's file I/O capabilities, the FPGA edge detection filter may therefore be inserted easily with a single shell command:

```
bash$ lav2yuv test.avi | yuvedgdet.bof | mpeg2enc -o output.mpg
```

where `lav2yuv` and `mpeg2enc` are programs distributed with the MJPEG Tools. In the above command, `lav2yuv` translates the source video `test.avi` into the YUV4MPEG2 raw video stream, which is then redirected to our FPGA edge detection filter. The filtered video is piped back to the software MPEG encoder `mpeg2enc` for the final encoding.

Comparing to a pure software implementation, performance of the above hardware-in-the-loop video processing is 18% slower as a result of I/O overhead. As with any other hardware acceleration scheme, the advantage of FPGA implementation is expected to be more prominent as more computation, such as MPEG encoding, is shifted to FPGA, amortizing performance degradations due to I/O.

Most importantly, this example demonstrates BORPH's unique approach to hardware/software co-execution. The use of standard file semantics for hardware/software communication is not only easy to understand for novel users, but also allows easy integration with existing commodity software, greatly improving productivity of designers.

5.4 Summary

This chapter has described briefly the industry standard hardware description language based design methodology for reconfigurable fabrics such as FPGA. Although sometimes regarded as cumbersome by some researchers, HDL design methodologies remain invaluable for low-level resource controlling.

Furthermore, we have also described the Simulink-based design flow developed at the BWRC and how it integrates with the run time system of BORPH. This design flow is suitable for high level application developments and for users with few prior experiences with FPGA application development.

Finally, development experiences from three different FPGA applications have been described to illustrate how various features of BORPH works together to ease their development efforts.

Chapter 6

Conclusions and Future Directions

This thesis has demonstrated that not only is it feasible, but it is essential to develop operating systems that on one hand provide systematic OS services to reconfigurable gateware designs, while on the other hand maintain backward compatibilities with conventional processor-based systems so as to ensure future success of reconfigurable computing research, allowing it to gain mainstream acceptance across multiple research domains.

BORPH demonstrated the concept of operating system support for reconfigurable computers on the BEE2 compute module. BORPH encapsulates FPGA hardware designs as running hardware processes and provides them with conventional OS services such as file system support. By integrating reconfigurable hardware processes into a conventional Linux system, BORPH provides a unified HW/SW runtime environment with a familiar UNIX interface. The concept of hardware process allows gateware designs to take an active role in the system. Instead of the traditional master-slave relationship with its controlling software, a hardware process may form a peer-to-peer relationship with any running software/gateware program. Furthermore, hardware processes interact with the rest of the system actively through standard UNIX file

I/O and passively through BORPH specific IOREG virtual file system.

There are multiple ways one can look at BORPH beyond an operating system: It is an abstraction model; it is an interface; it is an ideology.

BORPH presents a novel way to abstract a running FPGA design in a system using the conventional UNIX process model. Such an abstraction transforms a static FPGA design into an active running entity. It provides context for one to explain, appreciate and to reason about reconfigurable resources of a system. It also provides a framework in which many traditional FPGA research may be carried out. Topics such as task scheduling, I/O resource allocations, task swapping, take on new meanings under BORPH's UNIX process model.

BORPH extends the traditional software OS kernel/user interface concept to a spatial one that supports gateware designs executing on reconfigurable computers. Such kernel/user separation improves manageability of a reconfigurable computer while relieving gateware designers from cumbersome system management tasks, allowing them to focus on developing their actual applications. Furthermore, by confining a gateware design within the UNIX process boundary, BORPH introduces a coarse grain hardware/software runtime interface at this design language independent kernel boundary. Similar to the design of any other interface, the precise definition of an ideal gateware kernel/user interface is a moving target that requires constant refinements. Nonetheless, the current BORPH implementation has taken a first step towards that direction.

Finally, the development of BORPH is an ideology that through systematic op-

erating system support, the usability of reconfigurable computers may be improved. The lowered barrier-to-entry into the field of reconfigurable computing may therefore enable researchers from across different research domains to take advantage of reconfigurable computing technologies.

6.1 Future Directions

The work of BORPH has opened the door into operating system research for reconfigurable computing, allowing many traditional concepts in both computer sciences and FPGA designs to take on a new dimension.

The semantics for hardware processes remains to be explored. Concepts such as blocking, swapping, parallel file system access, and HW/SW signaling semantics need to be refined. Furthermore, the kernel/user interface of BORPH is a natural boundary to employ dynamic partial reconfiguration on a FPGA. The OS kernel should continue to execute while a user design is configured dynamically as a result of process starting, stopping, suspending, continuing, and swapping. Interactions of the runtime system with the compile time partial reconfiguration design flow is essential.

The concept of hardware process also enables future development in gateware design runtime debugging methodologies. Because of the close ties between the runtime OS kernel and a debugger, the development of one is likely to affect the development of the other.

Scaling of BORPH is going to be an exciting research topic. BORPH is currently implemented on the BEE2 module with 4 user FPGAs. Since no swapping

is performed, a maximum of 4 hardware processes may be executing at the same time. On one hand, BORPH may be scaled up to systems with plentiful of reconfigurable resources, stressing the limit of a centralized main OS kernel. On the other hand, high-end embedded systems may contain limited reconfigurable resources with tight real-time requirements, stressing the spatial and temporal resource allocation capabilities of the BORPH kernel.

Finally, as the BORPH abstraction model has demonstrated to be extremely useful, future reconfigurable computer architectures may be designed specifically to optimize performance for such model. For instance, communication between a user gateware design and the BORPH kernel has shown to be an overall system performance bottleneck in our current implementation. Future reconfigurable systems may therefore consider providing high performance native hardware support for gateware communications.

6.2 Closing Remarks

Gauging the usefulness of BORPH is a highly subjective matter. Taking away precious FPGA resources while imposing various limitations on FPGA designs in the name of operating system management is a difficult proposition for performance conscious hardware designers to appreciate. Nonetheless, positive experiences from current users have suggested that trading off resources for system manageability is indeed useful for high-level application designers.

As an illustration, FPGA research work at the Berkeley Wireless Research Center

has been on going since 2000. Multiple BEE and BEE2 systems have been setup for research uses within BWRC. However, only until BORPH is fully functional, and with the help of a mature Simulink design flow, when for the first time all FPGAs in the center are being utilized at the same time performing useful calculation.

It is the hope of the author that the work of BORPH will continue to open the door into reconfigurable computing for researchers from across different application domains, enabling them to perform research work that will benefit our world.

Bibliography

[1] S. Amarasinghe, M. l. Gordon, , M. Karczmarek, J. Lin, D. Maze, , R. M. Rabbah, and W. Thies, "Language and compiler design for streaming applications," *International Journal of Parallel Programming*, vol. 22, pp. 261–278, June 2005.

[2] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews, "Enabling a uniform programming model across the software/hardware boundary," in *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, April 2006, pp. 89–98.

[3] F. Balarin, P. D. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli, and K. Suzuki, *Hardware-software co-design of embedded systems: the POLIS approach*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[4] J. Becker and R. Hartenstein, "Configware and morphware going mainstream," *Journal of Systems Architecture*, vol. 49, pp. 127–142, September 2003.

[5] P. Bellows and B. Hutchings, "JHDL – an HDL for reconfigurable systems," in *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, 1998.*, 1998, pp. 175–184.

[6] Celoxica. [Online]. Available: http://www.celoxica.com

[7] C. Chang, J. Wawrzynek, and R. W. Brodersen, "BEE2: A high-end reconfigurable computing system," *IEEE Design & Test*, vol. 22, no. 2, pp. 114–125, 2005.

[8] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, 2002.

[9] Cray, "XD1 supercomputer." [Online]. Available: http://www.cray.com/products/xd1/

[10] K. Danne, R. Muehlenbernd, and M. Platzner, "Executing hardware tasks on dynamically reconfigurable devices under real-time conditions," in *16th International Conference on Field Programmable Logic and Applications (FPL'06)*, 2006, pp. 541–546.

[11] A. DeHon, "DPGA utilization and application," in *FPGA '96: Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays.* New York, NY, USA: ACM Press, 1996, pp. 115–121.

[12] A. DeHon, "Reconfigurable architectures for general-purpose computing," Ph.D. dissertation, Massachusetts Institute of Technology, 1996.

[13] A. DeHon, Y. Markovsky, E. Caspi, M. Chu, R. Huang, S. Perissakis, L. Pozzi, J. Yeh, and J. Wawrzynek, "Stream computations organized for reconfigurable execution," *Microprocessors and Microsystems*, vol. 30, pp. 334–354, 9 2006.

[14] A. Donlin, P. Lysaght, B. Blodget, and G. Troeger, "A virtual file system for dynamically reconfigurable FPGAs." in *Field Programmable Logic and Application, 14th International Conference, FPL 2004, Leuven, Belgium, August 30-September 1, 2004, Proceedings*, 2004, pp. 1127–1129.

[15] DRC computer, "Development system 2000." [Online]. Available: http://www.drccomputer.com

[16] S. Dydel and P. Bala, "Large scale protein sequence alignment using FPGA reprogrammable logic devices." in *Field Programmable Logic and Application, 14th International Conference, FPL 2004, Leuven, Belgium, August 30-September 1, 2004, Proceedings*, 2004, pp. 23–32.

[17] S. J. Eggers and R. H. Katz, "The effect of sharing on the cache and bus performance of parallel programs," in *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems.* New York, NY, USA: ACM Press, 1989, pp. 257–270.

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

[19] T. Fitzpatrick, "SystemVerilog for VHDL users," in *Design, Automation and Test in Europe Conference and Exhibition, 2004 (Vol 2). Proceedings*, April 2004, pp. 1334–1339.

[20] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented fpga computing in the streams-c high level language," in *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines.* Washington, DC, USA: IEEE Computer Society, 2000, p. 49.

[21] M. Gotz and F. Dittmann, "Reconfigurable microkernel-based RTOS: Mechanisms and methods for run-time reconfiguration," in *Reconfigurable Computing and FPGA's, 2006. ReConFig 2006. IEEE International Conference*, 2006, pp. 1–8.

[22] R. W. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *DATE '01: Proceedings of the conference on Design, automation and test in Europe*. Piscataway, NJ, USA: IEEE Press, 2001, pp. 642–649.

[23] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor." in *5th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '97), 16-18 April 1997, Napa Valley, CA, USA*, 1997, pp. 12–21.

[24] A. Krasnov, A. Schultz, G. Gibeling, P.-Y. Droz, and J. Wawrzynek, "RAMP Blue: A message-passing manycore system in FPGAs," in *To Appear in Proceedings of FPL 2007 - International Conference on Field Programmable Logic and Applications*, 2007.

[25] M.-H. Lee, H. Singh, G. Lu, N. Bagherzadeh, F. J. Kurdahi, E. M. Filho, and V. C. Alves, "Design and implementation of the MorphoSys reconfigurable computing processor," *The Journal of VLSI Signal Processing*, vol. 24, no. 2–3, pp. 147–164, 3 2000.

[26] P. H. W. Leong, M. P. Leong, O. Y. H. Cheung, T. Tung, C. M. Kwok, M. Y. Wong, and K. H. Lee, "Pilchard : A reconfigurable computing platform with memory slot interface," in *FCCM '01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 170–179.

[27] E. C. Lin, K. Yu, R. A. Rutenbar, and T. Chen, "A 1000-word vocabulary, speaker-independent, continuous live-mode speech recognizer implemented in a single FPGA," in *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*. New York, NY, USA: ACM Press, 2007, pp. 60–68.

[28] A. Lodi, C. Mucci, M. Bocchi, A. Cappelli, M. D. Dominicis, and L. Ciccarelli, "A multi-context pipelined array for embedded systems," in *16th International Conference on Field Programmable Logic and Applications (FPL'06)*, 2006, pp. 581–588.

[29] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson, "Effects of communication latency, overhead, and bandwidth in a cluster architecture," in *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*. New York, NY, USA: ACM Press, 1997, pp. 85–97.

[30] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins, "Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2004, p. 21224.

[31] S. M. Mishra, D. Cabric, C. Chang, D. Willkomm, B. van Schewick, A. Wolisz, and R. W. Brodersen, "A real time cognitive radio testbed for physical and link layer experiments," in *1st IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN 2005)*, Nov 2005, pp. 562–567.

[32] mjpegtools. [Online]. Available: http://mjpeg.sourceforge.net

[33] E. M. Ortigosa, P. M. Ortigosa, A. Cañas, E. Ros, R. Agís, and J. Ortega, "FPGA implementation of multi-layer perceptrons for speech recognition," in *Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal. Proceedings*, 2003, pp. 1048–1052.

[34] I. Page, "Closing the gap between hardware and software: hardware-software cosynthesis at Oxford," in *IEE Colloquium on Hardware-Software Cosynthesis for Reconfigurable Systems*. IEE, 1996, pp. 2/1–2/11.

[35] P. R. Panda, "SystemC: a modeling platform supporting multiple design abstractions," in *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*. New York, NY, USA: ACM Press, 2001, pp. 75–80.

[36] G. W. Reitwiesner, "The first operating system for the EDVAC," *IEEE Annals of the History of Computing*, vol. 19, no. 1, pp. 55–59, 1997.

[37] J. A. Rowson and A. Sangiovanni-Vincentelli, "Interface-based design," in *DAC '97: Proceedings of the 34th annual conference on Design automation*. New York, NY, USA: ACM Press, 1997, pp. 178–183.

[38] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. R. Taylor, "PipeRench: A virtualized programmable datapath in 0.18 micron technology," in *Proceedings of the IEEE Custom Integrated Circuits Conference, 2002*, 2002, pp. 63–66.

[39] H. K.-H. So and R. W. Brodersen, "Improving usability of FPGA-based reconfigurable computers through operating system support," in *16th International Conference on Field Programmable Logic and Applications (FPL'06)*, 2006, pp. 349–354.

[40] H. K.-H. So, A. Tkachenko, and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH,"

in *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM Press, 2006, pp. 259–264.

[41] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The raw microprocessor: A computational fabric for software circuits and general purpose programs," *IEEE Micro*, vol. 22, pp. 25–35, 3–4 2002.

[42] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: A survey," *Journal of VLSI Signal Processing*, vol. 28, no. 1, pp. 7–27, June 2001.

[43] T. Todman, G. Constantinides, S. Wilton, O. Mencer, W. Luk, and P. Cheung, "Reconfigurable computing: architectures and design methods," in *IEE Proceedings: Computer & Digital Techniques*, vol. 152, no. 2, March 2005, pp. 193–208.

[44] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed fpga," in *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 1997, p. 22.

[45] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink, "Design and programming of embedded multiprocessors: an interface-centric approach," in *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM Press, 2004, pp. 206–217.

[46] H. Walder and M. Platzner, "A runtime environment for reconfigurable hardware operating systems." in *Field Programmable Logic and Application, 14th International Conference, FPL 2004, Leuven, Belgium, August 30-September 1, 2004, Proceedings*, 2004, pp. 831–835.

[47] T. Wiangtong, P. Y. K. Cheung, and W. Luk, "A unified codesign run-time environment for the UltraSONIC reconfigurable computer." in *Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003, Proceedings*, 2003, pp. 396–405.

[48] G. B. Wigley, D. A. Kearney, and D. Warren, "Introducing ReConfigME: An operating system for reconfigurable computing," in *Proceedings of the 12th International Conference on Field Programmable Logic and Application (FPL'02)*. Springer, 2002.

[49] Xilinx, "Xilinx system generator." [Online]. Available: http://www.xilinx.com

[50] Xilinx, *XtremeDSP for Virtex-4 FPGAs User Guide*. [Online]. Available: http://direct.xilinx.com/bvdocs/userguides/ug073.pdf

[51] XtremeData, "XD1000 development system." [Online]. Available: http://www.xtremedatainc.com

[52] Z. Zhang, L. Dolecek, B. Nikolić, V. Anantharam, and M. J. Wainwright, "Investigation of error floors of structured low-density parity-check codes by hardware emulation," in *Proceedings of IEEE Global Communications Conference (GLOBECOM)*, November 2006.

[53] Z. Zhang, L. Dolecek, M. Wainwright, V. Anantharam, and B. Nikolić, "Quantization effects of low-density parity-check decoders," in *Proceedings of IEEE International Conference on Communications*, June 2007.