ANTIKERNEL: A DECENTRALIZED SECURE HARDWARE-SOFTWARE OPERATING SYSTEM ARCHITECTURE

By

Andrew D. Zonenberg

A Dissertation Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: COMPUTER SCIENCE

Examining Committee:

Bülent Yener, Dissertation Adviser

Christopher D. Carothers, Member

John F. McDonald, Member

Boleslaw Szymanski, Member

Rensselaer Polytechnic Institute
Troy, New York

April 2015
(For Graduation May 2015)

UMI Number: 3705663

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

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

UMI 3705663
Published by ProQuest LLC (2015). Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

© Copyright 2015
by
Andrew D. Zonenberg
All Rights Reserved
Released under CC-BY

CONTENTS

LIST OF FIGURES . . . viii

LIST OF LISTINGS . . . x

ACKNOWLEDGMENTS . . . xi

ABSTRACT . . . xii

1. Introduction . . . 1
   - 1.1 Motivation . . . 1
   - 1.2 Contributions . . . 2
   - 1.3 Conventions Used in This Thesis . . . 3
   - 1.4 Organization . . . 3

2. Prior Work . . . 4
   - 2.1 Introduction . . . 4
   - 2.2 Hardware/Software Security Relationships . . . 5
     - 2.2.1 Software Attacking Hardware . . . 5
     - 2.2.2 Software Attacking Software, Exploiting Hardware . . . 6
     - 2.2.3 Physical Attacks on Software, Exploiting Hardware . . . 7
     - 2.2.4 Hardware Attacking Software . . . 8
     - 2.2.5 Hardware Protecting Software . . . 9
   - 2.3 Review of Kernel Architectures . . . 10
     - 2.3.1 Library . . . 10
     - 2.3.2 Monolithic Kernel . . . 10
     - 2.3.3 Microkernel . . . 11
     - 2.3.4 Separation Kernel . . . 12
     - 2.3.5 Exokernel . . . 13
     - 2.3.6 Multikernel . . . 14
     - 2.3.7 Hypervisor . . . 14
     - 2.3.8 Conclusions . . . 16
   - 2.4 Operating Systems in Hardware . . . 17
     - 2.4.1 Hardware Acceleration . . . 17
     - 2.4.2 Security-Focused Designs . . . 19

2.4.3 Conclusions . . . 21
2.5 Network-on-Chip as a Design Paradigm . . . 22
2.6 Formal Verification . . . 24
2.7 Multithreaded CPU Architectures . . . 26
2.7.1 Overview . . . 26
2.7.2 Prior Implementations . . . 28
3. System Architecture . . . 30
3.1 Design Philosophy . . . 30
3.2 Networks . . . 31
3.2.1 RPC . . . 33
3.2.1.1 Packet Structure and Semantics . . . 33
3.2.1.2 Interrupt . . . 33
3.2.1.3 Call . . . 35
3.2.1.4 Network Status / Reserved . . . 37
3.2.1.5 Flow Control . . . 37
3.2.2 DMA . . . 39
3.2.2.1 Packet Structure and Semantics . . . 39
3.2.2.2 Memory Read . . . 40
3.2.2.3 Memory Write . . . 41
3.2.2.4 Command Status Notification . . . 41
3.2.2.5 Flow control . . . 42
3.3 System Information . . . 43
3.3.1 Name Server . . . 43
3.3.2 System Info Server . . . 44
3.4 Memory Management . . . 45
3.5 Processors and Threading . . . 47
3.5.1 Introduction . . . 47
3.5.2 GRAFTON . . . 47
3.5.2.1 Introduction . . . 47
3.5.2.2 ALU . . . 49
3.5.2.3 Memory Bus and L1 Cache . . . 49
3.5.2.4 MMU . . . 50
3.5.2.5 Coprocessor 0 RPC Interface . . . 52
3.5.2.6 RPC Function Calls . . . 55

3.5.2.7 Debug Support . . . 58
3.5.2.8 ELF Loader . . . 59
3.5.2.9 Performance . . . 59
3.5.3 SARATOGA . . . 60
3.5.3.1 Introduction . . . 60
3.5.3.2 Thread Scheduler . . . 62
3.5.3.3 Execution Units . . . 63
3.5.3.4 Register File . . . 64
3.5.3.5 L1 Cache . . . 65
3.5.3.6 MMU . . . 66
3.5.3.7 RPC Network Interface . . . 66
3.5.3.8 ELF Loader with Code Signature Checking . . . 71
3.5.3.9 Remote Attestation . . . 72
3.5.3.10 Performance and Layout . . . 73
4. Tools and Infrastructure . . . 78
4.1 Introduction . . . 78
4.2 Libjtaghal . . . 78
4.2.1 Layer 1: libjtaghal and jtagd . . . 78
4.2.2 Layer 2/3: JtagDebugController and nocswitch . . . 79
4.3 Splash . . . 81
4.3.1 Motivation . . . 81
4.3.2 Architecture . . . 82
4.3.2.1 Splashfiles . . . 83
4.3.2.2 Targets . . . 84
4.4 Test Cluster . . . 86
4.4.1 Introduction . . . 86
4.4.2 Software Construction . . . 87
4.4.3 Hardware Construction . . . 88
4.5 Constgen . . . 93
4.6 Nocgen . . . 95
4.6.1 Design Goals . . . 95
4.6.2 Operation . . . 96
4.7 Nocsniff . . . 100
4.7.1 Introduction . . . 100

4.7.2 Capture Core . . . 100
4.7.3 Packet Viewer . . . 101
4.8 RED TIN Logic Analyzer . . . 105
4.8.1 Motivation . . . 105
4.8.2 Capture Core . . . 106
4.8.3 Control Software . . . 109
5. Security Analysis . . . 111
5.1 Threat Model . . . 111
5.2 Requirements . . . 112
5.3 Prototype Verification . . . 112
5.3.1 Methodology and Goals . . . 112
5.3.2 Assumptions . . . 113
5.3.2.1 Physical Synthesis and Place-And-Route . . . 114
5.3.2.2 Network Topology . . . 114
5.3.3 RPC Network . . . 115
5.3.3.1 Layer-2 Packet Handling . . . 115
5.3.3.2 Layer-3 Packet Routing . . . 116
5.3.4 DMA Network . . . 117
5.3.4.1 Layer-2 Packet Handling . . . 117
5.3.4.2 Layer-3 Packet Routing . . . 118
5.3.5 Name Server . . . 118
5.3.5.1 General Correctness / Liveness . . . 119
5.3.5.2 Name Lookups . . . 119
5.3.5.3 Name Removal . . . 119
5.3.5.4 Name Insertion . . . 120
5.3.6 RAM controller . . . 120
6. Conclusions and Future Work . . . 122
6.1 Summary . . . 122
6.2 Conclusions . . . 122
6.3 Future Work . . . 123
6.3.1 Application Permissions . . . 124
6.3.2 Persistent Identifiers . . . 125
6.3.3 Additional Formal Verification . . . 125
6.3.4 Additional Peripherals . . . 126

6.3.5 Profile-Guided NoC Optimization 126
6.3.6 CPU Improvements 127
6.3.6.1 GRAFTON 127
6.3.6.2 SARATOGA 127
6.3.7 Defense in Depth 128
REFERENCES 129

LIST OF FIGURES

2.1 Scheduling of 5 threads in a barrel processor with a 5-stage pipeline . . 27
2.2 Scheduling of 3 threads in a barrel processor with a 5-stage pipeline . . 28
3.1 Example routing topology (simplified) 31
3.2 Example routing topology showing RPC and DMA 32
3.3 Structure of an RPC packet 33
3.4 Dataflow diagram for RPC interrupt transaction 34
3.5 Dataflow diagram for RPC call/return transaction 36
3.6 Dataflow diagram for RPC call/retry transaction 37
3.7 Timing diagram for RPC send with successful ACK 38
3.8 Timing diagram for RPC send with one retransmit 38
3.9 Timing diagram for RPC send with delayed ACKs 38
3.10 Structure of a DMA packet 39
3.11 Timing diagram for DMA send 42
3.12 Timing diagram for DMA send with delayed ACK 42
3.13 Block diagram of GRAFTON 48
3.14 Block diagram of SARATOGA 61
3.15 Floorplan of Xilinx XC7A200T 74
3.16 Overview of SARATOGA reference SoC 75
3.17 Detail of SARATOGA layout 76
3.18 Detail of SARATOGA floorplanning 77
4.1 JTAG framing example 80
4.2 Test cluster architecture (simplified for clarity) 87
4.3 Small FPGA cluster on desk 89
4.4 Large FPGA cluster on desk 90

4.5 FPGA cluster on rack 90

4.6 FPGA cluster on rack 91

4.7 Acrylic shim FPGA card removed from rack 92

4.8 Custom-built FPGA card removed from rack 92

4.9 Example nocsniff capture of application startup on SARATOGA 105

4.10 Shift register LUT (only four bits shown for clarity) 108

4.11 Shift register LUT configured for partial reconfiguration 109

4.12 Screenshot of RED TIN control application 109

LIST OF LISTINGS

3.1 Initializing division traps 49
3.2 Mapping a page on GRAFTON 51
3.3 Sending an RPC message on GRAFTON 52
3.4 Receiving an RPC message on GRAFTON 53
3.5 A complete RPC function call on GRAFTON 55
3.6 Sending a packed RPC message on SARATOGA 67
3.7 Sending an unpacked RPC message on SARATOGA 67
3.8 Receiving an unpacked RPC message on SARATOGA 69
4.1 Creating a C++ executable targeting three different platforms 84
4.2 Constgen input file 93
4.3 Constgen C/C++ output 94
4.4 Constgen Verilog output 94
4.5 Example NocGen board support file 96
4.6 Example NocGen script 97
4.7 Instantiating a sniffer in nocgen 100
4.8 Using a sniffer from Splash 101
4.9 Extended Doxygen syntax used by nocsniff 102

ACKNOWLEDGMENTS

First and foremost, I would like to thank Donald Knuth, Leslie Lamport, and all of the contributors to TeX and LaTeX. Without their software I would have experienced the unimaginable suffering of writing this dissertation in Microsoft Word.

I thank John McMaster for introducing me to the rabbit hole of semiconductor security. It goes a long way down, and this thesis is only the beginning.

My adviser, Prof. Bülent Yener, encouraged my research from the very beginning and gave me the initial ideas which inspired this research. Little did I know that first email during freshman orientation would lead to me pursuing a PhD with the same research group!

Many friends, colleagues, and research group members provided helpful comments, suggestions, and ideas. In no particular order I would like to thank Jeremy Blackthorne, Brennan Cozzens, Griffon Bowman, Kurt Rosenfeld, the members of RPISEC, and all of the regulars in the siliconpr0n.org IRC.

Clifford Wolf, the author of `yosys`, provided invaluable assistance in getting the formal correctness proofs to work. Between fixing bugs in the tool, answering questions about usage, and adding new features he saved me countless hours of frustration.

I also thank my parents for homeschooling me, and especially my father for leaving his C++ textbook where I could find it back in the summer of 2001. While the process may have been somewhat unorthodox, it does seem to have paid off.

Last of all, but certainly not least, I want to thank my wonderful fiancee, Allyson Miccio, for keeping me (somewhat) sane during the last two years and putting up with all of the late nights and time I spent working on research.

ABSTRACT

Security of monolithic kernels, and even microkernels, relies on a large and complex body of code (including both software and hardware components) being entirely bug-free. Most contemporary operating systems can be completely compromised by a bug anywhere in this codebase, from the network stack to the CPU pipeline's handling of privilege levels, regardless of whether a particular application uses that feature or not. Even formally verified software is vulnerable to failure when the hardware, or the hardware-software interface, has not been verified.

This thesis describes Antikernel, a novel operating system architecture consisting of both hardware and software components and designed to be fundamentally more secure than the existing state of the art. In order to make formal verification easier, and improve parallelism, the Antikernel system is highly modular and consists of many independent hardware state machines (one or more of which may be a general-purpose CPU running application or systems software) connected by a packet-switched network-on-chip (NoC).

The Antikernel architecture is unique in that there is no "all-powerful" software which has the ability to read or modify arbitrary data on the system, gain low-level control of the hardware, etc. All software is unprivileged; the concept of "root" or "kernel mode" simply does not exist so there is no possibility of malicious software achieving such capabilities.

The prototype Antikernel system was written in a mixture of Verilog, C, and MIPS assembly language for the actual operating system, plus a large body of C++ in debug/support tools which are used for development but do not actually run on the target system. The prototype was verified with a combination of simulation (Xilinx ISim), formal model checking (using the MiniSAT solver integrated with `yosys`), and hardware testing (using a batch processing cluster consisting of Xilinx Spartan-3A, Spartan-6, Artix-7, and Kintex-7 FPGAs).

CHAPTER 1
Introduction

1.1 Motivation

Computers are deeply integrated into our daily life and with the impending rise of the Internet of Things (IoT) this role is likely to increase even further. As we grow more dependent on them, and they become more interconnected, the potential damage caused by an attack becomes even greater.

Unfortunately, most contemporary operating systems are excessively complex and have a massive attack surface. Alternative designs such as microkernels improve on this but still leave potential failure points. While formal verification of the entire trusted computing base (TCB) can greatly reduce the number of attack vectors, the current state of the art in operating system verification assumes correctness of the hardware and hardware-software interface code, which can present an attractive entry (or persistence) vector for the adversary.

Application security can be compromised in countless ways, some of which do not involve arbitrary code execution (such as directory traversal in a web server). Rather than trying to detect them all, this thesis focuses on damage control. Antikernel's main goal is to protect the integrity of the OS and keep application-level security domains separate from one another. Needless to say, the more an application is able to compartmentalize itself into subsystems the more it can benefit from these protections.

The defensive measures introduced in this work operate at the system level and are orthogonal to application-layer mitigations such as ASLR, stack canaries, etc; they can be used instead of them or together as part of a defense-in-depth strategy. We have not implemented any application-layer countermeasures in our current prototype, since the focus of this research is to explore the security implications of the architecture rather than develop a production-ready system.

Unless otherwise stated, the threat model in this thesis is that of a remote attacker with no physical access to the target system, but who can communicate with

it arbitrarily through external interfaces (network etc). The goal is to prevent an attacker from escalating privileges, or limit the extent to which doing so is possible. Since many classical operating system designs are vulnerable to rootkits, which can intercept and tamper with system calls and IPC to make detecting a compromise much harder, methods for making rootkits difficult or impossible to deploy are of significant interest as well.

1.2 Contributions

This work unifies two previously orthogonal fields within computer science and engineering (hardware accelerators and operating system security) in order to create a new model for operating system architecture which can enforce OS security policy at a much lower level than previously possible.

In contrast to the conventional model of an OS kernel (a software component running on a general-purpose processor, which coordinates the various hardware and software components in the system) our system blurs or eliminates many of the classical boundaries between software, hardware, kernels, and drivers. Each hardware device includes state machines which implement low-level resource management and security for that particlar device, and provides an API via message passing *directly to userspace*. Applications software may either access this API directly (as in an exokernel) or through server software providing additional abstractions (as in a microkernel).

Perhaps most uniquely, there is no single piece of software or hardware in our architecture which corresponds to the kernel in a classical OS. The operating system is instead an emergent entity arising out of the collective behavior of a series of distinct hardware modules, which together provide all of the services normally provided by a kernel and drivers. By decentralizing to this extent, and creating natural chokepoints for dataflow between functional subsystems, we significantly reduce the portion of the system which is potentially compromised in the event of a vulnerability in any one portion, and render API-hooking rootkits impossible. This modular structure also allows piecewise formal verification of the system since the dataflow between all components is constrained to a single well-defined interface.

We have tested the feasibility of the architecture by creating a proof-of-concept implementation targeting a Xilinx FPGA, and report some experimental results including formal correctness proofs for several critical components. The implementation has been released as open source to encourage verification of our results and further research.

1.3 Conventions Used in This Thesis

`Monospaced font` is used to for processor instructions, executable or other file names, function names, and variable names.

Italic font is used for emphasis and direct quotes of third parties.

1.4 Organization

The remainder of this thesis is organized as follows:

Chapter 2 describes prior work in operating systems, security, and computer architecture which is relevant to the remainder of the discussion.

Chapter 3 describes the architecture of the prototype system, beginning with general design concepts and continuing on to details of core system components and processor microarchitecture.

The Antikernel system does not exist in a vacuum; many custom tools were created in order to ease design, implementation, or verification of the prototype. These tools, as well as the hardware infrastructure used for testing, are discussed in chapter 4.

Chapter 5 analyzes the security of the Antikernel architecture at a high level including a more detailed security model and discussion of how the prototype was verified for correctness. While the full source code for the machine-checked proofs is located in the source distribution for brevity, the high-level architecture of each proof is discussed.

Finally, chapter 6 contains concluding remarks and discussion of further work which is planned in order further improve Antikernel.

CHAPTER 2
Prior Work

2.1 Introduction

This chapter begins by introducing the relationships between hardware and software security in section 2.2. This material is central to the remainder of the discussion. We then review existing kernel architectures, including the separation kernel architecture, in section 2.3 and discuss their merits and weaknesses with special emphasis on security.

Operating systems can be made more secure by moving carefully selected components into hardware, which provides some protection against software-based exploits. Hardware acceleration of operating systems has been the subject of significant research in the past, but in nearly all cases the intention was to improve performance rather than security. Section 2.4 reviews existing implementations of operating systems and OS-related accelerators in custom hardware and examines the possible security benefits of these designs.

Components in a distributed operating system typically communicate through a packet-switched message passing interface such as a LAN. The SoC equivalent of this is a network-on-chip (NoC). Section 2.5 introduces the NoC paradigm and discusses the security aspects of various implementations.

The only way to achieve perfect security within a given threat model is to mathematically prove correctness of the system. Section 2.6 provides some examples of formal validation as applied to both hardware and operating system design and discusses the necessity of validation for a security-critical hardware OS.

At first glance, the goals of close integration between system components and minimizing communications overhead for performance, and explicit separation and compartmentalization for security, appear contradictory. Section 2.7 describes several multi-threaded CPU architectures and discusses how a barrel processor could be integrated with a NoC in a hardware OS to provide a compromise, which has low overhead but also strong compartmentalization between security domains.

2.2 Hardware/Software Security Relationships

Software and hardware engineers sometimes fall into the trap of considering their respective disciplines in isolation, ignoring the other. Unfortunately, many security issues cannot be understood - or even seen to exist - without considering both sides. We consider several different kinds of attack depending on whether the hardware is the source of the attack or the target.

2.2.1 Software Attacking Hardware

The first class of attack we consider is that of malicious software attacking the hardware. Any system which can potentially be damaged by software is at risk. Examples of dangerous features are the abilities to overwrite or erase firmware elsewhere on the board (as done by CIH, described in [1]), adjust voltage regulators or fans, or control external robots and machinery (as was exploited to great effect by Stuxnet, described in [2]).

This category also includes attacks in which malicious software causes hardware under its control to harm external equipment or personnel, for example causing a factory robot to cut up cars as they go down the assembly line. Sometimes these dangerous features were unintended side effect of a careless design, such as in the Therac-25 radiation therapy device.

The Therac-25 was intended to deliver low-energy electron beams or low-energy X-ray beams (produced by directing a very high energy electron beam onto a metal plate); delivering a high-energy electron beam to the patient should not have been possible. Due to a complex interaction between a race condition and integer overflow, however, certain sequences of keyboard input could lead to the machine delivering a massive radiation overdose. This occurred at least six times during the machine's deployment, as documented in the subsequent accident investigation [3], and resulted in the deaths of several patients. Although these were tragic accidents rather than deliberate malice , it is easy to see that the impact of malware on such a system could be catastrophic.

This was far from the only time in which real-world physical harm was caused by a software bug. Toyota's engine control units for certain car models contained

a bug ([4]) which could cause the control loop to freeze with the throttle set to maximum. This was implicated in several fatal crashes.

Given that the vast majority of industrial control systems are computer-controlled in some way, the potential for damage is immense. A German steel mill was recently damaged ([5]) in what appears to be a deliberate and carefully planned attack perpetrated by attackers with detailed knowledge of the plant's control systems. Similar operations may have been conducted as early as the Cold War - a CIA campaign is alleged in [6,7] to have been behind a 1982 oil pipeline explosion in Siberia.

Assuming that the ability to cause physical damage cannot be removed outright by hardware-based interlocks or safety mechanisms, mitigating these attacks requires that the hardware have some way to determine that the request is permitted by the operating system security policy. The classical way of doing this is to only permit potentially dangerous machine instructions (such as writes to I/O ports) to be executed in kernel mode, however several potential alternatives exist and are addressed later in this thesis.

2.2.2 Software Attacking Software, Exploiting Hardware

These attacks involve malicious software manipulating hardware in order to attack other software. An example of such an attack would be an unprivileged process using a peripheral (graphics card, etc) to DMA shellcode into a privileged process's address space (demonstrated by [8] and many others) or, as in [9], read encryption keys or other sensitive data.

Another possible attack of this category would be for an unprivileged process to exploit a bug in the CPU (for example incorrect handling of register renaming or pipeline forwarding after a mode switch) to affect the state of a privileged process.

In addition to actual silicon bugs, excessively complex architectures such as x86 are vulnerable to implementation bugs caused by subtle documentation errors or even differences in silicon between the two major vendors, Intel and AMD. CVE-2006-0744 (see details in [10]) is an example of a vulnerability which, while not strictly a silicon bug according to the vendor's claims (Intel's CPUs behaved as per

their published documentation) could result in memory corruption in OSes written to assume that the CPU adhered to the very slightly different AMD64 specification. The impact of this issue was massive - user-to-kernel privilege escalation on 64-bit Windows, Linux, Xen, FreeBSD, OpenBSD, and a host of other platforms. Software workarounds for the syscall handler were able to fix the issue but little imagination is required to see that the impact of a true silicon bug could be even more devastating.

2.2.3 Physical Attacks on Software, Exploiting Hardware

Early work on software security neglected the hardware, assuming that it was trustworthy, flawless, and in a safe environment. This has been repeatedly proven untrue (by [11–14] and countless others) with otherwise correct implementations of well-known and trusted algorithms like RSA and AES failing horribly due to the hardware being vulnerable to non-invasive attacks which induced a malfunction in the device to produce an incorrect ciphertext. By analyzing the resulting ciphertexts it is often possible to recover the plaintext, key, or both.

It is also possible to perform attacks merely by observing the externally visible state of the targeted system while it processes sensitive information. Differential power analysis, as described in [15, 16], is an example of such a technique, which exploits the fact that each gate in a system uses more power to change state than to remain idle. By correlating many traces of power over time, it is often possible to recover some or all of a cryptographic key.

In addition to the non-invasive attacks mentioned above, it is also possible to attack an embedded system by more invasive methods, such as physically placing probes on internal wires inside a chip to monitor signals, cutting or shorting wires (as demonstrated in [17]) with a focused ion beam (FIB), or using probes to inject voltages into wires. These are capable of defeating essentially all software-based security although physical antitamper techniques can increase the cost and difficulty of such an attack.

Semi-invasive attacks, as introduced by [18], are an intermediate class of attacks which involve removing the target IC from its packaging but not touching it. Lasers, ultraviolet light, or electromagnetic fields can be used to induce targeted

faults in the device; optical or EM sensors can also be used to observe internal state. Using these techniques, the author demonstrated in [19, 20] the ability to defeat the readout protection on a commodity 8-bit microcontroller (Microchip PIC12F683) using an inexpensive microscope, a UV germicidal lamp, and materials available at any department store: a Dremel tool, drain cleaner, a hot plate, nail polish, and nail polish remover... a one-time purchase of perhaps $500 plus a few dollars for each chip being attacked.

2.2.4 Hardware Attacking Software

The other side of IC security involves protecting the end user of the device from malicious modifications made to the hardware. This can involve anything from slightly thinning a wire (so that it fails from electromigration long before the intended service life is up) to adding a complex backdoor consisting of many gates which, for example, causes a firewall appliance to disable filtering when a packet containing a certain bit sequence is received. A group from NYU Poly explored this topic in great detail in [21], presenting the first taxonomy of hardware-based malware.

The author has previously suggested in [22] that FPGAs may be inherently more secure than ASICs in this regard because FPGAs are so flexible at fabrication time. While "blunt instrument" attacks, such as making the entire device fail early, are still possible, high-level attacks targeting the business logic of a particular device would be difficult to implement since the attacker has no way of knowing which logic cells will do what. Although the attacker could potentially duplicate backdoor logic for every general-purpose logic cell, this would add a large amount of overhead and would likely be detected. (Particularly paranoid designers could even buy a lifetime supply of their chosen FPGA before writing firmware. Since the code did not exist at the time the chip was manufactured, there is no possibility of it containing a targeted backdoor. This greatly reduces the portions of the device which need to be audited.)

Hard IP cores present a more attractive target to the attacker (similar to an ordinary ASIC), since their function is known in advance. An attacker could include

logic in each of the gigabit serial transceivers, for example, that would send data to the FPGA's internal reconfiguration port (such as Xilinx's ICAP, described in [23]) if a particular "magic" bit sequence were observed on the input. By sending a packet containing the trigger sequence to a network port, writing it to a temporary file on a SATA disk, or otherwise causing it to be sent through the transceiver the attacker could then overwrite portions of the FPGA with malicious logic.

Another potential attack is to backdoor the design tools. FPGA vendors release new versions of their toolchains fairly frequently, and these downloads could potentially be altered in transit by a well-equipped attacker (or compromised at the source by attacking the vendor's network or inserting an agent on the development team). Since the synthesis and mapping tools have access to the RTL source, high-level structures such as adder trees and known soft IP blocks can be identified and, through subgraph isomorphisms, used to identify known algorithms such as AES. (This is not too much of a stretch since FPGA synthesis tools already detect structures such as state machines to optimize synthesis). Backdooring of the toolchain can be done at any time through software attacks on existing workstations and can even be customized to a particular target, as demonstrated by [24]. In this case, however, the toolchain was not the source of the attack: the RTL source of the target system was compromised and a backdoored toolchain was used to prevent the malware from showing up in synthesis reports.

2.2.5 Hardware Protecting Software

The same relationships can be used defensively as well. For example, protected memory provides a hardware-enforced policy to prevent processes from writing to each other's memory. IOMMUs can mitigate the DMA attack described above.

Hardware interlocks on physically dangerous hardware can provide a vital sanity check to prevent malfunctioning or malicious software from causing harm. In the case of the Therac-25, the software bugs could have been rendered harmless if the electron beam's high-power mode was physically disabled when the X-ray generation target was not in the path of the beam. [1]

[1] Such a safety feature was, in fact, present in earlier Therac models. While the same software bug was present in those systems it caused the machine to silently shut down rather than exposing

Another potential defensive strategy, the primary focus of this thesis, is to move portions of the OS kernel into hardware. By hard-wiring these components, and only permitting state changes through a well-defined interface, it becomes impossible for their internal state to be tampered with in violation of policy. Performance improvements are possible as well, by reducing the latency of system calls, reducing the number of context switches due to interrupts, and allowing some hardware drivers to execute in parallel with applications without using CPU resources.

2.3 Review of Kernel Architectures

2.3.1 Library

The simplest, albeit least secure, design for an embedded platform is the outright lack of an OS. The application is merely linked with a few support libraries for controlling hardware to form a single executable. This is rare in systems more complex than an 8/16 bit microcontroller running a few thousand lines of code, because most larger systems require features such as tasks or threads. (An example system of this would be a single-threaded C program linked with the target MCU's peripheral library.) Systems using a library as the only OS can be completely compromised by a vulnerability anywhere in the entire codebase, but the hardware tends to be so simple and cost-sensitive that running a full operating system is impractical or impossible. On the bright side, most such platforms have limited communications capabilities and are thus difficult to compromise remotely. As a consequence such systems are beyond the scope of this thesis.

2.3.2 Monolithic Kernel

The next logical advance in architecture is to separate the kernel from application code. Since system state is protected against accidental or malicious modification from userspace, stability and security are improved significantly. Most systems using monolithic kernels separate userspace applications from one another to some extent (for example, by preventing applications run by different users from writing to each others' memory) but drivers and the core of the kernel run in the same

the patient to the full beam energy.

address space with no protection against a buggy or rogue module tampering with another module's state. Embedded Linux (and its descendants, such as Android and μcLinux) uses a monolithic kernel, as do VxWorks [25] and Windows CE [26].

Google's Android operating system is based on the monolithic Linux kernel but attempts to provide strong isolation between applications by running each application as a separate Linux user account (see security model documentation in [27]). Permissions are assigned to applications after confirming with the user during installation. Fine-grained access control is not typically possible as a user must accept or reject all privileges at once, although this is a failure of the user interface rather than the security model and several third-party add-ons attempt to provide this capability. Since it is based on Linux, there is no isolation between drivers and kernel modules.

2.3.3 Microkernel

The lack of driver isolation mentioned above is solved to some extent by the microkernel architecture. Only the bare minimum of the operating system runs in privileged mode to provide a few core services (typically memory management, interprocess communication, and threading) while hardware drivers and other OS services (such as filesystems, network stacks, etc) run in userspace. This provides protection against the simple case of a driver corrupting system state directly, and potentially allows malfunctioning drivers or subsystems to be restarted or terminated without catastrophic effects on overall system stability.

Unfortunately, however, real systems tend to be substantially more complex. Any hardware device capable of DMA has the potential to overwrite (or observe) critical kernel state. While IOMMUs (such as used by Intel VT-d) can help mitigate these issues they are rarely used for this purpose; most use of IOMMUs to date has focused on allowing virtual machines to access peripherals more efficiently.

Microkernels are commonly used on low-end microcontrollers, such as μC/OS-II [28] and FreeRTOS [29] but are less popular on larger systems. There are many potential reasons for this ranging from developer familiarity (developers are likely to be most familiar with monolithic kernels) to performance (microkernels require

more context switching, which typically comes with substantial runtime overhead).

QNX [30], L4 [31], and Symbian (now defunct) are among the few microkernels which saw common use on higher-end processors. Some other modern platforms, such as the XNU kernel [32] used in iOS, are hybrid designs which borrow some features from microkernels but do not go far enough to be considered true microkernels. As would be expected this offers a compromise between the compartmentalization of a full microkernel and the performance of a monolithic kernel.

2.3.4 Separation Kernel

Separation kernels, as introduced by Rushby in [33], provide even stronger guarantees of isolation than microkernels. The kernel is re-invented as a distributed system in which each module manages its own security policy and is independent of all of the other portions of the system, and modules communicate only through a well-defined interface which limits the interactions to those allowed by a global policy.

Rushby's paper makes the point that allowing *any* program to be given ultimate trust for any reason (for example, setuid root on Linux for the "ping" program so it can send raw ICMP packets) admits them to the trusted computing base (TCB) and thus that they, and all of their dependencies, are now critical to the security of the system. A single security bug in the "ping" executable could allow an unprivileged local user, or potentially even a remote attacker, to gain root access. In an appropriately decentralized system the "ping" executable could be replaced by an otherwise unprivileged process whose only distinguishing characteristic is the ability to send raw ICMP frames. Exploiting a bug in the ping command would then grant only the ability to display text on the calling process's tty and send ping packets; other activities such as file system operations would not be possible.

Despite the obvious security benefits, separation kernels have not been widely deployed. As with microkernels, the most likely reasons are the unfamiliar architecture (even for developers used to microkernels) and the perceived performance overhead.

The Mathematically Analyzed Separation Kernel (MASK), developed jointly

by Motorola and the National Security Agency (NSA) for a cryptographic SoC known as the Advanced Infosec Machine (AIM), is one of the first separation kernels to be deployed in a production system. The intended purpose was allowing the AIM to process data at multiple classification levels simultaneously without any risk of sensitive data leaking to a less secure domain. While the actual code and chip design for the AIM are classified, the operating system architecture was described by Motorola [34] in an unclassified paper.

Integrity [35] is a commercial separation-kernel based platform. It provides fixed time slices for each application to ensure hard-realtime performance, and also provides memory quotas to ensure that critical applications will always have at least some minimal amount of memory available. The system is available both as a standalone RTOS and as a hypervisor which can run more conventional guest operating systems.

2.3.5 Exokernel

Another unusual OS architecture, MIT's Exokernel [36,37] is focused on performance but has several interesting properties when studied from a security viewpoint. Rather than basing the OS on many layers of abstraction and forcing applications to work at the highest level, drivers are simplified to the extreme. An exokernel driver simply divides its resource into pieces (slices of CPU time, disk sectors, RAM pages, and so on), provides an interface for programs to request and release these resources, and enforces security policy by preventing one program from altering another's data. As with microkernels, higher level services such as filesystems and network stacks are typically implemented as userspace servers or libraries. (It should be noted that the exokernel architecture itself does not specify whether the lowest level drivers are integrated into the kernel as in a monolithic design, or in separate processes as in a microkernel.)

The original intention of the exokernel architecture was to improve performance. For example, if an application needs temporary storage on disk for transient data, there is no reason to impose the overhead of a file system. Instead, the application can request a raw sector directly from the disk driver and use it. Even

virtual memory paging is implemented at the application layer and used only when needed. The authors of [37] report massive performance improvements, up to two orders of magnitude in some of their tests.

As with separation kernels, exokernels have not been widely deployed and have mostly remained research curiosities. The author believes that this is unfortunate, as the decentralized structure of an exokernel can be made very similar to a separation kernel (although none of the original papers on the architecture explore this possibility). Since the TCB of an exokernel is so tiny the attack surface is drastically reduced compared to a typical monolithic kernel or even a microkernel.

2.3.6 Multikernel

The multikernel, developed by Microsoft Research [38], is an OS that can scale to large numbers of cores - potentially even using different ISAs - by reinventing the computer as a distributed system using concepts from supercomputing. Classical abstractions like shared memory and system calls are discarded and replaced with message passing between cores.

Barrelfish, the prototype implementation of the multikernel architecture, can run on both x86 and ARM. Each core runs an exokernel-esque "CPU driver" that provides context switching within that core. Cores are divided into a 2D grid network; one core of each CPU serves as the gateway to other CPU sockets or remote nodes accessed over Ethernet.

Performance testing of Barrelfish shows results comparable to, if not better than, contemporary operating systems on the same hardware for most metrics.

2.3.7 Hypervisor

Instead of attempting to secure the kernel against attack, one can also take a very different approach: Run multiple kernels in individual virtual machines on top of a hypervisor. As long as the hypervisor is able to reliably enforce separation between the VMs, individual security domains may be kept separate and compromise of one domain cannot be extended to compromise of the others. An unfortunate side effect of this approach is that a separate copy of the kernel and core services

is required for each security domain, which increases the hardware requirements for the host.

Qubes [39] is a virtualization-based compartmentalized OS intended for desktop computing. The OS implements many separation-kernel style isolation capabilities while maintaining backward compatibility with most Linux applications by means of re-using existing off-the-shelf components. Drivers which are considered vulnerable to compromise, such as the TCP/IP stack, run in dedicated service VMs. The major flaws of the architecture are reduced performance due to virtualization overhead and the fact that it relies on the virtualization extensions of the (massively complex) x86 architecture to provide isolation.

The authors of Qubes are clearly aware of this, saying "*There is no such thing as a 100% secure OS, at least on x86 COTS hardware. The software and hardware is simply too complex to analyze for all potential errors in case of x86 platforms... It is hard not to think that there might be a bug in a CPU that could e.g. allow for unauthorized ring3 to ring0 privilege escalation. Interestingly, no such bugs have ever been publicly disclosed*" (page 42). They believe, however, that their architecture does provide a significant improvement over existing alternatives.

Although embedded platforms have made fairly limited use of virtualization to date, the ARM architecture does have virtualization extensions and there have been some attempts (such as [40, 41]) to virtualize smartphones. One proposed use case is to run both a general purpose operating system for running end-user applications and a dedicated RTOS for hard-realtime radio baseband functions on the same CPU. Separation of personal and work applications on the same hardware is mentioned as another scenario.

While virtualizing several independent operating systems may be a practical option for systems requiring only a handful of security domains, it scales poorly since a separate copy of the kernel (and associated driver infrastructure) is required for each security domain. On an embedded device with fairly minimal amounts of memory and CPU capacity, it is unlikely that running each functional subsystem in its own VM will be practical.

ARM TrustZone [42,43] can be thought of as a hardware virtualization system

integrated into the CPU that supports exactly two guests: trusted and untrusted. The trusted guest is able to use all features of the CPU and access all peripherals without restriction; the untrusted guest is prevented from accessing a configurable set of memory address ranges or peripherals. While perhaps suitable for simpler applications such as DRM with only one “secure” module, it does not support large-scale compartmentalization with many independent security domains. It is possible in principle to run a software-based hypervisor in the trusted mode, but this essentially negates any of the security that would otherwise be gained. In addition the secure OS is often excessively complex and susceptible to vulnerabilities of its own, such as CVE-2013-3051 [44].

2.3.8 Conclusions

Each architecture can be seen to have strengths and weaknesses. Monolithic kernels are efficient and easy to develop but have many potential points of failure. Microkernels are slower and can be harder to design, but system security and stability is improved. Separation kernels take the microkernel concept to a more extreme level and thus have the same traits as microkernels, but more so. Exokernels are tricky to design well, and can suffer from the same task-switching overhead problem as microkernels, but the performance gains from eliminating unnecessary abstractions can make up for this and more. In addition, since the core drivers of an exokernel are smaller and simpler than monolithic or even microkernel drivers, they are less likely to contain bugs with stability or security impacts - something which is often neglected by researchers focusing on the kernel itself. Hypervisors have even more overhead than microkernels since applications run under a kernel which is itself under the hypervisor, but they can easily run any number of guest operating systems (which need not be the same) in their own security domain.

The exokernel architecture appears very promising but still leaves some things to be desired. If the drivers run in kernel mode then the same security/stability issues of a monolithic kernel are present (although perhaps to a reduced extent, since the drivers are simpler). If the drivers run in userspace, then it is susceptible to the same performance issues of a microkernel. What if it were possible to get

the best of both worlds? We suggest that that moving the drivers into hardware, as described in the next section, may offer a solution to this problem.

2.4 Operating Systems in Hardware

There are many examples in the literature of operating system components being moved into hardware [2] however the majority of these systems are focused on performance and do not touch on the security implications of their designs at all.

Fundamentally, any hard-wired OS component has an intrinsic *local* security benefit over an equivalent software version - it is physically impossible for software to tamper with it. This brings an unfortunate corollary - it cannot be patched if a design error, possibly with security implications, is discovered. Extremely careful testing and validation of both the design and implementation is thus required.

Hardware OSes may not, however, provide any *global* benefits to security. If the hardware component does not perform adequate validation or authentication on commands passed to it from software, compromised or malicious software can simply coerce the hardware into doing its bidding. When evaluating a hardware accelerator's security properties it is important to keep this in mind.

2.4.1 Hardware Acceleration

[45] proposes a distributed OS built into a network-on-chip, or NoC. (The NoC paradigm is covered more comprehensively in the next chapter of this paper.) Rather than context switching, each task is given its own CPU. This completely eliminates the need for many conventional OS capabilities like memory protection or task scheduling and also provides the ability to do clock gating / speed adjustment per "process". The system is adapted to a modified version of the OSEK monolithic RTOS on an FPGA. Some automated system generation by parsing source code for each task is proposed but not implemented. This architecture has a massive

[2] This dissertation uses the term "hardware OS" to refer to a series of state machines implemented in silicon which provide operating system services to a computer. Some other authors use the same term to refer to a very different concept: a component of an operating system (which is typically implemented in software) responsible for managing partitions of an FPGA or other reconfigurable computing device.

overhead in silicon area (at least one CPU per security domain is required, probably several since no context switching is supported and many domains are likely to require more than one thread) which makes it somewhat impractical. While the architecture exhibits some of the properties of a separation kernel, it is unclear whether the interconnect provides any security (for example, preventing one processor from spoofing another's address in a message).

[46] proposes a basic RTOS which contains a simple hardware microkernel implementing a scheduler (supporting up to 16 tasks), 16 semaphores, and 4 timers. μC/OS-II was ported to this architecture (with a small software driver to provide API compatibility with the original RTOS). Task switching is still done by software as there are no register windows or hardware-based features for context switching. Significant performance benefits are realized compared to a software-only implementation but the security implications of the architecture are not addressed. Since there are no authentication or protection capabilities built into the accelerator, no security benefit is realized.

[47] implements an FPGA-based prototype of a hardware RTOS. Results are presented from synthesizing for 800nm CMOS, however it appears that the chip was never actually manufactured. The hardware component of the RTOS implements task scheduling, semaphores, interrupts, and timers. As with the above papers in this section there is no mention of security whatsoever and the lack of any authentication on the hardware-to-software interface means it provides no security benefit.

[48] describes a microkernel-based OS using a 2D mesh NoC. Each node is a CPU with a microkernel on it, running user processes and/or servers. They also examine the possibility of having the microkernel running on a single node and dismiss this due to poor performance. Security is mentioned as a benefit of microkernel-based systems in general but there is no discussion of the security aspects of their specific implementation. The lack of any actual hardware acceleration makes the system no more or less secure than an equivalent system using several physical PCs.

[49] presents a framework for codesign of a MP-SoC and the RTOS it runs. It includes hardware-based memory allocation and several other features (such as

basic IPC) and exposes this functionality to both CPU-based applications and hardware accelerators. Some of the RTOS modules are dynamically generated based on parameters such as the number of CPU cores in the system.

[50] proposes a "hardware OS kernel", or HOSK, which is connected to a conventional (unmodified) RISC processor and functions as an accelerator. The HOSK implements task scheduling, semaphores, and context pre-fetching but does not actually implement context switching (this is left to the CPU, since the HOSK has no direct access to the CPU register file). Swapping of paged-out contexts to external RAM is handled by the HOSK. Security is not addressed at all. Since the HOSK functions purely as an accelerator and imposes no limitations on the system, and does not appear to authenticate requests from the CPU in any way, security is not improved.

[51] describes BORPH, an operating system for a reconfigurable platform containing one or more CPUs and one or more reconfigurable components such as FPGAs. It introduces the concept of a "hardware process", which is functionally equivalent to a conventional operating system process and supports the same IPC methods as POSIX (shared files, pipes, shared memory, signals, and sockets) and can access conventional OS resources such as the filesystem. No special isolation/authentication mechanisms are described and the platform runs on top of a conventional Linux kernel so the architecture provides no real security benefits. The concept of having interchangeable hardware and software modules using the same API, however, is a promising one.

2.4.2 Security-Focused Designs

[52] presents the first implementation of a separation kernel in hardware. It describes a distributed OS based on a "time-triggered network on chip" (TTNoC) connecting a series of IP cores, each considered a separate partition within the system. The TTNoC appears to be a shared bus, more so than a packet-switched network, since it uses time-division multiple access (TDMA) to share the single medium between all hosts. Each IP core contains a "Trusted Interface Subsystem" or TISS, which ensures that the core cannot transmit out of its TDMA slot, and

external - potentially untrustworthy - logic. An FPGA-based prototype is presented. The network includes a global arbiter, the Trusted Network Authority or TNA, which sets up virtual circuits either statically at configuration time or dynamically via (signed) reconfiguration messages.

While the TTNoC provides the highly desirable property of complete and deterministic isolation between hosts - no traffic sent by any other host can ever impact the ability of another to communicate and thus there are no timing / resource exhaustion side channels - it also results in the lack of burst capabilities and does not scale well to systems involving a large number of hosts (in a system with N nodes each one can only use 1/N of the available bandwidth).

The proposed design requires a separate CPU core (or hard accelerator) for each security domain and does not appear to consider the possibility of sharing CPU resources between applications. As with [45] this area overhead is likely to be prohibitive for systems with more than a small number of security domains.

[53] describes a "zero-kernel operating system" or ZKOS. The general guiding principles of "no all-powerful component", "hardware-software codesign", and "safe design" are very similar to this work, as well as the conclusion that privilege rings are an archaic and far too coarse-grained concept, however their solution is somewhat different. Their system relies on "streams" (point-to-point one-way communications links) and "gates" (similar to a syscall vector, allows one security domain to call into another) for IPC and does not support arbitrary point-to-point communication. Threading and message passing are implemented in hardware.

There is no support (by design) for interrupts in ZKOS; instead hard real-time deadlines must be met by a single OS-controlled timer interrupt which ensures hard-realtime deadlines can be met. As a result of this, peripherals are expected to buffer data until polled and drop excess data if the interrupt is not called often enough.

The ZKOS architecture appears to be primarily software based with minimal hardware support and does not support hardware processes/drivers, which can greatly help efficient handling of hard real-time processing such as high-speed networking. An extremely fine-grained tagged memory scheme (permissions bits

associated with individual memory words) is used for access control at the cost of a significant (100% - 300%) memory overhead.

BiiN [54] was the result of a joint Intel-Siemens project to develop a fault-tolerant computer, which could be configured in several fault-tolerant modes including paired lock-step CPUs. The system's architecture was somewhat reminiscent of a multikernel since it uses message passing for most communication and can scale transparently to a network, although it predated Baumann's paper by twenty years and does support shared memory between processes on a single node.

A capability-based security system is used to control access to particular objects in memory or disk. It appears that the capabilities are managed by the "type manager" for the particular resource of interest, making it similar to a microkernel or exokernel. The system architecture advocates heavy compartmentalization with each program divided up as much as possible, and using protected memory between compartments (although the focus of this was reliability against hardware faults through means such as error correcting codes and lock-stepped CPUs, not security against deliberate tampering). No mention of formal verification could be found in any published documentation.

BiiN also included microcoded support for some operating system functionality, such as multiprocessing and remote procedure calls, and a programmable "channel processor" for accelerating I/O (although it appears that the channel processor does not enforce security policy). The company closed down in 1989 [55] as a result of poor management and failure to secure adequate market share.

2.4.3 Conclusions

Most older hardware-accelerated OS designs focused purely on performance; typically there was no attempt to improve security and the systems were no more secure than conventional ones. These systems often pushed boundaries in architecture, however, and demonstrated new ways of blurring the classical kernel-hardware interface which may be useful in secure systems.

Security-focused designs, conversely, tended to have massive overhead. The TTNoC from [52] requires an entire CPU core for each security domain, rendering

fine-grained compartmentalization impossible. ZKOS uses half of the system's RAM just to keep track of ownership tags and the authors expect this overhead to grow even more once they implement hardware garbage collection.

The work described in this thesis combines these two orthogonal approaches into a single unified whole, attempting to provide a reasonable level of performance with high security.

2.5 Network-on-Chip as a Design Paradigm

One of the simplest ways to enforce isolation between different components of a computer system is to actually build a distributed system in the form of a multiprocessor system-on-chip (MP-SoC) containing one CPU core for each security domain, as proposed in [45]. While this requires much more silicon area than a single time-slicing CPU, by the simple expedient of placing bus wires between cores where communication is allowed and not connecting other paths, it is possible to absolutely prevent any unintended leakage of information. By having a trusted "firewall" module on the link between two cores, it is even possible to limit the types of information which may be exchanged, for example to disable write commands to core X from core Y but allow them from Z. [3]

Unfortunately, for systems with a nontrivial number of security domains it is infeasible to physically provide all of the $O(n^2)$ point-to-point links from every domain to every other domain it may need to communicate with. Instead, some sort of shared communications medium is necessary. Since a separation kernel is conceptually a distributed system, and most large-scale distributed systems are connected via computer networks, the most obvious form of interconnect is a packet-switched network-on-chip (NoC).

The basic network-on-chip paradigm was first introduced in [56]. The paper suggests several possible trends for future NoCs, including the growing use of globally asynchronous / locally synchronous (GALS) systems implemented as source-

[3] The astute reader will note that the "absolute" isolation mentioned above is not truly absolute due to attacks like differential power analysis between cores being theoretically possible. These attacks would require tampering with the device during manufacture and are beyond the scope of this paper.

synchronous NoC interfaces. (On FPGAs, which tend to run at lower clock rates and have well-characterized pre-built clock trees, this is less of an issue; in an ASIC pushing the limits of process technology and lacking a regular structure reduction of clock skew is a much more complex problem). The paper also suggests the possibility that future deep-submicron NoCs may have high enough bit error rates (BERs) that including checksums and retransmit capabilities, as in LAN/WAN links, may be necessary. The paper does not discuss the security implications of the architecture at all.

[57] suggests that implementing a NoC in an FPGA is a waste of programmable logic resources, and that much more efficient design could be realized if the FPGA included a NoC as one of the hard IP blocks provided in the fabric. The paper also proposes an emulated FPGA model for researching NoC designs too big to fit in current hardware. Security is not mentioned in the paper at all.

In the same line of reasoning, [58] observes that due to the low logic capacity of FPGAs compared to ASICs, contemporary architectures tend to use very simple buses to avoid wasting a large fraction of the design's area on interconnect. The paper further suggests that simplified NoC implementations could get adequate performance for some applications while using a fraction of the area of a more complex router. While the complex design does perform better in some applications, the massive (4x-5x) increase in area does not justify the gains in many cases. A hard-wired NoC would avoid this issue.

Although they are not single-chip systems, supercomputers are also distributed computing platforms containing many coupled processors working toward a common goal with a very high bandwidth interconnect. IBM's Blue Gene/L supercomputer, as detailed in [59], deviates from the typical model of many cores connected by one shared bus/network and instead has multiple networks (five) specialized for specific tasks: the collective network (for global operations such as the `MPI_Reduce` or `MPI_Allreduce` calls in MPI, the Message Passing Interface), the global barrier network (for system-wide barrier synchronization), a 3D torus network for general-purpose message passing, gigabit Ethernet for interfacing to the file servers, and a system debug/setup network consisting of 100Mbit Ethernet and JTAG. The Blue

Gene also includes hardware support for multicast. A Blue Gene system can be divided into several independent partitions which have no access to one another. This provides the ability to run several different programs simultaneously on different parts of the system (each running essentially on "bare metal" with little to no OS overhead) and have strong isolation between them.

[60] addresses security, specifically memory accesses, in NoCs. The paper proposes having a firewall module (Data Protection Unit, or DPU) integrated into each NoC interface, blocking bad requests either at the source immediately as they are sent, at the target just before processing, or both. The firewalls are based on a ternary content-addressable memory (TCAM) mapping message header information to memory page access rights. Detailed analyses of area and power overhead for 130nm CMOS (ST HCMOS9GPHS) are presented. It appears that the chip was laid out for testing but not actually fabricated.

[61] is a commercial SoC interconnect fabric based on ARM's AMBA, with some NoC capabilities. The system includes firewalling to restrict access to main memory from unprivileged code, however the system appears to be targeting monolithic kernels and lacks the granularity required for separation kernel designs.

2.6 Formal Verification

Due to the massive setup costs involved, and near-impossibility of field updates, design of an ASIC-based hardware RTOS must be done very carefully. Very few of the papers cited in this chapter made any references to how the designs were tested, validated, or proven correct.

If all logic within the TCB is formally validated against a suitable security model then it is possible to say with absolute certainty that no attacks within that model can compromise the security of the system. (In practice, as mentioned earlier in this chapter, attacks which are outside the scope of the security model sometimes present themselves.)

If possible, applications should be validated as well. Unfortunately this is typically too labor-intensive for all but the simplest systems, but running under a provably correct operating system will still allow strong guarantees to be made as

to the worst-case failure mode. It may still be possible, for example, for an attacker to corrupt data in the back-end database if a web application is exploited, but not for him to corrupt backups or avoid leaving traces in a log file.

In [62], Intel proposes use of a formal validation methodology to prove correctness of critical components of a complex system, in this case the Pentium 4 processor. Conventional validation methods are used to rapidly catch the majority of bugs; once all known bugs have been caught a formal model is constructed and validated to prove complete correctness of the system. Intel used the described methodology to ensure 100% correctness of the ALU portion of the datapath, catching 18 bugs in the process which they believed would have escaped into production silicon had the design not been formally validated. There is no discussion of whether the MMU, privilege ring bits, and other similar logic were validated as well or not.

The very fact that malware exists on current generation x86 platforms clearly demonstrates that "formally validated" does not mean "secure" - Intel's validation efforts focused on ALU correctness rather than security. Since x86 CPUs are designed to have most OS security policy implemented in software, and merely provide isolation between user and kernel memory, the hardware is unable to prevent a privileged process from tampering with the kernel since it has no way to tell that the access is not authorized.

SeL4 [63] implements the first known formally validated operating system kernel, (a port of the L4 microkernel). The group assumes the correctness of the C compiler, approximately 600 lines of assembly language used for context switching and cache setup, and the hardware; everything else is proven correct. The kernel was initially developed as an abstract specification and modeled in a subset of Haskell, which was then used as input to theorem proving software. Once the Haskell model was proven correct, it was ported to optimized C and a proof of equivalency was created. Several design decisions were made in the implementation for ease of formal validation, for example minimizing the use of global variables.

[64] presents an example of the formal validation of a software-based separation kernel for use in security-critical military systems. The kernel consists of "over 3000 lines of C and assembly code" (the CPU architecture is not mentioned). The

methodology is fairly typical for formal validation - formulation of a state-machine based model of the system, creating a formal specification of the properties they wish to prove, automatic proof that the model satisfies said properties, then implementation of the system and proof that it is logically equivalent to the verified model.

It should be noted that partial validation of a system may lull one into a false sense of security. For example if the kernel of a software-based system were to be validated against a security model that assumes the hardware is correct, then a user-to-kernel privilege escalation bug discovered in the hardware, the entire system could be compromised. (In addition, the "600 lines of assembly" which the authors of SeL4 assumed to be correct could potentially be vulnerable to attacks such as the Intel sysret bug mentioned above). We believe that a truly secure platform can only be developed if the hardware and software components critical to security are developed in tandem and validated as a single entity.

2.7 Multithreaded CPU Architectures

2.7.1 Overview

The previous sections set out two apparently conflicting goals: Achieving a secure system requires isolation between security domains, however allocating a separate physical CPU to each security domain would make the SoC prohibitively large, power-hungry, and expensive. What if there were a way to get the best of both worlds?

Efficiently running multithreaded workloads on a single CPU has been the subject of great effort by computer architects. One of the more radical solutions is the *barrel processor*[4], which has traditionally been relegated to niche applications because of its poor single-threaded performance. The idea of a barrel processor is simple: maintain n thread contexts resident in the register file simultaneously, by using register windows or similar means, and context-switch between them every clock cycle.

[4]The name presumably comes from the image of a rolling barrel, where each plank represents a thread and each angular position represents a pipeline stage.

Generally, n is at least equal to the pipeline depth p of the processor. This allows the pipeline to be much simpler since it is known *a priori* that no data hazards can occur: If instruction 2 of thread A (denoted A2 in Fig. 2.1) depends on instruction 1, there is no hazard because $n-1$ other instructions from other threads execute in between them. By the time A2 gets to the register fetch stage, the results of A1 will have been committed to the register file. As a consequence most stall/forwarding/interlock logic in the pipeline can be eliminated, the critical path shortened, and the maximum operating frequency increased. Very long pipelines with little or no forwarding are the norm in barrel processors - the author considered 16 or even 32 pipeline stages for SARATOGA (Sec. 3.5.3) before deciding on 8 to simplify the design.

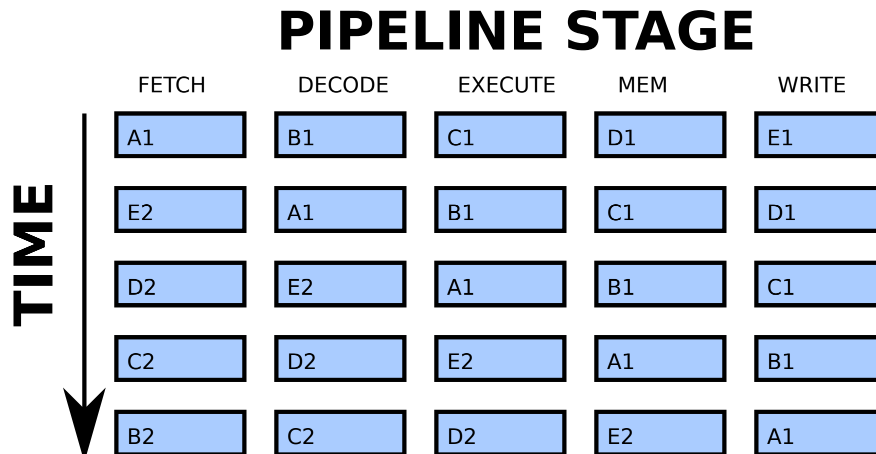

Figure 2.1: Scheduling of 5 threads in a barrel processor with a 5-stage pipeline

The major problem with a barrel processor is that, if the pipeline interlock / forwarding logic is removed, it is illegal for more than one instruction from a given thread to be in the pipeline at a time. This means that if less than p threads are active at any point in time, wait states must be added. In the example shown in Fig. 2.2 the CPU is running at a mere 60% efficiency.

This determinism can be seen as an advantage as well, however, since (ignoring the possibility of contention on multi-cycle I/O operations) each thread is guaranteed a minimum of the $\frac{1}{max(p,n)}$ of the total CPU cycles. This is highly beneficial for hard-realtime applications.

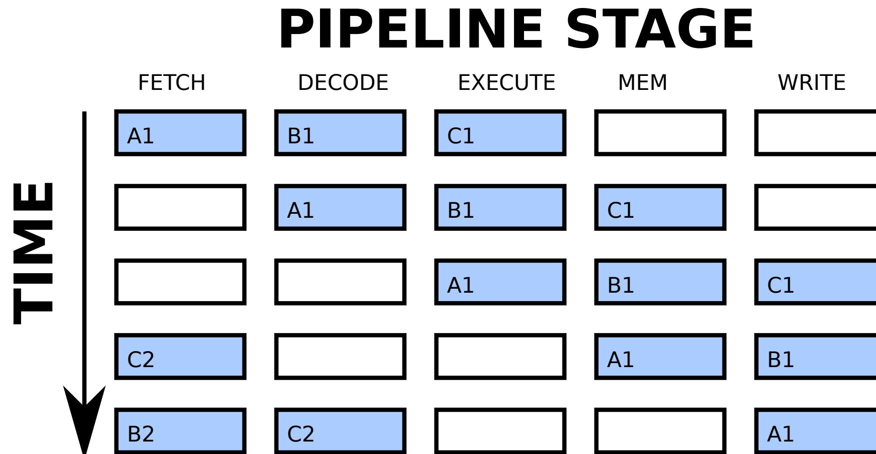

Figure 2.2: Scheduling of 3 threads in a barrel processor with a 5-stage pipeline

Graphics processing units (GPUs) have many features in common with barrel processors, the most notable being the massive number of registers (NVIDIA's Kepler microarchitecture, described in [65], has 65,536 registers per multiprocessor). A common optimization in GPU architecture is to hide cache miss latency by removing a thread from the run queue in the event of a cache miss and replacing it with another one, thus ensuring that the processor continues to execute instructions at full speed unless every single thread is blocking on memory. In most cases this means that branch prediction and out-of-order execution are unnecessary, permitting the processor to be even simpler.

Barrel processors are also very well-suited to applications with hard real-time constraints because the threads are almost completely independent. Aside from cache interference (which can be, optionally, eliminated by allocating a separate region of the cache to each thread context) and contention for peripherals, there is no way for a thread's performance to be negatively affected by another.

2.7.2 Prior Implementations

[66] describes a barrel-esque processor which has a large number of thread contexts sharing a relatively short (five stage) pipeline, ensuring that there is almost always a thread ready to run. The system is analyzed as part of a non-ccNUMA shared-memory system with local memory for each core and explicit message passing

between cores.

The XMOS XCore architecture [67] is a hardware multithreaded multicore processor architecture developed for high-performance embedded systems. Each core has eight hardware threads and a 4-stage pipeline; threads are scheduled in a round-robin barrel fashion such that if ≤ 4 threads are active each gets 1/4 of the CPU cycles and if more than 4 are active then each of the n threads gets $1/n$ of the CPU. The cores include basic secure-boot capability - OTP ROM not readable from outside the core and a fuse to disable JTAG. The intention is for the OTP ROM to contain an encryption key and a boot loader which loads an image from external Flash and decrypts it. Devices are available with 1 to 4 cores as of this writing. Full hardware operating system support is not provided but timers, locks and synchronizers are included.

The IBM Cell Broadband Engine [68] was originally developed for the PlayStation 3 game console and later used in many HPC systems such as the Roadrunner supercomputer. The Cell, fabricated on a 90nm IBM process, is a hybrid architecture containing a single 64-bit dual-threaded dual-issue in-order PowerPC core (Power Processing Element or PPE) plus eight SIMD cores (Synergistic Processing Elements or SPEs). The PPE has a barrel-esque design: instructions from two threads are issued alternately, which permits higher pipeline latency to be tolerated without an excessive branch misprediction penalty.

CHAPTER 3
System Architecture

3.1 Design Philosophy

The Antikernel architecture is intended to be more, yet less, than simply a "kernel in hardware". By breaking up functionality and decentralizing as much as possible we aimed to create a platform that allows applications to pick and choose the OS features they wish to use, thus reducing their attack surface dramatically compared to a conventional OS (and potentially experiencing significant performance gains, as in an exokernel). If the application in question does not store any data in the filesystem, for example, a vulnerability in the filesystem cannot be used to compromise it. It may also access raw disk/flash sectors directly with less overhead than if it were to go through a filesystem, while still being prevented from reading or tampering with data belonging to other code.

Antikernel does not have any APIs or system calls whatsoever; all OS functionality is accessed through message passing. To create a process, the user sends a message to the CPU core he wishes to run it on. To allocate memory, he sends a message to the RAM controller. Each of these nodes is self-contained and manages its own state internally (although nodes are free to, and many will, request services from other nodes).

There is no "all-powerful" software; all functionality normally implemented by a kernel is handled by unprivileged software or hardware. Even the hardware is limited in capability, for example the flash controller has no access to RAM used by the CPU. By formally verifying the isolation and interprocess communication, we can achieve a level of security which exceeds even that of a conventional separation kernel: even arbitrary code execution on a CPU grants no privileges beyond those normally available to userspace software. Escalation to "ring 0" or "kernel mode" is made impossible due to the complete lack of such privileges; unprivileged userspace runs directly on "bare metal".

3.2 Networks

At the highest logical level, an Antikernel-based system consists of a series of nodes (userspace processes or hardware peripherals) organized in a quadtree and connected by a packet-switched NoC with 16-bit addressing. Hardware and software components are indistinguishable to developers and are addressed using the same message passing interface. For the remainder of this thesis, NoC routing addresses are written in IPv6-style hexidecimal CIDR notation. For example the subnet consisting of all possible addresses is denoted 0000/0, 8002/16 is a single host, etc.

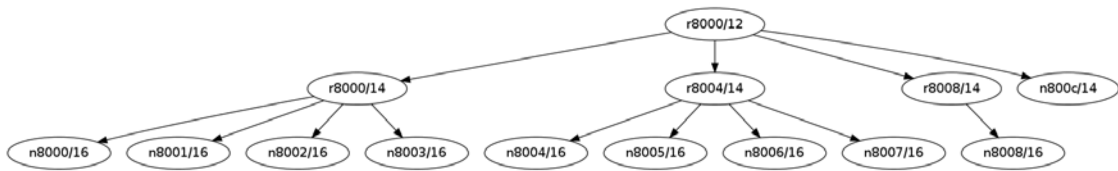

Figure 3.1: Example routing topology (simplified)

Each bottom-level leaf node (names prefixed with "n" in Fig. 3.1) is assigned a /16 subnet (in other words, a single address) and corresponds to a single hardware module. The next level nodes are routers for /14 subnets (names prefixed with "r"), followed by routers for /12 subnets, and so on. Routers are instantiated as needed to cover active subnets only; if there are only four nodes in the system the network will consist of a single top-level router with four children rather than an eight-level tree. Nodes may also be allocated a subnet larger than a /16 if they require multiple addresses: perhaps the node "n800c/14" is a CPU with support for four hardware thread contexts that runs each thread in its own security context and is allocated a /14 sized subnet so that the remainder of the system can distinguish between the threads.

One limitation of a strict quadtree design is that odd-sized subnet prefixes are not supported since each router has four downstream ports which cannot be combined: a node requiring two addresses must be assigned a /14 rather than a /15. While this does lead to some waste of address space, given the scaling trends observed in current and next-generation SoCs it is unlikely that any designs in the near future will have anywhere near 65536 uniquely addressable devices on chip. The architecture can, of course, be scaled to larger address sizes in the longer term

should the address size present a limitation.

"The network" is actually two parallel networks (each 32 bits wide) specialized for different purposes, as shown in Fig. 3.2. The RPC network transports fixed-size datagrams consisting of one header word and three data words, and is optimized for low-latency control-plane traffic. The DMA network transports variable size datagrams consisting of three header words and zero to 512 data words, and is optimized for high-throughput data-plane traffic. Each node uses the same address on both networks to ensure consistency, although individual nodes are free to only use one network and disable their associated port on the other (for example, nodes "n8002/16"and "n8005/16"). Entire routers for one network or the other may be optimized out by the code generator if all of their downstream ports are unused (for example, there is no RPC router for the subnet 8004/14 as all nodes in that subnet are DMA-only).

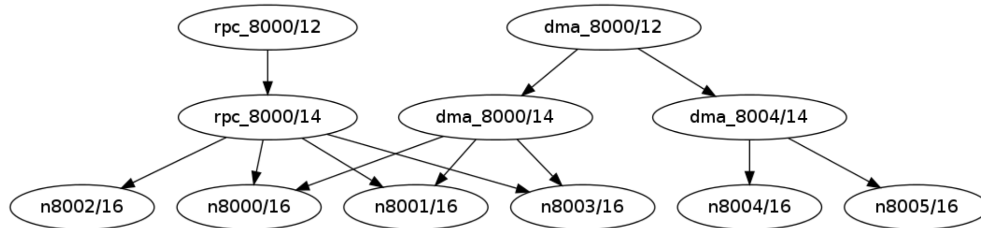

Figure 3.2: Example routing topology showing RPC and DMA

A full link for either network contains two independent unidirectional links, each consisting of a 32-bit data bus and several status flags. Since the prototype is FPGA-based all network links are system-synchronous, however they could fairly easily be converted to source-synchronous for a GALS ASIC.

Packets for both networks begin with a single-word layer-2 routing header containing the 16-bit source and destination node addresses, followed by protocol-specific layer-3 headers. There is no header field to distinguish RPC and DMA traffic; since the networks are physically distinct the protocol can be trivially determined from context.

Each network guarantees strict FIFO ordering, as well as reliable delivery, for any two endpoints. More formally, given three nodes A, B, and C, RPC packets sent from A to B will always arrive, and do so in the order that they were sent. There

is, however, no global temporal ordering implied for these packets with respect to RPC packets sent from A to C, or DMA packets from A to B.

3.2.1 RPC

3.2.1.1 Packet Structure and Semantics

An RPC message (Fig. 3.3) consists of the standard layer-2 routing header followed by an 8-bit call number indicating the operation to be performed ("call"), a 3-bit type field ("op"), and 85 bits of data divided into one 21-bit partial word and two 32-bit words. (The 21-bit field is often used as a 16-bit data halfword and five one-bit status flags, although its interpretation is up to the endpoints.)

Word \ Bit	31–16		15–0
0	Source node		Dest node
1	Call	Op	Data
2	Data		
3	Data		

Figure 3.3: Structure of an RPC packet

Any RPC transaction involves two nodes. The node which initiates the transaction is designated the master; the node responding to this packet is designated the slave. These roles are not fixed and any node may choose to act as a master or slave at any time.

3.2.1.2 Interrupt

The simplest kind of RPC transaction is is an interrupt (Fig. 3.4): a unidirectional notification from master to slave. RPC interrupts are typically sent to inform the slave that some long-running operation (such as a backgrounded DMA write to slow flash memory) has completed at the master, or that an external event took place (such as the arrival of a new Ethernet frame or a button press).

The `call` field of an interrupt packet is set to a value chosen by the master describing the specific type of event; the data fields may or may not be significant depending on the specific master's application-layer protocol. Since the field is 8 bits long any master may generate up to 256 uniquely distinguishable interrupts.

Note that the term "interrupt" was chosen because these messages convey roughly the same information that IRQs do in classical computer architecture. While the slave node may choose to interrupt its processing and act on the incoming message immediately, it may also choose to buffer the incoming message and handle it later. (If the slave runs out of buffer space it must process the message immediately or discard it since there is no way to request retransmission of an interrupt. Link-layer flow control may be used to prevent this problem as long as the slave does not need to receive messages from another node while it is busy.)

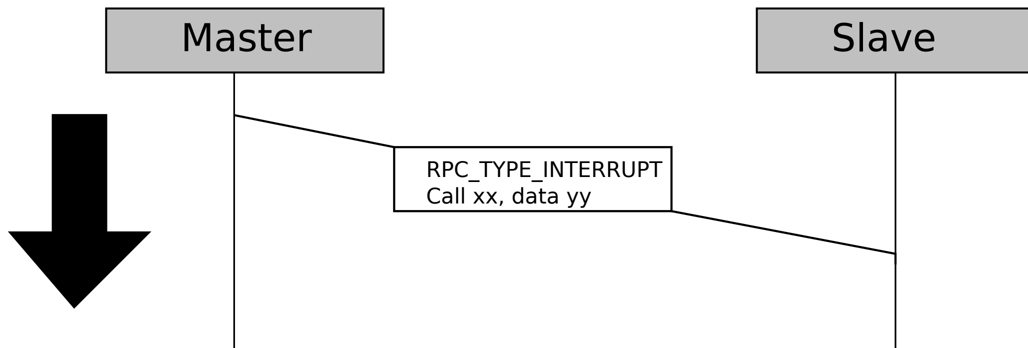

Figure 3.4: Dataflow diagram for RPC interrupt transaction

Devices are encouraged to avoid imposing tight hard-realtime deadlines on interrupt processing latency. This can be done in numerous ways, but one of the simplest is for the node itself to function as a DMA master. This is made quite easy since Antikernel's message system makes no distinction between hardware peripherals and software processes, so OS services are not limited to flowing one way (hardware provides services to software). Instead, all OS services, including the name server (Sec. 3.3.1) and dynamic memory allocation (Sec. 3.4), are exposed to hardware as well. This is discussed in greater detail in subsequent sections.

When Antikernel's Ethernet MAC (a Verilog module) initializes, for example, it queries the name server for the hostname `ram`, then sends a `RAM_ALLOCATE` request to allocate a page of physical memory. This address is stored in an internal register

until a new frame arrives.

When a frame arrives it is buffered in an internal FIFO and the Ethertype is checked against a list of registered layer-3 protocol handlers. If no match is found the packet is of no interest to the system and is dropped. If a match is found, however, the MAC will issue a DMA write to send the packet to RAM. Once the write completes it calls `RAM_CHOWN` to give the protocol handler access to the page, then sends a `ETH_FRAME_READY` interrupt to the handler to inform it that a new packet is ready for processing. The MAC then allocates a new page of physical memory and prepares to repeat the cycle for the next packet.

3.2.1.3 Call

The second major kind of RPC transaction is a function call: a request by the master that the slave take some action, followed by a result from the slave. This result may be either a success/fail return value (Fig. 3.5) indicating that the remote procedure call completed, or a retry request (Fig. 3.6).

The `call` field of a function call packet is set to a slave-dependent value describing one of 256 functions the master wishes the slave to perform. The meaning of the data fields is dependent on the slave's application-layer protocol.

A return packet (including a retry return) must have the same `call` value as the incoming function call request to allow matching of requests to responses. The meaning of the data fields is dependent on the slave's application-layer protocol.

The RPC call protocol is capable, in principle, of out-of-order (OoO) transaction processing (processing multiple incoming commands in the most efficient order, rather than that in which they were received) however no semantics are currently defined for matching reordered call and return packets and none of the nodes in the current prototype implement OoO processing. The simplest option for adding OoO would be to allow reordering of commands between nodes (in other words, two incoming calls from different nodes could be processed in any order) however not for one node (in other words, two incoming calls from the same node must be processed in the order they arrived in).

Retry return values may be sent by the slave at any time in response to an

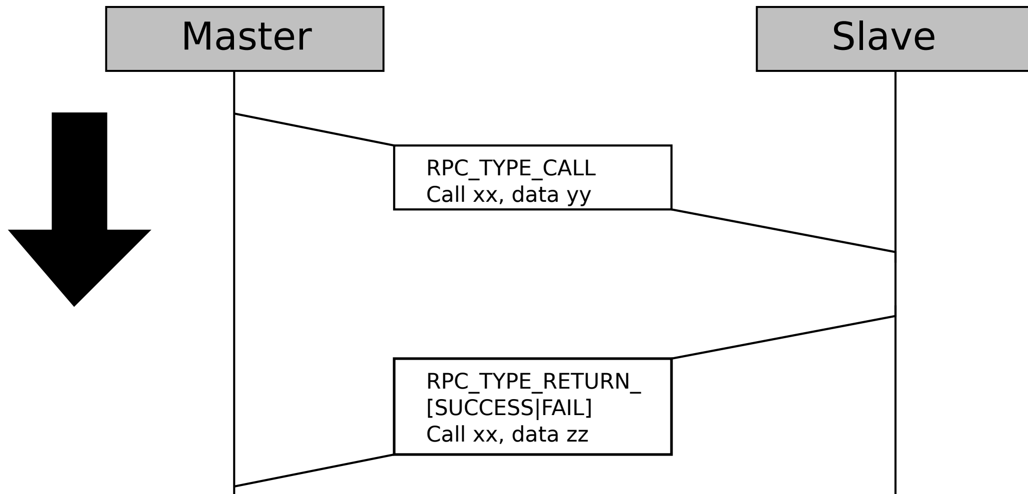

Figure 3.5: Dataflow diagram for RPC call/return transaction

incoming RPC call request, but are typically only sent if the slave is already processing another remote procedure call and has no free buffer space to save the message for future processing. The act of rejecting an RPC call with a retry is required to be idempotent - it does not result in any state change within the slave.

The master may treat a retry as a failure (if it cannot or does not wish to retry) or repeat the function call request at any time in the future. There is no limit on how many times a slave may reject an incoming RPC with a retry request, however masters may choose to impose an arbitrary limit on wait time or number of retries before timing out and declaring failure.

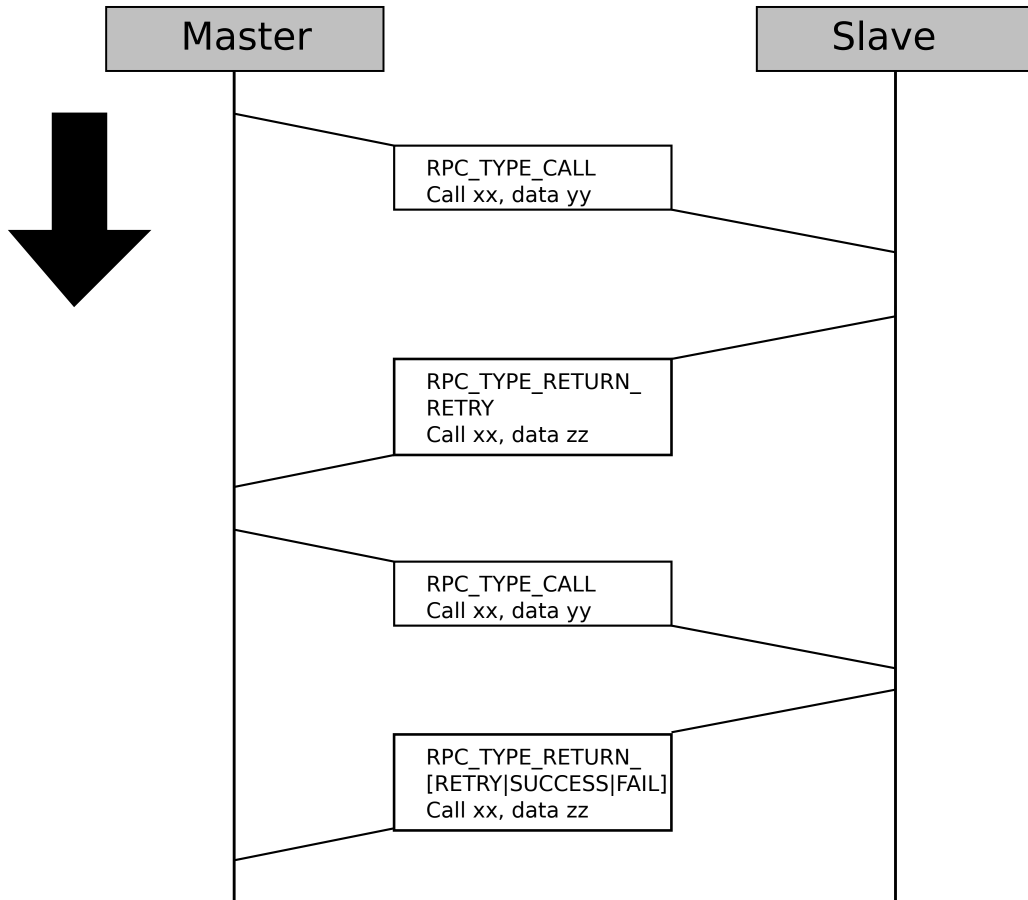

Figure 3.6: Dataflow diagram for RPC call/retry transaction

3.2.1.4 Network Status / Reserved

Two type codes are reserved for network status notifications (sent by a router or firewall to indicate that a packet could not be delivered because the destination port is not present or is blocked by firewalling). No protocol has been defined for these type codes yet and the current code does not implement them.

In addition, one type code is reserved for future expansion.

3.2.1.5 Flow Control

Flow control for the RPC network consists of a one-bit `tx_en` flag and a two-bit `tx_ack` flag. When the packet is sent, `tx_en` is asserted the same cycle as the first data word. The transmitter then blindly sends the remainder of the packet, and blocks until `tx_ack` is set (by the recipient) to a nonzero value, which may be an ACK (packet received successfully, Fig. 3.7) or NAK (recipient is busy, retrans-

mit required). In the event of a NAK, the transmitter must re-send the packet immediately (Fig. 3.8).

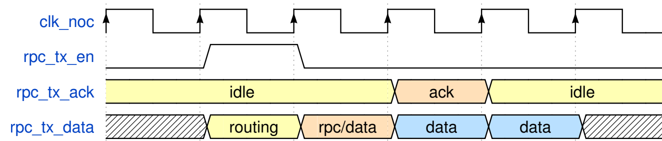

Figure 3.7: Timing diagram for RPC send with successful ACK

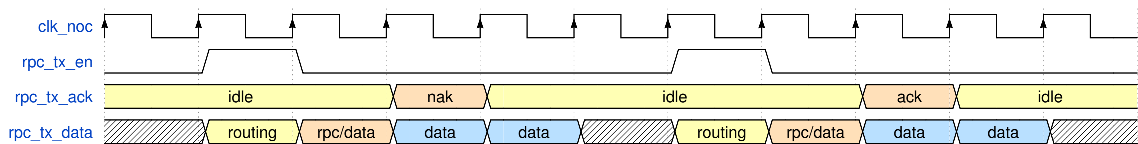

Figure 3.8: Timing diagram for RPC send with one retransmit

The RPC protocol will function over links with arbitrary latency (and thus register stages may be added at any point on a long link to improve setup times), however a round-trip delay of more than one packet time will reduce throughput since the transmitter must block until an ACK arrives before it can send the next packet (Fig. 3.9).

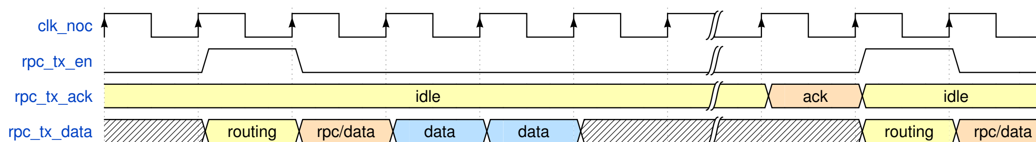

Figure 3.9: Timing diagram for RPC send with delayed ACKs

The RPC router is a full crossbar which allows any of the five ports to send to any other port, with multiple messages in flight simultaneously. Each exit queue maintains a round-robin counter which increments (mod 5) each time a packet is sent. In the event that two ports wish to send to the same destination simultaneously, the port identified by the round-robin counter is given max priority; otherwise the lowest numbered source port wishing to send is permitted to do so. This ensures baseline quality of service (each port is guaranteed 1/5 of the available bandwidth) while still permitting bursting (a port can use up to 100% of available bandwidth if all others are idle).

There are two implementations of the RPC network transceiver at this time. RPCv2Transceiver has an unpacked parallel interface intended to be used directly by state machine logic, and RPCv2RouterTransceiver has a SRAM interface which is intended to be used by routers or nodes requiring low-latency streaming processing of messages.

3.2.2 DMA

3.2.2.1 Packet Structure and Semantics

A DMA message (Fig. 3.10) consists of the standard layer-2 routing header followed by an opcode (read request, read data, or write data), data length, and a 32-bit address indicating the target of the DMA operation. This is then followed by message content for "read data" and "write data" packets. The "read request" packet has no data field. (One type code is left undefined for future expansion.)

Word \ Bit	3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0		
0	Source node	Dest node	
1	Op	Padding	Len
2	Address		
...	Data		
N+2			

Figure 3.10: Structure of a DMA packet

The maximum DMA payload size is 512 32-bit words, or one 2KB memory page. This size was chosen because it is a common size for FPGA block RAMs and is the smallest power-of-two size sufficient for storing an entire Ethernet frame.

The network-level packet format, routers, and transceivers do not enforce any word/page level alignment restrictions, however all nodes implemented to date require that operations be aligned to a 32-bit word boundary. Many nodes also refuse

to process multi-word read or write bursts that cross a 2KB page boundary, since access controls are done on a per-page basis and this would require multiple access checks and possible burst truncation (if access was granted to part of the range, but not the rest).

All write operations must be to an integral number of 32-bit words; byte masking is not supported. If byte-level write granularity is required this is typically implemented with a read-modify-write operation.

Since the DMA address field is 32 bits in size, a maximum of 4 GB may be addressed within a single routing address. Nodes requiring $>$ 4 GB of address space may be assigned to larger subnets; for example a SD card controller might use a /14 subnet (4 routing addresses) to permit use of cards up to 16 GB (4x 4 GB) in size.

When sending pointers between nodes, unless the node to which the pointer refers to can be inferred by context, it is necessary to send both the 16-bit routing address and the 32-bit pointer. The resulting 48-bit physical address uniquely identifies a single byte of data within the system.

Physical addresses are written as the routing address (either hostname or numeric forms are acceptable) followed by a colon and the full 32-bit pointer, such as ram:cdcd1234 or 8003:00000800.[5]

3.2.2.2 Memory Read

A memory read transaction consists of one packet from master to slave, with the type field set to "read request", the address set to the location of the data being requested, and the length set to the number of words being read. Note that the data field is *empty* in this one case, even though the length field is *nonzero*. This causes a slight increase in complexity for the transceiver cores and routers since the type must be checked before the actual packet size is known.

If the request is permitted by the slave's security policy, it responds with a packet of type "read data". Address and length are set as in the requested packet, and the data field contains the data being returned.

If the request is not permitted by policy, the slave returns an error code to

[5]While this notation appears similar at first glance to x86 segment:offset addressing, the routing and pointer fields in an Antikernel DMA address are fully independent and do not overlap.

the master (see section 3.2.2.4) so that it knows that no response is forthcoming. The master may then trigger an exception handler, reset, terminate the offending process, or take any other action it deems appropriate.

3.2.2.3 Memory Write

A memory write transaction consists of one packet from master to slave, with the type field set to “write data”, the address set to the location of the data being written, and the length set to the number of words being written.

If the request is permitted by the slave’s security policy, it responds with a “write complete” notification (see section 3.2.2.4). This allows the master to implement memory-fencing semantics for interprocess communication: to avoid potential race conditions, one cannot send the pointer to another node (or change access controls on it) until the in-flight write has completed.

If the request is not permitted, the slave returns an error code to the master (see section 3.2.2.4) so that it knows the underlying physical memory has not been modified. The master may then take any internal action it deems appropriate.

3.2.2.4 Command Status Notification

During initial architectural design it was decided that status messages (write done acknowledgements and access-denied errors) were control-plane traffic, and thus should travel on the RPC network as interrupt packets. By convention the code for “write done” is 0x80 and “access denied” is 0x81.

This was, in retrospect, a poor decision: In a softcore CPU the RPC network is typically under software control while the DMA network is controlled by the L1 cache. Requiring some RPCs to be handled by the cache increases the complexity of this logic substantially. Furthermore, any DMA-capable device which implements any security whatsoever (or allows writes) must also have a RPC connection, which increases the amount of logic used for interconnect.

In the future we intend to refactor the existing code to use the fourth (currently reserved) DMA packet type for memory status notifications. The type of status will be conveyed either in the reserved space in the header between the opcode and address, or in a single data word.

3.2.2.5 Flow control

In order to send a packet (Fig. 3.11), the transmitter asserts `tx_en` and places the layer-2 routing header on `tx_data`. It then waits for `tx_ack` to go high before sending the remainder of the packet headers and (if applicable) the body. If the receiver is not ready to accept new packets, it may delay assertion of `tx_ack` indefinitely (Fig. 3.12).

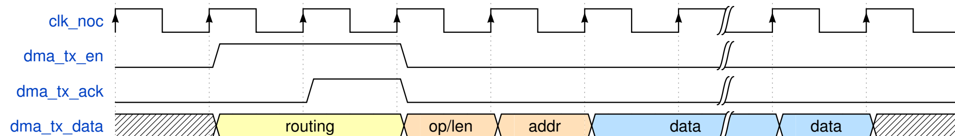

Figure 3.11: Timing diagram for DMA send

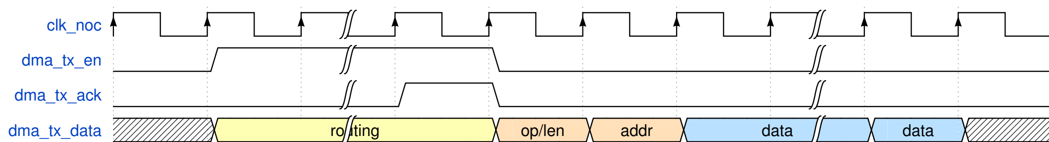

Figure 3.12: Timing diagram for DMA send with delayed ACK

The current DMA flow control scheme expects a fixed single-cycle latency between routers. We plan to extend this in the future in order to support variable latency for long-range cross-chip links, as was done for RPC.

As with RPC, there are two versions of the DMA router. `DMATransceiver` has a SRAM interface for data and an unpacked register interface for headers, and is intended for use by typical nodes. `DMARouterTransceiver` is intended for routers or streaming applications, and has one SRAM interface for data and another for headers.

The initial DMA router was based on a single buffer and a 5:1 mux to save area, however the limitations of this design quickly became apparent. The current DMA router uses the same arbiter and crossbar modules as the RPC router, although the buffers are somewhat larger (one 2KB block RAM plus one 32 bit x 3 word distributed RAM per port).

3.3 System Information

Any nontrivial operating system needs a way for nodes to query various properties of the system, enumerate services, etc. We provide this capability with two components: the system information server and the name server.

3.3.1 Name Server

The name server is always located at RPC address 8000/16 and functions much like a DNS server, providing mappings between hostnames and NoC addresses. [6] Nodes can supply either a hostname to be converted to an address, or an address to be converted to a hostname. It is also possible to enumerate the entire name table to discover all nodes in the system.

Both hardware and software nodes may query the name table to determine the addresses of other nodes. Hardware nodes may optionally "bake" the address of another hardware node in at synthesis time to reduce boot time, however if the node being queried is software-based (or may change addresses, perhaps due to cosimulation) the address lookups must be dynamic.

The initial name table (a list of hardware nodes and their addresses) is loaded into ROM at synthesis time. As software-based applications start up, they may optionally register their address and hostname with the name server. In order to prevent a malicious or compromised application from registering a name belonging to a legitimate service, the entire RPC request, including the source address and requested hostname, is signed.

The current prototype uses a pre-shared key cryptosystem (HMAC-SHA256) to reduce FPGA gate count and shorten compile times for tests. The system developer generates a name registration request, signs it with the secret key, and then places the signature into the source of the application before compiling it. This does of course mean that reverse engineering of one device would reveal the key used for all other devices in the family, so larger scale deployments would need to use per-device keys or public key cryptography to maintain security.

[6] Host names are ASCII strings, up to 8 characters long, which uniquely identify a given node in the NoC. Typically this is the same as the Verilog instance name of the node in the top-level HDL file.

One issue with the current protocol is that the network address (process ID) the application will run as must be known at the time that the request is signed. This requires that services wishing to register themselves with the name server must start in a known, defined order since SARATOGA gives out process IDs on a first come first served basis. We are exploring possible solutions to this issue but have not yet come up with a satisfactory one.

The name server provides several different read-only queries which require no authentication: `NAMESERVER_FQUERY` (forward query, given a hostname return the address if found), `NAMESERVER_RQUERY` (reverse query, given an address return the hostname if found), and `NAMESERVER_LIST` (name table listing, iterate over the entire name table and return the Nth entry).

Name registration requires multiple RPC calls so a mutexing scheme is used. The node first issues a `NAMESERVER_LOCK` call, which returns successfully if the mutex was granted. If the mutex is already held the application is asked to retry the lock request. The mutex includes a timeout to prevent deadlocking: if 2^{20} clock cycles elapse without any calls to the critical section, the mutex automatically unlocks.

Once a lock is granted, the node sends four `NAMESERVER_HMAC` calls, each one specifying an index from 0 to 3 and a 64-bit chunk of the HMAC signature, followed by a `NAMESERVER_REGISTER` call, which actually registers the name with the server and releases the lock. This function will fail if the HMAC signature is invalid, or if the name table is full. Both `NAMESERVER_HMAC` and `NAMESERVER_REGISTER` fail if the requesting node does not own the mutex.

3.3.2 System Info Server

The system information server always has the hostname "sysinfo" and currently provides three RPC methods on all Antikernel systems: get a 64-bit serial number for the board or FPGA die, get the period of the core NoC clock, and get the number of NoC clock cycles required to reach a given frequency. A simple 32-bit iterative (one bit per clock) divider core is used to aid in these computations.

In addition, several more functions are currently supported only on Xilinx 7 series FPGAs. These allow querying the current value of several system health

paraneters (die temperature, core voltage, auxiliary voltage, RAM voltage) as well as the normal ranges of these parameters.[7]

3.4 Memory Management

One of the most critical services an operating system must provide is allowing applications to allocate, free, and manipulate RAM. In the minimalistic environment of an exokernel there is no need for an OS to provide sub-page allocation granularity, so we require nodes to allocate full pages of memory and manage sub-page regions (such as for C's `malloc()` function) in a userspace heap. If a block larger than a page is required, the node must allocate multiple single pages and map them sequentially to its internal address space.

In order to simplify operating system implementation and security proofs, as well as reduce the potential for race conditions or unintended data corruption in application code, Antikernel's memory management enforces a "one page, one owner" model. Shared memory is intentionally not supported, however data may be transferred from one node to another in a zero-copy fashion by changing ownership of the page(s) containing the data to the new user. It is of course important for the sending node to ensure that its write caches, if any, are flushed and all writes have committed in order to ensure coherency. (In a software-based node this would typically be handled by the C runtime library's `munmap(2)` function.)

The Antikernel memory management API is extremely simple, in keeping with the exokernel design philosophy. It consists of four RPC calls for manipulating pages ("get free page count", "allocate page", "free page", and "change ownership of page") as well as DMA reads and writes. A "write complete" RPC interrupt is provided to allow nodes to implement memory fencing semantics before `chown(2)`ing a page.

The data structures required to implement this API are extremely simple, and thus easy to formally verify: a FIFO queue of free pages and an array mapping page

[7] In order to prevent these from being used to implement side channel attacks, the update rate of the sensors will be deliberately capped at a few Hz. Further research is necessary to choose an update rate which provides usefully current data to fan controllers etc, while not allowing one node to perform side channel attacks on another using the sensor readings.

IDs to owner IDs. When the memory subsystem initializes, the FIFO is filled with the IDs of all pages not used for the internal metadata and the ownership array records all pages as owned by the memory manager.

Requesting the free page count simply returns the size of the free list FIFO.

Allocating a page fails if the free list is empty. If not, the first page address on the FIFO is popped and returned to the caller; the ownership records are also updated to record the caller as the new owner of the page.

Freeing a page is essentially the allocation procedure run in reverse. After checking that the caller is the owner of the page, it is zeroized to prevent any data leakage between nodes, then pushed onto the free list and the ownership records updated to record the memory manager as the new owner of the page.

Changing page ownership does not touch the free list at all; after verifying that the caller is the owner of the page the ownership records are simply updated with the new owner.

DMA reads and writes perform ownership checks and, if successful, return or update the contents of the requested range. The current memory controller API requires that all DMA transactions be aligned to 128-bit (4 word) boundaries, be a multiple of 4 words in size, and not cross 2KB page boundaries.

The current prototype codebase contains two compatible implementations of the Antikernel memory management API: `BlockRamAllocator` (backed by on-die block RAM, parameterizable size) and `NetworkedDDR2Controller` (backed by DDR2, currently fixed at 128MB capacity with a 16-bit bus). A parameterizable-depth allocator backed by DDR3 is planned, but has not yet been implemented.

A standard Antikernel system must instantiate one of these with the hostname "ram". This memory is used as the default storage for large temporary data unless a different storage node is explicitly requested at the time of allocation. Since Antikernel's architecture is inherently NUMA, multiple memory controllers may be instantiated without causing problems as long as full 48-bit pointers are used to avoid ambiguity.

3.5 Processors and Threading

3.5.1 Introduction

We created two softcore CPUs during the development of Antikernel. Both are in-order pipelined processors which use variants of the MIPS-1 instruction set, however they use different approaches for interfacing with the NoC and thus binaries for one cannot run on the other (although C programs which do not use any advanced processor-specific features can be compiled against the libc for one or the other with little to no change).

GRAFTON, the first to be designed (Sec. 3.5.2) is optimized for minimal gate count at the expense of performance, can only run a single process (there is no support for context switching), and is missing support for many important features like code signing. It can, however, fit in a much smaller space: we are currently using it on a Xilinx XC6SLX25 (15K LUT6s) to run the LAN interface on a managed power distribution unit in our lab.

SARATOGA, the more recent processor (Sec. 3.5.3) is optimized for performance above size, and has support for an arbitrary power-of-two number of hardware thread contexts. It is substantially larger than GRAFTON, but has a deeper pipeline and has much higher peak performance.

3.5.2 GRAFTON

3.5.2.1 Introduction

The first prototype of Antikernel used a custom-developed 32-bit CPU named GRAFTON, after a nearby town. GRAFTON is a 5-stage-pipelined, single-issue in-order processor heavily based on the MIPS-1 ISA but eliminating interrupts and the standard coprocessors. While GRAFTON is intended to be compatible with C compilers designed to target the MIPS-1 instruction set given appropriate flags and linker scripts, no portability of binary software between GRAFTON and an actual MIPS processor is implied or intended.

The core uses only one clock domain so the CPU clock and NoC clock must be the same frequency. This limits the maximum operating frequency of the entire SoC as the GRAFTON core was designed with size, rather than performance, as

the primary goal.

While limited cooperative multitasking is theoretically possible, a GRAFTON core can only use a single NoC address so no security compartmentalization is possible within one core. Although designs requiring multiple threads can be split among multiple GRAFTON cores, this is inefficient and the SARATOGA core (Sec. 3.5.3) will give better performance in this use case.

The typical programming model for GRAFTON is event driven: the processor sleeps until an RPC interrupt arrives, handles it, then goes back to sleep. During event handling the processor can send and receive RPCs, perform memory accesses, etc. If an RPC interrupt arrives while the application is waiting for an RPC function call to return, the application may choose to queue it in RAM for further processing or handle it immediately.

The MIPS-1 instruction set allows for up to four coprocessors with 32 32-bit registers each. The `mtcx` (Move To Coprocessor X) and `mfcx` (Move From Coprocessor X) allow a 32-bit word to be moved between a coprocessor register and a general purpose register. The MIPS ISA defines purposes for these registers, however the `gcc` compiler does not use them so it is possible to redefine these registers without breaking compiler compatibility. GRAFTON uses coprocessor 0 to interface to the MMU as well as the RPC network, trace port, and software divider (see Fig. 3.13).

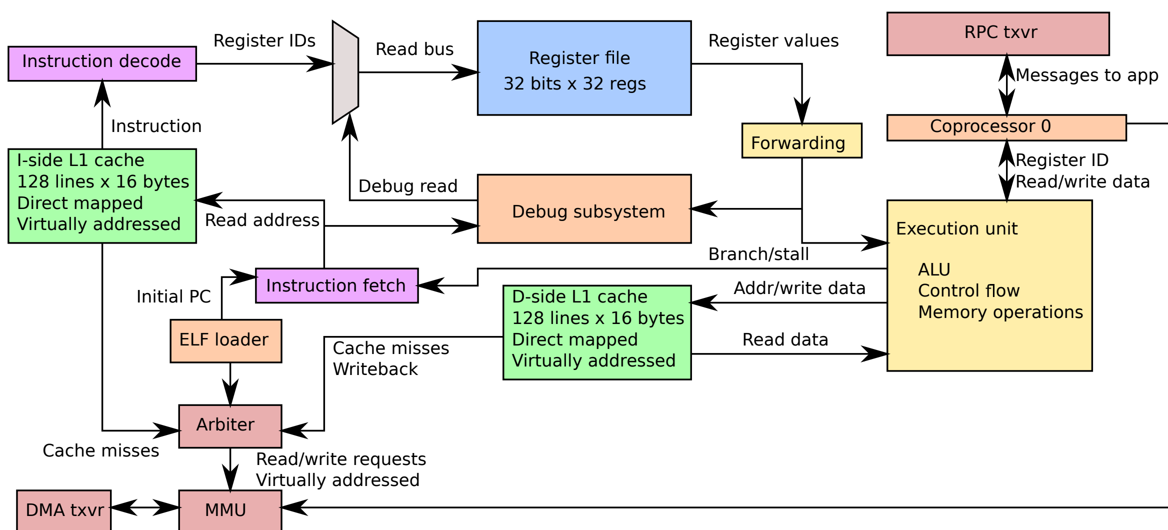

Figure 3.13: Block diagram of GRAFTON

3.5.2.2 ALU

GRAFTON implements all of the standard MIPS-1 integer ALU instructions, including multiplication using the FPGA's DSP slices, however there is no support for floating point since design of an efficient IEEE754 FPU was deemed out of scope for a thesis focusing on security.

To reduce the area consumed by the core, division is implemented by a software trap handler rather than hardware. Before the trap handlers are set up (Listing 3.1) any attempt to use the `div` or `divu` instructions will result in a null pointer dereference and segmentation fault.

Once the trap handlers are set up, executing a divide will result in a jump to the appropriate (signed or unsigned) division handler, with `k0` set to the address of the instruction after the divide. The dividend and divisor are stored in the `hi` and `lo` registers respectively and are accessed by the trap handler with the `mflo` and `mfhi` instructions. After the division completes the handler stores the remainder and quotient back into `hi` and `lo` using `mthi` and `mtlo` and jumps to `k0`.

The standard C runtime library for GRAFTON includes a simple iterative division routine based on the shift-and-subtract algorithm. This function uses the stack for saving registers, so it cannot be used before the stack has been initialized.

Listing 3.1: Initializing division traps

```
1  la      t0, SignedSoftDivisionHandler
2  mtc0    t0, div_handler
3  la      t0, UnsignedSoftDivisionHandler
4  mtc0    t0, divu_handler
```

3.5.2.3 Memory Bus and L1 Cache

GRAFTON has a split memory bus with separate instruction and data buses leading to separate virtually-addressed, direct-mapped caches organized as 128 lines of four 32-bit words each. This gives a total size of 2KB instruction and 2KB data cache plus tag bits.

Both caches have single cycle latency for cache hits and do not use the output register on the block RAMs, which limits the maximum operating frequency of the

core to 40-75 MHz (in a densely packed -2 speed, and loosely packed -3 speed, Spartan-6 FPGA respectively).

The cache banks have 32-bit read/write buses to an arbiter/controller which uses the MMU (see Section 3.5.2.4) to translate the 32-bit virtual address being fetched/flushed into a 48-bit virtual address and sends it out to the DMA network. A fixed priority scheme is used for arbitration: D-side writes have max priority, followed by D-side reads, and finally I-side reads. All read/write transactions consist of a single 4-word cache line.

The current cache does not have an explicit flush feature. Instead, flushing the D-side cache is accomplished by a software-based flush routine which sequentially fetches the first 2KB of the boot sector of flash into a register and discards it. Since the cache is 2KB and direct mapped, this guarantees that every cache entry will be evicted. We plan to eventually add a dedicated flush instruction which allows a selected range of the cache to be flushed in a more efficient fashion.

3.5.2.4 MMU

GRAFTON has a very simple MMU located between the L1 cache and the DMA network interface, which provides programmable translation for 512 2KB pages (1MB) of virtual address space starting at virtual address 0x40000000. All memory accesses outside this range, or to pages for which no physical address is mapped, result in a segmentation fault.

Since the page table is so small it is held entirely in block RAM on the FPGA - in other words the page table and the TLB are one and the same. Each entry contains three bits specifying read/write/execute access, the 16-bit NoC address of the mapped device, and the 21-bit physical address of the device-side page being mapped (the low 11 bits must be zero since mappings must be page aligned).

The mappings for a single page may be updated by executing the `syscall` instruction with `a0` set to `SYS_MMAP`. The mapping to configure is set by the `mmu_page_id`, `mmu_phyaddr`, `mmu_nocaddr`, and `mmu_perms` registers (see Listing 3.2).

Since permission checks are done by the destination of a DMA transfer, updating translations between physical and virtual addresses is an unprivileged operation.

Software can map any physical address to any virtual address it wants; the first cache miss will trigger a segmentation fault if the DMA request is denied. One somewhat counterintuitive side effect of this design is that if a page which is not readable in hardware is mapped as read-write, and the first cache miss to that address is a read, it will be loaded into the L1 cache with read-write permissions. This means that attempts to write to the memory will appear to succeed, triggering a segmentation fault only when the cache line is flushed and the cache controller attempts to write to the actual physical page.

When a segmentation fault occurs the CPU halts and asserts a 1-bit "segfault" flag. External logic may use this to reset the CPU or take other action.

Listing 3.2: Mapping a page on GRAFTON

```
1  # Pre-stack memory mapping
2  # No use of data RAM (and, consequentially,
3  # saved-temp registers) is permitted.
4  # a0 = Virtual base address of page
5  # (automatically rounded down to 2KB boundary)
6  # a1 = Physical address within the host to map
7  # (automatically rounded down to 2KB boundary)
8  # a2 = Network address of the host to map
9  # a3 = Permissions to map
10 .globl MmapHelper
11 MmapHelper:
12
13         mtc0        a0, mmu_page_id
14         mtc0        a1, mmu_phyaddr
15         mtc0        a2, mmu_nocaddr
16         mtc0        a3, mmu_perms
17
18         li          a0, SYS_MMAP
19         syscall
20
```

```
21      jr          ra
22      nop
```

3.5.2.5 Coprocessor 0 RPC Interface

Coprocessor 0 registers 1 through 7 (`rpc_dst_addr`, `rpc_src_addr`, `rpc_callnum`, `rpc_type`, `rpc_d0`, `rpc_d1`, and `rpc_d2` respectively) contain unpacked versions of a 32-bit RPC message. While this makes assembly-language message passing code very clear and easy to read, it is inefficient for sending unrelated messages since seven registers, rather than four, need to be loaded to send a message. Replying to a message with only a few fields changed, however, is very efficient and does not require complex bitmasking to do a read-modify-write of one field.

To send a message, the `a0` register is loaded with the value 0x0001 (`SYS_SEND_RPC`) and a `syscall` instruction is executed (Listing 3.3). There is no transmit FIFO; a send blocks if another send is still in progress and hasn't finished yet.

Listing 3.3: Sending an RPC message on GRAFTON

```
1  # Sends an RPC message
2  # Optimized routine to minimize pipeline delays
3  # a0 is a pointer to an RPC message
4  .globl SendRPCMessage
5  SendRPCMessage:
6
7       # First load needs a nop to bootstrap
8       lw          t0, 4(a0)
9       nop
10
11      # From here on, interleave to reduce need for nops
12      lw          t1, 8(a0)
13      mtc0        t0, rpc_dst_addr
14      lw          t2, 12(a0)
15      mtc0        t1, rpc_callnum
```

```
16      lw        t3, 16(a0)
17      mtc0      t2, rpc_type
18      lw        t4, 20(a0)
19      mtc0      t3, rpc_d0
20      lw        t5, 24(a0)
21      mtc0      t4, rpc_d1
22      mtc0      t5, rpc_d2
23
24      # Do the actual send
25      li        a0, SYS_SEND_RPC
26      syscall
27
28      jr        ra
29      nop
```

When a message is received, it is pushed into a FIFO capable of holding 256 messages (4KB). At any time, application software may load the `a0` register with the value 0x0002 (`SYS_RECV_RPC`) and execute a `syscall` instruction. The receive blocks if a message is not present in the FIFO; a nonblocking read opcode (`SYS_RECV_RPC_NONBLOCK`, 0x0003) is defined but not currently implemented. A `nop` operation is required in the delay slot after a `SYS_RECV_RPC` operation executes.

Once a message is received the application may read from the coprocessor 0 registers holding the message body. In order to pop the FIFO and allow receipt of another message, software loads `a0` with the value `SYS_RPC_DONE` (0x0004) and executes a `syscall` instruction (Listing 3.4).

Due to the fact that DMA writes use the RPC network for synchronization, race conditions may occur if a dirty cache line is flushed between the `SYS_RECV_RPC` and `SYS_RPC_DONE` operations. Unless application code is absolutely certain that no cache misses will happen, it is necessary to avoid memory write transactions during this period. (This is one of several shortcomings of GRAFTON that were addressed in the design of SARATOGA.)

Listing 3.4: Receiving an RPC message on GRAFTON

```
1  # Receives an RPC message.
2  # Blocks until a message is available.
3  # a0 is an output pointer to an RPC message
4  .globl RecvRPCMessage
5  RecvRPCMessage:
6
7       # Save the pointer
8       move        t7, a0
9
10      # Block until we get the message
11      li          a0, SYS_RECV_RPC
12      syscall
13      nop         # delay slot
14
15      # It is critical that no memory write transactions
            occur
16      # before we issue SYS_RPC_DONE to avoid race
            conditions
17      # or message corruption. The CPU will automatically
18      # acknowledge these messages and may overwrite the
19      # coprocessor registers in the process.
20
21      # Store it into registers
22      mfc0        t0, rpc_src_addr
23      mfc0        t1, rpc_dst_addr
24      mfc0        t2, rpc_callnum
25      mfc0        t3, rpc_type
26      mfc0        t4, rpc_d0
27      mfc0        t5, rpc_d1
28      mfc0        t6, rpc_d2
```

```
29
30      # Pop the FIFO
31      li          a0, SYS_RPC_DONE
32      syscall
33
34      # Write to memory
35      sw          t0, 0(t7)
36      sw          t1, 4(t7)
37      sw          t2, 8(t7)
38      sw          t3, 12(t7)
39      sw          t4, 16(t7)
40      sw          t5, 20(t7)
41      sw          t6, 24(t7)
42      nop
43
44      jr          ra
45      nop
```

3.5.2.6 RPC Function Calls

While the RPC network is intended to implement remote procedure call semantics, the low-level API provided by GRAFTON only allows sending and receiving of raw NoC frames. A simple C wrapper can provide a high-level function call interface.

Listing 3.5: A complete RPC function call on GRAFTON

```
1 /**
2     @brief Performs a function call through the RPC
         network.
3
4     @param addr       Address of target node
5     @param callnum    The RPC function to call
```

```
6       @param d0        First argument (only low 21 bits
            valid)
7       @param d1        Second argument
8       @param d2        Third argument
9       @param retval    Return value of the function
10
11      @return zero on success, -1 on failure
12   */
13 int RPCFunctionCall(
14      unsigned int addr,
15      unsigned int callnum,
16      unsigned int d0,
17      unsigned int d1,
18      unsigned int d2,
19      RPCMessage_t* retval)
20 {
21      //Send the query
22      RPCMessage_t msg;
23      msg.from = 0;
24      msg.to = addr;
25      msg.type = RPC_TYPE_CALL;
26      msg.callnum = callnum;
27      msg.data[0] = d0;
28      msg.data[1] = d1;
29      msg.data[2] = d2;
30      SendRPCMessage(&msg);
31
32      //Wait for a response
33      while(1)
34      {
35          //Get the message
```

```
36          RecvRPCMessage(retval);
37
38          //Ignore anything not from the host of interest.
39          //Save interrupts for future processing.
40          //Note that incoming RPC function calls, or
41          //pipelining multiple outstanding RPC calls,
42          //are not supported in this implementation.
43          if(retval->from != addr)
44          {
45              if(retval->type == RPC_TYPE_INTERRUPT)
46                  PushInterrupt(retval);
47              continue;
48          }
49
50          //See what it is
51          switch(retval->type)
52          {
53              //Send it again
54              case RPC_TYPE_RETURN_RETRY:
55                  SendRPCMessage(&msg);
56                  break;
57
58              //Fail
59              case RPC_TYPE_RETURN_FAIL:
60                  return -1;
61
62              //Success, we're done
63              case RPC_TYPE_RETURN_SUCCESS:
64                  return 0;
65
66              //We're not ready for interrupts, save them
```

```
67                  case RPC_TYPE_INTERRUPT:
68                       PushInterrupt(retval);
69                       break;
70
71                  default:
72                       break;
73             }
74
75        }
76   }
```

3.5.2.7 Debug Support

In order to allow synchronizing of a logic analyzer to certain regions of code, or measuring the execution time between certain events, the CPU has a 1-bit "trace flag" brought out to a top-level pin, which is asserted for one clock cycle during the execution of a `syscall` instruction when `a0` is set to `SYS_TRACE`.

GRAFTON includes a JTAG debug core as a synthesis-time option, along with a gdbserver so that standard debugging tools may be used with it. The debug core provides numerous standard debug functions such as "connect", "halt", "resume", and "read registers", and allows clearing of the "segfault" flag. Since GRAFTON typically executes from flash or ROM it is not possible to set breakpoints by writing to the code segment, so two hardware instruction breakpoints are provided. There is no support for data read/write breakpoints in the current version of GRAFTON.

Due to the node-level security provided by Antikernel, it is not possible for the JTAG debugger to simply read or write memory or I/O resources which belong to the application being debugged by sending NoC messages directly to the peripheral. Instead, the debugger sits on the D-side virtual address bus between the CPU core and D-side L1 cache. This effectively allows debugger memory accesses to NAT through the L1 cache and become indistinguishable from application memory accesses.

There is no authentication or security on the current debug core since it is

intended to be used in development hardware only (production devices would disable it at synthesis time). If necessary, it would be fairly simple to implement a password or other authentication system to prevent unauthorized debugging.

3.5.2.8 ELF Loader

In order to eliminate the need for a boot loader or complex ROM image creation, GRAFTON includes a bare-bones ELF loader implemented as a simple microcoded state machine (around 200 lines of Verilog including whitespace, comments, and debug print statements).

The loader reads the first 2KB of the executable into a SRAM buffer, then verifies the ELF magic number and walks through the ELF header to locate the ELF program headers. It then loops over the table of program headers and, if the header is marked as loadable and has a nonzero size on disk, adds an entry to the TLB (marked read-execute) for each 2KB page in the header.

The loader does have a number of limitations. It does not currently support the `.data` segment needed for read-write global variables initialized to nonzero values (initialized global constants in `.rodata`, or globals initialized to zero in `.bss`, are fully supported). All data stored in program headers must be aligned to 2KB boundaries, and the program header table must fit within the first 2KB of the executable. All libraries must be statically linked or moved to server applications, rather than dynamically linked.

After the loader finishes executing $pc is set to `e_entry` which is the virtual address of `_start`, the first instruction in the program.

Other than magic-number sanity checks, no signature checks or verification of the executable are performed. This is addressed in SARATOGA.

3.5.2.9 Performance

GRAFTON was designed for size rather than speed. The current version tops out at 75 MHz on a Xilinx Spartan-6 FPGA (-3 speed) and 110 on a Xilinx Artix-7 (-2 speed). On the Artix, resource usage is approximately 1700 flipflops, 2700 LUTs, 1100 slices, and 11 block RAMs for the CPU core alone. A minimal reference system

(CPU, RAM, ROM, JTAG debug bridge, name server, system info, and the required NoC routers) uses 6400 flipflops, 8100 LUTs, 2900 slices, and 30 RAMs.

Initial Dhrystone benchmark performance on GRAFTON was abysmal (0.035 DMIPS/MHz). We added a set of hardware performance counters and determined that the I-side L1 cache had terrible performance (a miss rate of over 5% and 80 cycle latency on misses), and that the benchmark spent the overwhelming majority of its time thrashing the cache and waiting for instruction fetches. Enabling compiler optimizations (`gcc -O2`) shrunk the working set size significantly, reducing the cache miss rate to 1.95%. After some optimizations to reduce L1 miss latency to 29 cycles, Dhrystone performance reached 0.205 DMIPS/MHz.

It appears that the poor benchmark performance is largely, if not entirely, due to the simplistic cache design (and not to overhead from security checks). We plan to improve the cache in the future, perhaps increasing the size to more than a single block RAM and/or making it associative, and explore ways to further reduce the latency of misses.

3.5.3 SARATOGA

3.5.3.1 Introduction

The next-generation CPU for Antikernel is called SARATOGA (after another nearby town). SARATOGA is a high-performance dual-issue in-order barrel processor using a modified version of the the MIPS-1 instruction set with an 8-stage pipeline and a parameterizable number of hardware threads (the default is 32, but any power of two greater than or equal to eight is legal). This produces the net effect of 8 virtual CPUs at 1/8 the core clock rate, time-sharing the same two execution units.

While SARATOGA is intended to be compatible with C compilers designed to target the MIPS-1 instruction set given appropriate flags and linker scripts, no portability of binary software between SARATOGA and an actual MIPS processor is implied or intended.

SARATOGA has eight pipeline stages: two of instruction fetch (IFETCH0 / IFETCH1), two of instruction decode and register fetch (DECODE0 / DECODE1),

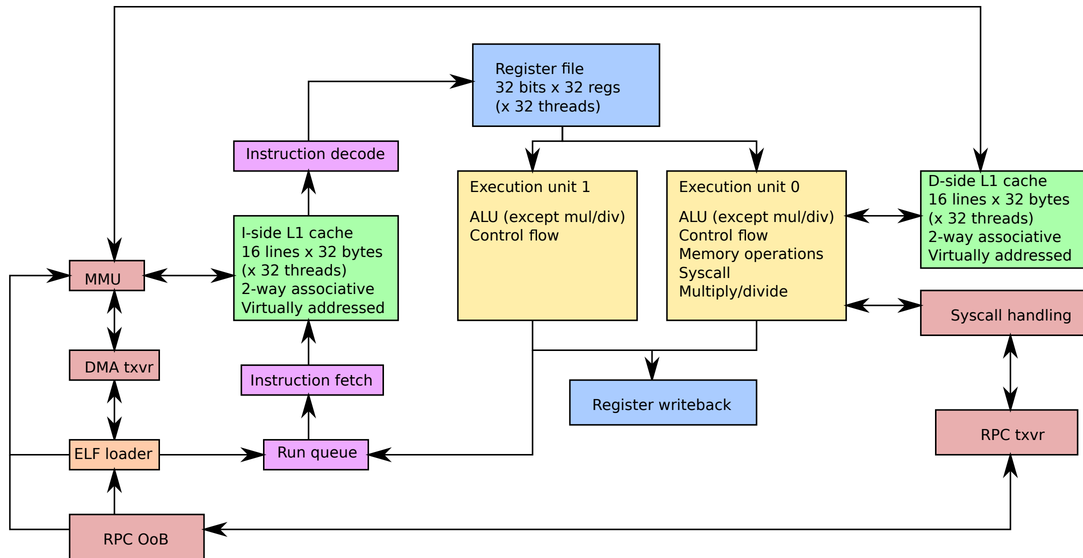

Figure 3.14: Block diagram of SARATOGA

and four of execution (EXEC0 / EXEC1 / EXEC2 / EXEC3).

The scheduler decides what thread to run next during EXEC3, looks up the program counter, and dispatches a 64-bit read on the I-side bus; the L1 cache has two-cycle read latency so the instructions are available (in the event of a cache hit) during the DECODE0 cycle.

During DECODE0 the register IDs are parsed out by combinatorial logic and fed out to the register file. The decode/schedule logic then determines which instruction(s) to execute. If the first instruction is a cache miss, the CPU stalls regardless of whether the second hit or missed (since there is no support for out-of-order execution). If both instructions hit, it will attempt to issue both. If the second instruction uses the result of the first, there is a data hazard and the second instruction must wait until the following cycle. Memory operations can only execute on unit 0 and cannot be dual-issued with any other instruction in the current design; improving this is planned for a future microarchitecture revision. The `syscall` instruction uses all ports of the register file and thus uses both execution units. Multiply/divide instructions can only execute on unit 0; there is no sense in dual-issuing them since the MIPS-1 ISA has dedicated result registers which cannot be shared by two concurrent multiply/divide operations. (If and when the processor is extended to support a “multiply or divide to general-purpose register” instruction such as found

in MIPS32, this scheduling restriction will be removed for those opcodes.)

Unlike an actual MIPS processor, SARATOGA does not have branch delay slots (since successive instructions in an application actually execute many cycles apart). If the first instruction in a pair is a jump the second does not execute at all; if the second is a jump the first executes normally. The jump-and-link instruction returns to the instruction 8 bytes after the jump as in the actual MIPS architecture. The instruction in the delay slot may be anything (since it never executes), however it is recommended that it be a `nop` to improve code readability. This results in code compatibility with MIPS C compilers and assemblers as long as they are instructed to not use delay slots (for example the `.set noreorder` directive in the GNU assembler, or the `-fno-delayed-branch` flag to gcc).

Branch operations begin in EXEC0 and the results from both execution units are resolved during EXEC1, so that final writeback to the program counter can occur during EXEC2.

3.5.3.2 Thread Scheduler

Threads in SARATOGA are scheduled using a circular linked list implemented as a Verilog module, which supports atomic (single cycle) inserts and deletes with 2-cycle pipeline latency.

The processor begins in the idle state with no threads running. A free-list of thread IDs is initialized to contain every valid thread ID, and a bitmap of thread IDs is initialized to the "unallocated" state.

When a "create new thread" request is received from the ELF loader (see section 3.5.3.8) the free list is popped, the bitmap is updated to reflect that this thread ID is allocated, and the thread ID is now available for use (but is not yet scheduled).

Once the ELF loader has finished initializing the thread, if everything was successful, it issues a "run" request to the scheduler. The provided thread ID is checked to make sure it's not already in the run queue, then inserted into the linked list.

During execution, the CPU reads the current thread from the linked list and

schedules it for execution if possible, then goes on to the next thread in the linked list the following cycle. If the thread is already in the pipeline (which may be true if less than 8 threads are currently runnable) then it waits for one cycle and tries again. If the thread is not in the run queue at all (which may be true if the thread was just canceled, or if no threads are currently runnable), then the CPU goes to the next thread and tries again the next cycle.

To delete a thread, it is removed from the linked list and pushed into the free list, and the bitmaps are updated to reflect its state as free. The linked-list pointers for the deleted thread are not changed; this ensures that if the CPU is about to execute the thread being deleted it will correctly read the "next" pointer and continue to a runnable thread the next clock cycle. (There are no race conditions possible due to the multi-cycle latency of the allocate and free routines; by the time the freshly deleted thread can be re-added to the run queue the CPU is guaranteed to have continued to a runnable thread.)

The architecture allows for a thread to remove itself from the run queue without terminating (although the thread management API does not currently provide a means for doing this). This will allow threads blocking on long-running I/O operations to place themselves in a "sleep" state from which they can quickly awake, but which does not take up space in the CPU pipeline that other threads could potentially be using.

3.5.3.3 Execution Units

SARATOGA has two execution units connected to separate ports on the register file. Both are copies of the same Verilog module however some functionality is left unconnected (and thus optimized out) in execution unit 1.

Each execution unit takes in two values from the register file during EXEC0, and outputs one value to the register file during EXEC3.

During EXEC0, unit 0 may also dispatch an RPC send or receive, or a memory transaction. Results from RPC receives (if data is available), as well as memory operations (in case of an L1 cache hit) are available during EXEC2.

The multiply-divide unit consists of a switchable signed/unsigned multiplier

which returns results during EXEC3. The divide instructions are currently unimplemented, but we intend to add a fully pipelined iterative divider that can accept a new input every cycle and returns results a fixed 32 cycles later. Given the 8-stage barrel design this leads to an apparent latency of 4 instructions (if 8 or less threads are active) or less. If all 32 threads are active, divisions will never block.

All other ALU operations complete by EXEC1.

3.5.3.4 Register File

SARATOGA's register file is logically 128 bytes (32 32-bit registers) for each thread context. For the default of 32 threads, this comes out to 4096 bytes, or 4 KB. This size would, at first glace, seem to fit comfortably in one Xilinx 7-series FPGA block RAM, or two Spartan-6 FPGA block RAMs.

Unfortunately, the register file must be capable of reading four registers and writing two every clock cycle. Conventional FPGA block RAM uses dual-port 8T SRAM cells and thus cannot access more than two words per clock. Adding R additional read ports can be done with trivially with $O(R)$ area overhead by creating R individual dual-port memories sharing one write port but with separate read ports.

This does not solve the problem of multiple write ports, however. Several proposed techniques, such as live value tables [69], were evaluated and rejected before settling on the XOR-based technique described in [70]. This technique allows dual port SRAM arrays to be scaled up to arbitrary numbers R of read ports and W of write ports with $O(W^2 + WR)$ area overhead, and $O(logW)$ critical path delay.

The heart of the algorithm is quite simple and takes advantage of the cancellation property of XOR (any value XORed with itself equals zero). A matrix consisting of $W + R$ rows by W columns of dual port SRAMs is constructed; to read from port r the desired address is read from all W SRAMs in row r of the grid and the outputs are fed into a W-input XOR tree. The output of the XOR tree is the desired value.

Writing a value to port w is a two-step process. First, the desired address in row $R + w$ of the matrix is read back and the values are XORed together, yielding the old value. This is then complemented, XORed with the new value, and written

back to all SRAMs in column w of the grid. When the new values are XORed the result will be the new value.[8]

We have implemented this memory as a fully parameterizable Verilog module which supports arbitrary numbers of read and write ports, arbitrary sized memory arrays, and can use either block or LUT RAM. The number of pipeline registers is parameterizable from zero (for LUT RAM only) to two.

In addition to the register file, smaller instances of the same multiport memory module have found themselves useful in various support roles throughout the processor (such as managing linked-list pointers in the thread scheduler, where the "next" and "prev" pointers for two different nodes need to be updated in one clock cycle to ensure atomicity of inserts and deletions).

3.5.3.5 L1 Cache

The L1 cache for SARATOGA is split into fully independent I- and D-side banks, and is fully parameterizable for levels of associativity, words per line, and lines per thread. The default configuration is 2-way set associative and 16 lines of 8 32-bit words, for a total size of 1KB instruction and 1KB data cache per thread (plus tag bits) and 32KB overall. The cache is virtually addressed and there is no coherency between the I- and D-side caches. (Since software cannot write to instruction memory, and the instruction bus lacks a write port, this was deemed unnecessary.)

The current cache is quite small per thread, which is likely to lead to a high miss rate, but this is somewhat made up for by the ability of multithreading to hide latency - if all 32 threads are active, a 32-cycle miss penalty can be incurred without causing more than a single cycle of stalling. We have not yet implemented performance counters (as in GRAFTON) for measuring cache performance; after this is added there are likely to be numerous optimizations to the cache structure (possibly including speculative prefetching of upcoming cache lines after a miss).

As with GRAFTON the cache lacks an explicit flush mechanism in the current

[8]Note that the value in cell $(R+w, w)$ is overwritten and thus not included in the feedback XOR calculation for writes. Since the cells along this diagonal are never read they can be optimized out during the code generation process, resulting in a final area cost of $W(W+R-1)$ SRAM blocks.

prototype. We intend to add one in the near future and are currently researching efficient flush strategies.

3.5.3.6 MMU

In order to speed prototyping a very simple MMU was created, consisting of a software-controlled TLB with no external page tables. It supports a parameterizable number (the default is 32) of 2KB pages of virtual memory per thread, mapped consecutively starting at virtual address 0x40000000. Each thread has a fully independent virtual address space, meaning that the total amount of virtual memory addressable by all threads combined in the default configuration is 32*32 = 1024 pages, or 2 MB. This has been sufficient for initial prototyping; a full MMU with a TLB and external page tables in RAM is planned for the future but was deemed beyond the scope of the thesis.

Since the MMU is accessed by software using an abstracted API (the `OOB_OP_MMAP` RPC call on the OoB management interface) it will be possible to make fairly extensive changes to the internal MMU and TLB structure without breaking software compatibility.

Each TLB entry consists of a valid bit, three permission flags (read/write/execute), a 16-bit node ID, and a 21-bit physical address within that node (all mappings must be page aligned). This allows the full 48 bits of physical address space to be used.

3.5.3.7 RPC Network Interface

The SARATOGA processor requires an RPC subnet with one address per thread context, plus an extra address for out-of-band (OoB) management. Since the thread count must be a power of two, and subnet sizes must be a power of two, this results in the processor consuming $2N$ addresses for N threads.

The first address of the subnet is used as the OoB management address. This ensures that any node which queries the name server for the hostname of the processor will get the management address, regardless of the number of threads supported. The high half of the subnet is used for thread addresses; all other addresses in the low half of the subnet are unused and incoming packets are ignored.

In order to send an RPC message, the high half of the `a0` register must be set to `SYS_SEND_RPC_BLOCKING`.[9] The low half of `a0`, as well as `a1`, `a2`, and `a3`, are then loaded with a 4-word RPC message. (The high half of `a0` is used as the opcode since this would normally be the source address of the packet, but this is added by the hardware). A `syscall` instruction then actually sends the message.

This calling convention was chosen such that sending an RPC message would appear to the compiler and user, as much as possible, to be an ordinary function call. Since the standard C calling convention for MIPS passes arguments in `a0`, `a1`, `a2`, and `a3`, a trivial wrapper function can be created (listing 3.6) which simply sets the high half of `a0` and invokes `syscall` before returning.

Listing 3.6: Sending a packed RPC message on SARATOGA

```
1 //void SendRPCMessage(unsigned int a, unsigned int b,
      unsigned int c, unsigned int d)
2 SendRPCMessage:
3         lui       t0, SYS_SEND_RPC_BLOCKING
4         andi      a0, a0, 0xffff
5         or        a0, a0, t0
6         syscall
7         jr        ra
```

If the calling code prefers to store the various message fields unpacked in a `struct`, as was done with GRAFTON, the code becomes slightly more complex since some bitmasking and shifting is required to generate the packed form of the message (listing 3.7).

Listing 3.7: Sending an unpacked RPC message on SARATOGA

```
1 /*
2         @brief Sends an RPC message
3
4         @param a0          Pointer to a RPCMessage_t object
```

[9] Note that the "blocking" part of the opcode name does NOT mean that the syscall will block until the message has been sent. It will, however, block rather than failing if the transmit FIFO is full.

```
 5   */
 6  .globl SendRPCMessage
 7  SendRPCMessage:
 8
 9      //Load arguments from RAM
10      lw          t0, 4(a0)    //dst addr
11      lw          t1, 8(a0)    //callnum
12      lw          t2, 12(a0)   //type
13      lw          a1, 16(a0)   //d0
14      lw          a2, 20(a0)   //d1
15      lw          a3, 24(a0)   //d2
16
17      //Pack into registers as necessary
18      lui         a0, SYS_SEND_RPC_BLOCKING
19      andi        t0, t0, 0xffff  //mask off valid addr
           bits
20      andi        t1, t1, 0xff    //mask off valid callnum
           bits
21      andi        t2, t2, 0x7     //mask off valid type
           bits
22      or          a0, t0          //addr word
23      sll         t1, t1, 24      //callnum
24      sll         t2, t2, 21      //type
25      or          a1, a1, t1      //shifted callnum | d0
26      or          a1, a1, t2      //full op word
27
28      //Do the actual operation
29      syscall
30
31      //Done
32      jr          ra
```

```
33      nop
```

Receiving an RPC message is essentially the same process in reverse. The high half of `a0` is loaded with `SYS_RECV_RPC_BLOCKING`[10] and a `syscall` instruction is executed. When a message is ready, the four words written to (in order) `v1`, `v0`, `k0`, and `k1`.

As with the argument registers, this set was chosen very carefully. `v0` and `v1` are the standard MIPS return registers. If the message is received in a function using the standard MIPS calling convention, the "return value" of the function is the second word of the RPC message. This contains both the RPC packet type (success/failure state) as well as the first data word, and thus allows simple RPC function calls to be performed entirely in registers without using memory. If the full body of the RPC message is required, it may be either accessed through the other registers directly, or by an assembly-language wrapper function (Listing 3.8) which unpacks the message and stores it in a `struct`. [11]

Listing 3.8: Receiving an unpacked RPC message on SARATOGA

```
 1 /*
 2      @brief Receives an RPC message.
 3
 4      Blocks until a message is available.
 5
 6      @param a0        Pointer to an RPCMessage_t object for
            storing the result in
 7  */
 8 .globl RecvRPCMessage
 9 RecvRPCMessage:
10
11      //Save the pointer
```

[10]This call blocks until a message is received and currently has no non-blocking equivalent. We plan to add a "check for message and return immediately if none found" call in the future.

[11]The names `rpc_header`, `rpc_d0`, `rpc_d1`, and `rpc_d2` are aliases for `v1`, `v0`, `k0`, and `k1` defined in a header file for convenience.

```
12      move        t7, a0
13
14      //Blocking RPC receive
15      lui         a0, SYS_RECV_RPC_BLOCKING
16      syscall
17
18      //Extract fields and store to memory
19      sw          rpc_d2, 24(t7)           //d2
20      sw          rpc_d1, 20(t7)           //d1
21      srl         t0, rpc_d0, 24           //shift out
            callnum
22      srl         t1, rpc_d0, 21           //shift out type
23      li          t2, 0x1FFFFF             //mask for d0
24      and         t2, t2, rpc_d0           //mask out d0
25      andi        t1, t1, 0x7              //mask for type
26      srl         t3, rpc_header, 16       //mask for src
            addr
27      andi        t4, rpc_header, 0xffff   //mask for dst
            addr
28      sw          t2, 16(t7)               //d0
29      sw          t1, 12(t7)               //type
30      sw          t0, 8(t7)                //callnum
31      sw          t4, 4(t7)                //dst addr
32      sw          t3, 0(t7)                //src addr
33
34      //Done
35      jr          ra
36      nop
```

Since a new application starting up on a SARATOGA core does not know the management address of its host CPU *a priori*, we provide a means for doing so through the `syscall` instruction. At any time, an application may perform a

`syscall` with the high half of `a0` set to `SYS_GET_OOB` to set the `v0` register to the current CPU's management address. Aside from the RPC interface itself, all other CPU management operations are accessed via RPCs to the management address.

As of now a total of four operations are supported by the management interface. The first, `OOB_OP_GET_THREADCOUNT`, returns the number of available threads. `OOB_OP_CREATEPROCESS` creates a new process (detailed in the following section). `OOB_OP_MMAP` creates or modifies a page table entry (section 3.5.3.6). `OOB_OP_ATTEST` is used for remote attestation (section 3.5.3.9).

The `OOB_OP_MMAP` call is restricted to threads running on the processor (and will modify the calling thread's current page table state). All other calls are available to any node in the system.

3.5.3.8 ELF Loader with Code Signature Checking

SARATOGA's ELF loader is slightly more complex than GRAFTON's due to additional security features and multithreading support.

To create a new process, a node sends an `OOB_OP_CREATEPROCESS` call to the CPU's OoB management port, specifying the physical address of the executable to run. The management system begins by allocating a new thread context, returning failure if all are currently in use.

If a thread ID was successfully obtained, the ELF loader then issues a DMA read for `sizeof(Elf32_Ehdr)` bytes to the supplied physical address, expecting to find a well formed ELF executable header. If the header is invalid (wrong magic numbers, incorrect version, or not a big-endian MIPS executable file) an error is returned.

If the header is well formed, the loader then looks at the `e_entry` field to find the address of the program's entry point. This is fed into a FIFO of data to be processed by the HMAC-SHA256 hashing engine. (It is important to hash the entry point address, as well as the contents of all executable pages, in order to ensure that a signed application cannot be modified to start at a different address within the code, potentially performing undesired actions.)

The loader then checks the `e_phoff` field to find the address of the program

header table, which stores the addresses of all segments (both loadable and not) in the program's memory image. It loops over the program header table and checks the `p_type` field for each entry. If the type is `PT_LOAD` (meaning the segment is part of the loadable memory image) then the loader reads the contents of the segment and feeds them into the hashing engine, and stores the virtual and physical addresses in a buffer for future mapping. If the type is 0x70000005 (an unused value in the processor-defined region of the ELF program header type specification) then the segment is read into a buffer holding the expected HMAC-SHA256 signature of the executable.

After all loadable segments have been hashed, the signature is compared to the expected value. If they do not match an error is returned and the allocated thread context is freed.

If the signature is valid, the list of address mappings is then fed to the MMU. Note that the ELF loader is the only part of the processor which has permission to set the `PAGE_EXECUTE` permission on a memory page; permissions for pages mapped by software through the OoB interface are ANDed with `PAGE_READ_WRITE` before being applied. This means that it is impossible by design for any unsigned code to ever execute as long as the physical memory backing the executable cannot be modified externally (for example, by modifying the contents of an external flash chip while the program is executing). With appropriate choices of access controls for on-chip memory, and use of encryption to prevent tampering with off-chip memory, this risk can be mitigated.

After the initial memory mappings are created the program counter for the newly created thread is set to the entry point address from the ELF header and the thread is added to the run queue.

3.5.3.9 Remote Attestation

SARATOGA supports a simple form of remote attestation. When an application is loaded by the ELF loader, the signature is stored in a buffer associated with the thread ID. At any time in the future, any NoC node may ask the CPU (through the OoB management interface) to return the signature associated with a

given thread context.

This is done through the OoB RPC call `OOB_OP_ATTEST`. It takes two parameters (the thread ID and a word index), and returns two values (the requested 32-bit portion of the signature, and a revision counter). The revision counter increments atomically every time the signature buffer is written, and is used to avoid returning a mix of two hashes if a new executable is loaded during the attestation operation. If the counter is not the same for all 8 words of the hash, then the attestation must be repeated to gain valid data.

While there is currently no support for providing attestation to off-die entities (through some sort of cryptographic signature) this could certainly be added in the form of a node with an embedded key which retrieves a hash by calling `OOB_OP_ATTEST`, signs it, and sends it over a LAN/WAN link.

3.5.3.10 Performance and Layout

SARATOGA was built with speed as a higher priority than size, and the implementation performance reflects this. The SARATOGA processor is much more parameterizable than GRAFTON so it is important to specify the configuration when discussing size or performance. Unless otherwise stated all performance numbers are for the default configuration (32 threads, 2-way cache associativity, 16 lines of 8 words per cache bank).

The CPU can easily reach around 180 MHz on a Xilinx Artix-7 FPGA (-2 speed), and can be pushed to 200 with careful floorplanning of the L1 caches and register file. Area is 5700 flipflops, 6400 LUTs, 2570 slices, and 44 RAMs for the CPU itself. The reference system requires more NoC routers than the GRAFTON reference due to the larger subnet size (although in a system with more peripherals the overhead would be less significant as the intermediate routers would be serving other children as well); the reference system is 12800 flipflops, 15100 LUTs, 5900 slices, and 77 RAMs. This includes all of the support infrastructure seen in the GRAFTON reference system, as well as a nocsniff instance observing the CPU's RPC uplink to aid in debugging. (This would of course be omitted for security reasons in a deployed system.)

Overall, when comparing the CPUs at a high level, SARATOGA is 2.4x the area (measured by slices) for 3.63x the theoretical peak performance (dual-issue 200 MHz = 400 MIPS, vs single-issue 110 MHz = 110 MIPS). We cannot report on Dhrystone performance at this time as the divide instruction is not yet implemented.

The target FPGA for the current test system is a Xilinx XC7A200T, which has 134,600 6-input LUTs and 269,200 flipflops located in 33,650 slices, as well as 740 DSP blocks and 365 4KB RAM blocks (each usable as two separate 2KB blocks under some conditions). The programmable logic fabric forms an "H" shape (Fig. 3.15) around the sixteen high-speed serial transceivers, divided into a 2x5 grid of clocking regions. Smaller gaps in the fabric are present at the left side of the array, for the PCIe endpoint block and configuration/startup/support logic. In order to get the maximum possible performance out of SARATOGA it was necessary to carefully floorplan the logic around these obstacles.

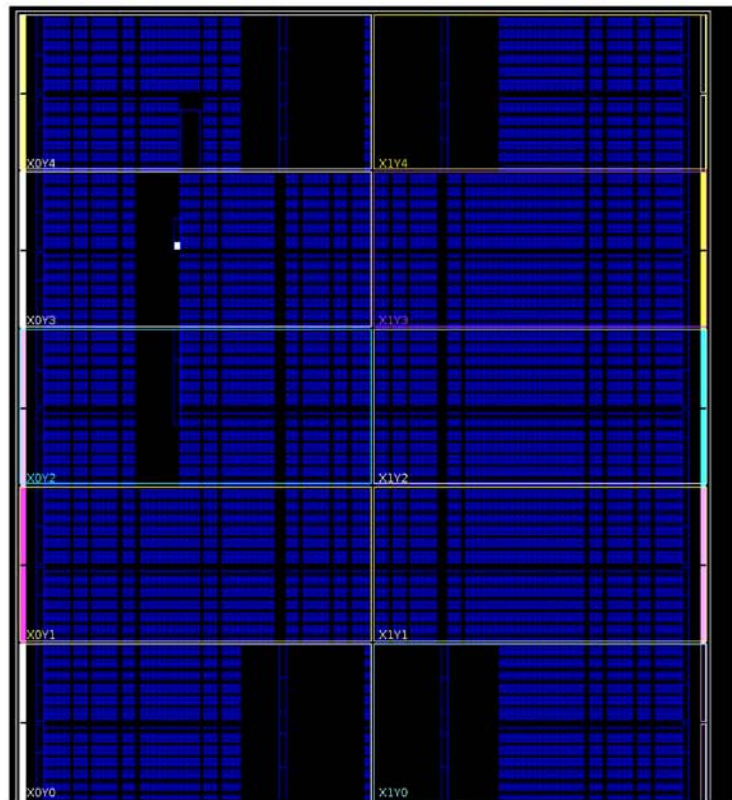

Figure 3.15: Floorplan of Xilinx XC7A200T

The physical layout for the current SARATOGA development system is shown

in Fig. 3.16. This view only shows the upper left portion of the FPGA; the remainder is unused in the current SoC.

The name server and system info core (yellow and green) are located in the upper left of the die, to the left of the transceiver block. The same area also contains the block RAM controller and firmware ROM (lavender) and the RPC and DMA routers for their subnet (purple).

Moving down and right, we get to the logic analyzer and packet sniffer used for testing the prototype (also purple), as well as their parent NoC routers (dark green). To the left, in cyan, are the parent RPC and DMA routers for the leaf routers. A second router pair (dark blue) connects these routers to the root routers (red). The root router itself then connects to the JTAG debug bridge (light green).

The CPU itself is located at the center of the device (bottom right of the figure), and is shown in yellow.

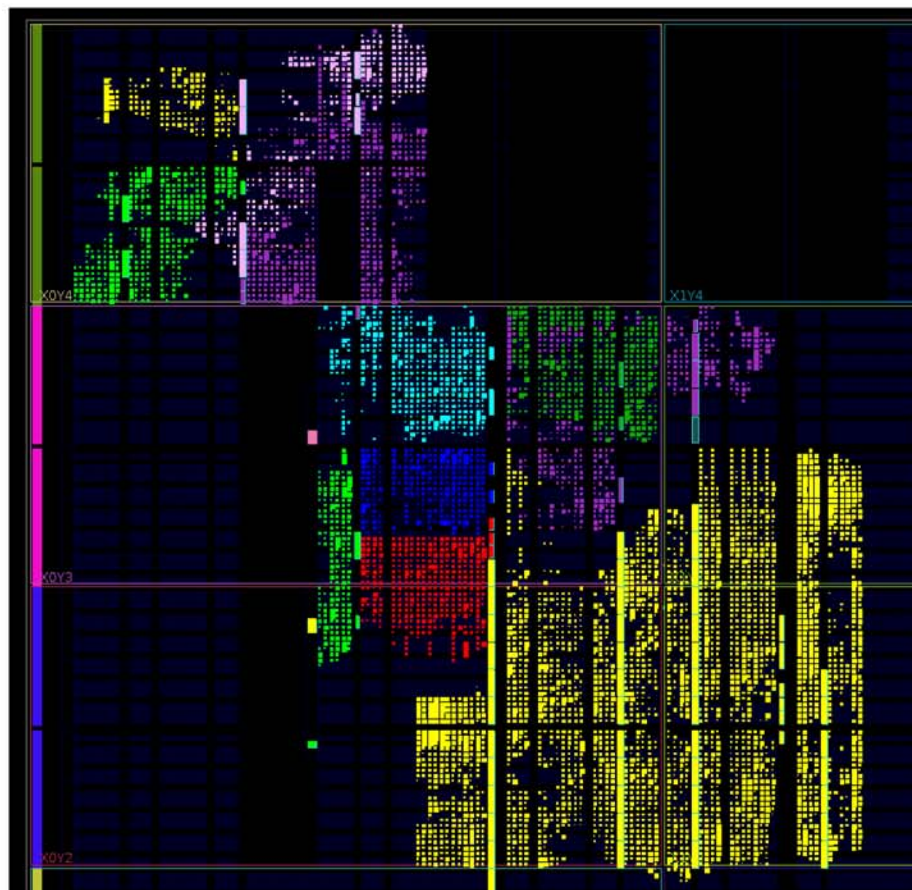

Figure 3.16: Overview of SARATOGA reference SoC

We now zoom in on the CPU, using a different color scheme (Fig. 3.17) to show more detail and hiding the surrounding logic.

The XC7A200T has slightly asymmetric spacing between columns of block RAM (large vertical rectangular blocks), which was critical to achieving timing closure at 200 MHz. By having the processor span the centerline of the device, where two RAM columns are slightly closer together, we were able to slightly reduce the latency between the cache tag matching logic and the output multiplexers.

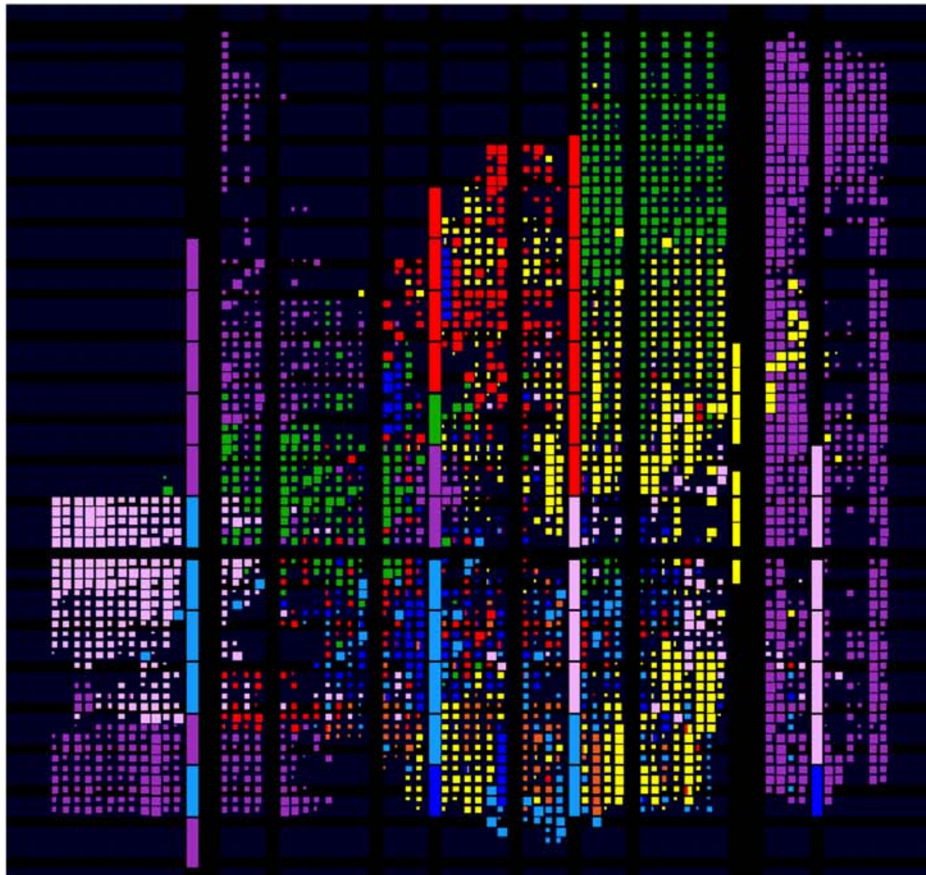

Figure 3.17: Detail of SARATOGA layout

At right, the two execution units (yellow) are located around the register file (lavender). The upper execution unit also connects to the multiply/divide unit registers (dark green rectangle, upper right) and the integer divider (purple), which takes up most of the right side of the CPU.

The D-side L1 cache (red) and the RPC interface (purple) are respectively located at the top center and left of the processor, next to execution unit 1, since

they are not used by unit 0.

Moving down, the I-side L1 cache (medium blue) is located at the bottom center and left of the processor, next to the program counters (orange) and instruction decoding logic (blue fabric logic). The remainder of the central area is occupied by the DMA transceiver and cache miss handling logic (dark green) and MMU (dark blue block RAMs at the very bottom).

The bootloader (pink rectangular block) is located in the lower left of the CPU, safely out of the way of critical paths in the execution units and caches but close to the NoC interfaces and program counter. The HMAC engine is visible in purple drirectly underneath. Scattered to the right is the thread scheduler (red).[12]

Initial design and implementation of the CPU relied on automatic placement however in order to achieve the 200 MHz performance target a substantial amount of manual tuning was necessary. This was done by specifying rectangular regions ("pblocks", in Xilinx parlance) of the FPGA and instructing the tools to place various parts of the CPU in each one. The current placement constraints are shown in Fig. 3.18.

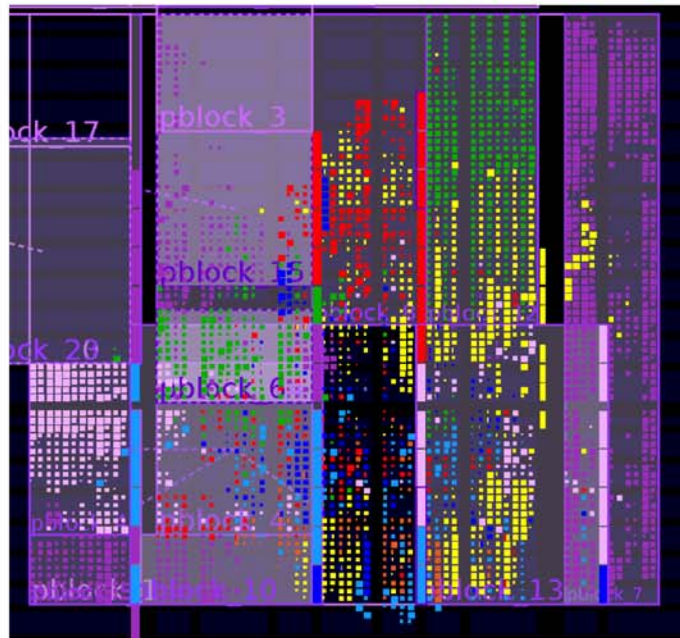

Figure 3.18: Detail of SARATOGA floorplanning

[12]The thread scheduler is one of the few major subsystems of the CPU to not be manually floorplanned so this location was chosen by the autorouter with no manual guidance.

CHAPTER 4
Tools and Infrastructure

4.1 Introduction

A system which cannot be tested or debugged is essentially useless. The difficulty with Antikernel is striking a balance between security and testability: it must be possible to confirm that the code is correct, and investigate failures if bugs are found, without opening the system up to attack.

The first step is to provide a means for external code to communicate with the nodes within the SoC. We chose to use the IEEE 1149.1 Joint Test Action Group (JTAG) boundary scan protocol for this as it is natively supported by the FPGA and requires no additional pins beyond those used to load the FPGA bitstream during development.

4.2 Libjtaghal

4.2.1 Layer 1: libjtaghal and jtagd

Early development and testing used Xilinx's `impact` tool for programming the boards however Xilinx does not provide any sort of SDK for using their JTAG adapter to perform in-circuit test operations aside from executing a fixed script in a SVF file so this was unsuitable for further work.

After examining and rejecting several existing tools such as `openocd` [71], `urjtag` [72], and `xc3sprog` [73] for several reasons, including inability to operate in a client-server environment and overly restrictive licenses, the author decided a new tool was warranted. This resulted in the creation of a library known as `libjtaghal` - the JTAG Hardware Abstraction Layer - and an associated `jtagd` service that bridges any adapter supported by `libjtaghal` out to a TCP socket.

The current lab environment runs one `jtagd` for each of our 20 FPGA development boards on a rackmounted server with an Intel i7 920 processor, connected via a tree of USB hubs to JTAG adapters using the FTDI FT232H chipset. The developer's computer then communicates with the `jtagd` over the LAN to load

bitstreams onto the board and run test cases. Due to scaling and performance limitations caused by the time-shared half-duplex USB bus, we are planning to make an FPGA-based debug card which implements the `jtagd` protocol in an FPGA and bridges a large number of JTAG ports directly to Gigabit Ethernet.

4.2.2 Layer 2/3: JtagDebugController and nocswitch

In order for test code running on a developer's computer to interact with the prototype, we needed some way to move RPC and DMA messages over the JTAG link. The initial implementation of this scheme required that the computer poll the board to confirm that transmit buffer space was free, then send a packet, then set a "push packet to network" bit in the status register once the packet was fully received. An analogous approach was used for receiving packets from the board. The end result was a half-duplex system whose performance was severely impacted by latency.

After running into bottlenecks using this interface for unit tests, we discarded the entire layer-2 encapsulation and started from scratch. The new system uses Ethernet-style framing in full-duplex system-synchronous mode and somewhat sidesteps the typical scan structure of JTAG.

Once the FPGA is configured, the host loads the FPGA's JTAG instruction register with the USER1 instruction, moves to the SHIFT-DR state, and stays there. The host clocks data into TDI constantly such that TCK is effectively free-running, sending idle code words (0x20080000) if there is no data to send (Fig. 4.1). On each rising edge of TCK the debug core shifts the incoming bit into the 32-bit JTAG data register and checks if DR equals a "magic" sync word (0x555555D5), which denotes the start of a data frame and allows the receiver to synchronize to 32-bit word boundaries; the same framing is used to send data from the board to the host.

In this example, two RPC messages are being sent, one in each direction. The start of an RPC or DMA message is denoted by a header word (0x60200000 in this example) which specifies the type of message, the frame length, a sequence number, and a "credit" counter indicating the number of words free in the remote node's ingress buffer as of that sequence number. By subtracting the number of words sent

since the packet with that sequence number ended (which are currently in flight) the sender can determine a lower bound on available space in the buffer and make sure not to send frames before the recipient is ready for them. (The idle code words contain credit counters in the low 11 bits, set to zero for clarity in this example, so that buffer state can be maintained even when no frames are being sent.)

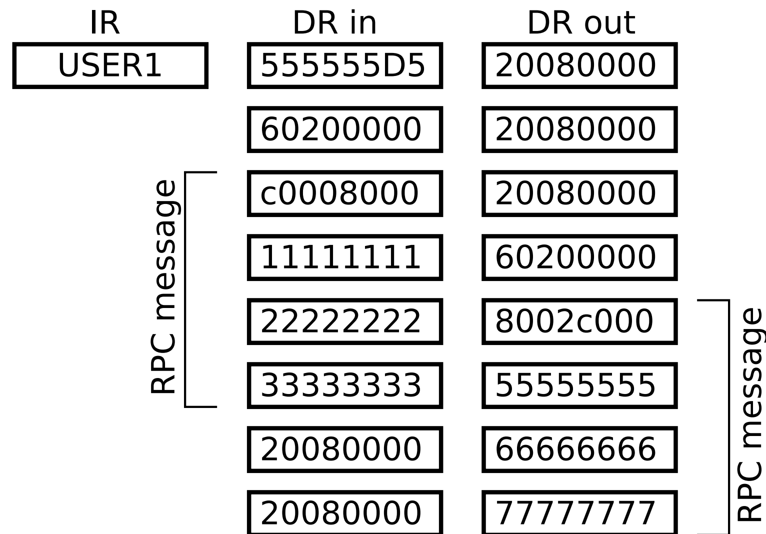

Figure 4.1: JTAG framing example

Since TMS remains at 0 and the JTAG state machine remains in SHIFT-DR during this entire period, there is no overhead of constantly sending UPDATE / CAPTURE commands and the transmit and receive datapaths may operate fully asynchronously (although sharing the same clock). At any time, the host is free to return the JTAG state machine to the idle state and perform ordinary boundary scan operations; the transfer will resume once USER1 is selected again. Single 32-bit words may be scanned individually if this high-speed mode is not supported by the test equipment in use.

While `libjtaghal` provides an API allowing applications to send and receive messages to the board by directly opening a JTAG connection or by communicating over TCP with a `jtagd`, this is not the common use case since it does not provide a means for multiple applications to communicate with the board simultaneously.

In order to fulfill this need, we created an application called `nocswitch` which acts as a PC-based NoC router. It connects to the target board through `jtagd`

and accepts TCP connections from applications on the host system (or potentially remote nodes in a LAN environment), assigning a unique address in the c000/2 subnet to each client. The `nocswitch` then takes packets arriving on any interface and forwards them out the appropriate port.

This allows a single FPGA system to exchange layer-2 frames over JTAG with our entire suite of development tools including an internal logic analyzer (section 4.8), gdbserver for our softcore CPU, and NoC packet sniffer (section 4.7). Both C++ (for PC-based unit tests) and Verilog (for hardware cosimulation) libraries are provided to interface with the `nocswitch` protocol.

Debug features can often present a potential point of entry for an attacker so it is important to secure them. While it is certainly possible for an implementation to fuse off the JTAG interface, Antikernel's compartmentalized security model should make this unnecessary since the debug bridge forces the high two bits of all incoming source addresses high (thus forcing them into the debug subnet c000/2). Since host-based nodes cannot impersonate on-chip entities the security model is not broken by the debug bridge (although of course nothing stops one host-based debug tool from impersonating another).

4.3 Splash

4.3.1 Motivation

Development of a system such as Antikernel poses some unique challenges: code is written for many languages across many architectures. A "standard" workflow would involve running one IDE for PC software (test cases and tools), one for embedded software (Antikernel applications and drivers), and one for FPGA development (the network, CPU, and hardware processes).

In such a workflow there are many consecutive build and test cycles moving from one IDE to the next, with the potential for erroneous results if one compile process was forgotten while testing in another tool. Furthermore, automated regression testing is difficult because most conventional build/unit testing frameworks cannot support cross-architecture dependencies and have poor support for testing on embedded platforms (which are heterogeneous and not interchangeable - each

test case can only run on a subset of the available boards).

A brief, but unsuccessful, attempt was made to use CMake [74] and CTest [75] to improve this workflow. While it was relatively easy to get native C++ compilation for the host system and some FPGA compilation working, and CMake is able to support cross-compilers, CMake does not support building C/C++ code for multiple instruction sets or operating systems in the *same* source tree. Furthermore, while CTest has a "parallel test" option, it does not support heterogeneous clusters.

Parallelizing of compile and test steps is absolutely critical when working with a codebase this large. The current prototype system consists of 80 C/C++ executables, 5 C/C++ shared libraries, 5 C/C++ static libraries, 31 FPGA bitstreams, 5 CPLD bitstreams, 18 simulation testbenches, 21 formal verification testbenches, and 54 hardware-in-loop (HiL) tests. Some of these are run multiple times; for example `jtagd` is compiled for x86-64 Linux, x86-64 Windows, and ARM Linux, and some of the HiL tests are run several times on different hardware configurations.

The only remaining option was to create a custom tool. After some study of existing open and closed source build systems to find good ideas, it was decided that the system would be modeled most closely on Google's in-house cloud build system, Blaze [76]. The result was a system called Splash - the closest thing to the opposite of "blaze" that we could think of, in keeping with the classic "A's not B" naming tradition.

4.3.2 Architecture

The core data structure in Splash is the `BuildGraph`. This a directed acyclic graph of `BuildGraphNode` objects, with edges from output nodes to the files and objects used to create them. For example, a `CppExecutableNode` would have edges to one or more `CppObjectNode` objects, which would have edges in turn to `CppSourceNode` objects.

Each `BuildGraphNode` object has a virtual function called `Update()`. This function recursively calls `Update()` on all dependencies, then computes a SHA-256 hash which uniquely describes a particular state of the node (hashes of dependencies, target architecture, contents of source files, compiler flags, compiler version, etc).

Since the hash describes the full state of the node, if an output file exists for a given hash then there is no need to re-generate it. This leads to an obvious optimization: `Update()` first checks if the corresponding output exists in `splashbuild/$HASH/` and returns immediately if so. Otherwise, a batch job is submitted to the cluster to generate it. If any of the node's dependencies are still building, the job is marked as depending on those jobs so that it will not be scheduled until they complete.

The BuildGraph object contains two lists of nodes: targets and tests. The default "build" operation issues an `Update()` call to all "target" nodes, waits until all submitted jobs have completed, then issues an `Update()` call to all "test" nodes and waits for completion. Another common operation is to rerun a test, rebuilding any required objects beforehand. (A future version of the code may optimize performance by dispatching build and test jobs simultaneously, such that tests begin running as soon as their respective dependencies are built, rather than waiting for the entire build to finish.)

Splash is designed in a modular fashion so that many external dependencies can be freely swapped out. For example, rather than using raw `libslurm` calls to submit jobs to the cluster, an abstract `Cluster` object is passed to each `Update()` call. As of now the only supported back end is SLURM however support could easily be added for other cluster managers.

4.3.2.1 Splashfiles

In order to save development time and avoid writing a parser for a custom domain-specific language, Splash was written as a C++ library; the "build scripts" are actually C++ source files which link to `libsplashcore`. When the `splash` binary initializes, it creates `BuildGraphNode` objects for each build script, updating them as needed.

Once the build binaries are created, `splash` calls `dlopen(2)` to load each one, then finds the `CreateNodes()` export in each build binary and calls them. After all nodes have been created, `CreateEdges()` in each Splashfile is called to create all of the edges between them.

After the entire `BuildGraph` has been created, `splash` then traverses the graph

and performs the requested operations.

In retrospect this led to what could be considered excessively complex build scripts. A possible future enhancement would be to make a scripting front end for libsplashcore which provides the same power with an easier user interface.

Re-creating the `BuildGraph` each build, including parsing dependencies, also leads to a significant performance penalty with large codebases. We intend to explore solutions involving in-memory caching to improve setup performance in the future.

4.3.2.2 Targets

Since Splash was designed with multi-architecture support from the ground up, every target is associated with an architecture and there are separate binary directories for each architecture. For example, the `jtagd` source files are inputs to the "jtagd-x86_64-linux-gnu" and "jtagd-arm-linux-gnueabihf" targets, which produce the binaries `splashbuild / x86_64-linux-gnu / jtagd` and `splashbuild / arm-linux-gnueabihf / jtagd` respectively.

A single Splashfile can create binaries for any supported architecture with the same ease as the host architecture, creating the appropriate compiler object by calling `CppToolchain::CreateCppToolchainCached()` with the desired architecture name. By looping over a list of architectures, the same code can be compiled for multiple architectures in one build script, as shown below.

Listing 4.1: Creating a C++ executable targeting three different platforms

```
1 SPLASHFILE_EXPORT void CreateNodes(BuildGraph* graph)
2 {
3     //Save some useful variables
4     string srcdir =
5         GetDirOfFile(CanonicalizePath(__FILE__));
6
7     //Find all of the source files
8     vector<string> source_files;
9     FindFilesByExtension(srcdir, ".cpp", source_files);
```

```
10
11      //Generic compiler settings
12      CppCompileFlagList cflags;
13      cflags.push_back(new CppStandardFlag(
14          CppStandardFlag::CPP_STANDARD_11));
15      cflags.push_back(new CppOptimizationLevelFlag(
16          CppOptimizationLevelFlag::OPT_LEVEL_NONE));
17      cflags.push_back(new CppDebugInfoFlag);
18      cflags.push_back(new CppProfilingFlag);
19
20      //Linker settings
21      CppLinkFlagList lflags;
22      lflags.push_back(new CppLinkLibraryByTargetNameFlag(
23          "jtaghal", graph));
24      lflags.push_back(new CppLinkProfilingFlag);
25
26      //Multiarch build
27      string arches[]=
28      {
29          "x86_64-linux-gnu",
30          "x86_64-w64-mingw32",
31          "arm-linux-gnueabihf"
32      };
33      for(size_t i=0; i<sizeof(arches)/sizeof(arches[0]); i
          ++)
34      {
35          string arch = arches[i];
36
37          CppToolchain* toolchain =
38              CppToolchain::
                  CreateDefaultToolchainForArchitectureCached
```

```
                (
39              arch);
40
41          CppLinkFlagList lflags_arch = lflags;
42          string ftd2xx_path =
43              FindSharedLibrary("ftd2xx", arch);
44          if(!ftd2xx_path.empty())
45              lflags_arch.push_back(
46                  new CppLinkLibraryByPathFlag(ftd2xx_path)
                    );
47
48          //Define the actual library
49          CppExecutableNode::CreateCppExecutableNode(
50              graph,
51              source_files,
52              toolchain,
53              cflags,
54              lflags_arch,
55              "jtagclient",
56              true);
57      }
58  }
```

4.4 Test Cluster

4.4.1 Introduction

Development of Antikernel was performed by writing code for a module, followed by testing using a mix of simulation, formal methods, and HiL methods to ensure correctness. A large suite of unit tests (well over a hundred separate tests) was used to confirm that newly written code worked correctly and did not introduce any regression bugs or break existing functionality.

Since many of the tests take a nontrivial amount of time to run (often due to bandwidth limitations on the JTAG link), and many FPGA development boards were available, parallelizing across the hardware seemed like an obvious way to speed up the testing.

4.4.2 Software Construction

The current test cluster (nicknamed "White Dwarf") consists of twenty FPGA/CPLD dev boards of ten distinct models ranging from tiny 32-macrocell CPLDs to 200K logic cell FPGAs, managed using the Simple Linux Utility for Resource Management (SLURM, described in [77]). Each type of board is assigned to a separate cluster partition for scheduling and each physical board has its own hostname. For example, our two Digilent Atlys boards [78] have hostnames `atlys0` and `atlys1`; a test case compiled for the Atlys platform is then submitted to the `atlys` partition and will run on whichever node becomes available first.

Since SLURM expects to run jobs on compute nodes that have full operating systems, some tweaking (inspired by the IBM BlueGene's architecture, [59]) of the usual SLURM workflow was needed: the `slurmd` process for each FPGA compute node runs on an associated x86 service node (Fig. 4.2), which may host `slurmd`s for many FPGA compute nodes.

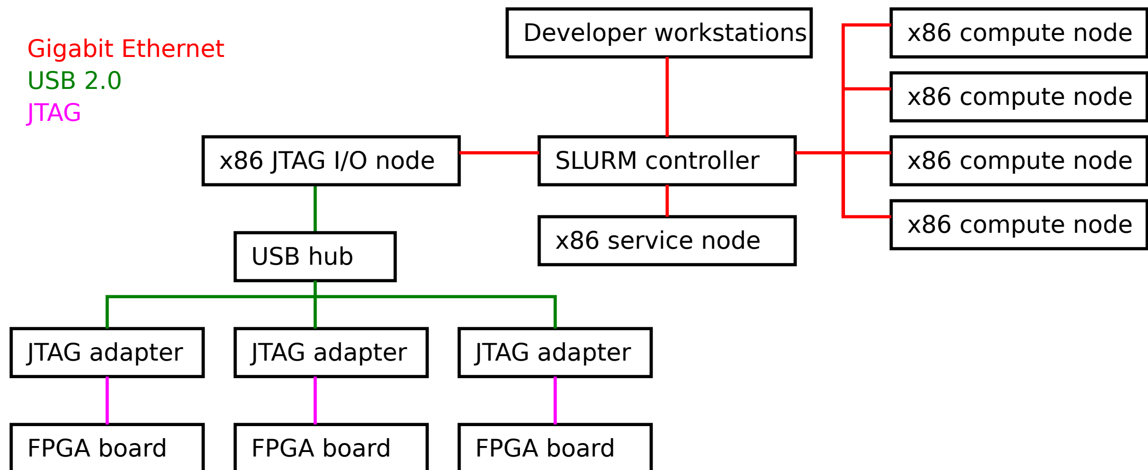

Figure 4.2: Test cluster architecture (simplified for clarity)

When a HiL test job starts, it begins to execute on the service node. The first thing to execute in our workflow is a C++ setup utility called `slurmwrapper`. This

tool takes a series of arguments from the `FPGAUnitTestNode` object in the test script specifying the bitstream to use, whether `nocswitch` or cosimulation are used, etc, as well as environment variables from SLURM providing information such as the node the current job was assigned to (so that it can disambiguate between multiple compute nodes sharing one service node).

After looking up the node ID, `slurmwrapper` looks in a text file to find the hostname and port of the `jtagd` for the associated compute node. This is currently a single x86 I/O node however we plan to create dedicated FPGA-based hardware for this purpose in the future.

If the test case specified a firmware image to load, `jtagclient` is then invoked to load it (this is generally the case for all tests except those testing the firmware loading algorithms themselves). Once firmware is loaded, `nocswitch` and `cosimbridge` are started if needed.

After all of the support servers are running, `slurmwrapper` then launches any debug tools (logic analyzer, NoC packet sniffers, gdbserver) needed for the run, and waits for them to report that initial configuration is done (for example, logic analyzer trigger conditions have been selected). The current `slurmwrapper` code assumes that the developer's workstation is also the service node, which prevents use of GUI debug tools on configurations where this is not the case, but a future version of the code will launch the debug tools via `ssh` on the node from which the job was submitted (presumably the developer's workstation).

At this point the test case itself is started, with the `jtagd` (if not using `nocswitch`) or `nocswitch` server's hostname and port, as well as the hostname/port to the UART console server (if any), as command-line arguments. Upon completion the exit code is noted and `slurmwrapper` waits for the debug tools to be closed. Before terminating, `jtagclient` is invoked a second time to reset the target board to a clean state.

4.4.3 Hardware Construction

The FPGA cluster was initially located on a desk in the author's office when it consisted of a handful of boards (Fig. 4.3) but quickly grew to the point of being

impractically large (Fig. 4.4). At that point, it was moved to a 19" rack (Fig. 4.5, 4.6) which also holds the I/O nodes and a small x86 compute cluster used for compiling the FPGA bitstreams.

When racking the cluster, the small boards were mounted to laser-cut 100 mm x 160 mm acrylic frames (Fig. 4.7) and inserted into a 3U Eurocard chassis, and all new designs were designed with the PCB in the same form factor (Fig. 4.8. The boards were oriented such that power, JTAG, and USB-serial connections were fed from the rear, and Ethernet from the front. The three largest boards (two Digilent Atlys and a Xilinx AC701) were placed horizontally in 1U server cases as they were more than 100mm long in the shortest dimension.

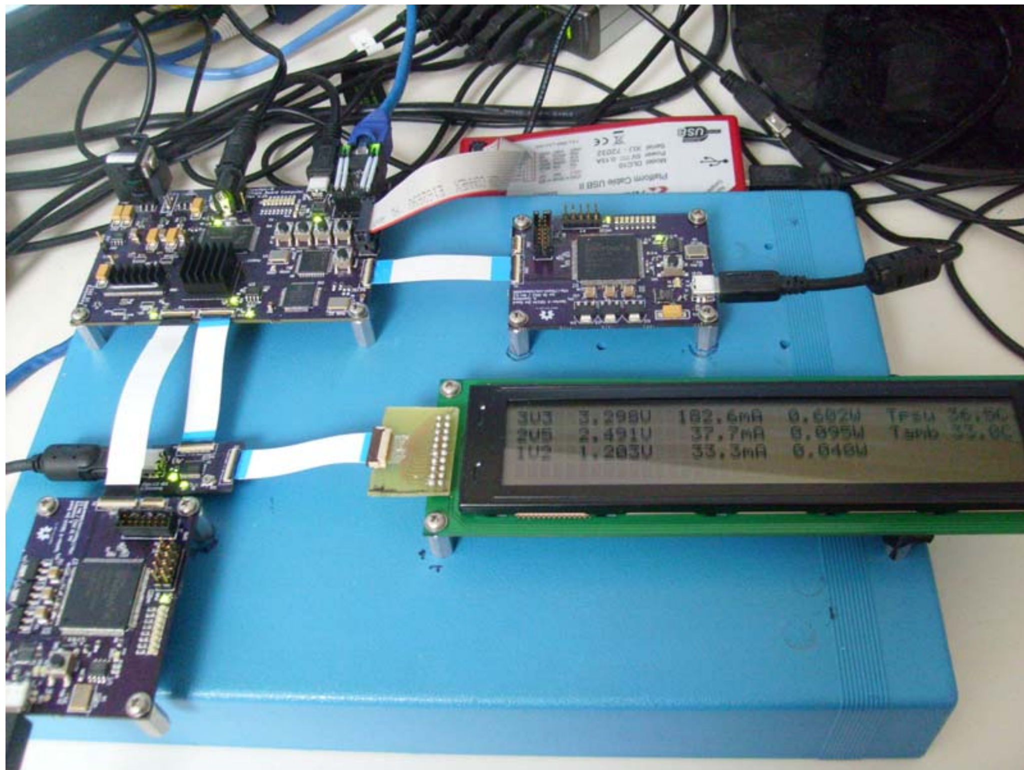

Figure 4.3: Small FPGA cluster on desk

Figure 4.4: Large FPGA cluster on desk

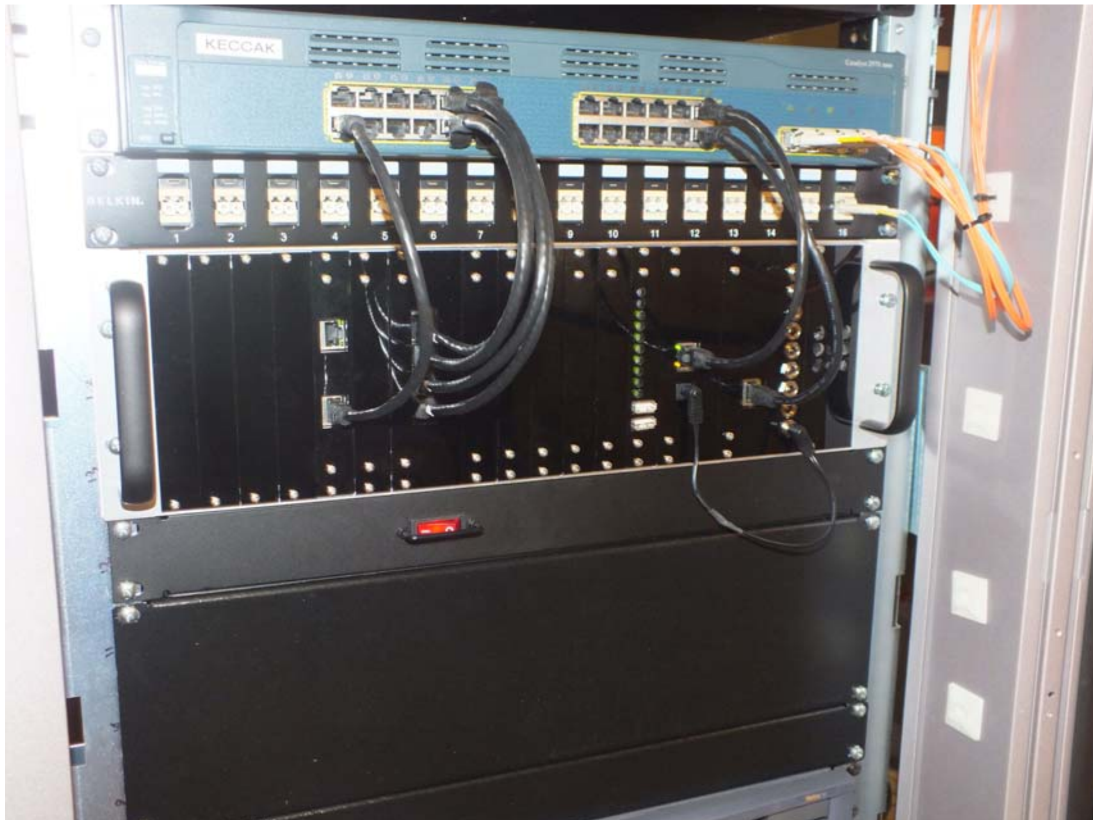

Figure 4.5: FPGA cluster on rack

Figure 4.6: FPGA cluster on rack

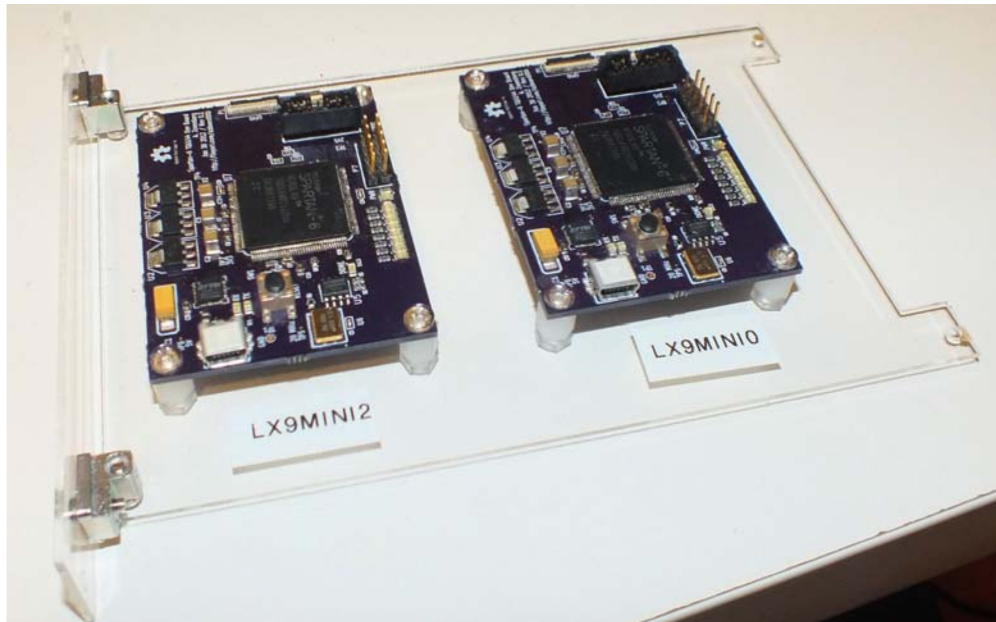

Figure 4.7: Acrylic shim FPGA card removed from rack

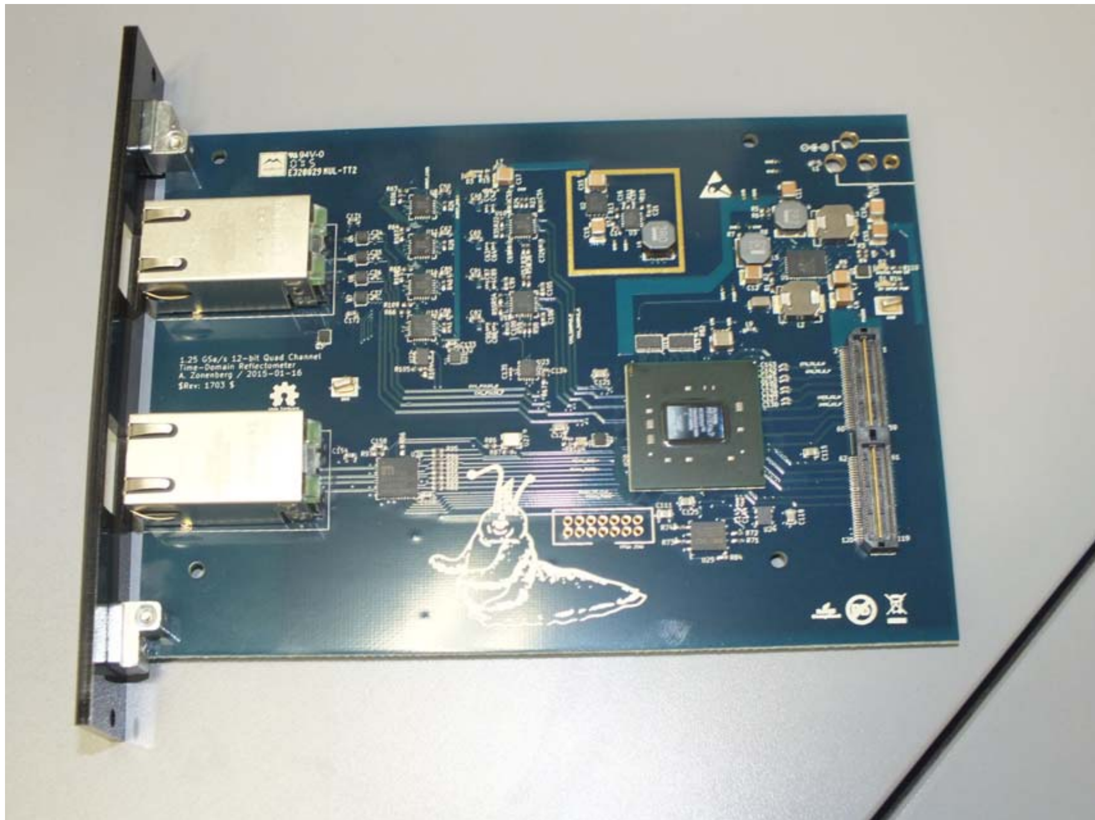

Figure 4.8: Custom-built FPGA card removed from rack

4.5 Constgen

One problem frequently encountered when co-designing an embedded hardware device and the associated software is managing large numbers of named constants (register IDs, bitfields, etc) across multiple languages. For example, in an Antikernel-based system each node on the RPC network has a series of named opcodes used in the Verilog implementation, but these same opcodes must also be used in C code by any application software wishing to use the services provided by that node. Furthermore, if the opcode values change for any reason (optimization of instruction decoding, for example) then the headers for both languages must be regenerated and the affected code recompiled.

The first attempt at solving this problem was a short shell script written in PHP which read a file containing tab-separated name-value pairs in either C or Verilog syntax (Listing 4.2), using any supported base, with or without length specifiers, and generated C and Verilog headers (Listings 4.3, 4.4). C++ style comments are supported in the input file but not propagated into the generated code.

Listing 4.2: Constgen input file

```
1 // @file
2 // @author Andrew D. Zonenberg
3 // @brief Opcode constants for SARATOGA OoB management
      port
4
5 OOB_OP_NOP                  8'h00
6 OOB_OP_GET_THREADCOUNT      8'h01
7 OOB_OP_CREATEPROCESS        8'h02
8 OOB_OP_MMAP                 8'h03
9 OOB_OP_ATTEST               8'h04
```

While the initial script was highly effective at solving the target problem, it was soon realized that allowing the named constants to be used in debug and build tools would allow even more powerful introspection capabilities. This resulted in

constgen being ported to C++ and integrated with Splash.

As of now the only supported output formats are C `#define` and Verilog `localparam` declarations, however we plan to support other formats (for example, C++ `enum`) in the future.

Listing 4.3: Constgen C/C++ output

```
1 ////////////////////////////////////////////////////////
2 // Generated automatically by CppConstgenNode from /nfs/
      home/azonenberg/Documents/local/programming/achd-soc/
      trunk/rtl/achd-soc/saratoga/
      SaratogaCPUManagementOpcodes.constants
3 // DO NOT EDIT
4 ////////////////////////////////////////////////////////
5
6 #ifndef
      CONSTGEN_INCLUDE_GUARD_SaratogaCPUManagementOpcodes
7 #define
      CONSTGEN_INCLUDE_GUARD_SaratogaCPUManagementOpcodes
8
9 #define OOB_OP_ATTEST            0x4     /* 8 bits wide */
10 #define OOB_OP_CREATEPROCESS    0x2     /* 8 bits wide */
11 #define OOB_OP_GET_THREADCOUNT  0x1     /* 8 bits wide */
12 #define OOB_OP_MMAP             0x3     /* 8 bits wide */
13 #define OOB_OP_NOP              0x0     /* 8 bits wide */
14
15 #endif //#ifndef
      CONSTGEN_INCLUDE_GUARD_SaratogaCPUManagementOpcodes
```

Listing 4.4: Constgen Verilog output

```
1 ////////////////////////////////////////////////////////
2 // Generated automatically by VerilogConstgenNode from /
      nfs/home/azonenberg/Documents/local/programming/achd-
```

```
    soc/trunk/rtl/achd-soc/saratoga/
    SaratogaCPUManagementOpcodes.constants
 3 // DO NOT EDIT
 4 ////////////////////////////////////////////////////////
 5
 6 localparam OOB_OP_ATTEST            = 8'h4;
 7 localparam OOB_OP_CREATEPROCESS     = 8'h2;
 8 localparam OOB_OP_GET_THREADCOUNT   = 8'h1;
 9 localparam OOB_OP_MMAP              = 8'h3;
10 localparam OOB_OP_NOP               = 8'h0;
```

4.6 Nocgen

4.6.1 Design Goals

During early development, the top-level Verilog modules were created by hand in a text editor. This did work, but manually instantiating all of the NoC routers, calculating PLL multipliers, assigning addresses to nodes, populating the ROMs for the name server, and creating pinout constraints was labor-intensive and error-prone. This was made worse by the fact that designs often needed to be re-targeted from one board to another due to outgrowing the capacity of the originally selected FPGA.[13]

We created a tool called `nocgen` to generate all of these elements automatically. As with `constgen`, it started out as a shell script but evolved into a C++ module for Splash. The inputs to `nocgen` are a script file (example: listing 4.6) and a board support file (example: listing 4.5).

By separating generic board data from the design-specific code, we eliminate the need to create a large number of nearly-identical constraint files for each of several unit tests running on the same hardware. Furthermore, porting of a given design to a different board which provides the same required interfaces (in this

[13]While targeting the highest-capacity FPGA in the cluster would largely eliminate this issue, it would result in much longer test run times since there would be no way to run tests in parallel. The best cluster utilization can be obtained if tests are balanced such that the total run time for the tests assigned to each cluster partition is about the same.

example, a clock and a UART without flow control) is a matter of changing one line in the Splashfile. Building the same design for several compatible boards can even be done in one Splashfile by looping over an array of board support file names (as was done for our indirect Flash programming firmware).

4.6.2 Operation

The first step in `nocgen` processing is to read and parse the board support file (`*.bsp`). This specifies the device being targeted and (in the case of JTAG) the position of the device in the scan chain. (As of now the chain position is unused, however in the future this will be used to allow automated program/debug tools to match firmware images to devices in multi-FPGA boards.)

Listing 4.5: Example NocGen board support file

```
1  ////////////////////////////////////////////////////////
2  // Global stuff
3
4  //The target device
5  target  xc6slx9-2tqg144 scanpos 0
6
7  //20 MHz clock input
8  clock   clkin       input   P55 std LVCMOS33    freq 20M
9
10 ////////////////////////////////////////////////////////
11 // UART (no flow control)
12
13 signal  uart_rx     input   P39 std LVCMOS33
14 signal  uart_tx     output  P40 std LVCMOS33
15
16 ////////////////////////////////////////////////////////
17 // SPI flash
18
19 signal  spi_cs_n    output  P38 std LVCMOS33
```

```
20 signal   spi_sck       output   P70 std LVCMOS33
21 signal   spi_data[0]   inout    P64 std LVCMOS33
22 signal   spi_data[1]   inout    P65 std LVCMOS33
23 signal   spi_data[2]   inout    P62 std LVCMOS33     pullup
24 signal   spi_data[3]   inout    P61 std LVCMOS33     pullup
```

The BSP also specifies the net name, pin location, and I/O standard of each signal connected to an FPGA pin, as well as additional metadata such as the frequency of a clock input and whether on-die terminators/pullups are required. This can be easily extended to specify setup/hold requirements for off-die devices however this is not currently implemented.

After the BSP is loaded, the main script (`*.nocgen`) is parsed. This contains a series of global commands (for example “top” which specifies the name of the top-level module being generated, or “namesrvr_key” which specifies the HMAC password used for signing writes to the name server).

Listing 4.6: Example NocGen script

```
1  // Top-level module name
2  top NetworkedUARTTestBitstream
3
4  //Pre-shared HMAC key for the name server to allow
       writing
5  namesrvr_key "ThisIsALongAndComplicatedPassword"
6
7  // Clock generation
8  pll MainClockPLL
9      inclk clkin
10     outclk clk_noc freq 100M duty 0.5 phase 0.0
11 endpll
12
13 // The UART
14 node uart NetworkedUART
```

```
15      input 1 uart_rx port uart_rx
16      output 1 uart_tx port uart_tx
17      rpc
18      dma
19  endnode
```

After the global commands come a series of blocks, each corresponding to a single module instance in the generated RTL. The most common block is the `node` block, which represents a node in the NoC. The user can specify the NoC hostname of the module, the Verilog module type, connections to the RPC and DMA network, and which top-level inputs and outputs from the BSP file should be connected to each port on the module.

Another common block type is the `pll` block, which instantiates a phase-locked loop (or equivalent clock synthesizer) and a global clock buffer for each output. All outputs are gated by the PLL lock signal to avoid glitching during startup.

Note that the the output clocks are specified as frequencies rather than multiply/divide values, and the input clock is identified by the net name from the BSP. This ensures that all PLL configuration values will be automatically recomputed if the design is re-targeted to a new board. The tool will iterate over possible multiply/divide values and report an error if the target frequencies cannot be created, or would require a VCO frequency outside the target device's allowed range. A minimum of one PLL is required for each design, in order to produce a signal called `clk_noc` which is used as the clock for all NoC nodes. Additional outputs on this PLL, and additional PLLs, may be instantiated as necessary for nodes requiring additional clocks.

In addition to the nodes specified in the script, two metadata nodes are automatically generated: `namesrvr`, a `NOCNameServer` object located at address 0x8000, and `sysinfo`, a `NOCSysinfo` object. These nodes are automatically populated with configuration data derived from the rest of the SoC: for example, the sysinfo service is configured to report the known frequency of `clk_noc`.

Once all of the `node` blocks have been read, the next step is to assign addresses to all nodes with either an RPC or DMA connection. This is currently done in a

greedy fashion: all nodes requiring /16 subnets are allocated addresses starting from 0x8000, in the order that they occur in the script file. All nodes requiring /14 subnets are then given addresses (rounding up to the nearest multiple of four as necessary), then /12, and continuing until all nodes have been given an address. (A future version of `nocgen` may incorporate packing optimizations to minimize the number of hops between nodes which communicate frequently, minimize the number of routers required by placing nodes with only one network connection at adjacent addresses, or strike a balance between these two goals.)

After all nodes have been given addresses, RPC and DMA routers are created in a quadtree. Routers with no children are automatically removed, however routers with one child remain to prevent network loops.[14]

A `JtagDebugController` is then attached to the root of the top-level router. We may add an option to disable this in the future if debugging of the generated design will not be required, as an additional layer of security or to provide a (tiny) reduction in resource usage.

After the full NoC topology is generated, all required ROM initialization files (for the name server, logic analyzers, etc) are produced. The top-level module Verilog code is generated and written to disk. Finally, the constraints file is created. As of now the only supported constraint format is UCF (for Xilinx ISE) however we intend to add support for Synopsys-style constraints in the future. This should allow designs to be ported to Xilinx's Vivado toolchain, or many other EDA vendors' toolchains, with no changes to the RTL or `nocgen` script file.

In the future we intend to refactor nocgen's handling of cryptographic key material to more clearly separate it from application code (perhaps by loading it from a separate file).

[14]If this were not done, a forwarding loop could be created. Consider the case of a router at 0x8000/16 directly under a router at 0x8000/12. A packet destined to 0x8005 would be forwarded up by the child router, down by the parent router, and continue circulating forever. We plan to address this issue in the future by configuring routers to drop packets coming in the upstream interface if the destination address is not in the local subnet, which would allow these redundant routers to be optimized out.

4.7 Nocsniff

4.7.1 Introduction

Since an Antikernel-based system is based on a network architecture, it is reasonable to believe that something analogous to existing network debugging tools could be used to aid in design and debugging. The obvious first choice is a packet sniffer.

We created a simple sniffer known as `nocsniff`, which sits on a NoC link between any leaf node and its parent router. Note that since the sniffer turns the point-to-point NoC link into a point-to-multipoint link, the assumptions made in section 5.3.2.2 are violated and thus none of the security proofs are valid for a system containing a sniffer; as a result once the Antikernel DRC tool is created it will flag any sniffer instances as violations of the isolation requirements. The sniffer is meant to be used during development and then disabled at synthesis time before deploying the system.

4.7.2 Capture Core

The capture core is a simple Verilog module which buffers incoming NoC frames in an internal block RAM based FIFO and streams in real time to a PC-based analysis application through the JTAG debug interface. It can be easily instantiated in `nocgen`, as shown in listing 4.7.

Listing 4.7: Instantiating a sniffer in nocgen

```
1  sniffer CpuSniff
2      target cpu
3  endsniffer
```

Due to the large difference in bandwidth between JTAG (66 Mbps raw signaling rate for a Xilinx 7-series FPGA, page 57 of [79]) and the NoC (25.6 Gbps for full duplex RPC+DMA at 200 MHz) it is clearly impossible to stream a full-speed packet burst at full line rate. The capture buffer in the sniffer (default size 2KB for each direction) must be large enough to handle the maximum expected traffic burst to avoid dropping packets; if this happens a dummy "overflow" frame is generated

so that the developer is made aware of the situation.

In order to reduce the risk of dropping interesting packets, we plan to implement filter rules in the capture core which will only capture packets matching the provided selector(s) and drop any others. This will allow efficient streaming capture of a few tens of Mbps of "interesting" traffic out of many Gbps of raw packets.

The current version of `nocsniff` does not capture DMA traffic, however this is planned for a future version.

4.7.3 Packet Viewer

The PC-side packet viewer is a gtkmm based application which connects to the target via a `nocswitch` server. FPGA unit test nodes in Splash can be configured to automatically start as many sniffers as required before launching the test case (listing 4.8).

Listing 4.8: Using a sniffer from Splash

```
1 auto node = FPGAUnitTestNode::CreateFPGAUnitTestNode(
2       /* stuff here */);
3 node->UseSniffer("CpuSniff");
4 node->UseSniffer("EthSniff");
```

The current viewer simply shows a live listing of packets updating in real time. Packets are automatically decoded as they arrive. First, the source and destination hostnames are looked up with the name server (caching mappings once looked up to avoid excessive load on the name server).

The viewer then examines the RPC packet type to determine which node's protocol to use. As a general rule the application-layer protocol is defined by the *source* node for interrupts and function returns, and the *destination* node for function calls. The `nocgen` file for the top-level module is then consulted to determine the Verilog module type corresponding to that hostname. If none is found, a simple hex dump of the packet is shown instead of a protocol decode.

If the Verilog module is successfully located, the packets are then decoded by comparing them against a protocol database generated from comments in the Verilog

source. In other words, the in-source documentation *is* the protocol decoder. This data-driven approach eliminates any need to waste developer time writing protocol decoders by hand.

At synthesis time a parser built into Splash scans each Verilog module for special comments using an extended form of Doxygen syntax, then builds the protocol database as an easily parseable text file. An example of these comments is shown in listing 4.9. (Note the reference to the `constgen` file in the @opcodefile directive, used to decode binary opcode values to human-readable mnemonics.)

Listing 4.9: Extended Doxygen syntax used by nocsniff

```
1  /**
2  @file
3  @author Andrew D. Zonenberg
4  @brief Block RAM based on-chip memory using the same API
       as NetworkedDDR2Controller.
5
6  @module
7  @opcodefile ../ddr2/NetworkedDDR2Controller_opcodes.
       constants
8
9  @rpcfn RAM_GET_STATUS
10 @brief Gets the current status of the RAM controller.
       Blocks until init is done.
11
12 @rpcfn_ok RAM_GET_STATUS
13 @brief    RAM status retrieved
14 @param    ready         d0[16]:dec Indicates if memory is
       fully initialized
15 @param    freepagecount d0[15:0]:dec Number of free RAM
       pages
16
17 @rpcfn RAM_ALLOCATE
```

```
18 @brief Allocates one page of memory.
19
20 @rpcfn_ok RAM_ALLOCATE
21 @brief    Memory allocated. The allocated page is zero-
      filled.
22 @param    addr d1[31:0]:hex Address of new memory page
23
24 @rpcfn_fail RAM_ALLOCATE
25 @brief    Out of memory.
26
27 @rpcfn RAM_CHOWN
28 @brief Change ownership of a page of RAM. The caller
      loses all rights to the page.
29 @param addr       d1[31:0]:hex      Address of page to
      chown
30 @param new_owner d2[15:0]:nocaddr New owner of page
31
32 @rpcfn_ok RAM_CHOWN
33 @brief    Ownership records updated.
34
35 @rpcfn_fail RAM_CHOWN
36 @brief Access denied. The caller probably didn't own the
      page.
37
38 @rpcfn RAM_FREE
39 @brief Free a page of RAM. The caller loses all rights to
      the page.
40 @param addr d1[31:0]:hex Address of page to free
41
42 @rpcfn_ok RAM_FREE
43 @brief    Memory freed.
```

```
44
45 @rpcfn_fail RAM_FREE
46 @brief Access denied. The caller probably didn't own the
       page.
47
48 @rpcint   RAM_WRITE_DONE
49 @brief    Write committed.
50 @param    len   d0[15:0]:dec Length of the written data
51 @param    addr  d1[31:0]:hex Address of written data
52
53 @rpcint   RAM_OP_FAILED
54 @brief    Access denied.
55 */
```

As a result of using the `nocgen` file for hostname → module type lookups, there is currently no support for protocol decodes of software-based services, as they do not exist as nodes in the nocgen file. A planned future fix is to use the `nocgen` file to find address ranges associated with a SARATOGA processor. The viewer can then use SARATOGA's `OOB_OP_ATTEST` call to look up the signature for a given process ID in that range, consult the Splash database to find the `SignedFirmwareNode` associated with that hash, and backtrack to find the associated `CppExecutableNode` object. This would then allow scanning of the relevant source files for Doxygen comments and protocol decoding off of them.

In the screenshot below (Fig. 4.9) the unit test `testcase` queries the CPU for the number of thread contexts, then creates a process from the signed ELF image at physical address 0 of the device `rom`. The application (`echofw`) then starts up, allocates and maps memory pages for stack and `.bss`, then registers itself with the name server.

Note that the reverse name lookups for `echofw` appear to violate causality, since the hostname is looked up successfully before it is even registered! This is a consequence of latency in the link: traveling over JTAG to the `jtagd` server, then over TCP to `nocswitch`, then over TCP to the viewer incurs one or two milliseconds

Start Clear Filter

Time	Source	Destination	Type	Length	Info	Summary
0.362s + 5829 clocks	testcase	cpu	RPC call	16	OOB_OP_GET_THREADCOUNT()	Get number of thread contexts
0.362s + 5837 clocks	cpu	testcase	RPC return: success	16	OOB_OP_GET_THREADCOUNT(pcount=32)	Thread count retrieved
0.364s + 48303 clocks	testcase	cpu	RPC call	16	OOB_OP_CREATEPROCESS(phyaddr=rom:00000000)	Create process
0.369s + 568 clocks	cpu	testcase	RPC return: success	16	OOB_OP_CREATEPROCESS(nocaddr=0x8060/echofw, tid=0)	Process created
0.369s + 1115 clocks	echofw	namesrvr	RPC call	16	NAMESERVER_FQUERY(hostname="ram")	Looks up the address for a given hostname.
0.369s + 1182 clocks	namesrvr	echofw	RPC return: success	16	NAMESERVER_FQUERY(addr=0x8002/ram)	Address found.
0.369s + 2321 clocks	echofw	ram	RPC call	16	RAM_ALLOCATE()	Allocates one page of memory.
0.369s + 2383 clocks	ram	echofw	RPC return: success	16	RAM_ALLOCATE(addr=0x00000000)	Memory allocated. The allocated page is zero-filled.
0.369s + 2717 clocks	echofw	cpu	RPC call	16	OOB_OP_MMAP(phyaddr=ram:00000000, vaddr=0x4000f800)	Write to page table
0.369s + 2724 clocks	echofw	cpu	RPC call	16	OOB_OP_MMAP(phyaddr=ram:00000000, vaddr=0x4000f800)	Write to page table
0.369s + 2738 clocks	cpu	echofw	RPC return: success	16	OOB_OP_MMAP()	Page table updated
0.369s + 2745 clocks	cpu	echofw	RPC return: success	16	OOB_OP_MMAP()	Page table updated
0.369s + 3185 clocks	echofw	ram	RPC call	16	RAM_ALLOCATE()	Allocates one page of memory.
0.369s + 3247 clocks	ram	echofw	RPC return: success	16	RAM_ALLOCATE(addr=0x00000800)	Memory allocated. The allocated page is zero-filled.
0.369s + 3581 clocks	echofw	cpu	RPC call	16	OOB_OP_MMAP(phyaddr=ram:00000800, vaddr=0x4000e800)	Write to page table
0.369s + 3588 clocks	echofw	cpu	RPC call	16	OOB_OP_MMAP(phyaddr=ram:00000800, vaddr=0x4000e800)	Write to page table
0.369s + 3602 clocks	cpu	echofw	RPC return: success	16	OOB_OP_MMAP()	Page table updated
0.369s + 3609 clocks	cpu	echofw	RPC return: success	16	OOB_OP_MMAP()	Page table updated
0.369s + 5995 clocks	ram	echofw	RPC interrupt	16	RAM_WRITE_DONE(len=8, addr=0x00000b80)	Write committed.
0.369s + 7316 clocks	echofw	namesrvr	RPC call	16	NAMESERVER_LOCK()	Acquire the write mutex
0.369s + 7378 clocks	namesrvr	echofw	RPC return: success	16	NAMESERVER_LOCK()	Write mutex granted for 2^20 clocks
0.369s + 9496 clocks	ram	echofw	RPC interrupt	16	RAM_WRITE_DONE(len=8, addr=0x00000780)	Write committed.
0.369s + 16154 clocks	echofw	namesrvr	RPC call	16	NAMESERVER_HMAC(blocknum=0, hmac=0x2fc68cbb6032bc15)	One block of the HMAC signature for a pending request
0.369s + 16217 clocks	namesrvr	echofw	RPC return: success	16	NAMESERVER_HMAC()	HMAC block accepted
0.369s + 22526 clocks	echofw	namesrvr	RPC call	16	NAMESERVER_HMAC(blocknum=2, hmac=0x595127af009a9855)	One block of the HMAC signature for a pending request
0.369s + 22589 clocks	namesrvr	echofw	RPC return: success	16	NAMESERVER_HMAC()	HMAC block accepted
0.369s + 24218 clocks	echofw	namesrvr	RPC call	16	NAMESERVER_HMAC(blocknum=4, hmac=0xa5575cbe16e8e730)	One block of the HMAC signature for a pending request
0.369s + 24281 clocks	namesrvr	echofw	RPC return: success	16	NAMESERVER_HMAC()	HMAC block accepted
0.369s + 25919 clocks	echofw	namesrvr	RPC call	16	NAMESERVER_HMAC(blocknum=6, hmac=0x0c696ef06f4f14ef)	One block of the HMAC signature for a pending request
0.369s + 25982 clocks	namesrvr	echofw	RPC return: success	16	NAMESERVER_HMAC()	HMAC block accepted
0.369s + 26882 clocks	echofw	namesrvr	RPC call	16	NAMESERVER_REGISTER(hostname="echofw")	Request name registration
0.369s + 82424 clocks	namesrvr	echofw	RPC return: success	16	NAMESERVER_REGISTER()	Name registration successful

Figure 4.9: Example nocsniff capture of application startup on SARATOGA

of latency. By the time the packets are parsed, a reverse name lookup is issued, and the query packet reaches the name server, upwards of 7 ms have elapsed and the hostname registration has completed.

4.8 RED TIN Logic Analyzer

4.8.1 Motivation

State-level debugging of large FPGA designs, or those which require access to complex off-chip peripherals, is typically not possible to do in simulation. While connecting a standard logic analyzer to the board is possible, this typically requires a large number of I/O pins and bulky, expensive high-speed connectors. The usual solution to this problem is an internal logic analyzer - an FPGA module which observes a number of signals, writes them to on-chip block RAM when a trigger event occurs, and connects to the developer's PC via some interface (typically JTAG).

All major FPGA vendors provide such tools, such as Xilinx's ChipScope ([80], included with the expensive professional versions of their toolchain or available for purchase separately) and Altera's SignalTap ([81], included at no extra cost in all versions of their design tools).

Unfortunately, neither of these tools could be used in the development of Antikernel. SignalTap is specific to Altera devices, and our lab is entirely stocked with Xilinx boards. ChipScope can only be used with Xilinx's proprietary programming cable (or license-built third-party ones), which costs over ten times as much as the custom JTAG adapters we built for use with libjtaghal. While our lab does have one Xilinx platform cable, constantly reaching behind the equipment rack to move it from one board to another would be impractical. Buying a sufficient number of them (and the associated software licenses) to support each FPGA board would be prohibitively expensive.

Furthermore, unit testing of some modules can be made significantly easier if internal state is observable. While ChipScope does provide a scripting API of sorts, the poorly documented Tcl example code did not appear to have much potential for building a solid, reliable unit testing suite.

The last nail in ChipScope's coffin, for our lab, was the unstable driver. The programming adapter worked correctly for a while after being plugged in, but after a period which ranged from hours to days it would stop responding. Unplugging and replugging the cable, or moving to a different USB port, was of no help; the only solution which would restore proper functionality was a full reboot of the host computer.

The end result was a tool which became known as RED TIN. (The name has no meaning, after a debate among research group members failed to produce any consensus the author picked an adjective and noun at random from a dictionary.)

4.8.2 Capture Core

On-chip data capture is handled by a `RedTinNocWrapper` instance. This module contains NoC transceivers, interface logic, and a `RedTinLogicAnalyzer` instance, which handles the actual data capture.

The `RedTinLogicAnalyzer` module contains a circular buffer of configurable width and depth for capture data, and a second buffer for timestamp values. When the LA is idle it writes to the capture buffer through a 3-stage flipflop shift register (to provide a very large setup margin and prevent the LA from dragging the DUT

into sub-optimal placement locations) every clock cycle, setting the timestamp to 1.

When a trigger event occurs, the LA enters capture mode. During capture mode it compares the second and third samples in the shift register, writing the incoming data to the capture buffer if they are different. A 32-bit repeat counter increments every cycle that the values are the same. When the values differ it is written to the timestamp buffer and reset to zero. This provides simple (run-length encoding) compression to allow long periods of inactivity to be stored in fewer words of the memory.[15]

The actual trigger logic takes advantage of the shift register LUT function found in most modern FPGAs (Fig. 4.10). This mode allows the SRAM cells of the LUT to be connected end to end as a shift register, with dedicated input/output, clock, and shift-enable pins. In order to allow smaller shift registers than the maximum (32 bits, in the case of a Xilinx Spartan-6 or 7 series device) the combinatorial inputs of the LUT can be used as an address selector, to connect the LUT output to any bit within the shift register.

Although the shift register address is typically held constant in order to create a fixed-length shift register, it is allowed to change at run time. Rather than thinking of the resulting structure as a variable-length shift register, it is instructive to think of it as a LUT whose contents can be shifted one bit at a time on demand (Fig. 4.11). By connecting the shift register input and clock to a control bus, and providing a means for gating the LUT output to avoid glitching, we get a 5-input combinatorial function whose behavior can be changed at run time by shifting a new truth table in one bit at a time over 32 cycles.

This capability is sometimes referred to as "poor man's partial reconfiguration", or "poor man's PR". It has numerous disadvantages over true partial reconfiguration (for example, it cannot change routing between LUTs, or alter the behavior of any hard IP blocks). It also has major advantages for our purpose: it uses general fabric routing and provides much more flexibility in layout than typical PR flows (which typically reconfigure a large rectangular tile of the FPGA all at once, losing

[15] In addition to capturing a new sample when the values change, a timeout function forces a new sample to be captured after a certain number of clock cycles, to prevent the LA from hanging for more than a few seconds if the device under test stops toggling signals for some reason.

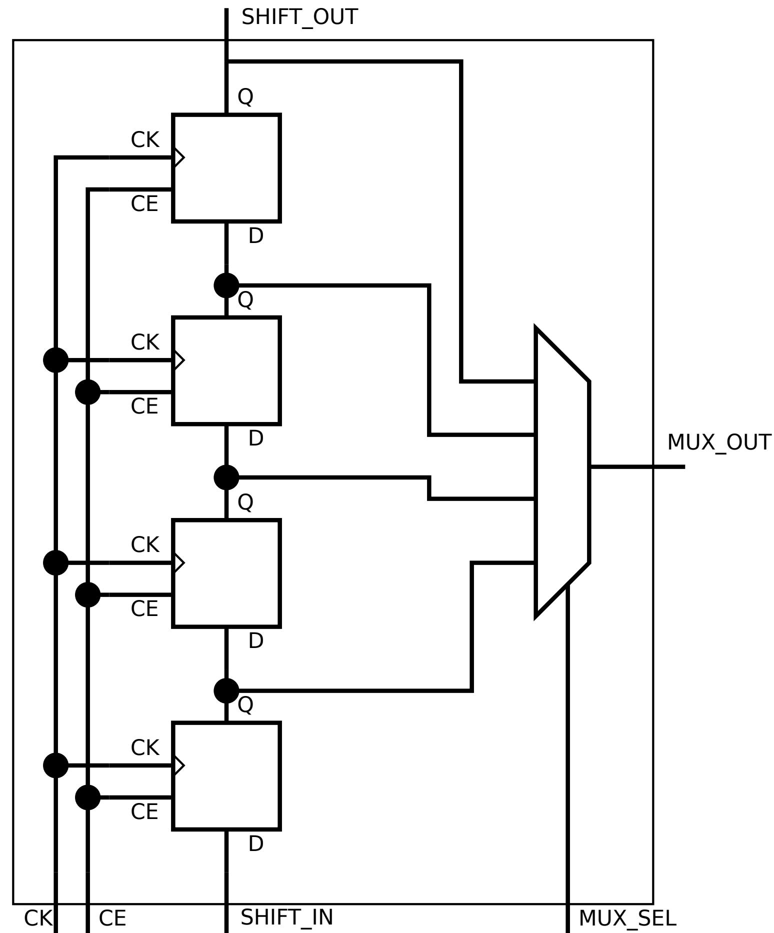

Figure 4.10: Shift register LUT (only four bits shown for clarity)

any state held in it). Vendor PR flows normally require the configuration bitstream for each reconfigurable region to be generated with their full development toolchain. The "poor man's PR" approach, on the other hand, can generate the LUT truth tables via arbitrary code since the addressing of bits in the shift-register LUT is well documented.

We use this reconfiguration technique in our trigger logic to create triggers for N input channels with N/2 reconfigurable LUTs. One input of each LUT is unused and tied low; the other four are connected to the current and previous state of two signals in the LA's input register. By generating an appropriate truth table in the PC-based control application and sending it to the LA via NoC frames, we can cause each LUT to return true when the desired trigger condition has been met. (Since both the current and previous state are stored, we can trigger on changes as well as

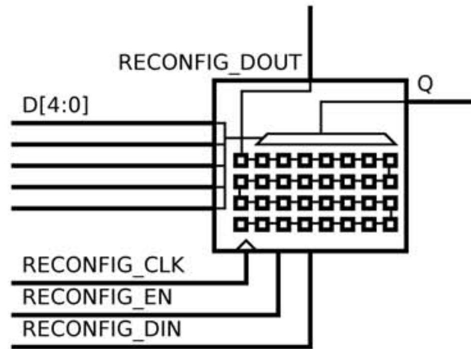

Figure 4.11: Shift register LUT configured for partial reconfiguration

specific values.) The outputs from all of the reconfigurable LUTs are then ANDed together, gated with an enable signal, and fed to the LA state machine to start the capture process.

In the future, we plan to combine the edge detector used for the run-length encoder with the trigger logic by using the fifth reconfigurable LUT input to select between two different truth tables. This should slightly reduce the number of LUTs consumed by the LA core.

4.8.3 Control Software

RED TIN's control software is a basic waveform viewer (Fig. 4.12) which displays data acquired from the capture core. It connects to a `nocswitch` server specified on the command line (or pops up a dialog box if none is specified) and connects to the requested capture core.

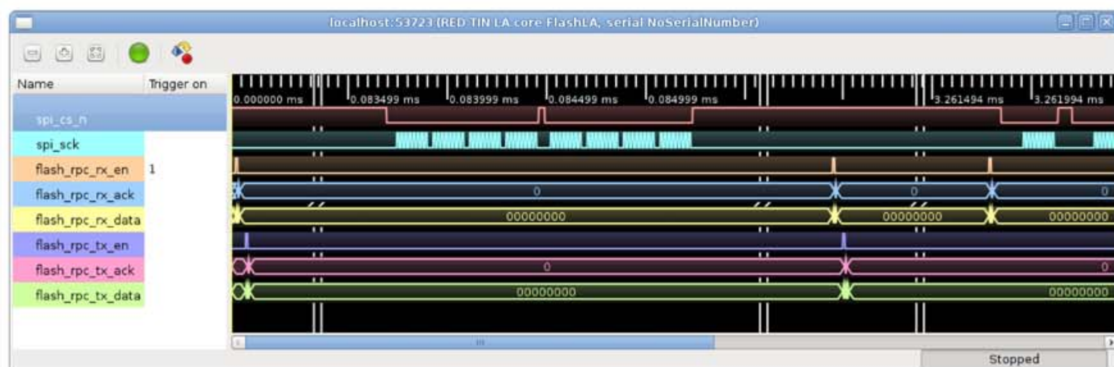

Figure 4.12: Screenshot of RED TIN control application

During initialization, it connects to the capture core and reads a small meta-

data ROM containing signal names and widths (typically generated by `nocgen`) via DMA messages. This is used to populate the list of channels in the user interface, as well as when generating trigger conditions.

In order to capture data the user simply enters a series of values into the "trigger on" column and presses the "start" button; a capture then begins when all of the trigger conditions are matched. The trigger conditions may be in any legal C or Verilog number format (enumerated constants are not yet supported). Entering no triggers results in an immediate, unconditional trigger.

When the button is pressed the software converts the trigger conditions into a series of truth tables, one per two channels, describing the conditions under which the trigger is matched. These truth tables are then sent in DMA messages to the capture core, and the control software then blocks until an RPC interrupt indicating a trigger event is received.

When a trigger event occurs, the viewer issues a series of DMA reads to read back the contents of the capture buffer. The resulting waveform is unpacked into arrays for each signal (using the width information read from the metadata ROM) and displayed to the user.

We spent a substantial amount of time tweaking the visualization to make it more attractive and useful. Although we have not conducted any sort of formal user study, selecting contrasting colors for adjacent signals (rather than displaying them all in one shade of green or blue as seems to be the *de facto* standard) made it much easier for the author to quickly follow signals in large, complex captures. We also collapse long periods with no transitions into a small space, so that captures with infrequent events can be scrolled quickly.

The viewer is a modular architecture and can support other signal sources beyond RED TIN through a library we call `libscopehal`. We have partial support for Rigol's DS1102D oscilloscope, although their USB interface appears to be somewhat unreliable and we had difficulty getting deep-memory captures to work properly.

CHAPTER 5
Security Analysis

5.1 Threat Model

Antikernel's primary goal is to enforce compartmentalization between userspace processes, and between userspace and the operating system. The focus is on damage control, rather than preventing initial penetration (although existing state-of-the-art techniques can be used to harden applications against this).

In particular, Antikernel aims to make all of the following scenarios impossible, given that an attacker has gained unprivileged code execution within the context of a userspace application or service: download a backdoor payload and configure it to run after system restart, modify executable code in memory or persistent storage, intercept/spoof/modify system calls or IPC of another process, read or write private state of another process, or gain access to handles belonging to another process by any means.

The attacker is assumed to be remote so physical attacks are not considered. Existing antitamper techniques can, of course, be used along with the Antikernel architecture to produce a system with some degree of robustness against physical tampering; but it is important to note that no physical security is perfect and an attacker with unrestricted physical access to the system is likely to be able to penetrate any security given sufficient time and budget.

For the scope of this chapter, we consider an abstract RTL-level model of the system with ideal digital signals in which it is not possible for the state of one register or input pin to observe or modify the state of another except if they are connected through combinatorial logic in the RTL netlist.[16]

[16] In practice it is sometimes possible for this property to be violated (for example by DRAM read disturbance, as described in [82, 83]). Such attacks exploit subtle layout-vs-schematic (LVS) mismatches which are not picked up by automated tools. While detecting these bugs is certainly an important task, it requires a level of solid-state physics better suited to a dissertation on electrical engineering so we leave it as an open research problem and focus on the computer science problem: ensuring safety of the pre-layout netlist.

5.2 Requirements

An Antikernel system must provide unforgeable attribution for all NoC communication between nodes and ensure that all NoC communications between nodes are authentic and unmodified. In order to prevent this security from being trivially bypassed it must also prevent information from leaking between nodes through channels other than the NoC (either through additional wires, or through state leakage within one node in violation of security policy). For example, the memory controller node must ensure that all memory is zeroized before allocating and that no node can read or write memory it does not own. A multithreaded CPU must prevent any thread from affecting the state of another except by means of sending it an RPC or DMA message.

While hardware nodes are implicitly trusted to be authentic (since they are physically immutable under this threat model and we assume the supply chain is secure), the functionality of software-based nodes is not known *a priori*. In order to authenticate a software-based node, we must have some way of uniquely identifying the code it is running.

The simplest way of doing this is to enforce mandatory code signing in the executable loader. If the system is incapable of marking any memory as executable at run time, or writing to executable pages, then we know that the currently executing code is the same as that which was loaded from ROM. If the signature on this code was verified when the executable was loaded, then we know that the currently running code is the same code which was signed. All we must do at this point is store the computed signature/hash for each executable as it is loaded and provide a means to securely query it at run time. We then have most of the core functionality required for remote attestation [84], but between NoC nodes rather than physically separate computers.

5.3 Prototype Verification

5.3.1 Methodology and Goals

We have performed fairly extensive verification on the current prototype system using a mix of simulation, HiL testing on our test cluster (section 4.4), and

formal methods. All tests are fully automated and run by Splash (section 4.3) before every commit.

The general verification methodology begins by creating at least one HiL test for each module or subsystem being verified, supplemented by simulation tests in some cases to speed the design cycle. The focus of this level of verification is catching obvious bugs that occur when the module is used as intended. 100% of the modules in the project receive at least this level of verification.

In addition, the most critical subsystems are provably verified against a formal model of the desired behavior. The choice of modules to verify is determined by several factors including their importance to the security model (the worse the impact of a bug, the more important provable correctness is) and their complexity (simpler modules are easier to prove correct).[17]

So far we have proven correctness properties on all of the interconnect and communications modules. (Details of exactly which properties were proven are described in the following sections.) Many subtle bugs were found and fixed in this process which would likely have gone undetected otherwise. One bug in the DMA transceiver's retransmit logic, triggered by NoC congestion at a precise time relative to the time the packet was sent, caused a glitch on the `dma_tx_en` signal which could have resulted in packet data being interpreted as headers and allowed spoofing. Multiple bugs in the name server's error handling were found which could have led to a hang (denial of service) when returning error codes from malformed commands.

5.3.2 Assumptions

No mathematical proof can be complete without stating the assumptions which it relies on. For something as complex as a correctness proof for a major subsystem of an operating system, the necessary groundwork is substantial. We describe the specific assumptions for our proofs in the following sections.

[17] While full verification of the entire implementation is of course desirable, and a goal we are working toward, it would likely require many additional man-years and would be impractical to do within the scope of a single doctoral dissertation. Additionally, several components of the design are still being optimized and improved, making a correctnesss proof of the current code a waste of time.

5.3.2.1 Physical Synthesis and Place-And-Route

All of the low-level proofs of correctness were performed on post-synthesis RTL netlists using `yosys`. We assume correctness of the temporal induction proof system in `yosys` and the SAT solver (MiniSAT by default, although different solvers can be configured at run time). In other words, we assume that if the post-synthesis netlist is inconsistent and one or more of the assertions in the netlist are violated, that the solver will correctly detect the error and declare the proof to not hold.

These proofs are only valid down to the RTL level for the current prototype. The actual synthesis and place-and-route (PAR) of the prototype systems were performed using Xilinx's proprietary tools; correctness of these tools and the FPGA silicon is assumed.[18]

A future implementation of our architecture targeting ASIC could be verified much more extensively at this level since it is possible to extract netlists from polygon-level layout and perform equivalence checking against the post-synthesis netlist. If `yosys` was used as the synthesis tool, assertion checking could be done on the actual post-synthesis RTL netlist, followed by equivalency checking against the post-PAR technology mapped netlist. This could be combined with post-fab micrography on selected dies, and image analysis to compare the manufactured silicon against the post-PAR polygons, to provide an extremely high degree of assurance that the actual manufactured silicon obeys the proven properties.

5.3.2.2 Network Topology

Most of the proofs later in this chapter assume that the network topology of the specific SoC being analyzed is "sane", in other words it adheres to the architecture described in the previous chapters.

More formally, it is assumed that the RPC network consists of a series of `RPCv2Router` objects connected in a quadtree, with nodes under the routers. All of these nodes must connect to the RPC network with either an `RPCv2RouterTransceiver` or an `RPCv2Transceiver` object, configured as a leaf node (with the exception of

[18] Since the FPGA microarchitecture is undocumented, equivalence checking on the actual FPGA bitstream would not be possible without extensive reverse engineering of the silicon. While an interesting problem, and one that the author is actively working on [85], it is beyond the scope of this thesis.

the multithreaded CPUs such as SARATOGA, which are treated as multiple nodes under one router for the scope of the network-level proofs).

The DMA network is assumed to consist of a series of `DMARouter` objects connected in a quadtree, and nodes under the routers. Each node must connect to the DMA network with a `DMATransceiver` or `DMARouterTransceiver` object, configured as a leaf node. As with the RPC network, multithreaded CPUs are a special case and handled separately.

Furthermore, it is assumed that if any node connects to both networks it uses the same address on each, and that there is no information flow between nodes outside of the NoC (for example, by wires that do not pass through a NoC router, or off-die paths on the printed circuit board).

All of the test SoCs created as part of this thesis were generated by the `nocgen` tool (Sec. 4.6), which is intended to enforce these requirements for the top-level module, however its correctness has not yet been proven. Verifying that any particular generated source file (and the instantiated modules) meet these requirements is relatively easy to do by inspection. In the future we intend to create a DRC tool which uses `yosys` to parse the actual RTL source for a particular SoC and verifies that all of the on-chip topology requirements are met.

5.3.3 RPC Network

5.3.3.1 Layer-2 Packet Handling

All four combinations of RPC transceivers (node or router at each end) for a layer-2 link were formally verified using `yosys`. The goal was to prove that regardless of what traffic enters a transceiver, it will always be interpreted properly at the other end, and there is no possibility of causing data to be interpreted as headers or vice versa.

Each test case instantiates one transmitter and one receiver of the appropriate types, as well as testbench code. `yosys` is then run on each testbench to synthesize to RTLIL intermediate representation, followed by invoking the SAT solver to prove the assertions in the testbenches. If the solver declares that all assertions pass, the proof is considered to hold.

While the testbenches are all slightly different due to the differences in interface between router and client transceivers, their basic operation is the same. When the test starts, all outputs are in the idle state and remain so in the absence of external stimuli. When a transmit is requested, the test logic stores the signals at the transceiver's inputs and asserts that the same data exits the receiver a fixed time later. The test also verifies that attempts to transmit while the receiver is busy block until the receiver is free (thus preventing dropped packets) and that the transceiver fully resets to its original state after sending a packet.

The testbenches may be found in the provided source distribution under /trunk/tests/RPCv2XXToYYChannelTest/ where XX and YY may be either "Client" or "Router".

5.3.3.2 Layer-3 Packet Routing

It is also necessary to prove that packets are correctly forwarded to the desired layer-3 destination by routers.

We can map the quadtree directly to routing addresses by allocating two bits of the address to each level of the tree. Each router simply checks if the high bits match its subnet, forwards out the downstream port identified by the next two bits if so, and otherwise forwards out the upstream port. It is easy to see by inspection that this algorithm will always lead to a correct tree traversal.

Since correct routing at the node level combined with a valid quadtree topology implies correct routing at the network level, and the proofs in the previous section show that link-layer forwarding is correct, the proof for correct end-to-end forwarding thus reduces to showing that `RPCv2Router` correctly implements the routing algorithm.

In order to reduce the search space for the solver the automated proof was split into two halves, split at the packet buffer SRAM.

The first half of the proof, `RPCv2RouterIngressQueueTest`, is responsible for showing that when an incoming packet arrives it is written correctly to SRAM, and the transceiver is kept in the "busy" state until the packet has been forwarded. The Verilog for this proof is located at /trunk/tests/RPCv2RouterIngressQueueTest.

The second half of the proof, `RPCv2RouterExitQueueTest`, is responsible for showing that, once a packet has arrived and been written to SRAM, it is forwarded to the correct location. The Verilog for this proof is located at /trunk/tests/R-PCv2RouterExitQueueTest.

5.3.4 DMA Network

5.3.4.1 Layer-2 Packet Handling

All four combinations of DMA transceivers (node or router at each end) for a layer-2 link were formally verified using `yosys`. The goal was to prove that regardless of what traffic enters a transceiver, it will always be interpreted properly at the other end, and there is no possibility of causing data to be interpreted as headers or vice versa.

Each test case instantiates one transmitter and one receiver of the appropriate types, as well as testbench code. `yosys` is then run on each testbench to synthesize to RTLIL intermediate representation, followed by invoking the SAT solver to prove the assertions in the testbenches. If the solver declares that all assertions pass, the proof is considered to hold. The basic structure of the proofs is similar to those for the RPC network.

It is important to note that due to the large maximum packet size (512 words) it was not possible to run the DMA network proofs to a steady state, thus the proof is not complete. The current proof is artificially limited to examining state for the first 64 cycles and shows that no assertions are violated during this time. Running the solver on each proof takes about ten minutes on a single CPU core and uses between three and ten gigabytes of RAM; given a sufficiently large amount of CPU time and RAM there is no reason why the proof cannot be extended until a steady state is reached. We plan to explore integrating a parallel SAT solver with `yosys` in order to run the full proofs on a cluster in the future, however this is beyond the scope of this thesis. We are also working with the developers of `yosys` to explore whether better performance can be obtained by using a SMT solver on a more abstract form of the netlist. Early results are encouraging however more work is required.

The testbenches may be found in the provided source distribution under /trunk/tests/DMAXXToYYChannelTest/ where XX and YY may be either "Client" or "Router".

5.3.4.2 Layer-3 Packet Routing

Aside from the transceivers, the majority of the DMA network router is identical to that of RPC, instantiating the same modules. The only changes were adding an additional SRAM buffer and multiplexer for each port since the DMA transceiver has separate memory channels for headers and packet bodies, as opposed to the single channel for RPC.

We believe that these changes are sufficiently non-intrusive that the probability of them containing a security-critical bug is very low. Although a full part-wise verification of the router (as was done for RPC) is certainly possible, and should be performed before an Antikernel system is actually deployed in a critical application, we believe that doing so at this stage of development would be an inefficient use of research time.

5.3.5 Name Server

Even if the layer-3 links between nodes are secure and packet misrouting is impossible, if a rogue node can somehow trick a target node into sending its traffic to the wrong address then MITM attacks can still occur. This can be done by causing the name server to report the wrong address when queried.

Avoiding this requires proving two properties: First, the name server must always return the correct entry (if one exists) from its table when queried, or an error if none exists. Second, the name server must only insert names into the table, or remove them, if authorized by system security policy.

The top-level `NOCNameServer` module consists of several RAM blocks, an RPC transceiver, an HMAC-SHA256 engine, a "target matching" system (which compares outputs of the RAMs against a value being searched for), a mutex, and the main control state machine.

We assume correctness of the RAM and prove correctness of the transceiver separately. The mutex, target matching logic, and HMAC cores have undergone

conventional validation but do not have correctness proofs as of this writing. Several correctness properties have been proven on the control state machine, as described in the following sections.

5.3.5.1 General Correctness / Liveness

We currently have a partial liveness proof on the name server, which shows that the `NAMESERVER_LIST` and `NAMESERVER_LOCK` calls will always terminate in constant time and return the name server to the idle condition. The proof also shows that these two calls will always behave as specified by the formal model, and will never modify any state outside that which was requested.

The remaining opcodes (namely `NAMESERVER_FQUERY`, `NAMESERVER_RQUERY`, `NAMESERVER_HMAC`, and `NAMESERVER_REGISTER`) are not yet tested as the proof is still in progress - the name server contains by far the most complex state machine we have attempted a formal correctness proof of to date.

This testbench may be found in the provided source distribution under /trunk-/tests/NOCNameServer_control/.

5.3.5.2 Name Lookups

We have verified correct operation of the name server's lookup functionality via conventional verification techniques, including automated unit testing, but have not yet completed a formal correctness proof. Although nearly every test uses the name server to some extent, the tests specifically intended to verify its correctness are located in the source distribution at /trunk/tests/NameServer and /trunk/test-s/NocSwitchNameServer.

5.3.5.3 Name Removal

Name removal is easy to do securely: The request to remove a name must come from the node itself. Since the network is trusted, receiving a packet from 0x8003 asking to delete the name corresponding to 0x8003 implies that this operation was requested by the node in question.

Proof: The current implementation of the name server does not support deleting a name once registered. Therefore, it is vacuously true that the delete function

does not violate security policy.

5.3.5.4 Name Insertion

Names for hardware nodes that are "baked" into the name table at logic synthesis time require no further authentication since the source code of the SoC is trusted implicitly (and an attacker has no way to modify these addresses).

Names being registered by a random NoC node at run time, however, are not inherently trusted. In order to prevent malicious name registrations, the name server requires a HMAC-SHA256 signature to be presented and validated before the name can be registered.

As with name lookups, we have tested this functionality using conventional verification techniques but have not yet completed a formal correctness proof.

5.3.6 RAM controller

There are two implementations of the RAM controller API, `BlockRAMAllocator` and `NetworkedDDR2Controller`.

We have an extensive test suite, located in the source distribution at /trunk-/tests/NetworkedDDR2ControllerTest, which performs a series of tests designed to find all of the main classes of malfunction we could think of.

In one test, numerous pages are allocated by one NoC node and another attempts to access them by various means, before and after changing permissions. This is intended to detect errors in permissions handling.

Another test allocates the entirety of RAM and verifies that each page pointer is returned once, and only once. This test is intended to detect errors in which the free list double-counts, loses a page, or otherwise becomes inconsistent.

Yet another test allocates all of RAM again, fills it with a periodic sequence of bits (including the page ID), reads it back, and confirms that the data is as expected. This is intended to detect collisions (such as a stuck address line) in which two addresses refer to the same physical memory. The test then frees all but the first page of memory and verifies that the first page is unmodified. It then re-allocates the freed pages, reads them all back, and confirms that they contain

zeroes rather than the old data. This is intended to detect errors in which data fails to be sanitized upon freeing, or the wrong pages are sanitized.

In order to ensure interoperability, the same compiled test binary is run on bitstreams containing both RAM controller implementations and is expected to work seamlessly with both.

While we have not yet performed formal verification of either, a correctness proof of `BlockRAMAllocator` (the simpler one) is planned for the near future.

CHAPTER 6
Conclusions and Future Work

6.1 Summary

The overall goal of this research was to determine whether moving operating system functionality into hardware is a practical means for improving operating system security. This involved the creation of an initial prototype which can serve as a basis for future research in the area.

Since the current prototype is primarily a proof of concept, many useful subsystems (such as the networking stack and filesystem) are missing major features or entirely absent. Although many of the core components (such as the NoC) have been formally verified, many higher-level components and peripherals have received basic functional testing only and the full system should be considered research-grade.

In order to encourage the widest possible use, the source code for the current prototype (including all of our internal tools) has been released as open source under the 3-clause BSD license, which grants any person permission to use it for both commercial and academic applications without needing to release the source to their software. As of this writing there is an on-line source browser available on the author's project website [86].

6.2 Conclusions

We have defined a high-level architecture, Antikernel, for an operating system which freely mixes hardware and software components as equal peers connected by a packet-switched network. The architecture takes the ideal of "least required privilege" to the extreme by having each node in the network be a fully encapsulated system which manages its own security policy, and only allows access to its internal state through a well-defined API.

The architecture draws inspiration from numerous existing operating system architectures, such as the microkernel (minimal privileged functionality with most services in userspace), the exokernel (drivers as very thin wrappers around hardware

providing nothing but security and sharing), and the separation kernel (enforcing strong isolation between processes except through a defined interface).

Additionally, the highly modular structure of an Antikernel system is (by design) very amenable to formal verification. Since there is no way for data to flow between nodes other than via the NoC, and the NoC infrastructure has been proven to correctly route packets, each node can be verified independently. If we define security of the entire system as the condition where all security properties of each node are upheld, we can then prove security by proving security of the interconnect, as well as proving that every node's security policy is internally consistent (in other words, policy cannot be violated by sending arbitrary messages to the NoC interface or any external communications interfaces).

We have tested the feasibility of the architecture by creating an initial prototype and formally verifying correctness of several subsystems within the prototype. While the simplistic cache model of our current prototype CPUs has led to poor performance in early benchmarks, this is not an inherent limitation of the architecture and we are working to improve the caches.

We hope that this work will serve to inspire future research at the intersection of computer architecture and security, and lead to more convergent full-stack design of critical systems. Blurring the lines between hardware and software appears to be a promising architectural model and one warranting further study. By releasing all of our source code we hope to encourage future work building on top of our design. We intend to continue active development and refactoring over the coming years as well.

6.3 Future Work

While the current prototype does show that hardware-based operating systems are practical and can be highly secure, it is far from usable in real-world applications. Many features which are necessary in a real-world operating system could not be implemented due to limited manpower so effort was focused on the most critical core features such as memory and process management.

6.3.1 Application Permissions

One feature lacking in the current Antikernel prototype is the ability for only a specific subset of applications to be granted access to important APIs or devices (for example, the camera or network stack) in a secure fashion which is controlled by the operating system. This is rarely used in desktop operating systems, where it is typically assumed that any software can access anything that the current user can, but is common in mobile platforms such as Android.

We have considered several possible implementations for this; one of the most promising is to add an extra section to SARATOGA's extended ELF format. This section would contain a list of scripted RPC messages which would be sent by the ELF loader (from the application's address) before the application could begin executing code, in order to set up a defined state when the application started. These messages could be used to instruct a peripheral (or firewall core on the NoC) to drop any further packets from that address, or to disable certain features, or to impose limits on maximum RAM, disk, or CPU usage.

Since the list of messages would be located in the executable headers, it could be easily inspected by a user when deciding whether to install or trust the application. The list would be covered by the same signature as the rest of the executable so any tampering would result in the application failing to run. As long as the semantics of the messages were correctly implemented (for example, firewall cores were proven to actually drop packets when instructed to) and the CPU was proven to always send the messages before executing any application code, the resulting sandbox would in principle be impossible to escape. [19]

Another related open problem is how to destroy the sandbox (delete firewall rules and quota limits) once the contained application has terminated. Clearly the application cannot send a message to destroy its own sandboxing, as this would allow trivial escapes. Another obvious idea is for the CPU to send a message to the firewall nodes from the OoB management address, however in Antikernel's decentralized model all nodes are equal and the number (and addresses) of CPUs may not be

[19] If the sandboxed application had permission to send messages to another node which did have access to the target resource, and which had an input parsing vulnerability, it would potentially be possible to escalate privileges horizontally.

known to the firewall so it would have no way to authenticate the message.

One promising solution to this problem is to have the "create firewall rule" message somehow specify the address of the CPU that the application is running on. The CPU could then send a series of scripted "remove firewall rule" messages from the OoB address, which could not be spoofed by the application. This, however, poses the (currently unsolved) problem of how the firewall can trust that the address in question is actually the host CPU, and not a collaborating piece of malware.

A very similar problem is that of ensuring that there are no resource leaks. When an application terminates the RAM controller (for example) should have some way of knowing that it needs to release all of the memory allocated to that node. Since there is no harm in allowing an application to free its own memory, however, these messages can simply be sent from the application's own address during the shutdown process or by C runtime library cleanup code.

6.3.2 Persistent Identifiers

In order to implement features such as file systems (which persist data across power cycles) it is important to be able to reliably identify an application over multiple reboots so that ownership records can be accurately maintained. Furthermore, name server registrations as currently implemented require that the address being registered be part of the signed packet.

The current prototype simply relies on the initialization code starting all applications in the same order (and thus receiving the same thread ID since these are allocated in FIFO order every boot). A more stable system for binding IDs is, however, desirable.

6.3.3 Additional Formal Verification

As of this writing, neither of the memory controller implementations have been formally verified. While fairly extensive automated testing has been performed, and they appear to work correctly, full proofs were not possible in the time available.

No part of SARATOGA or GRAFTON (other than of course the NoC transceivers) has been formally verified to date. While SARATOGA's architecture was designed

to minimize the risk of accidental data leakage between thread contexts, until full verification is completed we cannot rule out the possibility that such a bug exists.

Eventually we would like to verify that the CPUs themselves correctly implement the semantics of our reduced MIPS-1 instruction set. If we then compiled our application code with a formally verified C compiler (such as CompCert C [87, 88]) we could have full equivalency proofs from C down to RTL. [20] This could then be combined with verification of the C source code, resulting in fully verified correct execution from application software all the way down to RTL.

The current `nocgen` tool empirically seems to generate correct Verilog however it has not been formally verified and there is currently no way to automatically verify that its output is correct. There is also no way to automatically verify that third-party IP has a valid transceiver configuration without time-consuming source code auditing. We plan to create a DRC tool (as discussed in section 5.3.2.2) which will be run as part of the build process and ensure that the actual RTL netlist adheres to the network topology requirements for the security proofs to hold.

6.3.4 Additional Peripherals

The prototype codebase contains a basic IPv6 offload engine as a hardware module, as well as an ICMP module (which currently only handles SLAAC and inbound ping requests) however there currently is no support, in either hardware or software, for upper-level protocols. We intend to create a software-based TCP and UDP stack which runs on SARATOGA over the coming months.

Needless to say, many more peripherals could be created as well.

6.3.5 Profile-Guided NoC Optimization

The current `nocgen` tool assigns nodes to addresses in a greedy fashion. This is rarely optimal.

To optimize for size, `nocgen` should consider which nodes use only RPC or only DMA connections, and pack these into as few subnets as possible so that as

[20] The current CompCert compiler does not support the MIPS instruction set - only x86, ARM, and PowerPC. We plan to explore adding formally verified MIPS code generation to this or another verified C compiler in the future.

many routers as possible can be optimized out (due to having no children).

To optimize for performance, the design should be synthesized with instrumented routers and profiled to determine which nodes talk to each other the most. This will then be combined with optimization goals (hints describing which paths are most important to reduce latency on) in order to create a NoC in speed-critical links are as short as possible. Latency optimization can have dramatic effects: the author managed to achieve a 32% speedup on GRAFTON's Dhrystone benchmark score by manually swapping two nodes, thus shaving two hops off the link between the CPU and ROM.

6.3.6 CPU Improvements

6.3.6.1 GRAFTON

The GRAFTON processor works well and is currently in use running the control interfaces on some of our internal test equipment. The L1 cache, however, has terrible performance (I-side miss rates in excess of 5% on the Dhrystone benchmark) and we plan to explore options for getting better cache performance without massively increasing the required die area.

6.3.6.2 SARATOGA

As with GRAFTON, the SARATOGA processor is largely functional but has substantial room for improvement. The current microarchitecture has a single load-/store unit, which also has a control hazard prohibiting issuing a load or store with *any* other instruction. As a result the IPC figure for any code containing a significant fraction of loads and stores can be no better than 1.0.

We plan to redesign the load/store system such that ALU instructions can be dual-issued alongside memory instructions. We also intend to add a second load/store unit, which will permit two simultaneous memory operations (two loads, two stores, or one of each) to be issued during the same cycle as long as they both target the same cache line.

The current microarchitecture requires one hardware thread context for every thread, which can substantially increase the size of the register file for larger systems.

We plan to allow the CPU to be allocated a subnet with more addresses than hardware threads (for example, 128 addresses and 32 hardware threads). The CPU could then allocate a page of physical memory from the OoB address and page contexts in and out of the register file as needed. Since this memory would be owned by the CPU it would be impossible for any other node in the system to tamper with contexts during swapping.

While the current microarchitecture does implicitly hide I/O and memory fetch latency to some degree (by thread switching immediately after each instruction) there is some wasted performance since the blocking thread is still on the run queue and thus wasting one cycle out of every N. A future improvement to the architecture could remove threads blocking on L1 misses or waiting for incoming RPCs from the run queue, preventing them from using any cycles whatsoever until an event occurs which results in them becoming runnable again.

We have not yet done extensive performance benchmarking on SARATOGA due to the lack of hardware performance counters. We plan to add a synthesis-time option to create per-thread performance counters which record the number of cache hits, misses, time spent waiting on cache misses, instructions issued, instructions dual-issued, and possibly additional metrics.

6.3.7 Defense in Depth

Our current prototype is intended to be a proof of concept for hardware-based compartmentalization at the OS level. As a result, we do not incorporate any of the numerous defensive techniques in the literature for guarding against physical tampering, fault injection, or software-based exploits targeting userland software.

Further work could explore integrating some of this work with Antikernel. Possible defense-in-depth techniques include error-correcting RAM, encryption of off-chip memory and flash, SEU protection, redundant logic or lock-step CPUs for glitch protection, hardware sensor monitoring, and control flow integrity protection.

REFERENCES

[1] Sophos Ltd. (1998, Jun. 30) *Sophos warns of new PC paralyser*. [Online]. Available: https://www.sophos.com/en-us/press-office/press-releases/1998/06/pr_uk_19980630cih.aspx (Accessed 2015-04-09).

[2] N. Falliere *et al.* (2011, Feb. 1) *W32.Stuxnet Dossier*. Symantec Corp., Mountain View, CA. [Online]. Available: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf (Accessed 2015-04-09).

[3] N. Leveson and C. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, Jul 1993.

[4] J. Hirsch and K. Bensinger. (2013, Oct. 26) *Toyota settles acceleration lawsuit after $3-million verdict*. [Online]. Available: http://www.latimes.com/business/autos/la-fi-hy-toyota-damages-20131026-story.html (Accessed 2015-04-09).

[5] British Broadcasting Corporation. (2014, Dec. 22) *Hack attack causes 'massive damage' at steel works*. [Online]. Available: http://www.bbc.com/news/technology-30575104 (Accessed 2015-04-09).

[6] W. Safire. (2004, Feb. 2) *The Farewell Dossier*. [Online]. Available: http://www.nytimes.com/2004/02/02/opinion/the-farewell-dossier.html (Accessed 2015-04-09).

[7] G. W. Weiss, "Duping the soviets: The farewell dossier," *Stud. Intelligence*, vol. 39, no. 5, pp. 121–126, 1996. [Online]. Available: https://www.cia.gov/library/center-for-the-study-of-intelligence/kent-csi/vol39no5/pdf/v39i5a14p.pdf (Accessed 2015-04-09).

[8] D. R. Piegdon. (2007, Apr. 12) *hacking in physically addressable memory: a proof of concept*. [Online]. Available: http://eh2008.koeln.ccc.de/fahrplan/attachments/1067_SEAT1394-svn-r432-paper.pdf (Accessed 2015-04-09).

[9] F. Witherden. (2010, Sep. 7) *Memory Forensics over the IEEE 1394 Interface*. [Online]. Available: https://freddie.witherden.org/pages/ieee-1394-forensics.pdf (Accessed 2015-04-09).

[10] CERT Program. (2012, Jun. 12) *Vulnerability Note VU649219: SYSRET 64-bit operating system privilege escalation vulnerability on Intel CPU*

hardware. [Online]. Available: http://www.kb.cert.org/vuls/id/649219 (Accessed 2015-04-09).

[11] D. Boneh *et al.*, "On the importance of eliminating errors in cryptographic computations," *J. Cryptology*, vol. 14, pp. 101–119, 2001.

[12] A. Moradi *et al.*, "A generalized method of differential fault attack against aes cryptosystem," in *Cryptographic Hardware Embedded Syst. - CHES 2006*, Yokohama, Japan, 2006, pp. 91–100.

[13] A. Barenghi *et al.*, "Low voltage fault attacks to aes and rsa on general purpose processors." *IACR Cryptology ePrint Archive*, vol. 2010, p. 130, 2010.

[14] A. Barenghi *et al.*, "Fault attack on aes with single-bit induced faults," in *Inform. Assurance Security (IAS), 2010 6th Int. Conf.*, Atlanta, GA, Aug 2010, pp. 167–172.

[15] P. Kocher *et al.*, "Differential power analysis," in *Advances Cryptology - CRYPTO99*, Santa Barbara, CA, 1999, pp. 388–397.

[16] E. Brier *et al.*, "Correlation power analysis with a leakage model," in *Cryptographic Hardware Embedded Syst. - CHES 2004*, Boston, MA, 2004, pp. 16–29.

[17] C. Helfmeier *et al.*, "Breaking and entering through the silicon," in *Proc. 2013 ACM SIGSAC Conf. Comput. Commun. Security*, Berlin, Germany, 2013, pp. 733–744.

[18] S. P. Skorobogatov, "Semi-invasive attacks - A new approach to hardware security analysis," Univ. Cambridge, Comput. Lab, Cambridge, England, Tech. Rep. UCAM-CL-TR-630, Apr. 2005. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-630.pdf (Accessed 2015-04-09).

[19] A. Zonenberg. (2011, Mar. 3) *Silicon Exposed - Microchip PIC12F683 teardown*. [Online]. Available: http://siliconexposed.blogspot.com/2011/03/microchip-pic12f683-teardown.html (Accessed 2015-04-09).

[20] A. Zonenberg. (2014, Apr. 4) *CSCI 4974 / 6974 Hardware Reverse Engineering Lecture 14: Invasive attacks*. [Online]. Available: http://security.cs.rpi.edu/courses/hwre-spring2014/Lecture14_InvasiveAttacks.pdf (Accessed 2015-04-09).

[21] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware trojans," *Computer*, vol. 43, no. 10, pp. 39–46, Oct. 2010.

[22] A. Zonenberg. (2014, Sep. 17) *Silicon Exposed: Threat modeling for FPGA software backdoors* . [Online]. Available: http://siliconexposed.blogspot.com/2014/09/threat-modeling-for-fpga-software.html (Accessed 2015-04-09).

[23] Xilinx Inc. (2014, Nov. 14) *7 Series FPGAs Configuration User Guide v1.9*. San Jose, CA. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf (Accessed 2015-04-09).

[24] A. Tamoney, D. Kouttron, and A. Radocea. (2008, Oct. 5) *RPI Team Report: CSAW 2008 Embedded Systems Challenge*. [Online]. Available: http://isis.poly.edu/~vikram/rpi.pdf (Accessed 2015-04-09).

[25] Wind River. (2015) *VxWorks*. Alameda, CA. [Online]. Available: http://www.windriver.com/products/vxworks/ (Accessed 2015-04-09).

[26] Microsoft Corp. (2006) *Kernel Overview (Windows CE 5.0)*. Redmond, WA. [Online]. Available: https://msdn.microsoft.com/en-us/library/aa450566.aspx (Accessed 2015-04-09).

[27] Google. (2015) *System Permissions - Android Developers*. Mountain View, CA. [Online]. Available: http://developer.android.com/guide/topics/security/permissions.html (Accessed 2015-04-09).

[28] Micricum. (2015) *μC/OS-II*. Weston, FL. [Online]. Available: http://micrium.com/rtos/ucosii/overview/ (Accessed 2015-04-09).

[29] Real Time Engin. Ltd. (2015) *FreeRTOS*. London, England. [Online]. Available: http://www.freertos.org/ (Accessed 2015-04-09).

[30] QNX Software Syst. (2015) *QNX operating systems, development tools, and professional services for connected embedded systems*. Ottawa, Canada. [Online]. Available: http://www.qnx.com/ (Accessed 2015-04-09).

[31] TU Dresden. (2015) *The L4 Microkernel Family*. Dresden, Germany. [Online]. Available: http://os.inf.tu-dresden.de/L4/ (Accessed 2015-04-09).

[32] A. Singh. (2003, Dec.) *XNU: The Kernel*. [Online]. Available: http://osxbook.com/book/bonus/ancient/whatismacosx/arch_xnu.html (Accessed 2015-04-09).

[33] J. M. Rushby, "Design and verification of secure systems," in *Proc. 8th ACM Symp. Operating Sys. Principles*, Pacific Grove, California, 1981, pp. 12–21.

[34] W. Martin, P. White, F. S. Taylor, and A. Goldberg, "Formal construction of the mathematically analyzed separation kernel," in *Automated Software Eng., 2000. Proc. ASE 2000. 15th IEEE Int. Conf.*, Grenoble, France, 2000, pp. 133–141.

[35] Green Hills Soft. (2015) *INTEGRITY Real-time Operating System*. Santa Barbara, CA. [Online]. Available: http://www.ghs.com/products/rtos/integrity.html (Accessed 2015-04-09).

[36] D. R. Engler *et al.*, "Exokernel: an operating system architecture for application-level resource management," in *Proc. 15th ACM Symp. Oper. Syst. Principles*, vol. 29, no. 5, Copper Mountain, CO, Dec. 1995, pp. 251–266.

[37] M. F. Kaashoek *et al.*, "Application performance and flexibility on exokernel systems," in *Proc. 16th ACM Symp. Oper. Syst. Principles*, vol. 31, no. 5, Saint-Malo, France, Oct. 1997, pp. 52–65.

[38] A. Baumann *et al.*, "The multikernel: A new os architecture for scalable multicore systems," in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Principles*, Big Sky, MT, 2009, pp. 29–44.

[39] J. Rutkowska and R. Wojtczuk. (2010, Jan.) *Qubes OS Architecture*. [Online]. Available: http://files.qubes-os.org/files/doc/arch-spec-0.3.pdf

[40] General Dynamics. (2015) *OKL4 Microvisor*. Pittsfield, MA. [Online]. Available: http://www.ok-labs.com/products/okl4-microvisor (Accessed 2015-04-09).

[41] Green Hills Soft. (2003) *INTEGRITY Multivisor: Virtualization Architecture for Secure Systems*. Santa Barbara, CA. [Online]. Available: http://www.ghs.com/products/rtos/integrity_virtualization.html (Accessed 2015-04-09).

[42] ARM Ltd. (2014) *TrustZone Technology*. [Online]. Available: http://www.arm.com/products/processors/technologies/trustzone.php (Accessed 2015-04-09).

[43] Genode Labs. (2014) *An Exploration of ARM TrustZone Technology*. Dresden, Germany. [Online]. Available: http://genode.org/documentation/articles/trustzone (Accessed 2015-04-09).

[44] NIST. (2013, Apr. 15th) *CVE-2013-3051*. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-3051 (Accessed 2015-04-09).

[45] M. Engel and O. Spinczyk, "A radical approach to network-on-chip operating systems," in *System Sciences, 2009. HICSS '09. 42nd Hawaii Int. Conf.*, Honolulu, HI, Jan. 2009, pp. 1–10.

[46] S. Nordstrom *et al.*, "Application specific real-time microkernel in hardware," in *Real Time Conf., 2005. 14th IEEE-NPSS*, Miami, FL, Jun. 2005, p. 4.

[47] T. Nakano *et al.*, "Hardware implementation of a real-time operating system," in *Proc. 12th TRON Project Int. Symp.*, Tokyo, Japan, 1995, p. 34.

[48] W. Hu, J. Ma, B. Wu, L. Ju, and T. Chan, "Distributed on-chip operating system for network on chip," in *Comput. Inform. Technology (CIT), 2010 IEEE 10th Int. Conf. on*, Bradford, England, Jul. 1 2010, pp. 2760–2767.

[49] V. J. Mooney. *et al.*, "A hardware-software real-time operating system framework for socs," *Design Test Comput., IEEE*, vol. 19, no. 6, pp. 44–51, Nov. 2002.

[50] S. Park *et al.*, "A hardware operating system kernel for multi-processor systems," *IEICE Electron. Express*, vol. 5, no. 9, pp. 296–302, 2008.

[51] H. Kwok-Hay So *et al.*, "A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph," in *CODES+ISSS '06: Proc. 4th Int. Conf. Hardware/Software Codesign Syst. Synthesis*, Seoul, South Korea, 2006, pp. 259–264.

[52] A. Wasicek *et al.*, "A system-on-a-chip platform for mixed-criticality applications," in *Object/Component/Service-Oriented Real-Time Distributed Comput. (ISORC), 2010 13th IEEE Int. Symp.*, Carmona, Spain, May 2010, pp. 210–216.

[53] A. Thomas *et al.* (2013, Jan 10) *Towards a Zero-Kernel Operating System.* [Online]. Available: http://www.infsec.cs.uni-saarland.de/~hritcu/publications/zkos_draft_jan10_2013.pdf (Accessed 2015-04-09).

[54] BiiN Corporation. (1988, Jul) *BiiN Systems Overview.* Portland, OR. [Online]. Available: http://bitsavers.informatik.uni-stuttgart.de/pdf/biin/BiiN_Systems_Overview.pdf (Accessed 2015-04-09).

[55] New York Times. (1989, Oct 21) *COMPANY NEWS; Intel and Siemens End BiiN Venture.* [Online]. Available: http://www.nytimes.com/1989/10/21/business/company-news-intel-and-siemens-end-biin-venture.html (Accessed 2015-04-09).

[56] L. Benini and G. De Micheli, "Networks on chips: a new soc paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, Jan. 2002.

[57] R. Hecht *et al.*, "Dynamic reconfiguration with hardwired networks-on-chip on future fpgas," in *Field Programmable Logic Applicat,, 2005. Int. Conf.*, Tampere, Finland, Aug. 2005, pp. 527–530.

[58] G. Schelle and D. Grunwald, "Exploring fpga network on chip implementations across various application and network loads," in *Field*

Programmable Logic Applicat., 2008. FPL 2008. Int. Conf., Heidelberg, Germany, 2008, pp. 41–46.

[59] A. Gara *et al.*, "Overview of the blue gene/l system architecture," *IBM J. Research Develop.*, vol. 49, no. 2.3, pp. 195–212, Mar. 2005.

[60] L. Fiorin, L *et al.*, "Secure memory accesses on networks-on-chip," *IEEE Trans. Comput.*, vol. 57, no. 9, pp. 1216–1229, Sep. 2008.

[61] A. Wingard. (2005) *A Non-Blocking Intelligent Interconnect for AMBA-Connected SoC's*. [Online]. Available: http://www.yumpu.com/en/document/view/37653902/a-non-blocking-intelligent-interconnect-for-amba-ocp-ip (Accessed 2015-04-09).

[62] T. Schubert, "High level formal verification of next-generation microprocessors," in *Proc. 40th Annu. Design Automation Conf.*, Anaheim, CA, USA, 2003, pp. 1–6.

[63] G. Klein *et al.*, "sel4: formal verification of an os kernel," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Principles*, Big Sky, MT, 2009, pp. 207–220.

[64] C. L. Heitmeyer *et al.*, "Formal specification and verification of data separation in a separation kernel for an embedded system," in *Proc. 13th ACM Conf. Comput. Commun. Security*, Alexandria, Virginia, USA, 2006, pp. 346–355.

[65] NVIDIA Corporation. (2012) *NVIDIA Kepler GK110 Architecture Whitepaper*. Santa Clara, CA. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf (Accessed 2015-04-09).

[66] J.-M. Mäkelä and V. Leppänen, "Towards programming on the moving threads architecture," in *Proc. 11th Int. Conf. Comput. Syst. Technologies*, Sofia, Bulgaria, 2010, pp. 137–142.

[67] D. May. (2009) *The XMOS XS1 Architecture*. [Online]. Available: https://www.xmos.com/en/download/public/The-XMOS-XS1-Architecture(1.0).pdf (Accessed 2015-04-09).

[68] J. A. Kahle *et al.*, "Introduction to the cell multiprocessor," *IBM J. Research Develop.*, vol. 49, no. 4.5, pp. 589–604, Jul. 2005.

[69] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for fpgas," in *Proc. 18th Annu. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Monterey, CA, 2010, pp. 41–50.

[70] C. E. LaForest *et al.*, “Multi-ported memories for fpgas via xor,” in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Monterey, CA, 2012, pp. 209–218.

[71] OpenOCD Project. (2014) *Open On-Chip Debugger*. [Online]. Available: http://openocd.sourceforge.net/ (Accessed 2015-04-09).

[72] UrJTAG Project. (2014) *UrJTAG - Universal JTAG library, server, and tools*. [Online]. Available: http://urjtag.org/ (Accessed 2015-04-09).

[73] A. Rogers and U. Bonnes. (2014) *xc3sprog*. [Online]. Available: http://xc3sprog.sourceforge.net/ (Accessed 2015-04-09).

[74] Kitware. (2015) *CMake*. Clifton Park, NY. [Online]. Available: http://www.cmake.org/ (Accessed 2015-04-09).

[75] Kitware. (2015) *CTest 2.8.8 Documentation*. Clifton Park, NY. [Online]. Available: http://www.cmake.org/cmake/help/v2.8.8/ctest.html (Accessed 2015-04-09).

[76] C. Kemper. (2011, Aug 18) *Build in the Cloud: How the Build System Works*. [Online]. Available: http://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.html

[77] M. A. Jette *et al.*, “Slurm: Simple linux utility for resource management,” in *Lecture Notes Comput. Science: Proc. Job Scheduling Strategies Parallel Process. (JSSPP) 2003*, Seattle, WA, 2003, pp. 44–60.

[78] Digilent Inc. (2012) *Atlys Spartan-6 FPGA Development Board*. [Online]. Available: http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,836&Prod=ATLYS (Accessed 2015-04-09).

[79] Xilinx Inc. (2015, Mar 18) *Artix-7 FPGAs Data Sheet: DC and AC Switching Characteristics v1.18*. San Jose, CA. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds181_Artix_7_Data_Sheet.pdf (Accessed 2015-04-09).

[80] Xilinx Inc. (2015) *ChipScope Pro and the Serial I/O Toolkit*. San Jose, CA. [Online]. Available: http://www.xilinx.com/tools/cspro.htm (Accessed 2015-04-09).

[81] Altera Corporation. (2012) *SignalTap II with Verilog Designs*. San Jose, CA. [Online]. Available: ftp://ftp.altera.com/up/pub/Altera_Material/12.1/Tutorials/Verilog/SignalTap.pdf (Accessed 2015-04-09).

[82] Y. Kim *et al*, “Flipping bits in memory without accessing them: An experimental study of dram disturbance errors,” in *Comput. Architecture (ISCA), 2014 ACM/IEEE 41st Int. Symp.*, Minneapolis, MN, Jun 2014, pp. 361–372.

[83] C. Evans. (2015, Mar. 9) *Project Zero: Exploiting the DRAM rowhammer bug to gain kernel privileges*. [Online]. Available: http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html (Accessed 2015-04-09).

[84] G. Coker *et al.*, "Principles of remote attestation," *Int. J. Inform. Security*, vol. 10, no. 2, pp. 63–81, 2011.

[85] A. Zonenberg. (2014, Mar. 31) *Getting my feet wet with invasive attacks, part 1: Target recon* . [Online]. Available: http://siliconexposed.blogspot.com/2014/03/getting-my-feet-wet-with-invasive.html (Accessed 2015-04-09).

[86] A. Zonenberg. (2015, Apr. 25) *Antikernel source repository*. [Online]. Available: http://redmine.drawersteak.com/projects/achd-soc/repository (Accessed 2015-04-09).

[87] S. Blazy *et al.*, "Formal verification of a C compiler front-end," in *FM 2006: Int. Symp. Formal Methods*, vol. 4085, Hamilton, Ontario, 2006, pp. 460–475.

[88] S. Boldo *et al.*, "A formally-verified C compiler supporting floating-point arithmetic," in *ARITH, 21st IEEE Int. Symp. Comput. Arithmetic*. Austin, TX: IEEE Computer Society Press, 2013, pp. 107–115.