UNIVERSITY OF CALIFORNIA, SAN DIEGO

Design and Architecture of Automatically-generated Energy-reducing Coprocessors

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

in

Computer Science (Computer Engineering)

by

John Morgan Sampson

Committee in charge:

Professor Steven Swanson, Co-Chair
Professor Michael Taylor, Co-Chair
Professor James Buckwalter
Professor Lawrence Larson
Professor Dean Tullsen

2010

DEDICATION

To all those willing to wait for the fruits of their labor.

EPIGRAPH

When the facts change, I change my mind. What do you do, sir?

—John Maynard Keynes

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Many people have contributed directly and indirectly to my progress along the journey that lead me to construct this document. They are too many to name, and any attempt to impose a strict ordering on their contributions would be a fool's errand. I wish to thank those at UC Berkeley, UC San Diego, and at HP labs, my advisors, my colleagues, my family, my significant others, all those who let me kvetch incessantly, and all those who ever let me cook for them. You are all a part of my journey, and you are all appreciated.

Chapters 1, 2, 3, 7, and 8 contain material from "Conservation cores: reducing the energy of mature computations", by Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson and Michael Bedford Taylor, which appears in *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. The dissertation author was the secondary investigator and author of this paper. The material in Chapter 1 is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or email `permissions@acm.org`.

Chapters 1, 4, 5, 6, 7, and 8 contain material from "Energy-Delay Optimized Accelerators for Irregular Code", by Jack Sampson, Ganesh Venkatesh, Nathan Goulding, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor, which has been submitted for possible publication by IEEE in *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. The dissertation author was the primary investigator and author

of this paper.

VITA AND PUBLICATIONS

2000-2003	Teaching assistant University of California, Berkeley
2002	B. S. in Electrical Engineering and Computer Science University of California, Berkeley
2003-2010	Research assistant University of California, San Diego
2005	Internship HP Labs Palo Alto, California
2006	Internship HP Labs Palo Alto, California
2008	C. Phil. in Computer Engineering University of California, San Diego
2010	Ph. D. in Computer Engineering University of California, San Diego

PUBLICATIONS

Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, Michael Bedford Taylor, "Conservation Cores: Reducing the Energy of Mature Computations", *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2010.

Jack Sampson, Ruben Gonzalez, Jean-Francois Collard, Norm Jouppi, Mike Schlansker and Brad Calder, "Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers", *Proceedings of the 39th Inernational Symposium on Microarchitecture (MICRO)*, December 2006.

Christophe Lemuet, Jack Sampson, Jean-Francois Collard, Norm Jouppi, "The Potential Energy Efficiency of Vector Acceleration", *Proceedings of the 2006 ACM / IEEE conference on Supercomputing (SC)*, November 2006

Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Osvaldo Colavin, Brad Calder, "Unbounded Page-Based Transactional Memory", *Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006

Erez Perelman, Marzia Polito, Jean-Yves Bouguet, John Sampson, Brad Calder, Carole Dulong, “Detecting Phases in Parallel Applications on Shared Memory Architectures”, *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, April 2006

Lieven Eeckhout, John Sampson, and Brad Calder, “Exploiting Program Microarchitecture Independent Characteristics and Phase Behavior for Reduced Benchmark Suite Simulation”, *In Proceedings of the 2005 IEEE International Symposium on Workload Characterization*, October 2005

Jeremy Lau, Jack Sampson, Erez Perelman, Greg Hamerly, Brad Calder, “The Strong Correlation Between Code Signatures and Performance”, *Proceedings of the 5th International Symposium on Performance Analysis of Systems and Software*, March 2005

ABSTRACT OF THE DISSERTATION

Design and Architecture of Automatically-generated Energy-reducing Coprocessors

by

John Morgan Sampson

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California, San Diego, 2010

Professor Steven Swanson, Co-Chair
Professor Michael Taylor, Co-Chair

For many years, improvements to CMOS process technologies fueled rapid growth in processor performance and throughput. Each process generation brought exponentially more transistors and exponentially reduced the per-transistor switching power. However, concerns over leakage currents have moved us out of the classical CMOS scaling regime. Although the number of available transistors continues to rise, their switching power no longer declines. In contrast to transistor counts, power budgets remain fixed due to limitations on cooling or battery life. Thus, with each new process generation, an exponentially decreasing fraction of the available transistors can be simultaneously switched. The growing divide between available transistors and utilizable transistors leads to a *utilization wall*.

This dissertation characterizes the utilization wall and proposes *conservation cores* as a means of surmounting its most pressing challenges. Conservation cores, or *C-Cores*, are application-specific hardware circuits created to reduce energy consumption on computationally-intensive applications with complex control logic and irregular memory access patterns. C-Cores are drop-in replacements for existing source code, and make use of limited reconfigurability to adapt to software changes over time. The design and implementation of these specialized execution engines pose challenges with respect to code selection, automatic synthesis, choice of programming model, longevity/robustness, and system integration.

This dissertation addresses many of these challenges through the development of an automated conservation core toolchain. The toolchain automatically extracts the key kernels from a target workload and uses a custom C-to-silicon infrastructure to generate 45 nm implementations of the C-Cores. C-Cores employ a new pipeline design technique called *pipeline splitting*, or *pipesplitting*. This technique reduces clock power, increases memory parallelism, and further exploits operation-level parallelism. C-Cores also incorporate specialized energy-efficient per-instruction data caches called *cachelets* into the datapath, which allow for sub-cycle cache-coherent memory accesses.

An evaluation of C-Cores against an efficient in-order processor shows that C-Cores speed up the code they target by 1.5×, improve EDP by 6.9× and accelerate the whole application by 1.33× on average, while reducing application energy-delay by 57%.

Chapter 1

Introduction

For many years, improvements to CMOS process technologies fueled rapid growth in processor performance and throughput. Each process generation brought exponentially more transistors and exponentially reduced the per-transistor switching power. With these ample and efficient resources, computer architects were able to increase the number of cores per processor and employ increasingly sophisticated mechanisms to improve the performance of each core.

In recent years, however, the benefits of newer process generations have changed. Although the number of available transistors continues to rise, their switching power no longer declines. Concerns over leakage currents have moved us out of the classical CMOS scaling regime. We cannot employ the additional transistors from newer process generations if their use violates the power budget: Pragmatic concerns, such as thermal management, cost of ownership, and battery life, hold power budgets fixed. Thus, with each process generation, an exponentially decreasing fraction of the available transistors can be simultaneously switched. We term this growing divide between available transistors and utilizable transistors the *utilization wall.* The homogeneous multi-core approach to designing processors is ill-suited to face this challenge. However, the leakage-limited scaling regime introduces a new landscape in which heterogeneous designs with specialized hardware hold significant promise.

Faced with the utilization wall, designers find each new process generation leaves them with undesirable choices. They could allow the area budget to scale down with power, but that would signal the end of exponentially increasing circuit integration. As this increasing integration, commonly known as Moore's Law, has powered the past several decades of architectural innovation, this is not an acceptable option. For a fixed area budget, either some of that area must lay fallow at any given time, *dark silicon*, or the area must contain underutilized transistors, *dim silicon*. Such underutilization is most commonly accomplished via underclocking, which limits performance. Heterogeneous designs offer a means to exploit dark silicon: Only cores that are well matched to the current set of computations are active, and all others remain dark. As the current set of computations changes, cores can move between active and dark status, running threads on the most closely matching silicon.

Specialization and heterogeneity are appealing from both power and performance perspectives, and both are challenging from a system perspective. Producing a sufficient diversity of specialized hardware to provide high coverage for a non-trivial target workload requires significant design automation. Likewise, the hardware produced must reduce energy for serial and irregular code, as well as parallel code, or the technique will be of limited applicability. Custom ASICs and accelerators for highly parallel computations are well studied, but many open questions remain for the systematic generation of hardware for irregular code regions.

This dissertation characterizes the utilization wall and proposes *conservation cores* as a means of addressing the challenges of dark silicon. Conservation cores, or *C-Cores*, are application-specific hardware circuits created for the purpose of reducing energy consumption on computationally intensive applications with irregular code bases. C-Cores improve the energy efficiency of irregular codes by converting dark silicon into a collection of energy-saving, application-specialized cores. Each C-Core is a drop-in replacement for a region of code in the source application. These cores are produced by an automated toolchain which transforms regions of C source into C-Cores and transparently modifies the original applications to use the C-Cores. C-Cores, unlike many other approaches, can target nearly arbitrary code regions.

Conservation cores have a different goal than conventional application-specific circuits, and we differentiate between C-Cores and the more common *accelerators* along several axes. Accelerators focus on improving performance, at a potentially worse, equal, or better energy efficiency. As their name suggests, designers rarely deploy accelerators for code where performance does not greatly benefit from customized hardware. Conservation cores, on the other hand, focus on energy reduction. Serial and irregular codes are valid targets for energy reduction via conservation cores, even if performance benefits are limited. Conservation cores aim to always reduce energy consumption and to accelerate where practical. Conservation cores that are also accelerators are possible: Chapters 4 and 5 explore techniques for improving the performance of C-Cores while maintaining energy efficiency.

Even if, for a given C-Core, the hardware specialization that provides energy efficiency does not translate into better performance, energy efficiency can translate directly to better throughput. Under the utilization wall, such specialized, energy-efficient processors can increase parallelism by reducing the per-computation power requirements and allowing more computations to execute under the same power budget. Therefore, while single-threaded performance is important, especially for difficult to parallelize irregular code bases, performance should not be purchased at the expense of forsaking energy efficiency.

Shifting the focus from performance-at-all-costs to efficiency allows C-Cores to target a broader range of applications than accelerators. C-Cores, unlike most accelerators, can target both parallel and serial portions of an application. Accelerators provide benefits for codes with large amounts of parallelism and predictable communication patterns, as these codes map naturally onto hardware. Thus, parallelism-intensive regions of code that are hot (i.e., occupy a high percentage of running time) are the best candidates for implementation as accelerators. On the other hand, C-Cores are parallelism-agnostic: Hot code with a tight critical path, little parallelism, and/or very poor memory behavior is an excellent candidate for a C-Core: C-Cores can greatly reduce the number of transistor toggles required to execute that code, saving energy. For instance, our results show that C-Cores can deliver significant energy savings for irregular, integer applications (e.g., MCF from SPEC 2006) that would be difficult to automatically accelerate with specialized hardware.

Over the course of this dissertation, we will examine the utilization wall, conservation cores, and how these conservation cores rise to the challenges of a leakage limited scaling regime. Chapter 2 examines the origins and implications of the utilization wall. We discuss the theory behind classical and leakage limited CMOS scaling, and present our own empirical results for the utilization wall. We then discuss how the unique challenges of the utilization wall lead us to develop the C-Core approach.

Chapter 3 describes the system architecture for C-Core enabled systems, our toolchain for automatically creating and compiling for C-Cores, and presents and evaluates a prototype C-Core architecture. The toolchain automatically extracts the key kernels from a target workload and uses a custom C-to-silicon infrastructure to generate 45 nm implementations of the C-Cores. It also automates the process of evaluating and compiling for C-Core enabled systems. We use this toolchain to produce and evaluate a set of prototype C-Cores from multiple versions of several applications. We show that our prototype C-Cores provide significant energy and energy-delay savings over an efficient, in-order MIPS processor.

Chapters 4 and 5 describe enhancements to C-Core performance and energy efficiency. Our prototype C-Cores offer substantial energy savings, but have room to improve on performance and energy efficiency in three key areas: Memory ordering enforcement, reconfiguration mechanisms, and load-use latency all benefit from targeted optimizations. To improve C-Core performance we employ a new pipeline design technique called *pipeline splitting*, or *pipesplitting*. This technique reduces clock power, increases memory parallelism, and further exploits ILP. To reduce the energy, area, and operator delay costs incurred by the software adaptation mechanisms from [VSG^{+}10] we exploit a more nuanced form of reconfiguration. We also explore incorporating specialized energy-efficient per-instruction data caches called *cachelets*, which allow for sub-cycle cache-coherent memory accesses. The first two of these advances are presented in Chapter 4, and the third is the focus of Chapter 5.

Chapter 6 analyzes the hardware produced by our automated toolchain, and explores the impact and challenges of scheduling operations on C-Cores. We discuss why operation scheduling is important for C-Cores and why traditional modulo scheduling techniques are not applicable. We examine the limitations of a block-based execution and scheduling model in the face of small basic blocks, and highlight possible avenues for improving block size and exposed parallelism.

Chapter 7 surveys other approaches to energy reduction, accelerator architectures, and automated hardware generation. It also discusses differences in technique from related work for our implementations of scheduling, caching, and other components of our approach. Through these comparisons, we place conservation cores in the context of broader efforts in hardware specialization and highlight novel aspects of the C-Core approach.

Finally, in Chapter 8 we summarize the contributions of this dissertation, including the introduction of the utilization wall and the design and evaluation of C-Cores and C-Core enhancing techniques.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under NSF CAREER Awards 06483880 and 0846152, and under NSF CCF Award 0811794.

This chapter contains material from "Conservation cores: reducing the energy of mature computations", by Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson and Michael Bedford Taylor, which appears in *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. The dissertation author was the secondary investigator and author of this paper. The material in this chapter is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or

This chapter contains material from "Energy-Delay Optimized Accelerators for Irregular Code", by Jack Sampson, Ganesh Venkatesh, Nathan Goulding, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor, which has been submitted for possible publication by IEEE in *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. The dissertation author was the primary investigator and author of this paper.

Chapter 2

The Utilization Wall

In this chapter, we examine the nature of the utilization wall in detail, and motivate heterogeneous multiprocessors with application specialized cores as a means of dealing with its consequences. We show how scaling theory predicts that the transition from classical to leakage-limited regime will lead to the utilization wall. We then demonstrate how these theoretical models predict current practice, confirming the exponential nature of the utilization wall. Finally, we discuss how heterogeneous multi-core processors address the challenges posed by utilization wall in ways that homogeneous multi-core processors do not.

2.1 The Utilization Wall

In this section we provide the theoretical background for the utilization wall and then provide evidence of how it is already affecting designs. We show that scaling theory predicts the utilization wall to be an exponentially worsening problem. We then lend support to this prediction with our own experiments at 90 and 45 nm process nodes.

2.1.1 Theory

The utilization wall arises because of a breakdown of the classical CMOS scaling described by Dennard [DGR+74] in his 1974 paper. The "Classical Scaling" column in Table 2.1 shows the key elements of the classical CMOS scaling model. Associated with each transition between CMOS processes is a geometry scaling factor S, typically $1.4\times$. Classical CMOS scaling holds that transistor capacitances decrease roughly by this factor of S with each process shrink. At the same time, transistor switching frequency rises by S and the number of transistors on the die increases by S^2. Until recently, it has been possible to scale supply voltage by $1/S$, leading to constant power consumption for a fixed-size chip running at full frequency. Scaling the supply voltage requires that we also scale the threshold voltage proportionally if we wish to maintain switching frequency. Historically, this was not an issue because leakage, although increasing exponentially, was not significant in process nodes above 130 nm.

In the current process scaling regime, transistor densities and speeds continue to increase with Moore's Law. Unfortunately, to curtail leakage currents, limits on threshold voltage scaling now prevent supply voltage scaling. As a result, full chip, full frequency power is now rising as S^2. The introduction of 3D CMOS technology will exacerbate this trend further. Table 2.1 summarizes these trends. The equations in the "Classical Scaling" column governed scaling up until 130 nm. The equations in the "Leakage Limited" column govern scaling at 90 nm and below.

2.1.2 Practice

The result of these trends is a technology-imposed *utilization wall* that limits the fraction of the chip we can use at full speed at one time. The effects of the utilization wall are already indirectly apparent in modern processors: Intel's Nehalem provides a "turbo mode" that powers off some cores in order to run others at higher speeds. Another strong indication is the divergence between native transistor switching speeds and processor frequencies. Although the former have continued to double every two process generations, the latter have not increased substantially over the last 5 years.

To quantify the current impact of the utilization wall, we synthesized, placed, and routed several circuits using the Synopsys Design and IC Compilers.

Table 2.1: **The utilization wall** The utilization wall is a consequence of CMOS scaling theory and current-day technology constraints, assuming fixed power and chip area. The Classical Scaling column assumes that designers can lower V_t arbitrarily. In the Leakage Limited case, constraints on V_t, necessary to prevent unmanageable leakage currents, hinder scaling and create the utilization wall.

Param.	Description	Relation	Classical Scaling	Leakage Limited
B	power budget		1	1
A	chip size		1	1
V_t	threshold voltage		$1/S$	1
V_{dd}	supply voltage	$\sim V_t \times 3$	$1/S$	1
t_{ox}	oxide thickness		$1/S$	$1/S$
W, L	transistor dimensions		$1/S$	$1/S$
I_{sat}	saturation current	WV_{dd}/t_{ox}	$1/S$	1
p	device power at full frequency	$I_{sat}V_{dd}$	$1/S^2$	1
C_{gate}	**capacitance**	WL/t_{ox}	**1/S**	**1/S**
F	**device frequency**	$\frac{I_{sat}}{C_{gate}V_{dd}}$	**S**	**S**
D	**devices per chip**	$A/(WL)$	$\mathbf{S^2}$	$\mathbf{S^2}$
P	**full die, full frequency power**	$D \times p$	**1**	$\mathbf{S^2}$
U	**utilization at fixed power**	B/P	**1**	$\mathbf{1/S^2}$

Table 2.2: **Experiments quantifying the utilization wall** We use Synopsys CAD tools and TSMC standard cell libraries to evaluate the power and utilization of a 300 mm^2 chip filled with 64-bit adders, separated by registers. We use these operators to approximate active logic in a processor.

Process	90 nm TSMC	45 nm TSMC	32 nm ITRS
Frequency (GHz)	2.1	5.2	7.3
mm^2 Per Op.	.00724	.00164	.00082
# Operators	41k	180k	360k
Full Chip Watts	455	1225	2401
Utilization at 80 W	17.6%	6.5%	3.3%

Table 2.2 summarizes our findings. For each process, we used the corresponding TSMC standard cell libraries to evaluate power and area. We filled a 300 mm^2 chip with 64-bit operators to approximate active logic on a microprocessor die. Each operator is a 64-bit adder with registered inputs and outputs, which runs at its maximum frequency in that process. In a 90 nm TSMC process, running a chip at full frequency would require 455 W. This means that only 17.6% of the chip could be simultaneously active in an 80 W budget. In a 45 nm TSMC process, a similar design would require 1225 W, resulting in just 6.5% utilization at 80 W. This shows a reduction of 2.6× attributable to the utilization wall. The equations in Table 2.1 predicted a larger, 4× reduction. The difference is due to process and standard cell tweaks implemented between the 90 nm and 45 nm generations.

2.2 Implications of the Utilization Wall

Table 2.2 also extrapolates to 32 nm based on ITRS data for 45 and 32 nm processes. ITRS roadmap projections and CMOS scaling theory suggest that this percentage will decrease to less than 3.5% in 32 nm, and will continue to decrease by almost half with each process generation. Thus, the utilization wall is getting exponentially worse, roughly by a factor of two, with each process generation.

The remainder of the transistor budget must be either left unused, leaving *dark silicon*, or purchased at the expense of underutilizing all transistors in the design. The latter renders the whole processor *dim silicon*. For scaling existing multi-core processor designs, designers have choices that span a variety of design points. However, the best they can do is exploit the factor of S (e.g., 1.4×) reduction in transistor switching energy that each generation brings. Regardless of whether designers a) increase frequency by a factor of 1.4×, b) increase core count by 1.4×, c) increase core count by 2×, and reduce frequency by 1.4×, or d) some compromise of the three, the utilization wall ensures transistor speeds and densities are rapidly out-pacing the available power budget to switch them.

The situation is brighter for less conventional designs. Provided that only a subset of the transistors are active at the same time, designers can still harness all of the available transistor budget at full speed. In a homogeneous design, turning off one compute unit in favor of an identical one offers few benefits. In contrast, for a heterogeneous design, there are clear benefits to turning off one compute unit in favor of turning on another that is more specialized for the current task.

2.3 Using Heterogeneity to Scale the Utilization Wall

Heterogeneity and specialization are effective responses to the utilization wall and the dark silicon problem. Increasingly, specialized processors offer large energy savings with little to no opportunity cost: The silicon area that they consume would otherwise go unused because of the utilization wall. Thus, specialized silicon can trade cheap area for valuable energy efficiency.

Specialization is especially profitable in extremely power-constrained designs. This includes the mobile application processors that power the world's emerging computing platforms, including cell phones, e-book readers, media players, and other portable devices. Mobile application processors differ from conventional laptop or desktop processors. They have vastly lower power budgets – often less than 375 mW – and usage is heavily concentrated around a core collection of applications. Mobile designs already tend to be heterogeneous platforms. Mobile designers reduce power consumption, in part, by leveraging customized low-power hardware implementations of common functions such as audio and video decoders and 3G/4G radio processing. These computations are highly parallel and exceptionally well-suited to traditional accelerator or custom ASIC implementations.

The remaining code (user interface elements, application logic, operating system, etc.) resembles traditional desktop code and is ill-suited to conventional, parallelism-centric accelerator architectures. This code has traditionally been of limited importance. However, the rising popularity of sophisticated mobile applications suggests this code will become more prominent and consume larger fractions of device power budgets. As a result, applying hardware specialization to frequently-executed irregular code regions will become a profitable system-level optimization. Likewise, the resemblance between these code regions and traditional desktop applications means that hardware beneficial to either should benefit both.

For parallel and highly regular regions of code, traditional accelerator approaches will continue to apply. Many already produce low-power hardware, or offer sufficient performance to trade performance for power. However, to address the challenges of the utilization wall, we must provide specialization for more than just those portions of code trivially mapped into hardware. Similarly, to be a scalable solution, we must be able to map new codes onto new or existing specialized hardware as workloads change and applications evolve. Our approach to addressing the utilization wall is the construction of a fully automated toolchain that produces specialized execution engines called conservation cores that act as drop-in replacements for existing code. In the next chapter, we will explore both the conservation core approach and the conservation core toolchain.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under NSF CAREER Awards 06483880 and 0846152, and under NSF CCF Award 0811794.

This chapter contains material from “Conservation cores: reducing the energy of mature computations”, by Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson and Michael Bedford Taylor, which appears in *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. The dissertation author was the secondary investigator and author of this paper. The material in this chapter is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for com-

This chapter contains material from "Energy-Delay Optimized Accelerators for Irregular Code", by Jack Sampson, Ganesh Venkatesh, Nathan Goulding, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor, which has been submitted for possible publication by IEEE in *Proceedings of the 43rd annual International Symposium on Microarchitecture (MICRO).* The dissertation author was the primary investigator and author of this paper.

Chapter 3

The Design of Conservation Core Enabled Systems

In the previous chapter, we saw how growing transistor counts, limited power budgets, and the breakdown of voltage scaling conspire to create a utilization wall that limits the fraction of a chip that can run at full speed at one time. This leads to an increase in dark silicon with every process generation. Likewise, we saw indications that heterogeneous systems that can translate increasing transistor budgets into increasing levels of specialization may be able to exploit the dark silicon. Our approach to heterogeneous system design is built around automatically generated specialized processors that focus on reducing energy and energy-delay for code regions with irregular control flow and memory patterns. We call these processors conservation cores, or *C-Cores*,

Incorporating C-Cores into multi-processors, especially at a scale large enough to save power across many applications with multiple hot spots, raises a number of challenges:

1. **C-Core Selection** In order to build C-Cores, we must be able to identify which pieces of code are the best candidates for conversion into C-Cores. The code should account for a significant portion of runtime and energy, and stem from a relatively stable code base.

2. **Automatic synthesis** Designing numerous C-Cores by hand is not scalable,

so it must be possible to synthesize C-Cores automatically and correctly, without significant human intervention.

3. **Programming model** It should not be necessary to rewrite applications to make use of C-Cores. The system must utilize them automatically.

4. **Longevity** Conservation cores should remain useful even as the code they are designed to replace evolves.

5. **System integration** Since C-Cores should work seamlessly with existing code, the C-Core hardware and memory model must be tightly integrated with the rest of system.

In this chapter, we develop the conservation core architecture and show how our design addresses programming model and system integration challenges. We present our vision for how C-Cores will be selected, created, integrated, used, and adapted to workload and application changes over time. We call this the C-Core's "life cycle," and present our toolchain which automates each of the stages in that life cycle. We describe our toolchain in detail and evaluate the prototype C-Cores that it produces. Subsequent chapters will focus on improving both the performance and energy efficiency of these initial prototype C-Cores.

3.1 Conservation cores: System overview

This section provides an overview, describing the composition of a prototypical C-Core system. Architectures based on specialized hardware must address three core issues: 1) how the specialized core maintains coherence with the host system, 2) how the specialized core integrates with more general processing resources to handle code that does not justify building specialized hardware, and 3) how the system withstands changes to the software that it targets. We address these three core questions in turn, discussing the sharing of our coherent L1, the movement of execution between CPU and C-Core, and our approach to handling target application program changes.

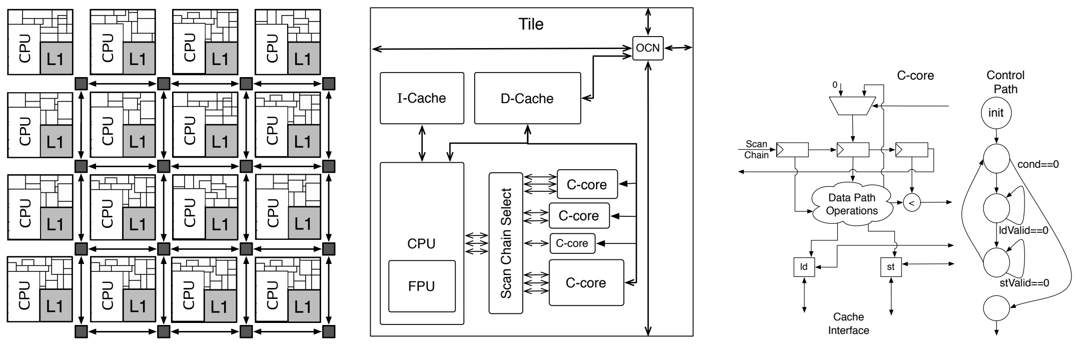


Figure 3.1: **The high-level structure of a C-Core-enabled system** A C-Core-enabled system (a) is made up of multiple individual tiles (b), each of which contains multiple C-Cores (c). Conservation cores communicate with the rest of the system through a coherent memory system and a simple scan-chain-based interface. Different tiles may contain different C-Cores. Not drawn to scale.

3.1.1 Basic chip-level architecture

A C-Core-enabled system includes multiple C-Cores embedded in a multi- or many-core tiled array like the one in Figure 3.1(a). Each tile of the array contains a general purpose processor (the "CPU"), cache and interconnect resources, and a collection of tightly-coupled C-Cores. Collectively, the C-Cores and CPU share the tile's resources, including the coherent L1 data cache, the on-chip network interface, and FPU.

Within a tile (Figure 3.1(b)), the C-Cores interface to the host CPU via a direct, multiplexed connection to the L1 cache. They also connect directly to the host CPU through a collection of scan chains. These allow the CPU to read and write all state within the C-Cores. The CPU uses these scan chains for passing arguments, for context switching, and for patching the C-Cores. The scan chain interface is covered in more detail in section 3.3.2. These facilities allow the system to reconfigure a C-Core to run future and past modified versions of the source code that was used to generate the C-Cores. Most data transfer occurs through the coherent L1 cache connection.

A shared, coherent cache is an attractive option for conservation cores because of their focus on irregular code. With irregular accesses and control flows unsuited to modulo scheduling, C-Cores are sensitive to memory latency as well as bandwidth. Caching can greatly reduce average memory access time, especially for dependent loads. As C-Cores already target code with irregular control flows, the variability in memory access time doesn't add significant additional complexity.

By having C-Cores share the same path to memory as the processor, drop-in semantics are easier to maintain. Likewise, C-Cores are trivially kept coherent with processor memory because the accesses go to the same cache. For applications with cache-friendly access patterns, caching also improves energy efficiency over directly accessing main memory.

3.1.2 Execution model

Each C-Core targets frequently executed, or hot, regions of an application. It achieves energy and power savings by creating specialized hardware datapaths that eliminate much of the overhead in conventional processor pipelines. Overheads reduced or removed include instruction fetch, register file accesses, and bypassing. These datapaths are controlled by a set of state machines that closely mirror the control flow graph of the source program. This mirroring allows for precise replication of the same semantics that the code would have if it were executing on the CPU. The shared, coherent L1 makes this as true for memory state as for control flow.

Portions of applications not important enough to be supported by C-Cores continue to run on the CPU. The CPU also serves as a fallback to support applications that were not available at the time of the manufacture of the chip. Similarly, exceptional behavior is supported with traps to the CPU. Execution shifts back and forth between the CPU and various C-Cores as an application enters and exits the code regions that C-Cores support.

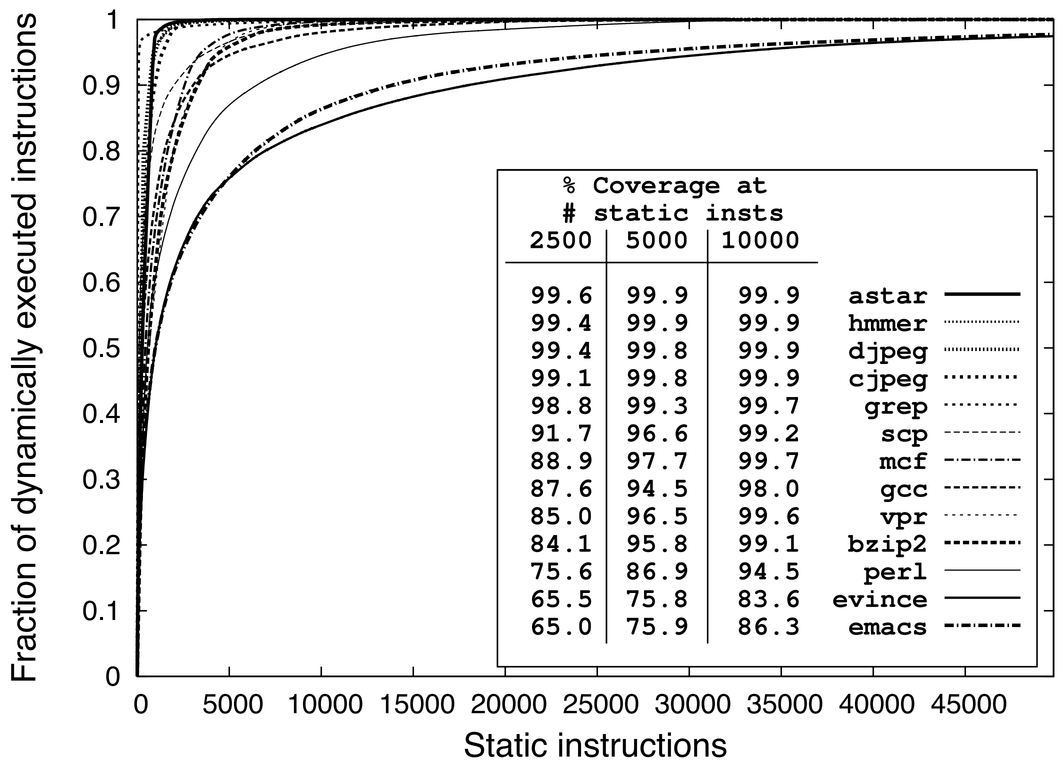
3.1.3 Future proofing

Although the C-Cores are created to support existing versions of specific applications, they also need to support newer versions that are released after the original C-Cores were synthesized. To do this, we implement all three reconfiguration mechanisms described in [VSG+10]. The C-Cores include reconfiguration bits which allow the behavior of C-Cores to adapt to commonly found changes in programs. Small changes, such as replacing one constant value in a program with another, are handled within the C-Core via reconfiguration. Larger changes are handled by forcing traps to software on certain CFG edges. Section 3.4 briefly discusses how the patching process fits into our automated toolchain and how these reconfiguration bits are configured. However, the details of the patching algorithms are outside the scope of this dissertation and can be found in [VSG+10].

3.2 The C-Core Life Cycle

For the C-Core approach to be successful, we must be able to map most execution onto C-Cores. Furthermore, for C-Cores to achieve high coverage in a reasonable amount of area, a relatively small fraction of the application's static instructions must account for a large fraction of execution. Fortunately, this is true for many programs. Figure 3.2 shows the fraction of dynamically executed x86 instructions (y-axis) covered by the number of static x86 instructions (x-axis) for a broad-ranging set of applications. These applications include SPECCPU2006 integer benchmarks astar and hmmer, desktop applications evince, emacs, grep, gcc, perl, and scp, and five applications for which we will construct C-Cores prototypes in this chapter. For many applications, the curve is quite steep. This means that converting a relatively small number of static instructions into hardware will cover a very large fraction of execution.

The C-Core approach must be scalable in design effort as well as area. We will address this through automation. As workloads change, new applications and new versions of applications can be run through our automated toolchain. Drop-in semantics allow the specialized hardware to be used transparently, without manual code changes. While this approach imposes some constraints on potential optimizations, especially on the memory system, it makes the conservation core approach scalable in effort.



% Coverage at # static insts

2500	5000	10000	
99.6	99.9	99.9	astar
99.4	99.9	99.9	hmmer
99.4	99.8	99.9	djpeg
99.1	99.8	99.9	cjpeg
98.8	99.3	99.7	grep
91.7	96.6	99.2	scp
88.9	97.7	99.7	mcf
87.6	94.5	98.0	gcc
85.0	96.5	99.6	vpr
84.1	95.8	99.1	bzip2
75.6	86.9	94.5	perl
65.5	75.8	83.6	evince
65.0	75.9	86.3	emacs

Figure 3.2: **Dynamic coverage for given static instruction counts** We show the cumulative distribution of dynamic coverage (on the vertical axis) as a function of the number of static instructions executed. For many x86 programs profiled, a small number of static instructions cover much of dynamic execution. This implies that a small amount of specialized hardware could cover large portions of execution for each of these programs.

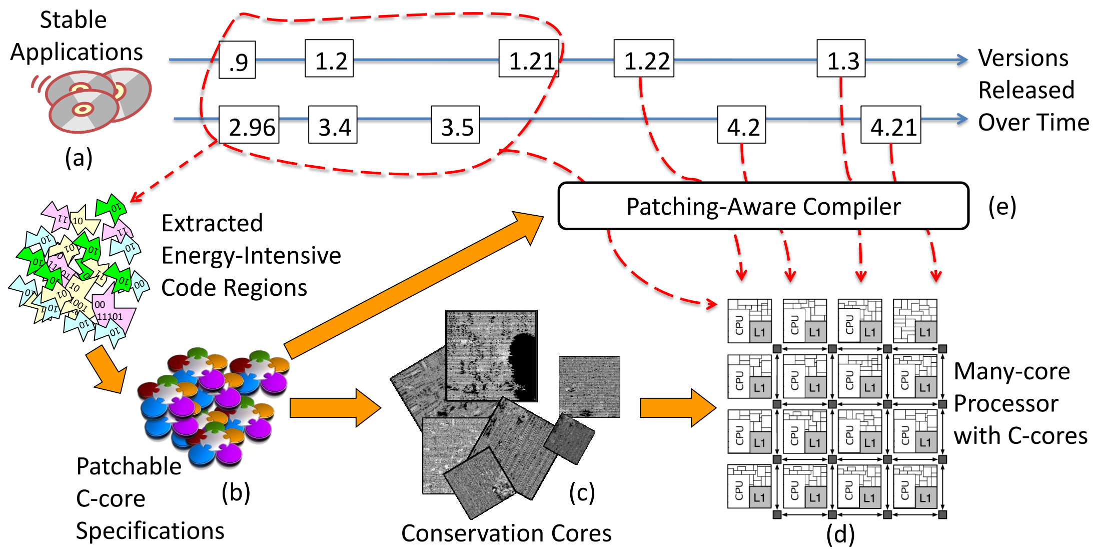

Stable Applications
(a)
.9
1.2
1.21
1.22
1.3
2.96
3.4
3.5
4.2
4.21
Versions Released Over Time
Patching-Aware Compiler
(e)
Extracted Energy-Intensive Code Regions
Patchable C-core Specifications
(b)
(c)
Conservation Cores
Many-core Processor with C-cores
(d)
CPU
L1

Figure 3.3: **The C-Core Life Cycle** A profiler (a) extracts a set of energy-intensive code regions from a corpus of stable applications in the processor's target domain. The toolchain generalizes these regions into patchable C-Core specifications (b), translates them into circuits via an automated synthesis infrastructure (c), and integrates them into a multi-core processor (d). A patching-aware compiler maintains a database of C-Core specifications and generates binaries that execute on a combination of C-Cores and the local CPU. The compiler (e) generates a patching configuration that allows the C-Cores to run future (and past) versions of the original applications.

While hardware release cycles are long, software release cycles are often much more rapid. Our design vision for C-Cores takes this into account in two key ways: First, we limit the non-recurring engineering costs for successive generations of C-Core enabled systems through automation. It is not feasible to rely on manual specification of every hot spot in every workload every time a new system is designed. Therefore, we have heavily automated this process. Second, we ensure that a C-Core enabled system can retain the utility of its C-Cores over the expected lifetime of the system. For this, we rely heavily on the patching mechanisms proposed in [VSG$^+$10], and automatic generation of the requisite patches via our compiler infrastructure.

Figure 3.3 depicts the generation of a many-core processor equipped with C-Cores and the toolchain needed to target them. The process starts with the processor designer characterizing the workload. This requires identifying applications that make up a significant fraction of the processor's target workload. The toolchain begins by extracting from these applications the most frequently used (or "hot") code regions (a). It then augments them with a small amount of reconfigurability (b) and synthesizes C-Core hardware using a 45 nm standard cell CAD flow (c). A single processor contains many tiles, each with a general purpose CPU and collection of different C-Cores (d).

In order to generate code for the processor, we extend a standard compiler infrastructure to support automatic code generation for C-Cores. In our case, we use a combination of LLVM [LA04], OpenIMPACT [Ope] and GCC. Any standard compiler infrastructure such as GCC, the Intel C++ compiler, or Microsoft Visual Studio would have been suitable. The compiler incorporates a description of the C-Core that the manufacturer has shipped silicon for. The compiler uses a matching algorithm to find similarities between the input code and the C-Core specifications (e). In cases where there are close matches, the compiler will generate both CPU-only object code and object code that makes use of the C-Core. The latter version of the code includes patching information that is downloaded into the C-Core via scan chains. This transfer occurs before the C-Core is invoked. The decision of whether to use the C-Core-enabled version of the code or the "software-only" version is made at run time, based on C-Core availability and other factors.

The next several sections describe how we turn our design vision into a concrete system. Section 3.3 describes the C-Core architecture. Sections 3.5, 3.6, and 3.4 detail our toolchain for automating the C-Core life cycle. We focus first on our system level architecture.

3.3 Conservation core architecture

This section describes the architecture of a C-Core-enabled system in more detail. We describe the organization of the C-Cores themselves and the interface between the C-Cores and the rest of the system.

3.3.1 Conservation core organization

Each C-Core acts as a drop-in replacement for a region of code. An individual C-Core comprises a datapath and control state machine derived directly from the code it targets. Figure 3.1(c) shows the architecture of a prototypical C-Core. The principle components are the datapath, the control unit, the cache interface, and the scan chain interface to the CPU.The scan chain interface provides access to internal state. Specialized load and store units share ports to memory and use a simple token-based protocol to enforce correct memory ordering. For simplicity of correctness in our C-Core prototypes, this protocol restricts the number of concurrent memory accesses to one. Later, in Chapter 4, we will investigate performance optimizations for the memory ordering interface.

Datapath and control By design, the C-Core datapath and control very closely resemble the internal representation that our toolchain extracts from the C source code. The datapath contains the functional units (adders, shifters, etc.), the muxes to implement control decisions, and the registers to hold program values across clock cycles.

The control unit implements a state machine that mimics the control flow graph of the code. Every basic block in the code's original CFG has at least one corresponding state in the C-Core's control unit. Where there are memory or other variable latency operations within a basic block, the basic block is partitioned into multiple basic blocks. One control state is then created for every basic block in the new CFG. The control unit tracks branch outcomes (computed in the datapath) to determine which state to enter on each cycle. The control path sets the enable and select lines on the registers and muxes so that the correct basic block is active each cycle.

The close correspondence between the program's structure and the C-Core is important for two reasons: First, it makes it easier to enforce correct memory ordering in irregular programs. The ordering that the control path's state machine enforces corresponds to the order that the program counter provides in general purpose processors. We use that ordering to enforce memory dependencies. Second, close correspondence enhances robustness. Structural similarity improves the likelihood that small changes in the source code (which are the common case) will require correspondingly small reconfiguration facilities from the hardware. This maximizes the probability that the patching mechanisms from [VSG+10] will provide sufficient flexibility to adapt to the application changes.

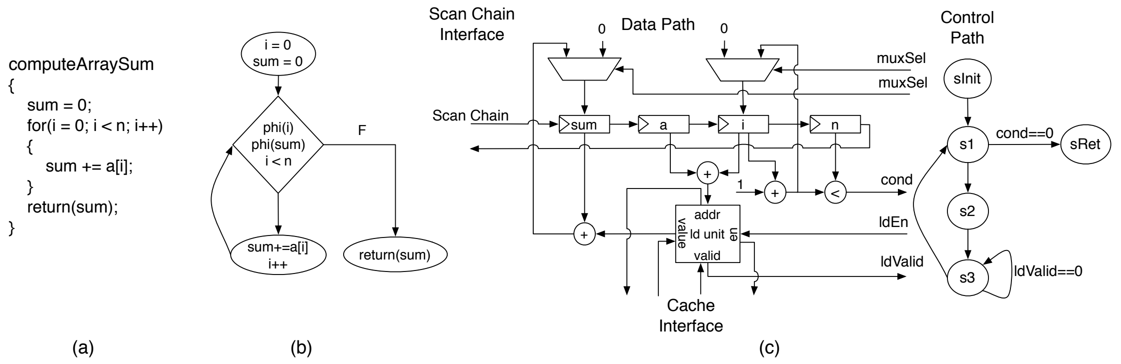
To maintain this correspondence and to reduce the number of registers required in the datapath, the registers in the C-Core datapaths adhere to SSA form: Each static SSA variable has a corresponding register. This invariant minimizes the number of register updates: Exactly one register value changes per new value that the program generates.

Memory interface and ordering Memory operations require special attention to ensure that the C-Core enforces memory ordering constraints. Our prototype conservation cores enforce these constraints by allowing only one memory operation per basic block. The C-Core only activates one basic block at a time, guaranteeing that memory operations within the C-Core execute in the correct order. During C-Core execution, the CPU is inactive and all CPU accesses will have completed before C-Core execution begins or resumes. Thus, as both C-Cores and the CPU access the same cache, ordering between CPU and C-Core memory accesses is trivially maintained, although this does limit potential memory optimizations for the C-Core.

The load/store units connect to a coherent data cache. This ensures that all loads and stores are visible to the rest of the system regardless of which addresses the C-Core accesses.

Cache accesses can take multiple cycles. Moreover, the number of cycles is variable. To address this, the toolchain adds a self-loop to the basic block that contains each memory operation and exports a “valid” line to the control path. When the memory operation is complete, it asserts the “valid” signal and control exits the loop and proceeds with the following basic block. The “valid” signal is similar to the memory ordering token used in systems such as Tartan [MCC+06] and WaveScalar [SSM+07].

Most of the communication between C-Cores and the CPU occurs via the



```
computeArraySum
{
   sum = 0;
   for(i = 0; i < n; i++)
   {
      sum += a[i];
   }
   return(sum);
}
```

Figure 3.4: **Conservation core example** An example showing the translation from C code (a), to the compiler's internal representation (b), and finally to hardware (c). The hardware schematic and state machine correspond very closely to the data and control flow graphs of the C code.

shared L1 cache. A coherent, shared memory interface allows us to construct C-Cores for applications with unpredictable access patterns. Conventional accelerators cannot speed up these applications, because they cannot extract enough memory parallelism. Such applications can be an excellent fit for C-Cores, however, as performance is not the primary concern. Since the CPU and C-Cores do not simultaneously access the cache, the impact on the CPU cycle time is negligible.

Multi-cycle instructions Conservation cores handle other multi-cycle instructions (e.g., integer division and floating point operations) in the same way as memory operations. Each multi-cycle instruction resides in a basic block with a self-loop and generates a "valid" signal when it completes.

Example Figure 3.4 shows the translation from C souece code (a) to hardware schematic and state machine (c). The hardware corresponds very closely to the CFG of the sample code (b). It has muxes for variables *i* and *sum* corresponding to the *phi* operators in the CFG. Also, the state machine of the C-Core is almost identical to the CFG, but with additional self-loops for multi-cycle operations. The datapath has a load unit to access the memory hierarchy to read the array *a*.

3.3.2 The CPU/C-Core interface

Aside from the cache, the only connection between the CPU and the C-Cores is a set of scan chains. These scan chains are the mechanism that allows the CPU access to all of the C-Core's internal state. The CPU side of the interface is shared among all C-Cores on the CPU's tile. The CPU can communicate via scan chains with only one C-Core at a time, with switching controlled by the CPU. The CPU uses the scan chains to install patches that will be used across many invocations, and to pass initial arguments for individual invocations. The scan chains also allow the CPU to read and modify internal C-Core state to implement exceptions.

Conservation core scan chains are divided into two groups, fixed-function and data. There are a small number of short, fixed-function scan chains for control, argument passing, and patch installation. The bulk of the scan chains provide access to datapath registers. A C-Core may have up to 32 such data scan chains.

The scan chains for arguments are short (just 64 bits) to make invocation fast, but the patch installation scan chains can be much longer. In our biggest C-Core prototype, the patch chains reach up to 12,772 bits. However, patch installation is infrequent, so the cost of accessing the scan chain is minor: In general usage, patches will only be installed once per program invocation. A special "master control" scan chain contains a single bit and allows the CPU to start and stop the C-Core's execution as needed.

Datapath scan chains allow the CPU to manipulate arbitrary execution state during an exception. Datapath scan chains range in length from 32 to 448 bits in our largest C-Core.

To access the interface, the CPU provides three new instructions: Move-From-ScanChain (MFSC), Move-To-ScanChain (MTSC), and ScanChain-Rotate-Left (SCRL). MFSC moves the 32 bits at the head of a scan chain into a general purpose processor register, and MTSC does the reverse. SCRL rotates a scan chain left by n bits. SCRL executes asynchronously but MFSC and MTSC have blocking semantics: They will wait for previously issued SCRLs on a scan chain to finish before returning a value from that scan chain. As the C-Cores are drop-in replacements for existing code, programs need not block if the correct C-Core is not available (i.e., if it is in use by another program or currently configured with the incorrect patch). The original CPU (software) implementation is still available, and the program can use it instead.

3.4 Patching conservation cores

When a C-Core-equipped processor ships, it can run the latest versions of the targeted applications without modification. We refer to this version as the *original*. When a new version of an application becomes available, our toolchain must determine how to map the new version of the software onto the existing C-Core hardware. We refer to the new version of the software as the *target*. The goal of the patching process is to generate a *patch* for the original hardware that will let it run the target software version.

The longevity provided by patching is an important aspect of the C-Core life cycle. However, the details of the patching process are outside the scope of this dissertation. Below, we briefly describe how we support the patching approach from [VSG$^+$10] in C-Cores.

3.4.1 Integrating patching support

The bold box in Figure 3.5 shows how the patching system fits into the toolchain. We work directly on the program's dataflow and control graphs, a representation that can be generated from either source code or a compiled binary. Our implementation of the reconfiguration technique described in [VSG$^+$10] provides C-Cores with three facilities to adjust their behavior after fabrication.

Configurable constants We generalize all hard-coded immediate values into configurable registers. This supports changes to the values of compile-time constants and the insertion, deletion, or rearrangement of structure fields.

Generalized single-cycle datapath operators To support the replacement of one operator with another, we generalize several operators. Any addition or subtraction is replaced by an adder-subtractor, any comparison operation by a generalized comparator, and any bit-wise operation by a bit-wise ALU. A small, four-bit configuration register is then added for each such operator, determining which operation is currently active.

Control flow changes In order to handle changes in the CFG's structure and changes to basic blocks that go beyond what the above mechanisms can handle, the C-Cores provide a flexible exception mechanism. The control path contains a bit for each state transition that determines whether the C-Core should treat it as an exception. This same mechanism is also used to handle system calls and other features not yet supported within our C-Core prototypes.

When the state machine makes an exceptional transition, the C-Core stops executing and transfers control to the general-purpose core. The exception handler begins by extracting current variable values from the C-Core via the scan-chain-based interface. It then executes the replacement code segment, transfers new values back into the C-Core, and resumes execution. The exception handler can restart C-Core execution at any point in the CFG, so exceptions can arbitrarily alter control flow and/or replace arbitrary portions of the CFG.

3.4.2 Patch generation

The patching algorithm developed by Venkatesh, et al. [VSG+10] proceeds in four stages: basic block mapping, control flow mapping, register remapping, and patch generation. The details and implementation of the patching algorithm are outside the scope of this dissertation, but we will briefly describe its operation: The algorithm operates at basic block granularity. It identifies potential matches, based on internal structure, from basic blocks in the target versions to those in the original. Then, the CFG and a register renaming scheme are used to further refine whether two blocks are a match. Matching connected subgraphs of the CFG are then identified as *hardware regions*, portions of the target CFG that will execute in hardware. Remaining portions of the CFG are considered *software regions* and entry into them will be treated as an exception. Once the patching algorithm has identified all such regions, our toolchain can begin patch generation.

At this point we have all the information required to generate a patch that will let the target code run on the original hardware. The patch itself consists of three parts:

- the configuration bits for each of the configurable datapath elements along with values for each of the configurable constant registers
- exception bits for each of the control flow edges that pass from a hardware region into a software region
- code to implement each of the software regions

The software region code is subdivided into three sections. First, the *prologue* uses the scan chain interface to retrieve values from the C-Core's datapath into the processor core. Next, the *patch code* implements the software region. The region may have multiple exit points, each leading back to a different point in the datapath. At exit, the *epilogue* uses the scan chain interface again to insert the results back into the datapath and return control to the C-Core.

3.5 C-Core selection

In order to evaluate a prototype C-Core-enabled system, we must first construct a set of C-Core prototypes. Over the course of the following sections, we will show how we have developed our automated toolchain to support each phase of a C-Core's life cycle. We also discuss the measurement and validation infrastructure built into our toolchain that provides the data used throughout this dissertation. We begin with the selection phase of the C-Core life cycle.

To select regions of code appropriate for transforming into C-Cores, we must accurately estimate the area and energy-saving potential of software to hardware conversion. This includes identifying key and constraining overheads. The overhead of transferring control to and from the C-Core is an important factor in C-Core selection. This overhead places a lower bound on how much work a C-Core must do to achieve net benefits. It also limits how frequently we can use the exception mechanism during patching without nullifying the C-Core's energy gains.

To quantify this cost, we created a C-Core that returns immediately and executed it in a tight loop. The results show that transferring control to and from the C-Core takes, on average, 317 cycles or 211 ns. Of this, the stub function that invokes the C-Core accounts for 63%. Using the scan chains to pass arguments and return values to and from the C-Core accounts for the remaining 37%.

The exception cost likewise affects selection criteria for C-Cores using the exception mechanism to perform system calls or other features currently not directly supported on C-Cores. To quantify this cost, we synthesized a simple C-Core that executed an empty loop. We compared the run time of the C-Core with a patched version of the same C-Core. The patched version used the exception mechanism to transfer control to the CPU and directly back again. The transfer takes 260 cycles (173 ns) on average.

With invocation and exception overheads known, we can filter the list of high-coverage code regions and target the appropriate levels of the call-graph. We have implemented a completely automated tool for C-Core selection and isolation, but it has so far proved difficult to fine-tune. For the results throughout this dissertation, we instead rely on a hybrid method for C-Core selection. After automatic profiling, there is a manual review of the suggested code regions, filtering out any problematic selections. The filtered list is then fed back into the toolchain, which will automatically perform all processing required to turn those selected code regions into C-Cores.

3.6 Automatic Synthesis of C-Cores

Our C-Core synthesis toolchain automatically transforms portions of C programs into silicon. In the previous section, we discussed how hot regions are profiled and marked for extraction. Once we have wrapped these functions with stubs, we compile them into an intermediate three-address representation. We split these functions into datapath and control segments, and then uses a state-of-the-art EDA tool flow to generate a circuit fully realizable in silicon. The toolchain also generates a cycle-accurate system simulator for the new hardware. We use the simulator for performance measurements and to generate traces that drive Synopsys VCS and PrimeTime simulation of the placed-and-routed netlist. Below, we describe these components in more detail.

3.6.1 Compilation Toolchain

Figure 3.5 summarizes the C-Core toolchain. The toolchain is based on the OpenIMPACT (1.0rc4) [Ope], CodeSurfer (2.1p1) [Cod], and LLVM (2.4) [LA04] compiler infrastructures. It accepts a large subset of the C language, including arbitrary pointer references, switch statements, and loops with complex conditions.

In the C-Core identification stage, functions or subregions of functions (e.g., key loops) are tagged for conversion into C-Cores based on profile information. The toolchain uses outlining to isolate the region and then uses exhaustive inlining to remove function calls. We pass global variables by reference as additional input arguments.

The C-to-Verilog stage generates the control and dataflow graphs for the function in SSA [CFR+89] form. This stage then adds basic blocks and control states for each memory operation and multi-cycle instruction. The final step of the C-to-Verilog stage generates synthesizeable Verilog for the C-Core. This requires converting ϕ operators into muxes, inserting registers at the definition of each value, and adding self loops to the control flow graph for the multi-cycle operations. Then, it generates the control unit with a state machine that matches the control flow graph. This stage of the toolchain also generates a cycle-accurate module for our architectural simulator.

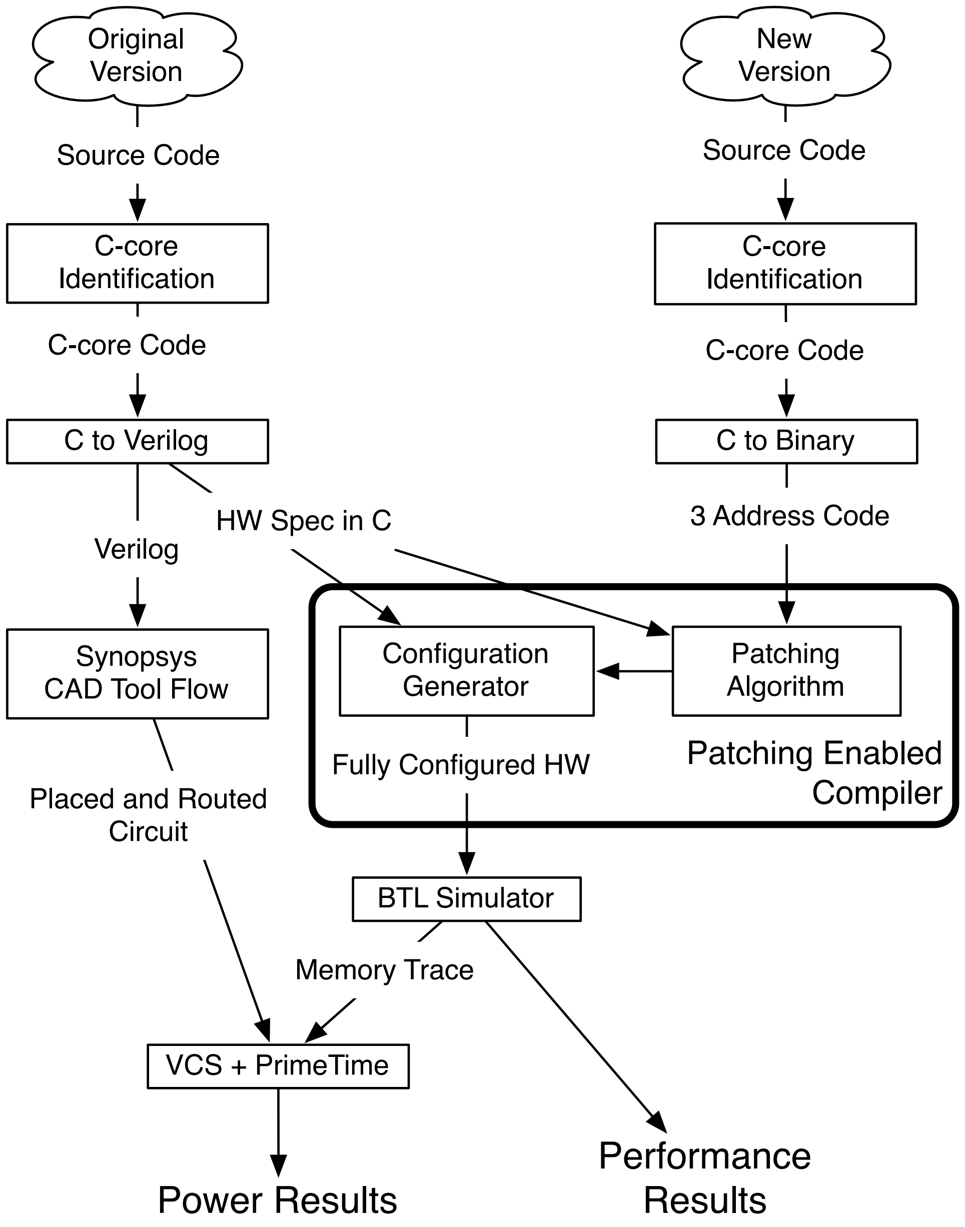


Figure 3.5: **The C-Core C-to-hardware toolchain** Our toolchain proceeds through hardware generation, patching, simulation, and power measurement over several stages. Program source enters our toolchain and passes through our C-Core selection stage, which identifies regions of code as C-Core candidates. Our C-to-Verilog compiler transforms these regions and produces a cycle-accurate simulator model for each region. Our simulator uses these models to produce traces, which we pass to VCS and PrimeTime to generate power results. The bold box contains the patch generation infrastructure based on our patching enabled compiler.

3.6.2 Simulation infrastructure

Our cycle-accurate simulation infrastructure is based on *btl*, the Raw simulator [TLM$^+$04]. We have modified btl to model a cache-coherent memory among multiple processors, to include a scan chain interface between the CPU and all of the local C-Cores, and to simulate the C-Cores themselves. The prototype C-Cores may operate at different clock frequencies from each other and from the core clock. These different cycle times are modeled in simulation by adjusting the duty cycle in clocking the C-Core.

In our prototype C-Core system model, the cache clock is synchronized to the C-Core when control is transferred to the C-Core. Minimum load-use latency for prototype C-Cores, as frequency is usually less than system frequency, is set at two C-Core cycles. In Chapter 4, we introduce a redesign of the C-Core pipeline such that all C-Cores operate synchronized with the system memory frequency, simplifying frequency modeling. Results in and after Chapter 4 presume a three-cycle load-use latency, matching that of the CPU.

3.6.3 Synthesis

For synthesis we target a TSMC 45 nm GS process using Synopsys Design Compiler (C-2009.06-SP2) and IC Compiler (C-2009.06-SP2). Our toolchain generates synthesizeable Verilog and automatically processes the design in the Synopsys CAD tool flow. The flow starts with netlist generation and continues through placement, clock tree synthesis, and routing before performing post-route optimizations. We specifically optimize for speed and power. We also make use of the Synopsys Module Compiler (C-2009.06-ICC-SP2), in order to generate technology-specific custom functional units for basic arithmetic (addition, multiplication, etc.) and bit-shifting. These modules have been optimized for speed and power usage.

Figure 3.6 shows a placed and routed standard cell layout for an automatically generated prototype C-Core from the MCF 2006 application. Over 50% of the area is devoted to performing arithmetic operations in the datapath, 7% is dedicated to the control logic, and 40% is registers. This circuit meets timing at clock frequencies up to 1412 MHz.

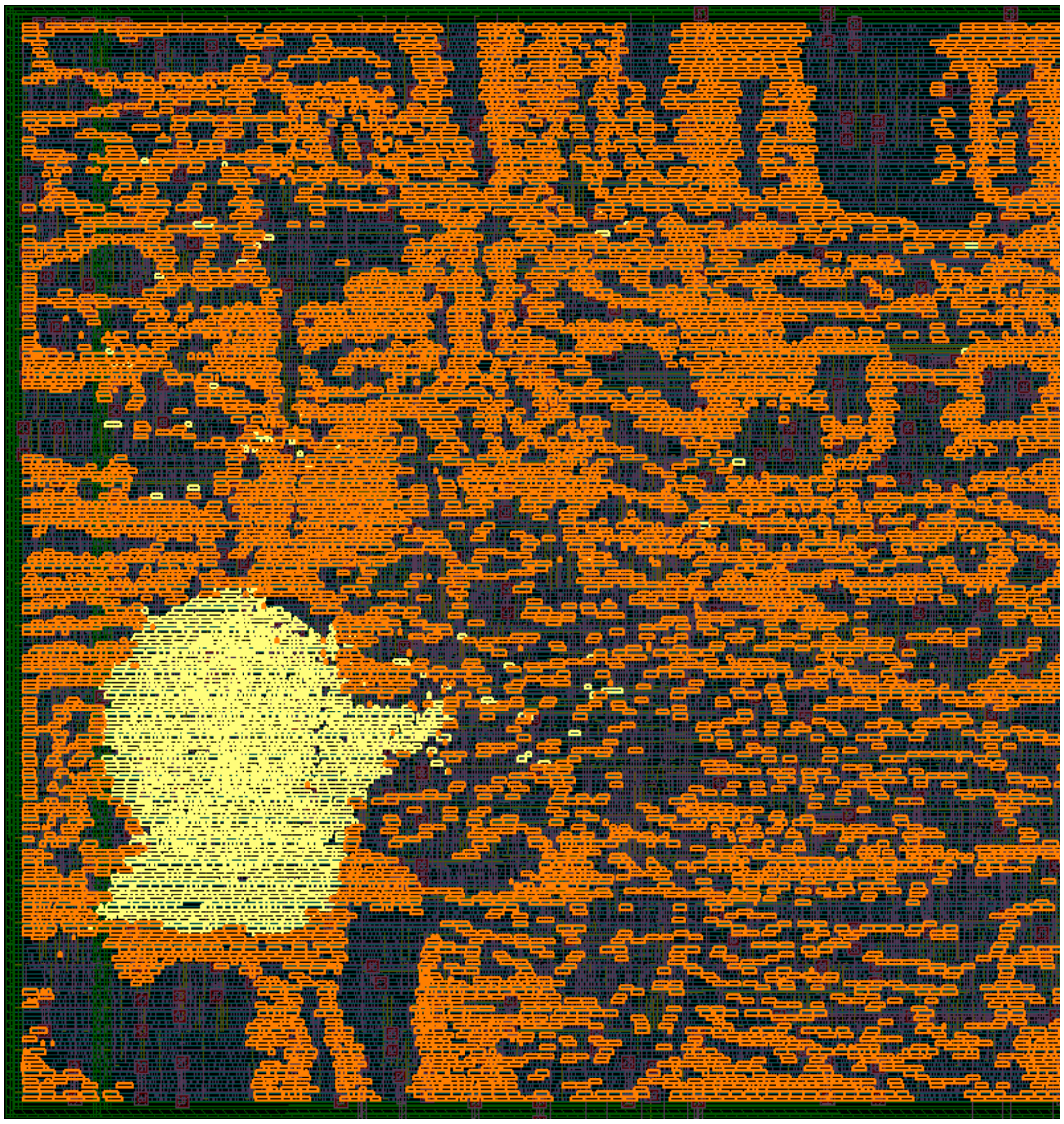


Figure 3.6: **MCF 2006 conservation core for primal_bea_mpp() function** The C-Core synthesizes to $0.077mm^2$, operates at speeds up to 1412 MHz, and provides 53% coverage for the application. The light gray elements are datapath logic (adders, comparators, etc.), dark gray elements are registers, and the white elements constitute the control path.

3.6.4 Power measurements

In order to measure C-Core power usage, our simulator periodically samples execution by storing traces of all inputs and outputs to the C-Core. Each sample starts with a "snapshot" recording the entire register state of the C-Core and continues for 10,000 cycles. The current sampling policy is to sample 10,000 out of every 50,000 cycles, and we discard sampling periods corresponding to the initialization phase of the application.

We feed each trace sample into the Synopsys VCS (C-2009.06) logic simulator. Along with the Verilog code our toolchain also automatically generates a Verilog testbench module for each C-Core. This testbench initiates the simulation of each sample by scanning in the register values from each trace snapshot. The VCS simulation generates a VCD activity file, which we pipe as input into Synopsys PrimeTime (C-2009.06-SP2). PrimeTime computes both the static and dynamic power for each sampling period. We model fine-grained clock gating for inactive C-Core states via post-processing.

3.7 Results

This section describes the performance and efficiency of our C-Core-enabled architecture and the impact of our C-Core prototypes on performance and energy consumption. We will also examine the effectiveness of our prototype C-Core-enabled architecture at the application level, focusing on system energy consumption. We will show that our C-Core prototypes succeed at greatly reducing energy for converted code regions, by up to 16×. Benefits at the system level are likewise substantial, but limited by coverage and other overheads. However, we will also see that, in their current state, our prototype C-Cores offer limited performance benefits. Improving on the design of our C-Core prototypes, optimizing for performance and increasing system efficiency, is the focus of Chapters 4 and 5.

3.7.1 Methodology

The results in this section, and throughout this dissertation, rely on outputs from the toolchain described in detail in section 3.6. The primary components of this toolchain produce full-system performance numbers, and provide detailed power and performance models for the C-Cores themselves. To model power for other system components, we derive processor and clock power values from specifications for a MIPS 24KE processor in TSMC 90 nm and 65 nm processes [MIP09], and component ratios for Raw reported in [KTMW03]. We have scaled these values for a 45 nm process and assume a MIPS core frequency of 1.5 GHz with 0.077 mW/MHz for average CPU operation. Finally, we use CACTI 5.3 [TMAJ08] for I- and D-cache power.

3.7.2 Energy savings

Figure 3.7 shows the relative energy efficiency, EDP improvement, and speedup of prototype C-Cores vs. a MIPS processor executing the same code. For fairness, and to quantify the benefits of converting instructions into our C-Core prototypes, we exclude cache power for both cases. The figure compares both patchable and non-patchable C-Cores to a general-purpose MIPS core for six versions of bzip2 (1.0.0−1.0.5), and two versions each of cjpeg (v1−v2), djpeg (v5−v6), mcf (2000−2006), and vpr (4.22−4.30). Table 3.1 summarizes the C-Core prototypes.

The data show that patchable C-Cores are, on average, 9.52× as energy-efficient as a MIPS core at executing the code they were built to execute. The non-patchable C-Core prototypes are even more energy efficient, but their inability to adapt to software changes limits their useful lifetime: Results from [VSG+10] showed that, without patching support, application specific hardware for four out of five applications was unable to run any later application version.

Table 3.1: **Conservation core prototype statistics** The prototype C-Cores we generated vary greatly in size and complexity. In the "Key" column, the letters correspond to application versions and the Roman numerals denote specific functions from the application that a C-Core targets. "LoC" is lines of C source code, and "% Exe." is the percentage of execution that each function comprises in the application.

C-Core	**Ver.**	**Key**	**LoC**	**Ops**	**% Exe.**	**Area (mm^2)**		**Frequency (MHz)**	
						Non-P.	Patch.	Non-P.	Patch.
bzip2									
fallbackSort	1.0.0	A *i*	231	647	71.1	0.128	0.275	1345	1161
fallbackSort	1.0.5	F *i*	231	647	71.1	0.128	0.275	1345	1161
cjpeg									
extract_MCUs	v1	A *i*	266	406	49.3	0.108	0.205	1556	916
get_rgb_ycc_rows	v1	A *ii*	39	68	5.1	0.020	0.044	1808	1039
subsample	v1	A *iii*	40	85	17.7	0.023	0.039	1651	1568
extract_MCUs	v2	B *i*	277	406	49.5	0.108	0.205	1556	916
get_rgb_ycc_rows	v2	B *ii*	37	68	5.1	0.020	0.044	1808	1039
subsample	v2	B *iii*	36	85	17.8	0.023	0.039	1651	1568
djpeg									
jpeg_idct_islow	v5	A *i*	223	432	21.5	0.133	0.222	1336	932
ycc_rgb_convert	v5	A *ii*	35	82	33.0	0.023	0.043	1663	1539
jpeg_idct_islow	v6	B *i*	236	432	21.7	0.135	0.222	1390	932
ycc_rgb_convert	v6	B *ii*	35	82	33.7	0.024	0.043	1676	1539
mcf									
primal_bea_mpp	2000	A *i*	64	144	35.2	0.033	0.077	1628	1412
refresh_potential	2000	A *ii*	44	70	8.8	0.017	0.033	1899	1647
primal_bea_mpp	2006	B *i*	64	144	53.3	0.032	0.077	1568	1412
refresh_potential	2006	B *ii*	41	60	1.3	0.015	0.028	1871	1639
vpr									
try_swap	4.22	A *i*	858	1095	61.1	0.181	0.326	1199	912
try_swap	4.3	B *i*	861	1095	27.0	0.181	0.326	1199	912

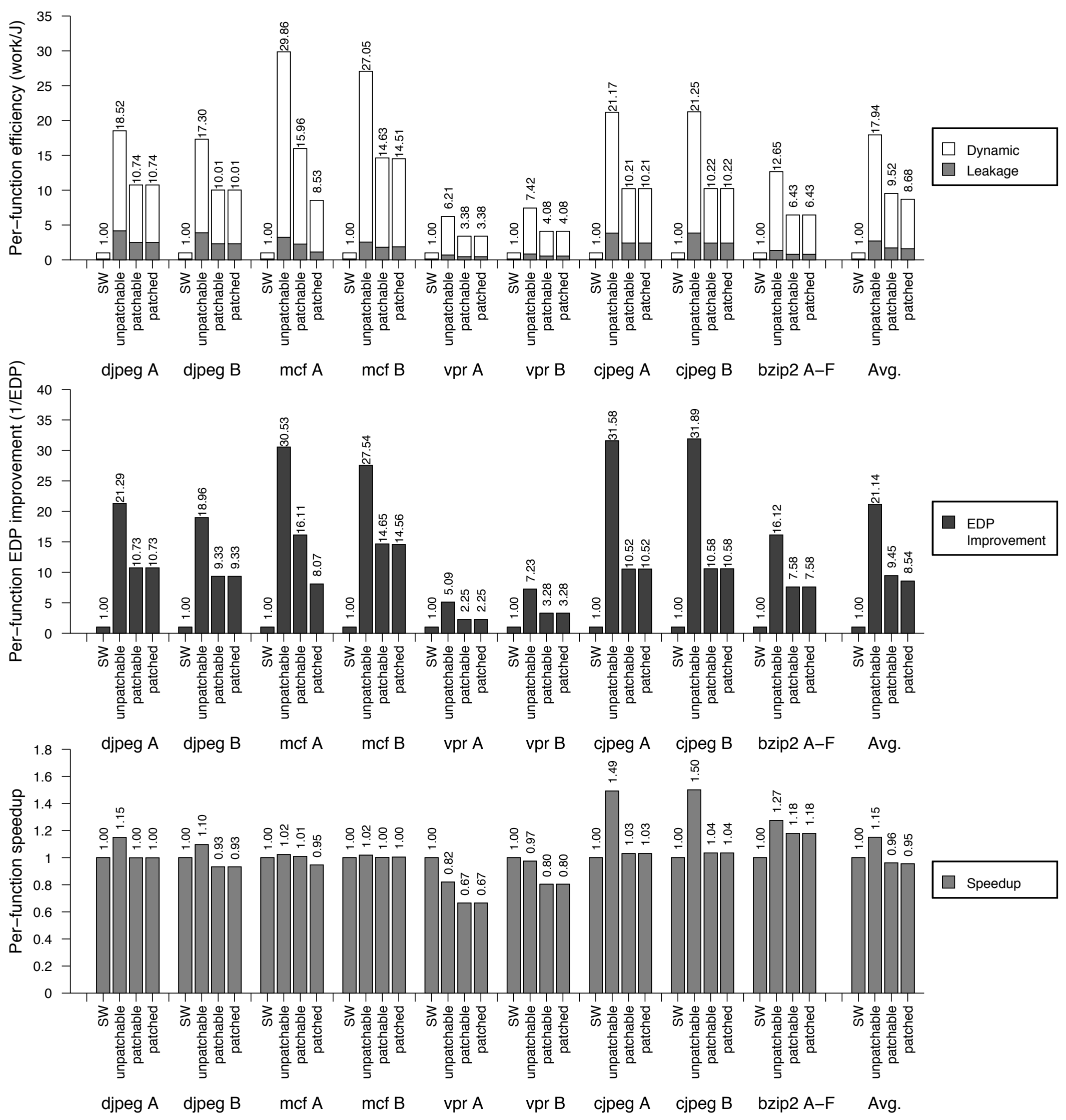


Figure 3.7: **Conservation core energy efficiency** Our patchable C-Cores provide up to 16× improvement in energy efficiency compared to a general-purpose MIPS core for the portions of the programs that they implement. The gains are up to 2× larger for non-patchable C-Cores, but their lack of flexibility limits their useful lifetime. Each subgroup of bars represents a specific version of an application (see Table 3.1). Results are normalized to running completely in software on an in-order, power-efficient MIPS core ("SW"). "unpatchable" denotes a C-Core built for that version of the application but without patching support, while "patchable" includes patching facilities. Finally, "patched" bars represent alternate versions of an application running on a patched C-Core. For all six versions of bzip2 (A-F), the C-Cores performance is identical.

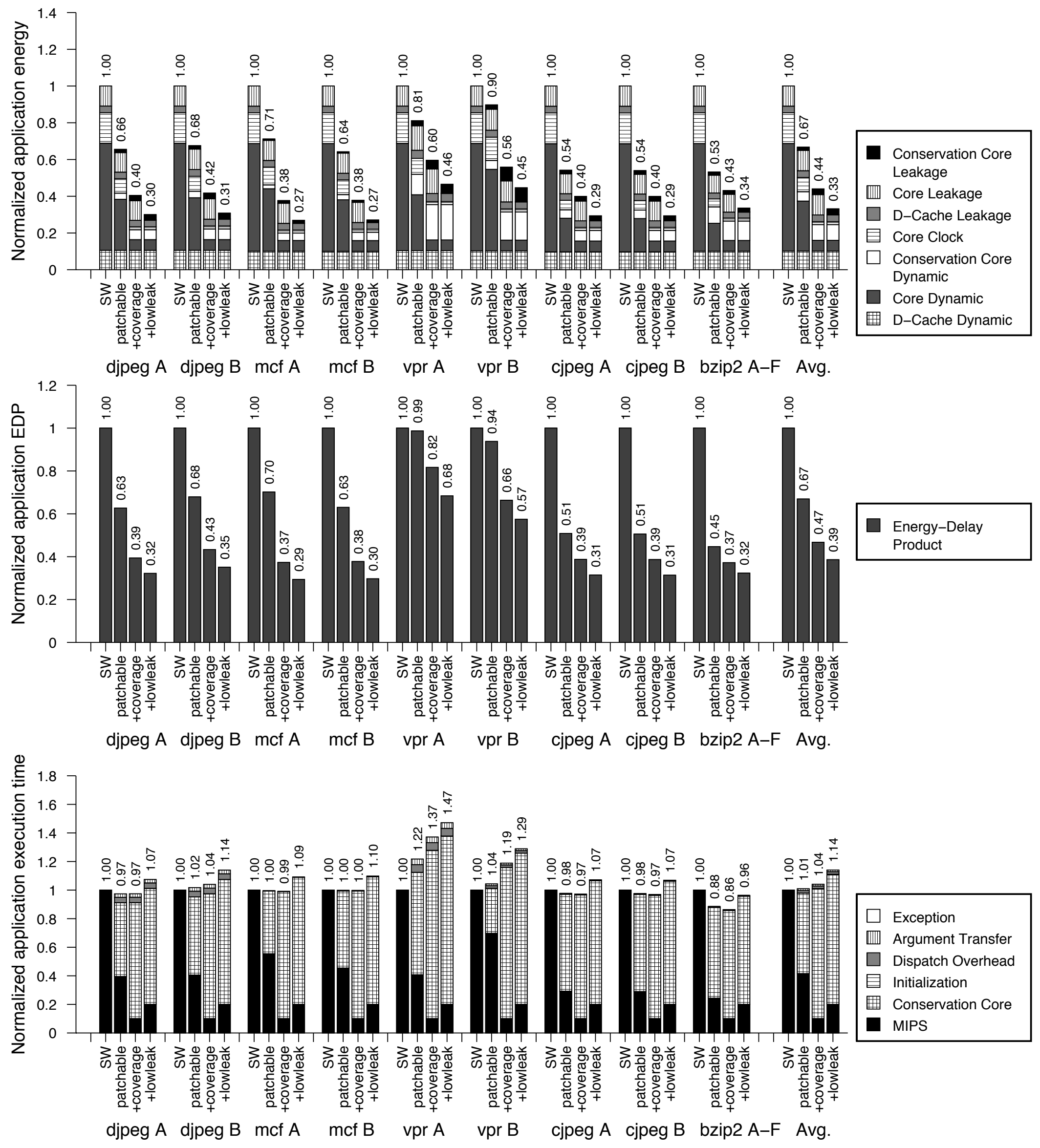


Figure 3.8: **Full application system energy, EDP, and execution time for C-Cores, and projections for potential improvements** These graphs show full application system energy, EDP, and execution time for C-Cores (lower is better). "SW" and "patchable" are as described in Figure 3.7. "+coverage" displays achievable improvements to energy reduction if 90% of the application can run in a C-Core. If a slower, lower-leakage process is used for the MIPS core in addition to improved coverage ("+lowleak"), even further improvements are possible. As in Figure 3.7, each subgroup of bars represents a specific version of an application, and for all six versions of bzip2 the C-Cores performance is identical.

3.7.3 System Efficiency

Figure 3.7 demonstrates that the C-Cores produced by our toolchain, even at a prototype stage of development, can provide large efficiency gains for individual functions. However, there are other components than C-Cores in the system, and portions of each application not executed by C-Cores. These other components have not been optimized by the introduction of C-Cores. As Amdahl's law indicates, collective improvements are limited by the degree to which an optimization is applicable.

In this section, we evaluate full-application, full-chip C-Core energy efficiency. Figure 3.8 shows the energy (top), energy-delay product (middle), and delay (bottom) for each of the applications. In each group, the first bar measures the performance of the program without using any C-Cores. The second measures performance with the patchable C-Cores described in the previous section. The data show that, at the application level, C-Cores save between 10% and 47% of energy compared to running on a MIPS core, and they reduce EDP by between 1% and 55%. Runtime varies, decreasing by up to 12% and increasing by up to 22%, but only increasing 1% on average.

The benefits for these C-Core-enabled systems are sizable. However, the efficiency improvements from the prototype C-Cores are moderated by two key factors. The first of these is coverage, the fraction of execution spent on C-Cores rather than on the processor. The second is the fact that the remaining parts of the system are largely untuned for use in this context. We examine both below.

Coverage Coverage in our prototype system is largely a function of toolchain maturity, and the results in subsequent chapters will see it improve with additional engineering effort. However, it also reflects certain more fundamental limitations: The amount of coverage that C-Cores can deliver is a function of the applications the system targets, the maturity of the C-Core toolchain in selecting and supporting hot code regions, and the overheads that C-Cores introduce. Absent overheads, Figure 3.2 from Chapter 2 indicated that it should be possible to achieve 90% coverage with a hardware budget covering less than 5000 static instructions per application for the applications in question. In practice, we have not yet achieved this level of coverage for all applications.

Several limitations of our prototype toolchain prevent us from achieving optimal coverage. One is the cost of transitioning between CPU and C-Core execution. A combined call and return overhead of 317 cycles, as mentioned in section 3.5, is sufficient for most important code regions, but does limit the utility of C-Cores for regions with frequently executed short functions or frequent exceptions. We continue to work on this through improvements to our automated code selection pass, and the associated inlining and outlining subpasses. Another limitation is our lack of support for floating point functions such as *exp* and *sqrt*. For instance, adding better floating point support would allow us to increase the coverage for vpr 4.30 from 27% to 54% improving energy savings.

Improving coverage is a continued effort, and a worthwhile goal. Projecting out to 90% coverage using the efficiencies for the currently covered regions for each benchmark, the "+coverage" bars in Figure 3.8 show our C-Core prototypes would provide energy savings ranging from 40% to 62%, and EDP savings from 18% to 63%. In Chapter 4, we will see improvements in C-Core selection and isolation increase average coverage from 59% to 75%. The same chapter also introduces benchmarks for which more than 90% coverage is achieved. Full application efficiency for these benchmarks is in line with projections.

Fixed costs As coverage improves, reducing the energy consumption of the other components in the system becomes more important. For instance, the contribution of the processor to leakage could be greatly reduced by switching to high-V_t transistors. With higher C-Core coverage, this would be quite sensible, as the performance penalties to the processor would only affect small portions of execution (i.e., 10%). This same approach can be applied to the instruction cache and other peripheral components. Shared components still heavily used by C-Cores, however, such as the data cache, could not be as aggressively modified without compromising performance.

The final bar in each group in Figure 3.8 shows projections for the impact of making these changes to the other system components. Rebalancing reduces the fixed cost overheads of instruction cache and processor leakage at the cost of approximately halving the performance of the software component of execution. Reducing these fixed costs would provide an additional 11% savings in energy and increases the total energy savings to 67% and total EDP savings to 61% on average. While the prerequisite coverage has not yet been achieved, even with the additional contributions described in subsequent chapters, these results indicate that a system rebalancing will eventually be fruitful.

3.8 Summary

As we run up against the utilization wall, reducing energy per operation becomes increasingly important. Conservation cores are a new class of circuits that aim to increase the energy efficiency of mature applications. C-Cores are automatically synthesized from C code and have built in support that allows them to evolve when new versions of the software appear. Data for 18 fully placed-and-routed C-Core prototypes show that they can reduce energy consumption by 10-47% and energy-delay by up to 55%.

In this chapter we have presented our vision for the C-Core life cycle and seen how conservation cores are selected, constructed, and evaluated by our automated toolchain. We have seen how this toolchain makes use of the patching mechanisms from [VSG$^+$10] to extend the lifetime of C-Cores. We have also seen how the decisions made in support of memory ordering and future-proofing affect the performance and energy efficiency of conservation cores. While our prototype C-Cores are significant energy savers, conservative design choices limit opportunities for performance optimization.

In the next chapter, we will look at extending and transforming our prototype C-Cores, which save energy, into C-Cores that both save energy and improve performance. In designing these new, accelerating C-Cores, we will retain all of the automation of the toolchain described in this chapter. We will, however, revisit those areas that presented performance and efficiency stumbling blocks for our prototype C-Cores. In particular, we will focus on how to preserve memory ordering in a datagraph execution engine and how to select which datapath elements are worth future-proofing.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under NSF CAREER Awards 06483880 and 0846152, and under NSF CCF Award 0811794.

This chapter contains material from "Conservation cores: reducing the energy of mature computations", by Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson and Michael Bedford Taylor, which appears in *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems.* The dissertation author was the secondary investigator and author of this paper.

Chapter 4

From Conservation to Acceleration

In Chapter 3 we introduced conservation cores and the toolchain that selects and produces them. Although our prototype C-Cores increased application energy-efficiency for arbitrary C programs by up to 2.1×, there were no substantial gains in performance. Many computing domains demand performance as well as energy efficiency. For instance, cell phones, e-book readers, media players, and other emerging computing platforms have driven demand for mobile application processors that deliver both energy efficiency and high performance for a core set of applications. These applications have increasingly larger sections of performance-critical, irregular code. While conventional accelerator techniques offer large returns on both power and performance, they only apply to computations with ample parallelism and predictable memory access patterns. C-Cores already improve energy efficiency for irregular code. Our goal, therefore, is to improve C-Core performance for irregular code regions without sacrificing energy efficiency.

This chapter describes a set of improvements to the C-Core architecture that improve both performance and energy efficiency. While the focus of this chapter is on performance, the goal of conservation cores remains reduction of energy and energy-delay product. Thus, we focus only on performance improving techniques that are neutral or beneficial to EDP. For clarity of exposition, we will refer to these improved C-Cores as *Improved Conservation Cores*, or ICCs, to differentiate between our prototype and optimized C-Cores. ICCs provide the same system interface as our prototype C-Cores, but differ in internal pipeline organization, memory ordering implementation, and reconfigurability mechanisms.

These new C-Cores improve significantly on prototype C-Core energy efficiency and performance by leveraging two architectural techniques. The first is a new pipelining technique called *pipeline splitting*, or *pipesplitting*, that allows the memory system clock to run much faster than the datapath clock. This split saves power in the datapath and improves memory performance. We also reduce the power and area costs, relative to our prototype C-Cores, of the reconfigurability that allows C-Cores to adapt to changes in the software they target. Below, we describe each of these features and highlight the unique challenges of accelerating irregular code.

The remainder of this chapter is organized as follows: Section 4.1 develops and explains the pipesplitting technique at the heart of ICC performance and efficiency. Section 4.2 presents our evaluation of pipesplitting. Finally, Section 4.3 summarizes the benefits of our C-Core redesign over both a baseline MIPS processor and our C-Core prototypes.

The remainder of this dissertation builds on the new C-Core design presented in this chapter. Further improvements to C-Cores, in the form of customized, distributed L0 caches embedded in the datapath called *cachelets* are explored in Chapter 5. Chapter 6 examines the challenges of operation scheduling in the face of pipesplitting and cachelets.

4.1 Pipeline splitting

Our redesigned C-Cores bridge the gap between the disparate requirements of memory and datapath using a novel pipelining scheme called pipeline splitting, or pipesplitting. Pipesplitting allows memory to run at a much higher clock frequency than the datapath. The fast clock effectively replicates the memory interface *in time* by exploiting pipeline parallelism. Meanwhile, the datapath runs at a slower clock rate, saving power and leveraging ILP by replicating computation resources *in space*. With pipesplitting, we execute faster and consume less energy than a general-purpose processor or our prototype C-Cores. Pipesplitting improves performance by enabling memory pipelining and exploiting ILP in the datapath. Pipesplitting reduces static and dynamic power because smaller, slower gates can be used in the datapath, and fewer pipeline registers are required. Whenever a C-Core is stalled waiting for memory, the datapath values will settle, reducing transistor switching energy.

Under pipesplitting, datapath operators are organized according to the basic blocks in a program's CFG, and one basic block executes for each pulse of the slow clock. The execution of a basic block begins with a slow clock pulse from the control unit. The pulse latches live-out data values from the previous basic block and applies them as live-ins to the current block. The next pulse of the slow clock, which will trigger the execution of the next basic block, will not occur until the entire basic block is complete. Different basic blocks operate at different slow clocks.

For each basic block, there is one control state, and each state contains multiple substates called *fast states*. A fast state corresponds to one cycle of the faster memory/system clock. The number of fast states in a given control state is based on the number of memory operations in the block and an estimate of the latency of the datapath operators. During the execution of the basic block, the control unit passes through fast states in order. Some fast states are associated with memory operations. These occur in pairs, one fast state for the request and another associated with the response. For the former, the ICC sends out a load or store request to the memory hierarchy. The ICC also includes a register to receive the result of loads. Unlike the registers between basic blocks, these registers latch values on fast clock edges. These are the only registers within a basic block. The ICC remains in a fast state associated with a memory response until the corresponding memory operation completes.

Figure 4.1 illustrates pipesplitting over one basic block. C source code for the original program is shown at top, and underneath that a timing diagram, control flow graph (CFG) and datapath for the implementation of that code. The

C code:
```
for (i=0; i<N; i++) {
  x = A[i];
  y = B[i];
  C[x] = D[y] + x + y + (x-1)*(y-1);
}
```

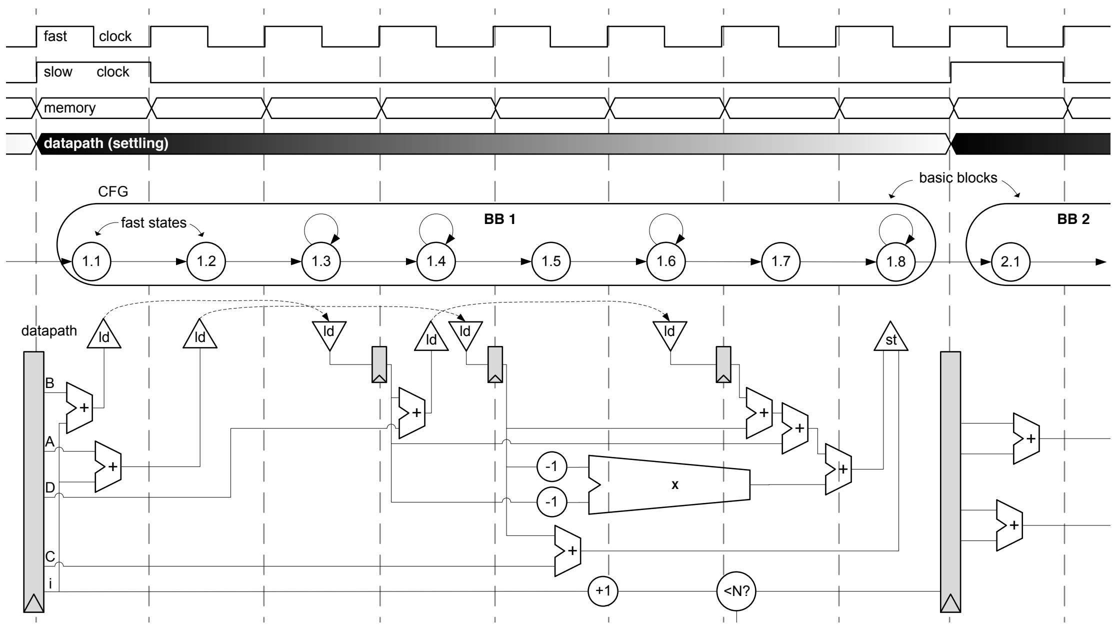


Figure 4.1: **Example datapath and timing diagram demonstrating pipesplitting** We show C source above a timing diagram, control flow graph, and datapath implementing that code. Under pipesplitting, non-memory datapath operators chain freely within a basic block, while memory operators and associated receive registers align to fast clock boundaries. The datapath contains arithmetic operators and load/store units for individual operations from the original program. The timing diagram shows how the datapath logic can take multiple fast cycles to settle while an ICC makes multiple memory requests.

datapath contains arithmetic operators and load/store units for individual operations from the original program. The timing diagram shows how the datapath logic can take multiple fast cycles to settle while an ICC makes multiple memory requests. The figure demonstrates how pipesplitting saves energy and improves performance. In a traditional pipeline, the registers at fast clock boundaries would latch all the live values in the basic block. Pipesplitting eliminates most of those registers, saving area and reducing clock energy. Eliminating registers also allows for more flexible scheduling of operations and removes the set-up, hold-time, and propagation delays that registers introduce. For the applications we examined, datapath register count was reduced, on average, by 20% relative to the prototype C-Cores.

4.1.1 Fast clock aligned operations

While most operations are only constrained by the slow clock, memory accesses and other long-latency operations must be scheduled with respect to the fast clock.

Pipelined memory operations ICCs enforce the memory ordering semantics that imperative programming languages require. ICCs require in-order completion of memory requests to reduce complexity and save power, but they also pipeline the interface to support memory parallelism and improve performance.

Every load and store occurs in two steps: request and response. A request consists of an address and, for stores, the value to be stored. When the datapath generates a new request, the ICC sends it to the memory hierarchy and continues performing other operations in parallel. In the response step, the control unit will wait if necessary for the load value or store confirmation. Load values are saved in fast-clock registers for use by the dependent operators in the datapath. By splitting memory accesses into two phases, an ICC can initiate up to one memory request (load or store) and receive up to one memory response (load value or store confirmation) on every cycle. Memory operations complete in order, but multiple outstanding requests can be in flight at any time.

Splitting memory operations and pipelining them gives ICCs advantages over our prototype C-Cores. Our prototype C-Cores had no smaller control subdivision than a basic-block control state. This precluded having multiple potentially blocking operations within the same basic block. As a result, our prototype C-Cores completely serialized memory accesses, whereas ICCs can issue memory operations in back to back cycles. This reduces the fragility of ICC performance with respect to changes in memory latency. Similarly, by splitting memory operations into two phases, ICCs move the stall points further back in the datagraph. This allows more independent operations to be scheduled alongside a memory access.

Long-latency operations Some non-memory operations, such as integer division and floating point operations, also have a long and/or variable latency. ICCs handle these long-latency operations just like memory requests: They wait in a specific fast state for a valid signal from the corresponding functional unit.

4.1.2 Pipesplitting implementation

Pipesplitting relies on a fast clock for memory and a different slow clock for the datapath of each basic block. The fast clock operates at the system frequency of 1.5 GHz. Due to variable-latency operations, such as cache accesses, the slow clocks are aperiodic. The slow clock edges correspond to progress through particular sets of fast states rather than fixed durations in time. The slow clock signals come from the ICC's control unit, which tracks the flow of execution through the ICC.

To determine how many fast states a control state will contain, an operation scheduler calculates a minimum execution time for the block, in number of fast clock cycles. This number is the maximum of the number of memory operations in the block and the critical path through the block divided by the fast clock period. For this calculation, the tool chain assumes that all memory operations will hit in the L1 cache.

Many signals in the basic block can safely take the entire minimum execution time to propagate through the block. However, the inputs to memory operations need to propagate more quickly because they must be ready on fast clock boundaries. For instance, in Figure 4.1, the path from input `i` through the increment and compare can take up to eight fast clock cycles. Meanwhile, the path from `B` to the first load must complete in a single cycle. Similarly, the result of the third load has just 2 cycles to propagate to the store in the last fast state. Our toolchain enumerates all of these multi-cycle constraints and passes them to the synthesis toolchain which enforces them.

As the benefits of pipesplitting are sensitive to the accuracy of these multi-cycle constraints, proper operation scheduling is important. To achieve maximum performance, the ICC scheduler must accurately estimate the number of fast states required for the critical path through each basic block. For simplicity, we schedule assuming cache hits. The scheduler must also assign memory operations, especially loads, to the earliest fast states in which their inputs will be ready. If the scheduler is too conservative, the ICC will waste time in unnecessary fast cycles, resulting in slower performance. On the other hand, if the scheduler is too aggressive, the back-end CAD tools will not be able to meet timing requirements. This will cause the ICC to run at a slower clock frequency than the system clock. Chapter 6 discusses ICC scheduling in more detail.

4.1.3 Reducing patching overheads

In order to remain useful across software versions, application-specific hardware must be able to adapt to changes in the code it supports. ICCs adapt to software changes by using improved versions of the patching mechanisms introduced in [VSG$^+$10]. Analysis of the programs in Table 4.1 showed opportunities to reduce patching overheads, especially in terms of the size of registers used to hold constant values: In our workloads, 87% of all compile-time constants can be represented by 8 or fewer bits. Thus, we can use smaller configurable registers to represent most constants with little risk of reducing generality. Bitwidth analysis of constants and improved heuristics for when to deploy configurable ALUs in lieu of fixed-function logic allow us to reduce patching area and energy overheads in ICCs by 43% and 70%, respectively, without significantly impacting the ability of ICCs to adapt to software changes. Size reducuctions for configurable constant registers account for 77% of the savings.

4.2 Evaluating pipesplitting

In this section we describe the workloads to which we apply ICCs and evaluate the impact of pipesplitting on ICC efficiency, performance, and energy-delay product. We show how redesigning C-Cores around pipesplitting gives ICCs performance and efficiency advantages over our initial C-Core design.

4.2.1 Methodology

We constructed and evaluated ICCs by extending the toolchain described in section 3.6. The basic toolchain is augmented with multi-cycle constraints and a pipesplitting-aware scheduler. We also further refine our model for interconnect delay to and from the shared L1 cache. These refinements lead us to model a three-cycle load-use latency for MIPS, our prototype C-Cores, and ICCs in this and following chapters.

We derive processor and clock power values for other system components from specifications for a MIPS 24KE processor in a TSMC 45 nm process [MIP10] and component ratios for Raw reported in [KTMW03]. We assume a MIPS core frequency of 1.5 GHz with 0.10 mW/MHz for CPU operation. We use CACTI 5.3 [TMAJ08] for I- and D-cache power.

Table 4.1 describes the 9 applications for which we have created ICCs. The applications are a superset of those examined in Chapter 3. However, our C-Core selection toolchain has matured and we examine only one version of each application. Thus, the number and selection of code regions transformed into C-Cores is not identical to that in Chapter 3. For comparison, we produce both prototype C-Cores and improved C-Cores for each selected code region.

Table 4.1: **ICC Workloads** We built 19 ICCs running at 1.5GHz for 9 irregular applications, covering the majority of execution. Patching optimizations significantly reduce area.

Workload	Description
bzip2 [SPE00]	Data compression algorithm
cjpeg [Gro]	JPEG image compression
djpeg [Gro]	JPEG image decompression
mcf [SPE00]	Single-depot vehicle scheduling
radix [WOT+95]	Sorting algorithm
sat solver [TH04]	Stochastic local search SAT solver
twolf [SPE00]	Placement & connection of transistors
viterbi [Emb]	Convolutional code decoder
vpr [SPE00]	Place and route algorithm

Workload	ICC count	Coverage %	Average Slow-clock MHz	Area (ICC) mm^2	Area (+Patch Opt.) mm^2
bzip2 [SPE00]	1	76	366.74	0.27	0.18
cjpeg [Gro]	3	75	116.73	0.31	0.18
djpeg [Gro]	3	77	85.32	0.33	0.21
mcf [SPE00]	3	82	302.41	0.28	0.17
radix [WOT+95]	1	94	120.38	0.17	0.10
sat solver [TH04]	2	66	215.20	0.30	0.20
twolf [SPE00]	4	49	252.20	0.20	0.13
viterbi [Emb]	1	98	259.07	0.22	0.12
vpr [SPE00]	1	61	684.93	0.37	0.23

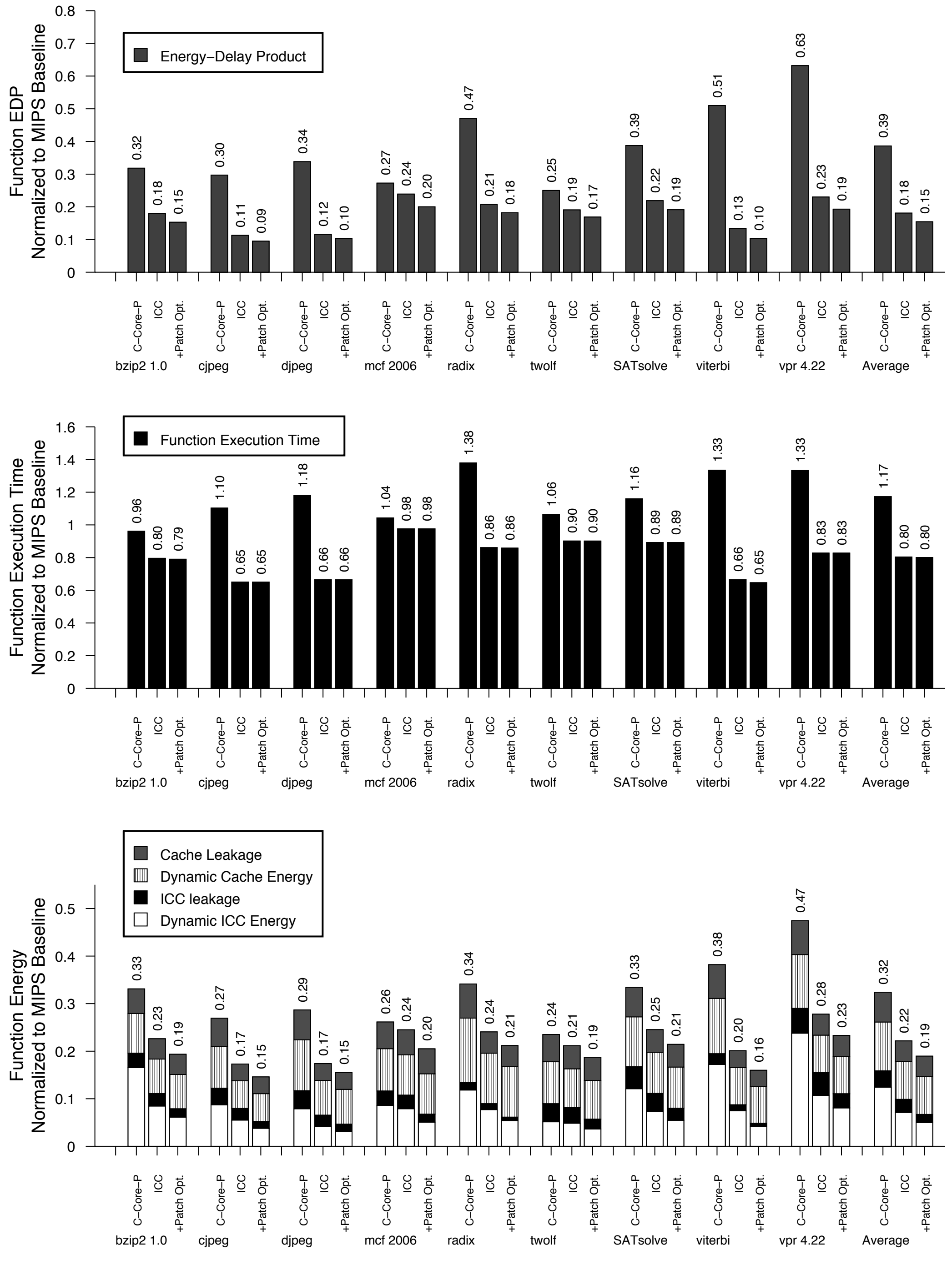


Figure 4.2: **ICC performance and efficiency** Pipesplitting allows the baseline ICC design to achieve lower latency (middle), and energy usage (bottom) compared to prototype C-Cores, resulting in significantly better energy-delay (top). Optimizing patching support decreases energy while maintaining performance and longevity.

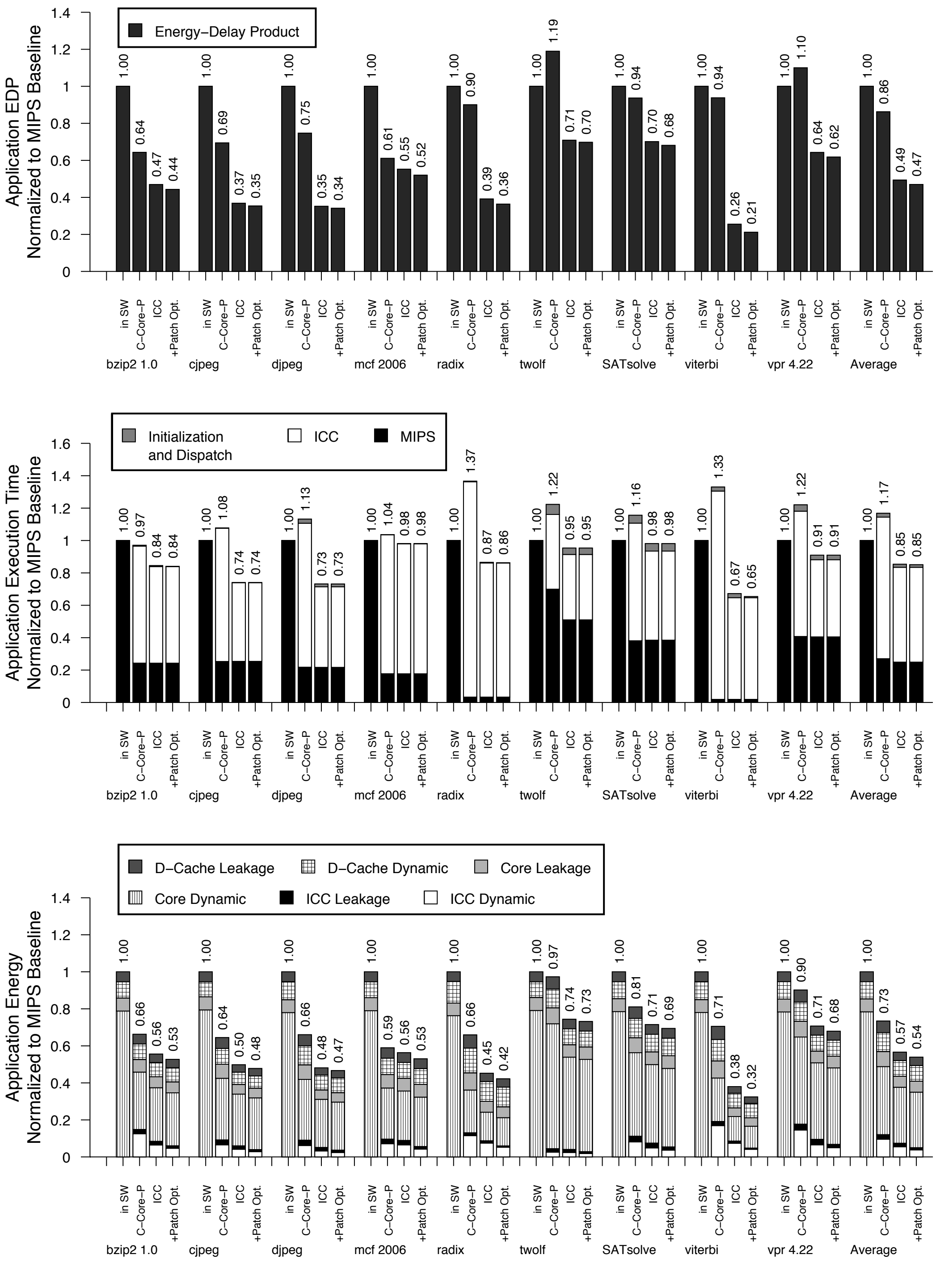


Figure 4.3: **Application performance and efficiency with ICCs** Application latency (middle), energy (bottom), and energy-delay (top) improvements from using ICCs can be large, with latency reductions of up to 35% and average EDP reductions of nearly 2x. The benefits of ICCs increase with application coverage.

4.2.2 ICC performance and EDP

We first evaluate pipesplitting and its associated scheduling and logic optimizations. Figure 4.2 shows energy-delay product (EDP), and its two components (execution time and energy), for the portions of the applications executed on ICCs. Results are presented normalized to the baseline single-issue low-power MIPS processor executing the same function. In addition to the baseline ICC design, we also present numbers for our prototype C-Cores and an ICC with reduced patchability overheads(*+Patch Opt.*).

While the C-Core prototypes have substantially lower EDP than the MIPS core, this remains a consequence of energy reduction, not performance. On average, the C-Core prototypes are actually slower than the MIPS core. The performance results differ from those in the previous chapter due to refinements in modeling interconnect delay to and from the shared L1 and changes to our C-Core selection stage.

The ICCs not only outperform both the MIPS baseline and prototype C-Cores, but they are substantially more energy-efficient than prototype C-Cores. On average, the ICC baseline has a speedup of 1.27 relative to MIPS and 1.47 relative to prototype C-Cores. The baseline ICCs reduce energy for covered execution by 78% over MIPS and by 33% over prototype C-Cores. ICCs with improved patching reduce energy by an additional 3%. We will use *+Patch Opt.* as the baseline ICC design for examining cachelets and scheduling in Chapters 5 and 6. The benefits of ICCs for non-memory energy are even larger: Total energy expended in the memory system is very similar among MIPS, prototype C-Cores and ICCs. Energy expended in the memory system accounts for more than half of total ICC energy.

Figure 4.3 shows how ICC performance and efficiency gains are translated to the application level, where ICCs offer an average EDP improvement of 51%. ICC calling overhead is shown to be a small fraction of total execution time, except in twolf, SATsolve, and vpr. Coverage is incomplete for several benchmarks, and thus application level speedups are notably lower than isolated speedups. For the two kernels, radix and viterbi, coverage is nearly complete, and energy savings are substantial. Even for benchmarks with more than 90% coverage, energy spent running on the CPU is greater than that spent on running on the C-Core. While coverage is similar for both viterbi and radix, EDP savings are much greater for viterbi than for radix because of lower speedup. Further exploration of the ICC memory system in Chapter 5 will show how this, and other differences in speedup, are heavily influenced by application memory characteristics.

4.3 Summary

In redesigning the C-Core pipeline around the disparate needs of the memory interface and datapath logic, we have moved from just reducing energy to reducing execution time as well. Through pipeline splitting, ICCs improve on prototype C-Cores in both performance and efficiency. Likewise, more targeted flexibility in patching reduces overheads for little loss in utility. With these combined facilities, ICCs reduce average EDP by 6.6× for covered code, and by 54% at the application level. ICCs, unlike prototype C-Cores, provide an average speedup for covered code of 1.27× over a baseline MIPS processor.

However, for C-Cores to achieve their full potential, we must pay special attention to further optimizing memory operations, which are common, high-latency operations and frequently on the critical path. While the pipelined nature of the ICC memory interface improves throughput over that of prototype C-Cores, load-use latency remains the same. In the next chapter, we explore augmenting ICCs with cachelets, a mechanism for reducing common case memory access time that can improve performance even further.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under NSF CAREER Awards 06483880 and 0846152, and under NSF CCF Award 0811794.

This chapter contains material from “Energy-Delay Optimized Accelerators for Irregular Code”, by Jack Sampson, Ganesh Venkatesh, Nathan Goulding, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor, which has been submitted for possible publication by IEEE in *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Cachelets

Pipesplitting improves efficiency for non-memory operations and increases memory throughput via pipelined memory access. However, pipesplitting alone does not reduce load-use latency. Excluding L1 misses, load-use latency is the largest single component of the critical path for a baseline ICC. We will show how ICCs can reduce this cost by providing select groups of communicating static memory operations very fast (sub-cycle load-use), tiny, specialized caches called *cachelets*. Cachelets reduce average memory delay by eliding accesses to the L1. They are especially useful for increasing the performance of dependent sequences of memory operations.

Memory latency is a critical component of execution time for irregular codes. The data for the baseline ICCs show that load-use latency, and equivalently, L1 hit time, in an ICC is a limiting factor for its performance. On average, L1 cache hits account for 30.8% of total time on the dynamic critical execution path for a baseline ICC.

In conventional processors, all loads and stores go to a single cache since as all load and store instructions execute on a small set of load/store functional units. In contrast, ICCs can optimize individual load and store operations independently, as each static memory operation has its own associated hardware. A cachelet serves a fixed subset of these operations, all of whose accesses go through the cachelet, and an ICC may have several cachelets.

In this chapter, we explore augmenting ICCs with cachelets to reduce memory latency. Cachelets contain one to four cache lines and are tightly integrated into the ICC data path. Lines in a cachelet are backed by an inclusive L1 and are fully coherent. Operations that have not been statically mapped to a cachelet communicate directly with the L1.

In the remainder of this chapter, we describe the cachelet architecture and evaluate ICCs augmented with cachelets. We will show that adding cachelets to ICCs significantly improves performance and also improves EDP. Section 5.1 provides an overview of the cachelet architecture and differentiates cachelets from an L0. Then, sections 5.2 and 5.3 present a simple solution for cachelet coherence and discuss mechanisms for cachelet selection. In section 5.4, we evaluate two cachelet selection strategies relative to unaugmented ICCs. We also compare against the limit case of an ICC whose L1 has the same access latency as a cachelet. Finally, section 5.5 summarizes the benefits of adding cachelets to ICCs.

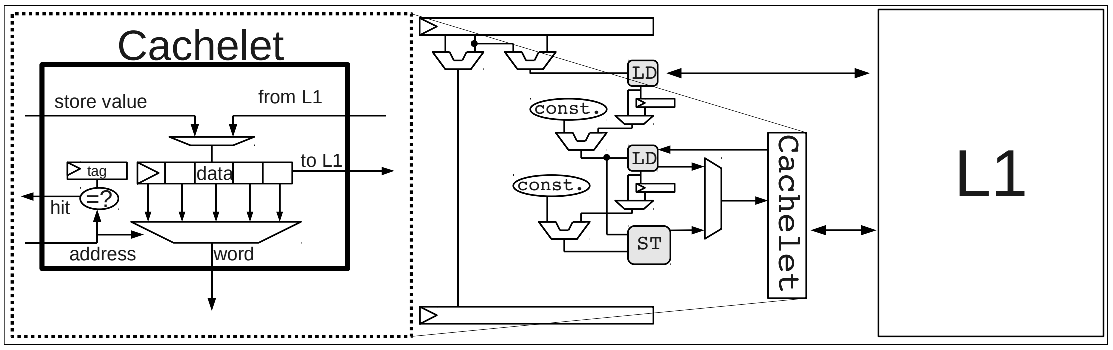


Figure 5.1: **Cachelet architecture** Cachelets have similar fundamental components to larger caches, but we implement them with latches rather than SRAM due to their small size and datapath integration. With cachelets, memory operations with good locality will be mapped to local, low-latency memories while other operations continue to interface directly to the L1. In this example, a one line cachelet serves one load and one store in a basic block, while the L1 services the third memory operation.

5.1 Cachelet architecture overview

The ICCs presented in Chapter 4 have a 3-cycle load-use latency to the L1, as does the MIPS processor used as a baseline throughout this dissertation. Our synthesis results show cachlet hit time to be half a memory clock cycle, reducing common case memory latency by 83%. This low latency is due to datapath integration and the small size of cachelets.

Cachelts offer physical locality, as they are closer to the datapath and separated by fewer layers of logic than the shared L1 interface. Cachelets also provide specialization for access streams. Each cachelet is associated with a particular disjoint subset of the static memory operators, allowing indpendent optimization of each operator or group of operators. The cachelet approach also allows memory instructions that do not benefit from cachelets to communicate directly with the L1. Operators whose access streams have poor spatial and temporal locality would quickly pollute a 1-4 line cachelet. The partitioning of cachelets by operator helps protect more cache-friendly regions of code from poor performers.

Figure 5.1 shows how an ICC with cachelets communicates with the L1 cache and shows the internal structure of a cachelet. In the figure, two communicating memory operations share a single, one-line cachelet, while a third accesses the L1. Internally, cachelets share many similarities with small full-scale caches, such as tags, comparators, and word select muxes, but they use latches rather than SRAMs to store data.

5.1.1 Cachelets vs. L0

In an ICC-enabled system, the L1 is shared, and the addition of an additional private level to the cache hierarchy may not seem immediately novel. However, the cachelet approach is quite different from simply adding a private L0 to each ICC. The two key differences are the tight integration of cachelets into the datapath and the use of multiple cachelets for each ICC.

In the C-Core architecture there is a path from every memory operation to a single external memory interface, which is time multiplexed. These paths to memory often lie on the critical timing paths in an ICC circuit. Accessing an L0 cache placed after the multiplexing of paths to memory is negligibly closer than accessing an L0 cache external to the ICC. Thus, in an ICC with dozens or even hundreds of memory operations, the L0 is still somewhat distant from source and dependent datapath operators. For small L0s, this means that the latency in getting to and from the L0 will be on par with accessing the L0 itself. As the primary motivation for adding this additional level of the memory hierarchy is reducing latency, this is not an optimal solution.

Cachelets, on the other hand, are associated with small, disjoint subsets of the static memory operators. They are built as part of the datapath, rather than as external entities. This allows ICCs to exploit the low access time cachelets provide without having to pay the transit costs of generalized access. Just as the operators in the datapath provide specialization for computation, cachelets provide specialization for locality of access not provided by a single L0.

Mapping disjoint sets of operators to different cachelets comes with its own challenges. Traditional coherence strategies presume a single cache at the closest level of the memory hierarchy, and are thus not directly applicable. Moreover, the choice of which operators map to a cachelet can have profound effects on its utility.

In the following sections, we present a simple coherence protocol for cachelets and explore alternatives for deciding what types of cachelets to instantiate.

5.2 Coherence

The ICC execution model requires a coherent memory system, so the coherence protocol must extend to cachelets. Making each cachelet a full-fledged cache from the protocol's perspective is not practical: The coherence controller and state machines for the cachelet would be much larger than the cachelet itself.

To provide cachelet coherence at minimal cost, we allow cachelets to "check out" cache lines from the shared L1 cache. To check out a cache line, the cachelet issues a fill request to the L1 cache. The L1 acquires exclusive access to the line and returns its contents to the cachelet. The cachelet now has exclusive access to the line. If another cachelet, the general-purpose core, or another processor in the system attempts to access that line, the L1 detects this and forcibly reclaims the line from the cachelet.

To perform a reclamation, the L1 freezes the ICC to prevent concurrent updates to the cachelet. It then copies the cachelet's contents back into the L1, invalidates the line in the cachelet, and completes the coherence request. The ICC can then continue execution, potentially re-acquiring the line if it needs it again.

The eviction process is a heavy-weight operation requiring halting ICC execution. We minimize this cost through profiling and careful assignment of cachelets to memory operations. Additionally, when an ICC finishes executing, the ICC implements a cachelet flush mechanism that writes back the contents of all dirty cachelets in the ICC and invalidates all lines in cachelets.

5.3 Cachelet selection

Judicious selection of which static memory operations to provide cachelets for is essential for good performance. Including too many cachelets negatively impacts ICC area requirements without significantly improving performance, whereas including too few limits performance gains. Likewise, we must avoid operation-to-cachelet mappings that would result in poor hit rates or frequent coherence traffic.

We have developed two strategies for selecting which cachelets to instantiate. The first strategy, called "private," performs an LRU-stack-based [BK75] cache simulation in which every memory operation has a dedicated cache. The simulation reveals how many lines the cachelet needs in order to significantly reduce the miss rate for that operation. The simulation includes coherence misses, so operations that share data with other memory operations are unlikely to receive a cachelet. The "private" strategy includes a cachelet if it would require fewer than 4 lines, and would have a hit rate of at least 66%.

The second strategy, called "shared," analyzes the communication patterns and assigns a shared cachelet to communicating sets of memory operations. It forms transitive closures of communication operations within an ICC: It partitions operations into sets such that, during an invocation of an ICC, no operation in one set accesses any line of memory accessed by any operation in another set. It uses the same LRU-stack analysis as in "private" to determine whether to include a cachelet and how big it should be.

5.4 Results

In this section, we discuss our infrastructure for selecting and modeling cachelets and evaluate the impact of cachelets on performance and EDP.

5.4.1 Methodology

Our evaluation of cachelets builds on the infrastructure used to produce the ICCs presented in Chapter 4. Several modifications are made to the C-to-Verilog stage of the toolchain which contains the operation scheduler. The pipesplitting-aware scheduler is extended to also be a cachlet-aware scheduler. We introduce new post-simulation profiling tools to perform cachelet selection and a feedback loop from profiling to rescheduling. We also employ a trace driven performance estimator for cachelets.

We derive processor and clock power values for other system components from specifications for a MIPS 24KE processor in a TSMC 45 nm process [MIP10] and component ratios for Raw reported in [KTMW03]. We assume a MIPS core frequency of 1.5 GHz with 0.10 mW/MHz for CPU operation. We use CACTI 5.3 [TMAJ08] for I- and D-cache power. We use the "+Patch Opt." ICC variant with optimized constant widths and degeneralized datapath operators as our baseline ICC. We use the same workloads examined in Chapter 4.

We model cachelets as arrays of latches and use values from measurements of arrays synthesized in our ASIC tool flow for cachelet area and energy. We extend existing ICC generation tools to output fine-grained scheduling information and address tracing to feed to our cachelet selection and evaluation tools.

Our tools perform cachelet selection based on annotated address traces from ICC simulation. The traces carry annotations indicating source operations and intra-block operation ordering. Cachelet selection information is then sent back to earlier stages of the ICC toolchain to generate new, cachelet-aware schedules. Our cachelet performance estimator reorders and then replays traces according to these new schedules to model cachelet-augmented ICC performance and energy.

5.4.2 Evaluating Cachelets

Using the two strategies described in section 5.3, we modeled the effects of adding cachelets to each of the ICCs. On average, each ICC included 8.4 cachelets for the "private" selection strategy, and 6.2 for "shared". In the "shared" case, each cachelet served an average of 10.3 memory operations. No single ICC utilized more than 28 total lines of cache across its cachelets, and on average used fewer than 16 total lines. Area overheads over the baseline ICC for the "private" and "shared" selection strategies are 13.4% and 16.8%, respectively.

Figure 5.2 shows the impact of cachelets on ICC performance (top), application performance (middle), and application EDP (bottom). The first bar in each series depicts a baseline ICC without cachelets (the "+Patch Opt." bar from Figures 4.2 and 4.3), and the second and third bars present the "private" and

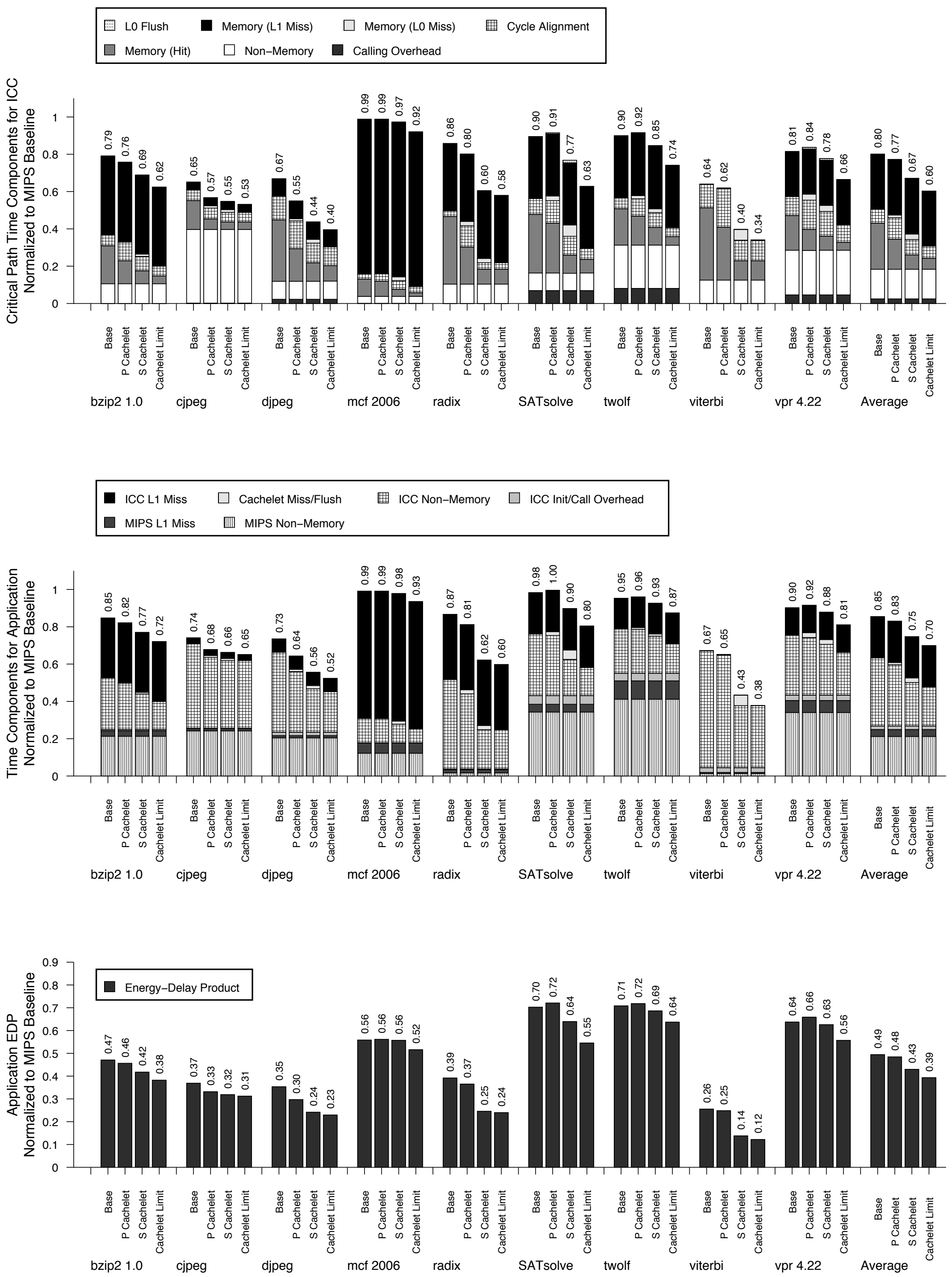


Figure 5.2: **Cachelet performance and efficiency.** The addition of cachelets greatly reduces latency and further improves EDP.

"shared" strategies, respectively. The fourth bar shows results for a limit study for cachelet benefits assuming a 0.5-cycle, 32-KB L1. This models the potential of having our entire L1 cache accessible with cachelet latency. This is unattainable, as access time for a cache of this size is larger than 0.5 cycles even in the absence of interconnect delay. Both the "private" and "shared" cachelet approaches offer performance benefits, but the "private" strategy covers fewer critical memory operations, due to frequent communication between memory operations. The "shared" strategy realizes much more of the potential that the limit study demonstrates.

Adding cachelets to the baseline ICC design reduces latency by 13%, application latency by 10%, and application EDP by 6% in addition to the gains from the baseline ICC over the MIPS core.

5.5 Summary

Cachelets provide significant latency reductions for several benchmarks. While they do not, in general, provide sufficient temporal re-use that they improve per-access energy, they still improve EDP. The ICCs in Chapter 4 provided average speedups of 1.27× and 1.17× over our baseline MIPS processor for covered code and applications, respectively. ICCs augmented with cachelets provide an average speedup of 1.5× for covered code and 1.33× for applications. They also provide improvements for covered code of 6.9× in EDP. At the application level, this translates to an average application EDP reduction of 57%.

As we will see in the next chapter, the introduction of cachelets changes scheduling as well as performance. This, and other scheduling artifacts, approaches and decisions are much more important for ICCs than for C-Core prototypes due simply to the greater size of scheduling windows provided through pipesplitting. In the following chapter, we examine the properties of automatically generated hardware and explore scheduling in detail.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under NSF CAREER Awards 06483880 and 0846152, and under NSF CCF Award 0811794.

This chapter contains material from “Energy-Delay Optimized Accelerators for Irregular Code”, by Jack Sampson, Ganesh Venkatesh, Nathan Goulding, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor, which has been submitted for possible publication by IEEE in *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Conservation Core Structure and Scheduling

In the previous chapter, we added cachelets, an entirely new hardware element, to our C-Cores. Cachelets and our modifications to patching support directly altered the hardware we use to build C-Cores. However, the introduction of new mechanisms is not the only source of hardware changes in C-Cores. Both pipesplitting and cachelets also produced more subtle effects, changing the amount of parallelism exposed to the scheduler. In this chapter, we discuss operation scheduling for conservation cores and analyze the hardware our automated synthesis toolchain produces.

Our toolchain produces hardware that varies in makeup both within and between applications. However, our use of basic blocks as the fundamental scheduling unit remains constant across all C-Cores. This choice, along with the stylized nature of automated synthesis, produces some common trends within these scheduling blocks. Understanding these trends and how they affect operation scheduling is important for understanding C-Core performance.

6.1 Scheduling for C-Cores

Our initial C-Core prototypes partitioned basic blocks to isolate memory operations. Therefore, there were few operations per state and the first generation of C-Cores were not particularly sensitive to scheduling decisions. For these initial C-Core prototypes, we used a single mean operation latency to schedule all low-latency operations. Operations were scheduled in the order they appeared in the intermediate SSA representation. For ICCs, we require a more considered approach to scheduling.

ICCs have larger datagraphs to schedule than our prototype C-Cores, and are also sensitive to the accuracy of latency estimations. This sensitivity stems from the implementation of the aperiodic slow clocks via multi-cycle constraints. The ICC scheduler must accurately estimate the number of fast states required for the critical path through each basic block and assign memory operations to fast states in which their inputs will be ready. Inaccurate latency estimations will not affect functional correctness, but can cause an ICC's fast clock paths to not meet timing. As we designed ICCs around the memory interface, an ICC that fails to meet timing requirements fails to uphold the interface contract.

Many approaches to generating specialized hardware, such as [CHM08, FKDM09, YGBT09] rely on modulo scheduling of loops. Many techniques for performing modulo scheduling exist([CLG02] surveys several of them), but they all share certain similarities. Modulo scheduling aims to produce a schedule such that when the operations in the schedule are repeated at fixed interval, there are no resource conflicts. Modulo scheduling is difficult to apply to code outside of loops, or to loops with control and data dependent resource usage. Modulo scheduling is not an ideal match for a cached memory interface and the irregular codes targeted by conservation cores. Such codes run counter to its preferences for regular control flow and memory timing. Similarly, pipesplitting does not allow concurrent executions of the same basic block. As a consequence of the utilization wall, the relative cost of area is lower, and hardware re-use is less important than the energy savings of hardware specialization. For conservation cores, we therefore focus on scheduling within the context of a single basic block.

Our approach to scheduling for ICCs takes into account both widely varying operator latencies and the impact of bit level parallelism. Individual operations can range in latency from 10 ps for a NAND to over 1000 ps for a multiply. Bit-level parallelism in back-to-back operations can result in a shorter latency than the sum of the individual latencies.

As an example of this, consider a multiply followed by an add. This is a common composite operation, known as multiply-accumulate, or MAC. At 45 nm, a single 32-bit add takes approximately 0.31 ns, and a single 32-bit multiply takes 1.12 ns, resulting in a naïve estimate of 1.43 ns for the combined operation. However, after CAD tool optimizations the chained multiply-plus-add operation takes only 1.14 ns, a savings of 20%. We approximate the effects of bit-level parallelism for scheduling via a lookup table of all sequences of two chained operators.

ICC scheduling operates on one basic block at a time, and performs two scheduling passes. The first pass schedules operations on an idealized hardware target with an arbitrarily wide memory interface and an unbounded number of functional units. In both the first and second pass, the scheduler may reorder loads with respect to other loads, but does not speculatively move loads before stores. We use the first pass to assign priorities to operations for the next pass. The scheduler computes priorities based upon the estimated aggregate operator latency between an operation and its latest finishing descendant operation. The second pass schedules operations given C-Core resource constraints. The scheduler imposes a limit of at most one memory request or other fast-cycle aligned operation per fast-cycle, and enforces alignment restrictions on operator inputs and outputs. This pass uses a priority list scheduler. Among mutually schedulable operations, the priority value from the previous pass determines which operation is scheduled first.

Figure 6.1 depicts a representation of the schedule for a single basic block in bzip2. Markers along the left hand side indicate fast-state boundaries, and each operation is annotated with an estimated latency. We show a subset of definite and possible dependencies with arcs connecting the critical scheduling path for each operation. A color code denotes operations categories: Loads are blue, stores are green, control operations are orange, and all others are beige. Memory operations are split-phase, and the request and receive phases are shown as distinct operations. In the pre-cachelet schedule, a fast-cycle aligned “Wait

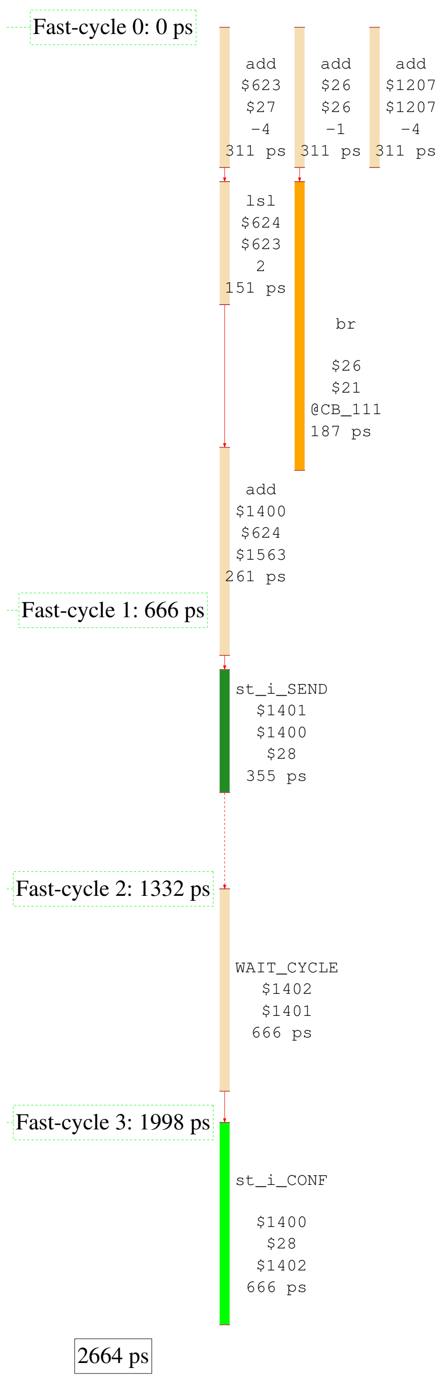


Figure 6.1: **Schedule for a basic block** Operations are shown annotated with estimated latency. Arcs are depicted showing the critical scheduling path for each operation.

State" dummy operation separates these operations to account for cache delay. We schedule address computations separately from the associated load or store, and thus we mark them as general operations, rather than memory operations.

Figure 6.1 is representative of many of the block schedules produced by our tool. Common trends include memory operations dominating the critical path through the basic block, operations spanning fast cycles, and operator chaining influencing estimated latency. As an example of the latter two, the scheduler estimates the *add* operation following the *lsl*(left-shift-logical) operation to have a lower marginal latency than similar operations dependent on values from previous blocks. This same operation also spans the first two fast-cycles. Another item of interest is that the control operation resolves more quickly than the critical path. While C-Cores cannot currently exploit inter-block parallelism, the presence of such cases indicates a potential for further optimizations.

6.1.1 Alignment

One source of additional overhead in C-Core schedules is fast-cycle alignment. Each slow-cycle must be an integer number of fast-cycles. ICCs therefore lose some potential performance due to the granularity of scheduling. A larger portion of alignment overhead is due to memory operations. Memory operations not mapped to cachelets have alignment restrictions on their address generation and, for stores, value stability. These signals must be ready a fixed time before the end of the cycle so that as to registers external to the C-Core can latch them. Failure to meet alignment restrictions delays a memory operation until the next fast-cycle in the schedule.

6.1.2 Rescheduling for cachelets

With the introduction of cachelets, scheduling for an ICC becomes an iterative process. Cachelet selection requires a profiling run, which in turn requires an initial schedule. Once the profiling run completes, the cachelet selection data feeds back into the scheduler. The scheduler assumes a cachelet hit time for those

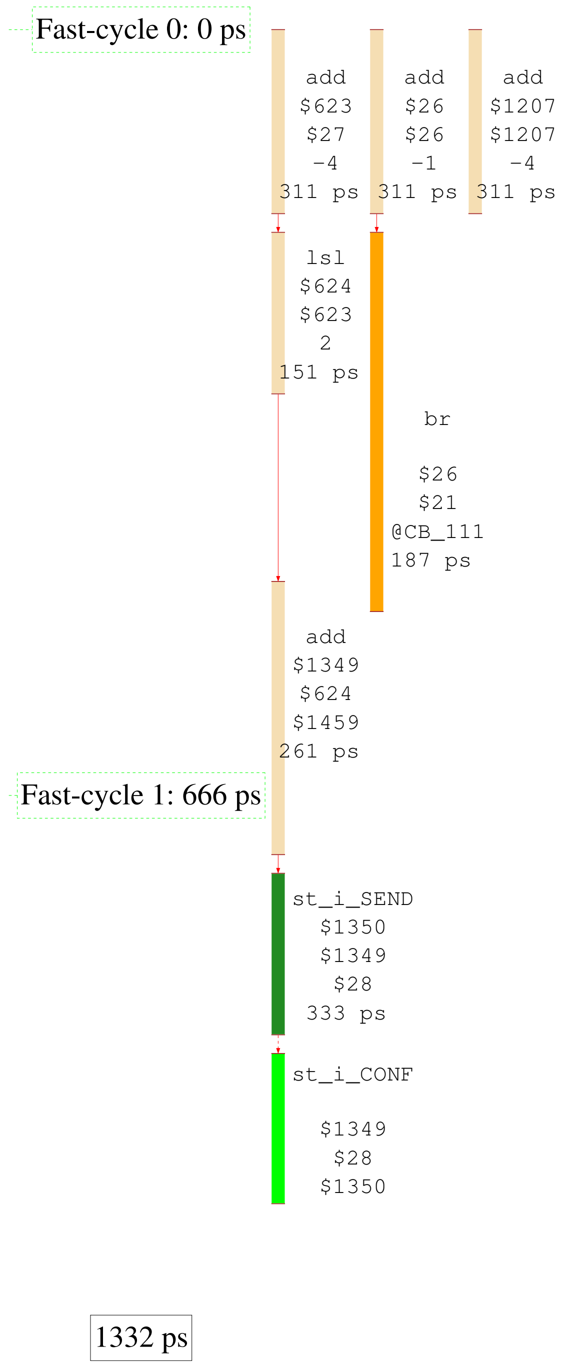


Figure 6.2: **Schedule for a basic block with cachelets** The addition of cachelets requires rescheduling under the assumption that cachelet accesses will be hits.

operations mapped to cachelets, and schedules unmapped operations assuming L1 hit times. This asymmetry between memory operation hit times can change which path through a basic block the scheduler estimates as the longest path in an idealized schedule. Thus, the introduction of cachelets can cause the reordering of independent loads relative to the pre-cachelet schedule. Figure 6.2 shows the schedule for the same basic block as Figure 6.1 after rescheduling for the addition of cachelets.

Intriguingly, the main benefit of cachelets, their low access latency, poses an interesting tension for the scheduler. Cachelet accesses, being sub-cycle, cannot overlap with other memory accesses in the default ICC architecture. Overall, cachelets benefit the schedule by reducing critical path latency. As seen in Chapter 5, this increases performance. However, the sub-cycle nature of cachelet access does not fit easily into the pipelined memory access scheme. The stall point of a cachelet receive is necessarily very close to the request point. In order to preserve the property of in-order memory completion, cachelet schedules cannot contain overlapping cachelet and direct L1 accesses. This limits the application of cachelets to operations with very high hit rates, as the scheduler can find very little work to overlap with cachelet access.

6.2 Analysis

In this section we examine the outcomes and limitations of current C-Core scheduling. The schedules for C-Cores reflect the irregular nature of their source code regions. To preserve hardware longevity, we also preserve software form. Irregular codes are characterized by frequent control decisions. These frequent control decisions produce a small average block size. The execution model for C-Cores is basic block oriented. Therefore, the average block size is a limitation on the ability of our scheduler, or any other intra-block scheduler, to expose parallelism.

Table 6.1 highlights key execution and scheduling characteristics of our conservation cores. It shows the number of states, and the dynamic averages of operation count, memory operations, and critical path length across scheduling

Table 6.1: **Properties of ICC schedules** For each application, we show the average properties of its C-Cores over execution. The "States" column shows the number of static scheduling states across all C-Cores for that application, and the next three columns show dynamic averages of memory operations per state, estimated critical path latency per state, and operations per state.

App	States	Dyn. Ops/ State	Dyn. Memory Ops/State	Dyn. Fast-cyles/ State (non-miss)	
				Base	Cachelet
bzip2	118	4.47	0.98	4.10	2.82
cjpeg	48	17.20	3.00	12.85	9.97
djpeg	36	24.29	7.69	17.58	9.20
mcf	95	3.71	1.76	4.96	3.81
radix	55	12.46	3.99	12.46	5.48
SATsolver	85	5.93	2.80	8.10	4.60
twolf	186	4.60	1.61	5.94	4.68
viterbi	36	7.64	2.46	5.79	3.07
vpr	367	2.12	0.52	2.20	2.17

states states. The third column, operations per state, indicates the average size of the instruction window available to the scheduler. For most applications, this window is of limited size.

Small block size limits the ability of the scheduler to expose more distant parallelism. Small block size is also detrimental to the pipelining of memory operations. The pipesplitting model requires all operations within a basic block to complete before the next block begins execution. This has the effect of flushing the memory pipeline after every basic block. While this greatly simplifies flow control, it restricts performance. In particular, allowing memory operations to span basic blocks would allow cross-iteration parallelism in inner loops.

In C-Cores, memory operations and their associated address computations often dominate the critical path through a basic block's schedule. Blocks with many memory operations are characterized by chains of memory operations on their critical path. These operations are not necessarily data dependent. As our compiler currently provides no static disambiguation support, and our memory interface is single issue and blocking, even independent memory operations produce distinctive schedules. As memory operations are the longest-latency operations we schedule, dependent memory operations produce the longest chains. The block in Figures 6.1 and 6.2 is an example of a schedule dominated by memory latency. Even with the latency reduction from cachelets, more than half of the raw operator latency on the critical path is through memory and address computation.

Figure 6.3 shows a more extreme example of a memory-constrained path through a block from the application djpeg. The level of detail has been reduced in order to fit in the space provided. This block highlights the distinct natures of the compute and memory resources C-Cores provide. Spatial computation provides plentiful datapath operators and abundant parallel computation in early fast-cycles. However, ICCs replicate memory resources in time, rather than space, and produce schedules with long chains of memory operations with generally decreasing datapath parallelism.

6.2.1 Memory and parallelism

From an energy standpoint, memory-constrained blocks are acceptable, as the values produced by operators early in the block will settle. From a performance standpoint, this is less optimal, as it indicates that the memory resource constraints dominate others, hinting at an imbalance in resource allocation. Naïvely increasing the width of the memory interface would likely improve performance, but at the cost of energy-efficiency and complexity. To understand the appropriate memory optimizations to apply, we must view the problem through the lens of automation.

Hand-crafted designs, such as [HQW+10], often benefit from knowledge of detailed program semantics. For an automated approach, it is often difficult to discern the precise semantics of applications and source code constructs from lower level representations. However, constructing conservation cores by hand for arbitrary and changing workloads is not scalable. The automation of this process is key to the success of the C-Core approach. Therefore, any knowledge we rely on regarding memory independence must stem from our toolchain.

Parallel memory access will only be energy efficient when co-scheduled memory operations are very rarely dependent or completely independent. Sometimes this is provable, through pointer analysis. In other cases, profiling can lend strong indications that given operations are independent. Energy efficiency does not pre-

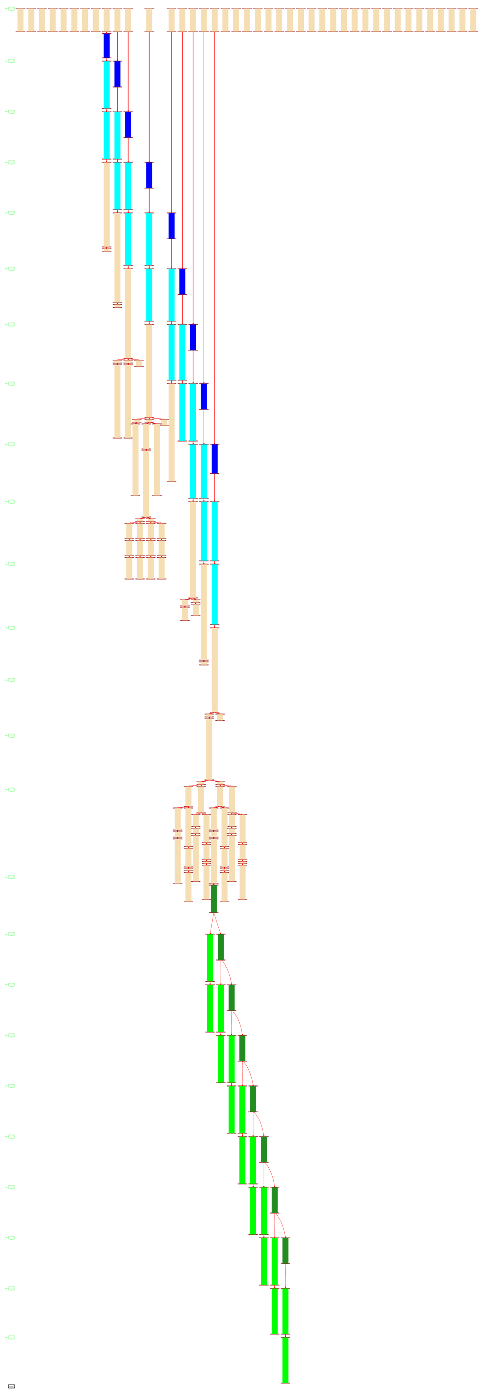


Figure 6.3: **Schedule for a memory-constrained block** This schedule for a block from djpeg shows the key features of memory-constrained blocks. Independent datapath operations cluster in early cycles, while both dependent and independent memory operations sequence their accesses to the memory interface over time.

clude speculative mechanisms, provided they are tuned to speculate only at very high confidence values. In both cases, the compiler infrastructure needed to fully estimate available parallelism is part of ongoing work. The degree of parallelism exposed will determine the aggressiveness of the hardware we will deploy to harness it.

Another key direction of ongoing work is reconsidering our choice of the basic block as our fundamental execution and scheduling unit. Working at the basic block level has allowed us to quickly develop energy-efficient hardware that is readily stylized and easy to reason about. With our current execution model and scheduling infrastructure, we achieve both performance and efficiency, but the above analysis shows that there remains untapped potential. As we have shifted our efforts to improving performance, the basic block has proved a limiting factor. For most C-Cores, the number of states is much greater than the number of loop bodies. This indicates that the basic block may not be the optimal unit for scheduling. Expanding the ability of our scheduler to expose parallelism is a key aspect of our ongoing efforts with conservation cores.

6.3 Future work

Conservation cores offer significant energy efficiency and, with targeted optimizations, improve performance. However, the internal organization of C-Cores can benefit from clever renovation. We have seen this once already with the introduction of pipesplitting. Likewise, the tuning and maturation of the C-Core toolchain is an ongoing process. At the time of the writing of this dissertation, there are several continuing efforts to improve our compilation and scheduling infrastructure to provide even greater performance and efficiency. Below, we briefly discuss near-term goals.

6.3.1 Unrolling

Our datapath uses a form of spatial computation, introducing hardware for every static operator. Loop unrolling will therefore increase the area requirements of conservation cores. However, it will also allow cross-iteration optimization within the scope of a single block. While we cannot apply unrolling indefinitely, there is room in our area budget for some degree of unrolling, especially for hot inner loops. Under the utilization wall, trading area for performance is usually a beneficial trade.

Support for unrolling requires changes in only one stage of our toolchain, the OpenImpact compiler. The version of the OpenImpact [Ope] compiler we build our toolchain on does not support loop unrolling in the optimization stages that produce the intermediate representation from which we derive C-Cores. As many previous efforts have studied loop unrolling, implementation is an engineering effort, rather than a research undertaking. Work to implement loop unrolling in our OpenImpact framework is ongoing as of the writing of this dissertation. Should this prove difficult, we are also considering source-level unrolling as part of C-Core selection transformations in LLVM [LA04].

6.3.2 Waves

Another way to use bigger blocks to expose more parallelism is to move from basic blocks to waves [SSM+07]. Waves, single entry, multiple exit CFG subgraphs, are a natural fit for our static-dataflow spatial execution model. The entire body of an inner loop will often be a single wave. Transitioning from basic blocks to waves is compatible with unrolling. Between the two techniques, our scheduler will have access to a greater portion of the inherent ILP in a benchmark.

Supporting waves in C-Cores requires C-Cores to implement some form of predication. The most straightforward support for C-Core predication is actually a non-speculative approach. We can transform comparator signals controlling state transitions into predicate lines that gate the control signals governing external interactions. Both the transformation and selecting the signals from the single valid case are straightforward. Similarly, merge points merely require multiplexed inputs. The chief challenge in moving from basic blocks to waves will be one of energy efficiency. Gating control signals prevents wrong-path execution from affecting the outside world. However, in a C-Core, we must also prevent wrong-path execution from unnecessarily toggling transistors in order to maintain energy efficiency.

6.3.3 Speculation

Moving to waves, or even hyperblocks, requires support for predication. However, we can apply even more optimizations if C-Cores add support for speculative execution. Given that saving energy is a C-Core design goal, any such speculation should be of a high-confidence variety. One can view cachelets as a form of high-confidence speculative optimization as their selection process is profile driven: Performance and energy efficiency, rather than correctness of execution path, are speculative. Another profile-driven speculative optimization is the integration of profile-derived disambiguation information into scheduling.

We ran an initial study using oracular disambiguation within basic blocks to speculatively hoist loads above stores in pre-cachelet ICC schedules. A few benchmarks, such as djpeg, benefited greatly from this, improving performance by up to 15%. However, most other benchmarks were insensitive, mostly due to the small size of blocks and lack of cross-iteration through-memory dependencies to be elided. Moreover, when we combined disambiguation with cachelets, the marginal benefit was small. The reordering of independent loads via uneven application of cachelets provided much of the benefit of within-block disambiguation. However, once we have implemented unrolling and moved from basic blocks to waves as our basic scheduling unit, we plan on reexamining disambiguation. As seen in Table 6.1, frequently executed blocks in djpeg, which improved with disambiguation, had several memory operations in each block. With waves and unrolling, we expect scheduling blocks to look more like those of djpeg. If the potential performance benefits are high, they may warrant the energy costs associated with an infrequently exercised rollback mechanism.

6.4 Summary

Over the course of this dissertation, we have seen how C-Cores save energy and, properly optimized, improve performance. In this chapter, we have analyzed key limiting factors in current C-Core performance and highlighted avenues for future optimizations. In the following chapter, we compare conservation cores to other approaches for generating specialized hardware. We also discuss how the techniques and mechanisms used to optimize conservation cores relate to other previously proposed techniques and mechanisms.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under NSF CAREER Awards 06483880 and 0846152, and under NSF CCF Award 0811794.

This chapter contains material from “Energy-Delay Optimized Accelerators for Irregular Code”, by Jack Sampson, Ganesh Venkatesh, Nathan Goulding, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor, which has been submitted for possible publication by IEEE in *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA).* The dissertation author was the primary investigator and author of this paper.

Chapter 7

Related Work

In this chapter we place C-Cores and systems containing them within the landscape of other hardware specialization efforts. We compare our goals, methods, and achievements with prior work. We examine these previous efforts from three perspectives: hardware specialization, heterogeneous system design, and the automation of hardware generation. We address each category in turn, and discuss work related to subsidiary techniques and methods.

7.1 Hardware specialization

Customizing hardware for a particular application can provide significant gains to energy or performance. In [HQW^+10], the authors develop an ASIC that uses $500\times$ less energy for the HD H.264 encoder than an aggressive out-of-order core. Both ELM [BDBS^+08] and the work in [HQW^+10] break down the relative energy contributions of I-cache, fetch, decode, register-file access, functional units, D-cache and control for average processor instructions, showing sources of inefficiency in general purpose processor computation. These areas are all targets for removal or reduction via hardware specialization. The energy benefits of C-Cores' removal of fetch and decode, replacement of register-files with distributed registers and merging basic blocks into complex meta-operations are consistent with their analysis.

Many other specialized hardware systems exist either to improve performance, or to reduce energy. Many of these, like Cell [Kah05], IRAM [PAC$^+$97], and EXOCHI [WCC$^+$07], have been designed as domain-specific accelerators. Specialized accelerators are a subject of increasing interest in recent computer architecture research. Recent work has targeted accelerators for computations such as cryptography [WWA01], signal processing [ECF96, GSM$^+$99], vector processing [ADK$^+$04, DLD$^+$03], physical simulation [Age], and computer graphics [nVi, ATI, OLG$^+$05]. Other vendors, such as Phillips [Phi97] and Equator [map01], provide non-configurable but specialized designs for media applications.

Stream processors, such as [ADK$^+$04, DLD$^+$03], focus on localizing communication, and therefore on reducing energy. These processors also provide significant speedups to target applications. However, stream processors require source programs to be formulated as a set of communicating kernels. C-Cores provide energy and performance benefits without stringent limitations on the structure of the source program. C-Cores thus support legacy applications as well as code that is not readily transformed into streaming, vector, or other of the highly parallel forms targeted by traditional accelerator architectures.

The unifying concept in the above designs is the parallelism inherent in their problem domains. In the case of programmable approaches, such as GPUs and physics accelerators, this parallelism is sufficient to hide the limitations of slow individual processing units and loose coupling between the accelerator and the host system. However, while such programmable platforms offer very high performance in their domain, they rely on SIMD parallelism and perform poorly in the face of irregular control flow. Thus, they are a poor match for the irregular codes that C-Cores target.

Many ASIC-like accelerators [CHM08, FKDM09, YGBT09] have focused on using modulo scheduling to exploit regular loop bodies that have ample loop parallelism and easy-to-analyze memory access patterns. VEAL [CHM08] presents a general approach for accelerating highly structured, non-speculative loops by using a combination of static and dynamic techniques to map modulo schedulable loops in native binary code onto a generic loop accelerator. The loop accelerator uses address generators to decouple the fetching of inputs and storing of results from loop body computation on a configurable array of functional units. For the limited class of loops covered, VEAL achieves much of the potential speedup of an infinitely wide accelerator while maintaining a small area footprint. The work in [FKDM09] and [YGBT09] design circuits with limited flexibility by incorporating limited programmability, or by merging multiple circuits into one, respectively.

In contrast to many of the above examples, our approach is general, rather than domain-specific. C-Cores differ in that they target the more general class of irregular, hard-to-parallelize computations that are not well-suited to modulo scheduling. We can target any C codebase once it reaches a minimal level of code stability, and our patching mechanisms allow continued utility from C-Core hardware even as codebases change. Additionally, unlike many accelerators, we prioritize energy reduction over performance improvement.

7.2 Heterogeneous systems

In order to run entire workloads, specialized hardware is frequently a component of a larger, heterogeneous system. There are many types of heterogeneous systems, ranging from general purpose processors with loosely coupled slave accelerators to single chip solutions directly targeting a specific domain. The C-Core approach designs heterogeneous multi-core processors with an aim of maximizing energy efficiency.

Recent work on single-ISA heterogeneous multi-core processors [KTR$^+$04, KFJ$^+$03, KTJ06, BRUL05, GRSW04, LM05] investigates the power and performance trade-offs for CMPs with non-uniform cores. In [KFJ$^+$03], the authors use phase-transition-driven partitioning to trade 10% of performance for a nearly 50% reduction in power consumption by moving execution between aggressive out-of-order cores and simpler, in-order cores. The work in [KTJ06] performs a design space search over many possible core types and combinations in a four core heterogeneous chip, optimizing for area and power. The authors find that, for optimal designs, no core in such a heterogeneous processor monotonically improves over the other cores included, and that for several power envelopes, most of the potential benefits are achievable with limited diversity.

The single-ISA approach uses heterogeneity to improve energy efficiency, but it does not leverage hardware specialization. It does not benefit from operator, communication, or control flow optimizations specific to a domain or application. While the presence of out-of-order cores allows for higher peak performance than C-Cores, the single-ISA approach is limited in peak energy savings by the efficiency of the core most closely matching the current computation. The conservation core architecture can deliver larger energy savings, as it reduces system energy by an average of 47% over even a simple, *in-order* core.

A range of commercial heterogeneous processors and research prototypes are available or have been proposed, including Sony's Cell [Kah05], IRAM [PAC+97], Merrimac [DLD+03] and Intel's EXOCHI [WCC+07]. These machines augment a general purpose processor with tightly coupled vector or stream co-processors to accelerate multimedia and other SIMD-rich applications. Approaches for coupling the general and more specialized cores vary. Cell [Kah05] couples a single, simple general purpose core with 8 identical SPE units, each with a private memory, and communicates via DMA transfers over a set of high bandwidth ring buses. EXOCHI [WCC+07], on the other hand, dispatches from an aggressive, out-of-order multi-core processor to an 8-threaded media processor and communicates through shared virtual memory. Meanwhile, loosely coupled heterogeneous systems comprising a general purpose multi-core processor and one or more GPUs have become ubiquitous on the desktop and other markets. Similarly attached specialized accelerators for physics [Age] or signal processing [CSX07] have also appeared in recent years.

These designs have features common to many heterogeneous platforms. They target a specific domain, retain general purpose host cores to run code outside their targeted domain, and require programmer intervention to take advantage of specialized resources. While EXOCHI provides a uniform abstraction for sequencing execution across heterogeneous execution engines, their framework still requires distinct source code for each target piece of hardware. Similarly, programs written for traditional microprocessors require rewriting to take full advantage of Cell's

SPEs or to run as stream kernels on Merrimac. While language extensions such as CUDA [NBGS08] and streaming frameworks such as Brook [BFH+04] attempt to ease the use of GPUs and other loosely coupled accelerators for "general purpose" computation (GPGPU), such approaches still require source code rewrites and focus on parallel computation.

C-Core-enabled systems also have a host processor, but otherwise differ from the above systems. A fundamental goal of the C-Core approach is that the presence of specialized hardware should be transparent to the programmer. C-Cores, unlike the above systems, require no manual source modifications in order to take advantage of specialized hardware.

Cell, EXOCHI, and IRAM all target vector parallelism in highly structured computations. Each C-Core similarly targets a very specific functionality, but the automated nature of the C-Core toolchain allows us to rapidly design as many C-Cores for as many domains as a given target system requires. While a C-Core enabled system is generated with a particular workload in mind, that workload can span multiple domains. The code regions that become C-Cores within the same system do not have to have internally uniform structure, and need not be similar to each other. C-Cores support both regular and irregular code regions, and are parallelism agnostic.

Previous work has also proposed generalist heterogeneous systems and combining cores that exhibit microarchitectural heterogeneity to improve performance or reduce power consumption on general purpose workloads. Designs such as Chimaera [YMHB00], Tartan [MCC+06], GARP [HW97], PRISC [RS94], and the work in [CBC+05] augment a general-purpose processor with reconfigurable logic. These approaches provide widely varying degrees of code coverage. Chimeara [YMHB00] provides a tightly integrated 9-input 1-output reconfigurable functional unit to accelerate common instruction sequences, and GARP [HW97] uses VLIW software pipelining techniques to implement pipelined versions of loops in an FPGA fabric, while Tartan [MCC+06] aims to map entire programs onto a hierarchical coarse-grained asynchronous fabric.

A reconfigurable ASIP-like approach is presented in [CBC+05], executing datagraphs on reconfigurable feed-forward hardware. The reconfigurable logic does not, however, support memory access or internal control, limiting the scope of new operators. Other operator customization approaches, such as Tensilica's Stenos [WKMR01] and OptimoDE [CZF$^+$04], provide configure-once, rather than reconfigurable, specialized datapath operators.

While the C-Core patching mechanism does provide a degree of reconfigurability and associated overheads, the application-specific nature of a C-Core still yields an energy efficiency much closer to an ASIC than to a reconfigurable fabric or co-processor. Depending on the workload, fine-grained reconfigurable fabrics do not always save power compared to efficient in-order cores. Reconfigurable logic does allow for greater potential functionality to be mapped into a smaller silicon area. However, estimates in [MCC$^+$06] showed that mapping entire programs into reconfigurable fabric was not practical without fabric virtualization, adding runtime overheads to reconfiguration. Moreover, under the utilization wall, area restrictions are not as constraining and energy efficiency is more pressing. C-Cores can also target larger regions of code than operator customization approaches, customizing control and communication between basic blocks as well as within them.

7.3 Automatically generated hardware

Specialized hardware delivers the greatest benefits when form, as communication paths, operator selection, or control specialization, most closely follows function. Software tends to contain many distinct patterns of computation, even within a single application [SPC01]. Thus, to maximize benefits, systems targeting diverse computations with specialized hardware must have a high degree of heterogeneity in the specialized hardware they employ. In order to scalably supply specialized hardware for many different software structures within a single design or even across designs requires some degree of design automation.

Previous efforts have attempted to automatically generate optimal architectures for constrained design spaces. Strozek and Brooks [SB06] improve the energy efficiency for a set of applications in the embedded space by automatically selecting a heterogeneous set of specialized cores from a well-defined design space. The cores are Pareto-optimal for the target workloads. PICO [ARK99] employs a similar approach to VLIW design space exploration, but limits heterogeneity to functional unit number and construction.

Our automated approach admits a much larger range of core designs, but sacrifices formal guarantees of optimality. Both approaches retain many features of general purpose processors, such as instruction fetch and instruction decode. Data from [BDBS$^+$08] show that, even if ISAs improve the encoding of instructions and specialize datapath operators, the process of fetching an instruction and decoding and retrieving its operands from a register file remains a significant energy overhead. Compared to [SB06] and [ARK99], C-Cores benefit from optimizations not available to programmable architectures. C-Cores eliminate instruction fetch for the most frequently executed code regions and employ distributed registers rather than a centralized register-file.

Work by Budiu on spatial computation (ASH) [BVCG04, BG03] and the associated CASH compiler [BG02] has similar goals in transforming C directly into hardware. However, the circuits produced are very different from C-Cores. ASH circuits are asynchronous designs, whereas C-Cores are synchronous. ASH produces heavily pipelined circuits with output latches for every operation, whereas C-Cores employ pipesplitting to transform each basic block into a composite operation, removing intermediate storage and transfer and allowing bit-level optimizations.

7.4 Techniques

In this section, we discuss techniques and mechanisms related to some of those used in developing conservation cores.

7.4.1 Bit-level parallelism

Our approach, pipeline splitting, uses operator chaining to execute the datapath operators unaligned to the cycle boundaries. Pipeline splitting schedules the operations of the sequential code to increase the potential for operation chaining, and then allows the operator chains to execute across many fast clock cycles. While estimating operation latency, the scheduler takes into account the speed-ups provided by bit-level parallelism across dependent operations, allowing for tighter packing of operations.

Several designs have leveraged the bit-level parallelism that pipesplitting exposes between datapath operations. The approach presented in [SA02] schedules multiple dependent operators back-to-back in the same cycle to help physical synthesis meet frequency targets. The approach in [PPM09] uses the technique to reduce register file accesses for sequential code regions. Finally, the work in [DDF+08] moves datapath operators across pipeline registers to prevent short path-related false positive timing errors. These techniques reschedule operators across just one or two cycles. Pipesplitting applies this technique more aggressively, eliminating most pipeline registers between datapath components. Furthermore, pipesplitting applies the technique only to arithmetic operators, leaving memory to run fully pipelined.

7.4.2 Cache specialization

C-Cores provide a higher-performing and more-efficient memory system, with pipelined access and integrated cachelets. The CHiMPS multi-cache architecture [PEB+09] uses several application-specific caches and enforces coherence via flushing, but the purpose, sizing, and implementation of CHiMPS multi-cache differs greatly from the cachelet approach. CHiMPS aggregates 4-KB block RAMs on an FPGA into caches backing different regions of memory in order to provide memory parallelism and to simplify the memory interface for a C-like programming model. In contrast, cachelets utilize small caches with between one and four lines that reduce the average hit time and access energy by eliding accesses to the L1.

7.4.3 Clock gating

C-Cores with pipesplitting offer interesting opportunities for clock gating, especially RTL and symbolic clock gating approaches [DIBM03, BDMM$^+$99]. In a C-Core, only one basic block is active at a time, and the output registers associated with that block are only clocked once per basic block execution. Thus, the activity factor for any slow-clock net is very low. Moreover, the pipesplitting model provides guarantees about the minimum number of fast-clock cycles before a register could possibly need to be clocked again given the currently executing block and fast-state. This may allow C-Cores to benefit from deterministic dynamic clock gating policies such as those in [LBC$^+$03]. We currently model fine-grained clock gating. However, we are investigating methods for providing sufficient information to our CAD tools to generate pruning logic closer to the root of the clock tree. Since the vast majority of registers in a C-Core should be clock gated in any given fast-clock cycle, substantial pruning near the root of the tree should be possible.

7.4.4 Dynamic voltage and frequency scaling

C-Cores aim to reduce energy, and many past proposals used dynamic voltage and frequency scaling (DVFS) either at chip-level [GCW95, PBB98], or finer granularity [IM02, SMB$^+$02], to achieve the same end. However, we do not consider DVFS as a particularly valuable addition or viable alternative to C-Cores. We came to this conclusion for two reasons: First, the same leakage concerns that spelled the end of classical CMOS scaling reduce the effectiveness of DVFS at advanced process nodes. Reducing the threshold voltage would produce large leakage currents. With a fixed threshold voltage, reductions in supply voltage will deeply reduce switching speed. In a leakage limited domain, running slower reduces power, but weakens the translation from lower power to lower energy. With each generation of voltage stagnation, C-Cores become more a more attractive solution and DVFS less so. Second, DVFS trades performance for power. While C-Cores are energy-oriented, they trade for power with idle transistors rather than performance, a property still in demand.

7.5 Summary

Conservation cores are specifically designed to address the challenges of the utilization wall, turning dark silicon into specialized hardware for irregular applications. This goal serves to differentiate C-Cores from other heterogeneous systems and domain specific efforts. Even where conservation cores use existing techniques, such as exploiting bit-level parallelism, their benefits derive from application at a different scale and for a different purpose. In the next and final chapter, we will summarize the benefits of conservation cores, and the contributions of this dissertation.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under NSF CAREER Awards 06483880 and 0846152, and under NSF CCF Award 0811794.

This chapter contains material from “Conservation cores: reducing the energy of mature computations”, by Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson and Michael Bedford Taylor, which appears in *ASPLOS ’10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. The dissertation author was the secondary investigator and author of this paper. The material in this chapter is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or email

This chapter contains material from “Energy-Delay Optimized Accelerators for Irregular Code”, by Jack Sampson, Ganesh Venkatesh, Nathan Goulding, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor, which has been submitted for possible publication by IEEE in *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. The dissertation author was the primary investigator and author of this paper.

Chapter 8

Summary

The utilization wall is upon us, and it will fundamentally change the way that we design future processors. We began this dissertation by characterizing the utilization wall, describing how it will lead to an exponentially expanding area of dark silicon. We have shown how the effects of the utilization wall are already apparent in modern processors and how homogeneous processors are ill-suited to addressing the challenges of the utilization wall. While traditional designs are limited, we have shown that heterogeneous designs with specialized hardware can make use of the burgeoning dark silicon area. Over the course of this dissertation, we have presented an approach for the design of such heterogeneous systems, a toolchain that embodies that approach, and evaluated the product of that approach, conservation cores.

As we run up against the utilization wall, we enter a regime in which reducing energy per operation becomes increasingly important. In Chapter 3 we introduced conservation cores, a new class of circuits that aim to increase the energy efficiency of key applications in a workload. Conservation cores provide an automated approach to having the most important pieces of the most important applications in a workload run on the hardware most specialized for them.

Conservation cores are the product of an automated C-to-silicon toolchain. Our toolchain synthesizes C-Cores from C code and builds in support that allows them to evolve when new versions of the software appear. This automation is a fundamental aspect of the conservation core approach. Automatic extraction of key code regions from source and drop-in semantics allow our toolchain to transparently improve applications with specialized hardware. Tansparency and automation provide scalability to the conservation core approach. Our toolchain already produces energy efficient and performant hardware, and the quality of that hardware continues to improve as the toolchain matures.

In Chapters 4 and 5 we showed the evolution of C-Cores from our initial energy-saving prototypes to their more performance optimized successors. Our improved C-Cores use three key techniques to reduce energy consumption and improve performance compared to both a general purpose processor and our C-Core prototypes. First, C-Cores use pipesplitting, a pipelining technique that allows them to use a high-speed memory system while running non-memory operations at a much lower frequency. Second, refined support for changes to the software that C-Cores implement improves energy efficiency and significantly decreases area. Finally, cachelets reduce L1 hit times while maintaining a coherent memory interface. Together, these techniques speed up the code they target by 1.5×, improve EDP by $6.9 times$ and accelerate the whole application by 1.33× on average, while reducing application energy-delay by 57%. In Chapter 6, we showed how the current operation scheduling approach intertwines with hardware design decisions and highlighted avenues for further toolchain improvement.

Overall, we have shown that conservation cores are a viable means of addressing the most pressing challenges presented by the utilization wall. C-Cores provide an architectural means to trade area, now inexpensive with the advent of dark silicon, for power. C-Cores improve energy efficiency for nearly arbitrary code regions. Thus, the approach can be applied to any computing domain. Furthermore, C-Cores offer sufficient performance on irregular applications to be deployed as part of a general-purpose heterogeneous system. This makes C-Cores an attractive design alternative, especially for power-constrained and energy-constrained platforms.

Acknowledgments

Portions of this research were funded by the US National Science Foundation under NSF CAREER Awards 06483880 and 0846152, and under NSF CCF Award 0811794.

This chapter contains material from "Conservation cores: reducing the energy of mature computations", by Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson and Michael Bedford Taylor, which appears in *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. The dissertation author was the secondary investigator and author of this paper. The material in this chapter is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or email `permissions@acm.org`.

This chapter contains material from "Energy-Delay Optimized Accelerators for Irregular Code", by Jack Sampson, Ganesh Venkatesh, Nathan Goulding, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor, which has been submitted for possible publication by IEEE in *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA).* The dissertation author was the primary investigator and author of this paper.

Bibliography

[ADK+04] Jung Ho Ahn, William J. Dally, Brucek Khailany, Ujval J. Kapasi, and Abhishek Das. Evaluating the Imagine Stream Architecture. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 14–25. IEEE Computer Society, 2004.

[Age] Ageia Technologies. PhysX by Ageia. `http://www.ageia.com/pdf/ds_product_overview.pdf`.

[ARK99] Shail Aditya, B. Ramakrishna Rau, and Vinod Kathail. Automatic architectural synthesis of VLIW and EPIC processors. In *ISSS '99: Proceedings of the 12th international symposium on System synthesis*, page 107. IEEE Computer Society, 1999.

[ATI] ATI website. http://www.ati.com.

[BDBS+08] James Balfour, William Dally, David Black-Schaffer, Vishal Parikh, and JongSoo Park. An energy-efficient processor architecture for embedded systems. *IEEE Comput. Archit. Lett.*, 7(1):29–32, 2008.

[BDMM+99] L. Benini, G. De Micheli, E. Macii, M. Poncino, and R. Scarsi. Symbolic synthesis of clock-gating logic for power optimization of synchronous controllers. *ACM Trans. Des. Autom. Electron. Syst.*, 4(4):351–375, 1999.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[BG02] Mihai Budiu and Seth Copen Goldstein. Compiling application-specific hardware. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 853–863, Montpellier (La Grande-Motte), France, September 2–4 2002.

[BG03] Mihai Budiu and Seth Copen Goldstein. Optimizing memory accesses for spatial computation. In *International ACM/IEEE Symposium on Code Generation and Optimization (CGO)*, pages 216–227, San Francisco, CA, March 23–26 2003.

[BK75] B. T. Bennett and V. J. Kruskal. Lru stack processing. *IBM Journal of Research and Development*, pages 353–357, July 1975.

[BRUL05] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 506–517, Washington, DC, USA, 2005. IEEE Computer Society.

[BVCG04] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 14–26, Boston, MA, October 2004.

[CBC$^+$05] Nathan Clark, Jason Blome, Michael Chu, Scott Mahlke, Stuart Biles, and Krisztian Flautner. An architecture framework for transparent instruction set customization in embedded processors. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 272–283. IEEE Computer Society, 2005.

[CFR$^+$89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM Press, 1989.

[CHM08] Nathan Clark, Amir Hormati, and Scott Mahlke. Veal: Virtualized execution accelerator for loops. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 389–400, Washington, DC, USA, 2008. IEEE Computer Society.

[CLG02] Josep M. Codina, Josep Llosa, and Antonio González. A comparative study of modulo scheduling techniques. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 97–106, New York, NY, USA, 2002. ACM.

[Cod] CodeSurfer by GrammaTech, Inc. http://www.grammatech.com/products/codesurfer/.

[CSX07] February 2007. `http://www.clearspeed.com/docs/resources/ClearSpeed_Architecture_Whitepaper_Feb07v2.pdf`.

[CZF+04] Nathan Clark, Hongtao Zhong, Kevin Fan, Scott Mahlke, Krisztian Flautner, , and Koen Van Nieuwenhove. OptimoDE: Programmable accelerator engines through retargetable customization. In *HotChips*, 2004.

[DDF+08] Ganesh Dasika, Shidhartha Das, Kevin Fan, Scott Mahlke, and David Bull. Dvfs in loop accelerators using blades. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 894–897, New York, NY, USA, 2008. ACM.

[DGR+74] R.H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. In *IEEE Journal of Solid-State Circuits*, October 1974.

[DIBM03] Monica Donno, Alessandro Ivaldi, Luca Benini, and Enrico Macii. Clock-tree power optimization based on rtl clock-gating. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 622–627, New York, NY, USA, 2003. ACM.

[DLD+03] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35. IEEE Computer Society, 2003.

[ECF96] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. RaPiD - reconfigurable pipelined datapath. In *FPL '96: Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pages 126–135. Springer-Verlag, 1996.

[Emb] Embedded Microprocessor Benchmark Consortium. Eembc benchmark suite. http://www.eembc.org.

[FKDM09] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In *HPCA: High Performance Computer Architecture.*, pages 313–322, Feb. 2009.

[GCW95] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *MobiCom*

'95: Proceedings of the 1st annual international conference on Mobile computing and networking, pages 13–25, New York, NY, USA, 1995. ACM.

[Gro] Independent JPEG Group. Library for jpeg image compression. http://www.ijg.org/.

[GRSW04] Ed Grochowski, Ronny Ronen, John Shen, and Hong Wang. Best of both latency and throughput. In *ICCD '04: Proceedings of the IEEE International Conference on Computer Design (ICCD'04)*, pages 236–243, Washington, DC, USA, 2004. IEEE Computer Society.

[GSM$^+$99] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *ISCA '99: Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 28–39. IEEE Computer Society, 1999.

[HQW$^+$10] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, pages 37–47, New York, NY, USA, 2010. ACM.

[HW97] John R. Hauser and John Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In Kenneth L. Pocek and Jeffrey Arnold, editors, *FCCM '97: IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21. IEEE Computer Society Press, 1997.

[IM02] Anoop Iyer and Diana Marculescu. Power and performance evaluation of globally asynchronous locally synchronous processors. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 158–168, Washington, DC, USA, 2002. IEEE Computer Society.

[Kah05] Jim Kahle. The CELL processor architecture. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, page 3. IEEE Computer Society, 2005.

[KFJ$^+$03] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *MICRO 36: Proceedings of the 36th Annual IEEE/ACM*

International Symposium on Microarchitecture, page 81. IEEE Computer Society, 2003.

[KTJ06] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 23–32, New York, NY, USA, 2006. ACM Press.

[KTMW03] Jason Sungtae Kim, Michael B Taylor, Jason Miller, and David Wentzlaff. Energy characterization of a tiled architecture processor with on-chip networks. In *International Symposium on Low Power Electronics and Design*, San Diego, CA, USA, August 2003.

[KTR$^+$04] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 64. IEEE Computer Society, 2004.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75. IEEE Computer Society, 2004.

[LBC$^+$03] Hai Li, Swarup Bhunia, Yiran Chen, T. N. Vijaykumar, and Kaushik Roy. Deterministic clock gating for microprocessor power reduction. *High-Performance Computer Architecture, International Symposium on*, 0:113, 2003.

[LM05] Jian Li and José F. Martínez. Power-performance considerations of parallel computing on chip multiprocessors. *ACM Trans. Archit. Code Optim.*, 2(4):397–422, 2005.

[map01] MAP-CA datasheet, June 2001. Equator Technologies.

[MCC$^+$06] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. Tartan: evaluating spatial computation for whole program execution. *SIGOPS Oper. Syst. Rev.*, 40(5):163–174, 2006.

[MIP09] MIPS Technologies. MIPS Technologies product page, 2008-2009. `http://www.mips.com/products/processors/32-64-bit-cores/mips32-24ke` , 2008-2009.

[MIP10] MIPS Technologies. MIPS Technologies product page, 2010. `http://www.mips.com/products/cores/32-64-bit-cores/mips32-24ke`, 2010.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–14, New York, NY, USA, 2008. ACM.

[nVi] nVidia website. http://www.nvidia.com.

[OLG+05] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, , and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[Ope] OpenImpact Website. http://gelato.uiuc.edu/.

[PAC+97] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, April 1997.

[PBB98] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pages 76–81, New York, NY, USA, 1998. ACM.

[PEB+09] Andrew Putnam, Susan Eggers, Dave Bennett, Eric Dellinger, Jeff Mason, Henry Styles, Prasanna Sundararajan, and Ralph Wittig. Performance and power of cache-based reconfigurable computing. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 395–405, New York, NY, USA, 2009. ACM.

[Phi97] TM1000 preliminary data book, 1997. `http://www.semiconductors.philips.com/acrobat/other/tm1000.pdf`.

[PPM09] Yongjun Park, Hyunchul Park, and Scott Mahlke. Cgra express: accelerating execution using dynamic operation fusion. In *CASES '09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 271–280, New York, NY, USA, 2009. ACM.

[RS94] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *MICRO*

27: Proceedings of the 27th annual international symposium on Microarchitecture, pages 172–180. ACM Press, 1994.

[SA02] Mukund Sivaraman and Shail Aditya. Cycle-time aware architecture synthesis of custom hardware accelerators. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 35–42, New York, NY, USA, 2002. ACM.

[SB06] Lukasz Strozek and David Brooks. Efficient architectures through application clustering and architectural heterogeneity. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 190–200, New York, NY, USA, 2006. ACM Press.

[SMB$^+$02] Greg Semeraro, Grigorios Magklis, Rajeev Balasubramonian, David H. Albonesi, Sandhya Dwarkadas, and Michael L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 29, Washington, DC, USA, 2002. IEEE Computer Society.

[SPC01] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[SPE00] SPEC. SPEC CPU 2000 benchmark specifications, 2000. SPEC2000 Benchmark Release.

[SSM$^+$07] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. The wavescalar architecture. *ACM Trans. Comput. Syst.*, 25(2):4, 2007.

[TH04] Dave A. D. Tompkins and Holger H. Hoos. Ubcsat: An implementation and experimentation environment for sls algorithms for sat and max-sat. In *In SAT*, pages 37–46, 2004.

[TLM$^+$04] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal.

Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *ISCA '04: Proceedings of the 31st annual International Symposium on Computer Architecture*, page 2. IEEE Computer Society, 2004.

[TMAJ08] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20, HP Labs, Palo Alto, 2008.

[VSG$^+$10] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 205–218, New York, NY, USA, 2010. ACM.

[WCC$^+$07] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 156–166, New York, NY, USA, 2007. ACM.

[WKMR01] Albert Wang, Earl Killian, Dror Maydan, and Chris Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 184–188. ACM Press, 2001.

[WOT$^+$95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.

[WWA01] Lisa Wu, Chris Weaver, and Todd Austin. Cryptomaniac: A fast flexible architecture for secure communication. In *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 110–119. ACM Press, 2001.

[YGBT09] S. Yehia, S. Girbal, H. Berry, and O. Temam. Reconciling specialization and flexibility through compound circuits. In *HPCA 15: High Performance Computer Architecture*, pages 277–288, Feb. 2009.

[YMHB00] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee. CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 225–235. ACM Press, 2000.