Generalized methods for application specific hardware specialization

by

Snehasish Kumar

M. Sc., Simon Fraser University, 2013
B. Tech., Biju Patnaik University of Technology, 2010

Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Spring 2017

Approval

Name:	**Snehasish Kumar**
Degree:	**Doctor of Philosophy (Computing Science)**
Title:	***Generalized methods for application specific hardware specialization***
Examining Committee:	**Chair:** Binay Bhattacharyya Professor

Arrvindh Shriraman
Senior Supervisor
Associate Professor
Simon Fraser University _____

William Sumner
Supervisor
Assistant Professor
Simon Fraser University _____

Vijayalakshmi Srinivasan
Supervisor
Research Staff Member,
IBM Research _____

Alexandra Fedorova
Supervisor
Associate Professor
University of British Columbia _____

Richard Vaughan
Internal Examiner
Associate Professor
Simon Fraser University _____

Andreas Moshovos
External Examiner
Professor
University of Toronto _____

Date Defended: November 21, 2016

Abstract

Since the invention of the microprocessor in 1971, the computational capacity of the microprocessor has scaled over 1000× with Moore and Dennard scaling. Dennard scaling ended with a rapid increase in leakage power 30 years after it was proposed. This ushered in the era of multiprocessing where additional transistors afforded by Moore's scaling were put to use. With the scaling of computational capacity no longer guaranteed every generation, application specific hardware specialization is an attractive alternative to sustain scaling trends. Hardware specialization broadly refers to the identification and optimization of recurrent patterns, dynamic and static, in software via integrated circuitry.

This dissertation describes a two-pronged approach to architectural specialization. First, a top down approach uses program analysis to determine code regions amenable for specialization. We have implemented a prototype compiler tool-chain to automatically identify, analyze, extract and grow code segments which are amenable to specialization in a methodical manner. Second, a bottom up approach evaluated particular hardware enhancements to enable the efficient data movement of specialized regions. We have devised and evaluated coherence protocols and flexible caching mechanisms to reduce the overhead of data movement within specialized regions.

The former workload centric approach analyses programs at the *path* granularity. Our observations show that analysis of amenability for specialization along the path granularity yield different conclusions than prior work. We analyse the potential for performance and energy improvement via specialization at the path granularity. We develop mechanisms to extract and merge amenable paths into segments called *Braids*. This allows for increased offload opportunity while retaining the same interface as path granularity specialization.

To address the challenges of data movement, the latter micro-architecture first approach, proposes a specialized coherence protocol tailored for accelerators and an adaptive granularity caching mechanism. The hybrid coherence protocol localizes data movement to a specialized accelerator-only tile reducing energy consumption and improving performance. Modern workloads have varied program characteristics where fixed granularity caching often introduces waste in the cache hierarchy. We propose a variable granularity caching mechanism which reduces energy consumption while improving performance via better utilization of the available storage space.

Keywords: hardware accelerators, program analysis, energy efficiency, caching, accelerator benchmark, coherence protocol

Dedication

Dedicated to Baba, Ma, Bukum and Tinki

Acknowledgements

I would like to thank my supervisors, Dr. Arrvindh Shriraman, Dr. Nick Sumner, Dr. Viji Srinivasan and Dr. Alexandra Fedorova for their guidance, motivation and support. I would like to thank my colleagues, Amirali Sharifian, Naveen Vedula, Steve Margerm and Dr. Apala Guha for their help and support. I would also like to thank Dr. Andreas Moshovos for serving as examiner on such short notice. Additionally, the departmental staff, Melissa and David, have been very helpful in the final stages of preparing the thesis. Finally, I would like to thank my lovely wife, Tinki, for her patience as I worked my way through graduate school instead of getting a real job.

Table of Contents

Approval ii

Abstract iii

Dedication v

Acknowledgements vi

Table of Contents vii

List of Tables xi

List of Figures xii

1 Introduction 1

1.1 The breakdown of technology scaling 1

1.2 Challenges 6

1.3 Approach 7

1.4 Dissertation Organisation 9

2 Background 11

2.1 Application Specific Hardware Specialization 14

2.2 Challenges 15

2.2.1 Challenge 1: What to specialize? 15

2.2.2 Challenge 2: How to specialize? 17

2.2.3 Challenge 3: Integration 18

2.3 Thesis Contributions 19

2.4 Relationship to published work 21

3 What to specialize – Extracting Accelerator Benchmarks from Microprocessor Benchmarks 23

3.1 Introduction 23

3.2 Motivation & Methodology 25

3.2.1 Acyclic Program Paths [15] 25

3.2.2 Selecting Paths to Characterize . . . 27
3.2.3 Extracting identified paths . . . 28
3.2.4 Metrics & ISA-independence . . . 29
3.2.5 Characterizing at the Path Level . . . 29
3.2.6 Benchmarks . . . 30
3.3 Characterization . . . 30
3.3.1 Making a case for Path-based Acceleration . . . 30
3.3.2 Characteristics Summary . . . 35
3.4 Path Characteristic Variability . . . 38
3.5 Path Derived Workload Suite . . . 42
3.5.1 Memory Address Entropy Analysis . . . 42
3.6 Related Work . . . 43
3.7 Conclusion . . . 44

4 How to specialize – Leveraging Program Analysis to Extract Accelerators from Whole Programs 45
4.1 Introduction . . . 45
4.2 Scope and Related Work . . . 47
4.2.1 Hardware Accelerator Perspective . . . 47
4.2.2 Compilers for VLIW processors . . . 50
4.3 BL-Path Accelerators . . . 53
4.3.1 Path Ranking . . . 54
4.3.2 BL-Path Properties . . . 55
4.4 BL-Path Expansion and Braids . . . 57
4.4.1 BL-Path Target Expansion . . . 57
4.4.2 Braids – Merging BL-Paths . . . 58
4.5 Execution Model . . . 61
4.6 Evaluation . . . 62
4.6.1 Performance . . . 63
4.6.2 Energy Evaluation . . . 65
4.7 Conclusion . . . 67

5 Integration – Coherent Cache Hierarchies for Accelerators 68
5.1 Introduction . . . 68
5.2 Background and Motivation . . . 72
5.2.1 Baseline Architectures . . . 73
5.3 *FUSION*: A Coherent Accelerator Cache Hierarchy . . . 76
5.3.1 Design Overview . . . 76
5.3.2 *FUSION* Architecture . . . 78
5.4 Toolchain and Benchmarks . . . 82

5.5 Evaluation . . . 85
5.5.1 Performance . . . 85
5.5.2 Energy . . . 87
5.5.3 Writeback vs Write-Through at L0X . . . 88
5.5.4 *FUSION-Dx*: Write Forwarding . . . 88
5.5.5 Larger AXC caches . . . 89
5.5.6 Address Translation . . . 89
5.6 Related Work . . . 90
5.7 Summary . . . 92

6 Integration – Adaptive Granularity Caching 93
6.1 Introduction . . . 93
6.2 Motivation for Adaptive Blocks . . . 95
6.2.1 Cache Utilization . . . 96
6.2.2 Effect of Block Granularity on Miss Rate and Bandwidth . . . 96
6.2.3 Need for adaptive cache blocks . . . 98
6.3 Amoeba-Cache: Architecture . . . 98
6.3.1 Amoeba Blocks and Set-Indexing . . . 100
6.3.2 Data Lookup . . . 101
6.3.3 Amoeba Block Insertion . . . 101
6.3.4 Replacement: Pseudo LRU . . . 102
6.3.5 Partial Misses . . . 102
6.4 Hardware Complexity . . . 103
6.4.1 Cache Controller . . . 104
6.4.2 Area, Latency, and Energy Overhead . . . 105
6.4.3 Tag-only Operations . . . 106
6.4.4 Tradeoff with Large Caches . . . 107
6.5 Chip-Level Issues . . . 108
6.5.1 Spatial Patterns Prediction . . . 108
6.5.2 Multi-level Caches . . . 109
6.5.3 Cache Coherence . . . 110
6.6 Evaluation . . . 110
6.6.1 Improved Memory Hierarchy Efficiency . . . 111
6.6.2 Overall Performance and Energy . . . 113
6.7 Spatial Predictor Tradeoffs . . . 116
6.7.1 Predictor Indexing . . . 116
6.7.2 Predictor Table . . . 116
6.7.3 Spatial Pattern Training . . . 118
6.7.4 Predictor Summary . . . 119

6.8 *Amoeba-Cache* Adaptivity . . . 119
6.9 *Amoeba-Cache* vs other approaches . . . 120
6.10 Multicore Shared Cache . . . 122
6.11 Related Work . . . 123
6.12 Conclusion . . . 124

7 Software Release **125**
7.1 Path Profiling . . . 125
7.2 Path Derived Workload Suite . . . 125
7.3 Needle . . . 126
7.4 Fusion Simulator . . . 127
7.5 Amoeba Simulator . . . 127

8 Future Work and Conclusion **128**
8.1 Concurrent and Future Work . . . 128
8.1.1 Macro Instructions from Sequentially Dependent Operations . . . 128
8.1.2 Eliminating the Load-Store Queue for Specialized Units . . . 129
8.1.3 Software specialization based on dynamic profiling . . . 129
8.1.4 Micro-Workload Generation . . . 131
8.2 Summary of contributions . . . 131

Bibliography **133**

List of Tables

Table 3.1 Workload Characteristics 34
Table 3.2 Path Predictability 38

Table 4.1 Comparison of sequential programs on spatial architectures 48
Table 4.2 Control flow Characteristics 49
Table 4.3 Path Characteristics 56
Table 4.4 Next Path Target Expansion 58
Table 4.5 Braid Characteristics 60
Table 4.6 System parameters 63

Table 5.1 Accelerator Characteristics 75
Table 5.2 System parameters 83
Table 5.3 Accelerator Execution Metrics 84
Table 5.4 Bandwidth in Flits (8bytes/flit) 88
Table 5.5 Inter-AXC forwarded blocks and percentage reduction in energy consumption per component 89
Table 5.6 Virtual memory table look up count 89

Table 6.1 Benchmark Groups 96
Table 6.2 Optimal block size. Metric: $\frac{1}{\mathbf{Miss-rate} \times \mathbf{Bandwidth}}$ 98
Table 6.3 *Amoeba-Cache* Hardware Complexity. 105
Table 6.4 % of direct accesses with fast tags 108
Table 6.5 Avg. # of *Amoeba-Block* / Set 113
Table 6.6 *Amoeba-Cache* Performance. Absolute #s. 118
Table 6.7 Predictor Policy Comparison 120
Table 6.8 Multiprogrammed Workloads on 1M Shared *Amoeba-Cache*% reduction in miss rate and bandwidth. Baseline: Fixed 1M. 123

List of Figures

Figure 1.1 Moore's Law [122] . . . 2
Figure 1.2 Moore's Cost Data [122] . . . 2
Figure 1.3 Transistor Size Scaling [25] . . . 3
Figure 1.4 Transistor Cost Scaling [123] . . . 3
Figure 1.5 Voltage Scaling [44] . . . 4
Figure 1.6 Frequency Scaling [44] . . . 4

Figure 2.1 Amdahl's Projection [6] . . . 12
Figure 2.2 Multicore Scaling [49] . . . 12
Figure 2.3 Intel Tick [7, 147] . . . 12
Figure 2.4 Cost/Transistor stops scaling [149] . . . 12
Figure 2.5 Performance increase more than technology scaling [44] . . . 13

Figure 3.1 Using program analysis to demarcate and extract code paths [15] for accelerators within CPU programs. . . . 25
Figure 3.2 Acyclic paths in a control flow graph . . . 26
Figure 3.3 Path Bias . . . 27
Figure 3.4 Benefits of Path-Based Execution. We have only shown a few workloads due to lack of space. Opcode histogram of paths within a function; % indicates exec coverage. . . . 31
Figure 3.5 Opcode Distribution. The 5 bars for each workload represent the top-5 hot paths (L-R), GEP=pointer access. . . . 32
Figure 3.6 **D1**: Total Ins, **D2**: Guards, **D3**: ϕ's Simplified, **D4**: Total Live Vals, **D5**: Path Predictability **D6**: Path Coverage . . . 39

Figure 4.1 Superblock and Hyperblock construction for overlapped paths. % indicates the relative frequency. . . . 51
Figure 4.2 The distribution of biased branches in the application. Applications not shown in the plot have 99% of the branches with each branch $> 80\%$ bias. . . . 52
Figure 4.3 Fraction of "cold" ops included in Hyperblocks. . . . 53
Figure 4.4 Path Coverage : Path weight (P_{wt}) by rank. . . . 54
Figure 4.5 Braid construction from BL-Paths . . . 59

Figure 4.6 Frame construction from Braid. 62
Figure 4.7 Performance Improvement 63
Figure 4.8 Net Energy Reduction for Braid 66

Figure 5.1 Offloading Sequential Program to Accelerators 69
Figure 5.2 Left: *SCRATCH* Architecture. Per-accelerator scratchpads into which DMA transfers data. Switches to a different accelerator Right: *SHARED*. Shared L1 cache between the accelerators in a tile. The Shared L1 cache is kept coherent with the host multicore through MESI protocol. Host shared L2 maintains inclusion with the accelerators shared L1X. 71
Figure 5.3 Top:*FUSION* Architecture. Bottom: Timeline for image processing example on *FUSION* and *FUSION-Dx*. 77
Figure 5.4 Left: ACC protocol servicing requests from accelerator and interaction with MESI. Right: ACC Protocol servicing forwarded requests from MESI. 80
Figure 5.5 Left: *FUSION* without write forwarding. Right: *FUSION-Dx*. ACC protocol with write forwarding. 82
Figure 5.6 Design tradeoffs in the accelerator cache hierarchy. X-Axis SC: *SCRATCH*, SH: *SHARED*, FU: *FUSION*. Y-Axis: All plots/fusion, lower is better and values are normalized to *SCRATCH* system. Note for the *SHARED* design, the L1X→L0XDATA represents response from shared L1X to AXC and L0X→L1XMSG represents requests from AXC to the shared L1X. For the *SCRATCH* design, there is only one link for data from L2 to the local scratchpad. 86
Figure 5.7 Comparing the benefits of LARGE (L0X:8KB,L1X:256KB) vs SMALL (L0X:4KB,L1X:64KB) 90

Figure 6.1 Cache designs optimizing different memory hierarchy parameters. Arrows indicate the parameters that are targeted and improved compared to a conventional cache. 94
Figure 6.2 Distribution of words touched in a cache block. Avg. utilization is on top. (Config: 64K, 4 way, 64-byte block.) 97
Figure 6.3 Bandwidth vs. Miss Rate. (a),(c),(e): 64K, 4-way L1. (b),(d),(f): 1M, 8-way LLC. Markers on the plot indicate cache block size. Note the different scales for different groups. 99
Figure 6.4 Amoeba-Cache Architecture. 100
Figure 6.5 Lookup Logic 101

Figure 6.6 Partial Miss Handling. Upper: Identify relevant sub-blocks. Useful for other cache controller events as well, e.g., recalls. Lower: Refill of words and insertion. 103
Figure 6.7 Amoeba Cache Controller (L1 level). 104
Figure 6.8 Serial vs Normal mode cache. 107
Figure 6.9 Spatial Predictor invoked on a *Amoeba-Cache* miss 109
Figure 6.10 Fixed vs. Amoeba (Bandwidth and Miss Rate). Note the different scale for different application groups. 112
Figure 6.11 Distribution of cache line granularities in the 64K L1 and 1M L2 Amoeba-Cache. Avg. utilization is on top. 114
Figure 6.12 % improvement in performance and % reduction in on-chip memory hierarchy energy. Higher is better. Y-axis terminated to illustrate bars clearly. Baseline: Fixed, 64K L1, 1M L2. 115
Figure 6.13 Spatial Predictor Performance Comparison 117
Figure 6.14 Effect of increase in block size from 64 to 128 bytes in a 1 MB cache 120
Figure 6.15 Relative miss rate and bandwidth for different caches. Baseline (1,1) is the Fixed-2x design. Labels: • Fixed-2x, ∘ Sector approaches. ∗: Multi$, △ Amoeba. (a),(b) 64K cache (c),(d) 1M cache. Note the different Y-axis scale for each group. 122

Figure 8.1 % Reduction in IR Instructions 130

Chapter 1

Introduction

Since the invention of the microprocessor, it is unlikely that the role of a computer architect has ever been as important as it is now. The breakdown of Dennard's scaling led to the multi-core revolution [57]. Mark Bohr, Intel Fellow and recipient of the IEEE Jun-ichi Nishizawa Medal [82], reflecting on the breakdown of Dennard scaling [25] stated –

> *... ours is a very inventive industry and new transistor technologies such as strained silicon, high-dielectrics, metal gates and multiple-gate devices have been or will be introduced to continue scaling. So although the letter of Dennard's Law can no longer be followed, it has gotten us very far over the past 30 years and the spirit is alive and well in transistor R&D facilities around the world.*

As of 2016, the breakdown of Moore's Law seems inevitable [4]. This marks the second time computer architects are called upon to devise novel technologies that deliver performance improvements and reduced energy consumption. Arguably, it is even more challenging than the crisis faced at the onset of the 21st century.

1.1 The breakdown of technology scaling

Gordon Moore's seminal work in 1965 defined the law that governed semiconductor cost scaling for the next five decades. Primarily interested in shrinking transistor costs, Moore formulated the law based on empirical data presented in his paper. The plot which extrapolates the exponential transistor count increase as a function of time is reproduced in Figure 1.1. Based on the observations from 1959 to 1965, he predicted a doubling of the transistor count every $\simeq 2$ years [122]. It is an impressive achievement of the semiconductor industry to have borne out five decades of scaling in accordance to Moore's law. The increased transistor count in each generation has been the primary driving force for increased computational efficiency.

Furthermore Moore describes cost scaling as shown in Figure 1.2. Moore stated –

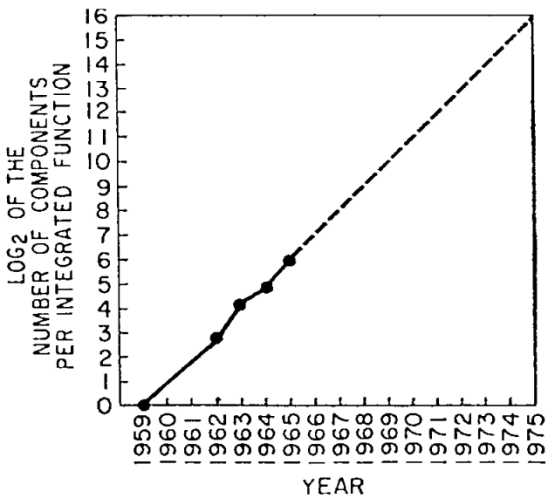


Figure 1.1: Moore's Law [122]

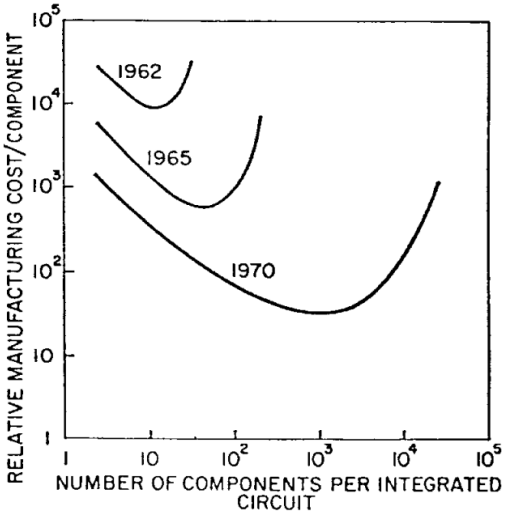


Figure 1.2: Moore's Cost Data [122]

> *"For simple circuits, the cost per component is nearly inversely proportional to the number of components, the result of the equivalent piece of semiconductor in the equivalent package containing more components. But as components are added, decreased yields more than compensate for the increased complexity, tending to raise the cost per component. Thus there is a minimum cost at any given time in the evolution of the technology."* [122]

Moore observed that the overall cost is dependent on two factors: the density of transistors on a single chip and the reliability of the fabrication process. The reliability is also known as the *yield rate*. Increasing the number of transistors for a given area makes them cheaper; however it increases the chance of defects. The presence of fabrication process induced defects on wafers reduces yield rate and drives up cost. This net effect results in an optimal number of components per integrated circuit, as shown in Figure 1.2. To maintain fabrication cost scaling of microelectronics as described by Moore, two sizes need to be considered a) the wafer size and b) the transistor size. A single wafer contains multiple *dies*, small block of semiconducting material on which a functional circuit is etched. As of 2016, the current industry standard wafer is 300mm in diameter. A simple estimation of the number of processing dies per wafer for the Haswell 4 core configuration at 22nm is $\simeq$ 400 dies (not accounting for fabrication defects).

Larger wafers are preferred, as a fixed number of steps are required to produce a single wafer, and cost is amortized. Additionally, defects are often found on the edges which also favours larger wafers to improve yield. There have been efforts to move to a 450mm size wafer, and estimated use in volume production is projected to start in 2020.

Figures 1.3 and 1.4 show the scaling of transistors in terms of a) size and b) cost from the year 1970 onwards. We can see that transistor sizes have scaled linearly on the log scale

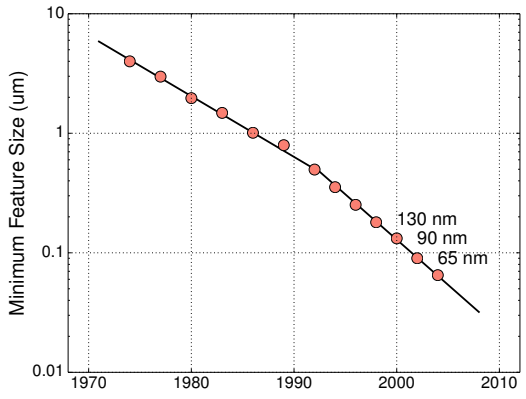


Figure 1.3: Transistor Size Scaling [25]

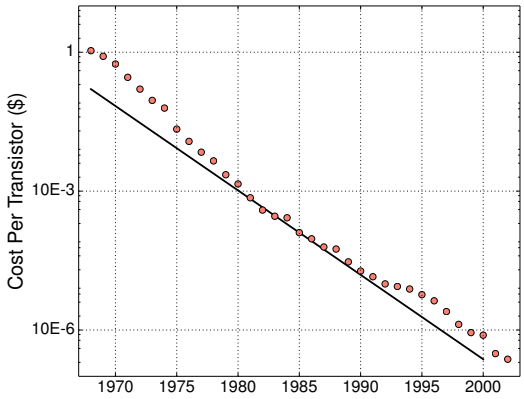


Figure 1.4: Transistor Cost Scaling [123]

as shown in Figure 1.3. Similarly, the number of transistors has doubled every two years in this time frame. The cost has decreased from 1 transistor for $1 in 1970 to 10^6 transistors for $1 in 2000 due to advances in VLSI technology [123].

Dennard Scaling: Moore's scaling for cost motivates the need to make smaller transistors but it is Robert Dennard who outlined the implications of semiconductor scaling in 1974 [46]. In his work, Dennard showed that as transistors get smaller, they switch faster and use less power. Thus power density remains constant in spite of increased transistor density. This implies that smaller chips consume less power whereas similar sized chips could run faster. Dennard's scaling has set the roadmap for the semiconductor industry for each generation of process technology, with a concrete transistor scaling formula to move each generation forward. From the early 70's with the invention of the microprocessor till the early 2000's, improvements in computational efficiency were primarily afforded by Dennard Scaling [46]. With Dennard scaling, the computing industry reaped the benefits of improved silicon fabrication methodologies year on year. A 1000× improvement in performance has been realized since 1970. The scaling described a 30% reduction in transistor size every 18 months. Their area shrinks by 50% thus doubling transistor density. A commensurate 30% reduction in supply voltage is also necessary to maintain reliability. Thus transistor density doubles while reducing delay by 30%. The power consumption is reduced by 50% and energy by 65% [26]. Additionally, the larger number of available transistors contributed to micro-architectural innovations.

In the early 2000s the industry found that the current scaling trends showed increased power consumption inconsistent with the trend forecast by Dennard. While Dennard scaling projections for transistor feature sizes held, the chip supply voltage did not. This in turn made it difficult to provide economically viable cooling solutions to new power hungry chips. Figure 1.5 shows the how supply voltage scales with respect to transistor feature size. On reducing the feature size from 0.13 µm to 0.03 µm there has been little change in the supply

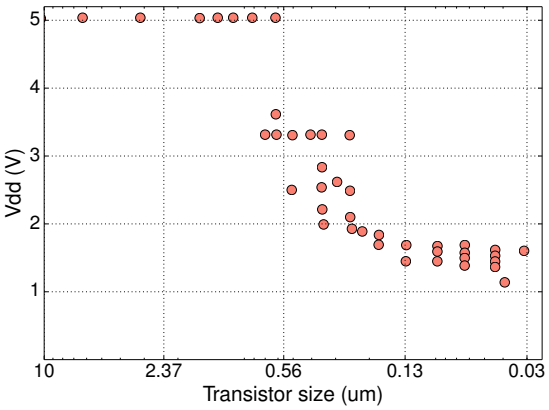


Figure 1.5: Voltage Scaling [44]

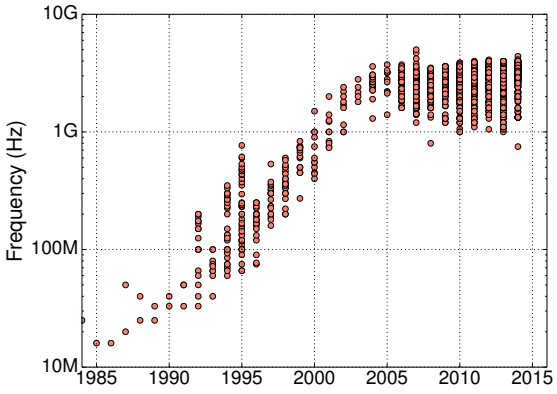


Figure 1.6: Frequency Scaling [44]

voltage. This manifests as a lack of threshold voltage (minimum gate-to-source voltage differential) scaling. Thus a significant amount of sub-threshold leakage current counts against the overall consumed power. The impact of this issue is reflected in Figure 1.6. The frequency of microprocessors reached a plateau in the early 2000s. In fact, even 15 years later, consumer Intel microprocessors are clocked in the 2.4-3.4 GHz range similar to their Pentium 4 microprocessor released in 2004 (Prescott).

Smarter utilization of silicon is imperative with the growing menace of the *Utilization Wall* [171]. The utilization wall is a consequence of CMOS scaling theory and current-day technology constraints, assuming fixed power and chip area. In 2010, researchers showed that less than 7% of the chip can be utilized in order to maintain power consumption below 80W. Should the entire chip be use, using the 45nm TSMC process, a staggering 1225W of power is consumed! The consequence of the utilization wall is *dark silicon*, silicon that cannot be powered-on at the operating voltage for a given thermal design power (TDP) constraint.

Implications of technology scaling breakdown: Dark silicon is not a new phenomenon. Rather the rapid increase, 2× per process generation, is the cause for concern. There is often circuitry available on the chip which is not used by all workloads. A simple example is the presence of hardware floating point units which are not used by the linux kernel [109]. To mitigate dark silicon, four approaches have emerged as potential candidates [166]. The first approach is to shrink the size of the chip itself. Compromising on performance to decrease the size of the chip is a non sequitur for chipmakers. The second approach is to wait for a breakthrough in semiconductor devices. Two candidates are Tunnel Field Effect Transistors (TFETS) [84], and Nano-Electro-Mechanical switches [41, 35]. Both of them hint at orders-of-magnitude improvements in leakage power consumption, but remain to be realized as economic alternatives. The third approach is increased usage of *dim silicon*, general purpose logic which is underclocked or used infrequently to meet the

constrained power budget. This introduces heterogeneity in the chip and has been explored by chipmakers. For example, NVidia used a "low power companion core" in the Tegra architecture [168]. While this approach counters the utilization wall, it is not a measure targeted at increasing performance. The final approach is the usage of *specialized* circuitry. This approach can provide much faster performance, much lower energy consumption or even both [171, 70]. The dark silicon is used to implement specialized co-processors. Execution migrates to a particular co-processor while power and clock gating the primary core. As of 2016, an increasing number of chips are dominated by specialized co-processors [66].

With the migration of execution across the chip, data movement becomes an additional cause for concern in specialized architectures. Using dedicated wires have long been deprecated in favour of on-chip networks [148, 43]. Published research by Intel in 2004 showed up to 50% of the dynamic power consumption in their microprocessor can be attributed to the network-on-chip (NOC) [112] The 80-core Intel TERAFLOPs chip expends 39% of the dynamic power per node in the NOC [75]. In the current generation, often the energy cost of data movement is as high as the computation itself leading to significant overheads in dynamic power consumption [11, 42]. In effect, wires have not scaled in power consumption with respect to the rest of the system.

To summarize, the implications of the breakdown of technology scaling are

1. **Favouring *specialization* to increase performance/watt**
   Hardware specialization is the use of circuitry or mechanisms which target a particular pattern in a workload. With knowledge of the pattern, a hardware specialized unit can provide faster performance and/or energy efficient execution. For example, vector units (SIMD) optimize for data parallel code segments within a program. With the transistors afforded by Moore's law scaling, multiple patterns can be targeted by different specialized units. We see this trend in modern, low power processors where more than 50% of the chip area is dedicated to specialized co-processors [66].
2. **Optimizing for data movement is necessary**:
   Von-Neumann architectures fetch data from memory to perform computation. However, due to deep memory hierarchies and increased efficiency for compute operations, a significant part of the dynamic energy consumption within a chip is due to the movement of data. Eliminating waste or unused data can reduce overall energy consumption, particularly in the memory hierarchy. With the use of specialized compute units, the problem is aggravated. Compute units consume less energy while data movement is increased as the execution of the workload migrates across the chip to utilize varied specialized hardware units.

1.2 Challenges

The use of specialized co-processors introduces three significant challenges.

What to specialize? Figuring out the target for specialization within a large workload can be a daunting task. Computer architects often use dynamic profilers to identify important functions. Profilers such as `gprof` [65] can identify the fraction of execution time consumed by each function. However, their insights are tied to a particular architecture and yield no information about the amenability for specialization.

Domain experts can be consulted to provide insight to the engineers as to which segments of an algorithm are amenable to specialization. This implies a large investment in time and money for the industry. Researchers have proposed the use of standardized kernels to study specialization [142]. However, it relies on manual culling of kernels and may not be representative of complicated real work workloads. Prior work focused on specialized co-processors [64, 127, 39, 68] have obliquely addressed this problem. Unfortunately, the regions of interest selected by the authors for specialization is constrained by their accelerator micro-architecture. Considerations regarding the amenability and profitability of selected regions further complicate the issue.

How to specialize? Often there is no clear distinction in prior works between what to specialize and how to specialize. For example, the DySER [64] and CCA [39] architectures target computation only in their specialized regions. Thus their choice of *what* to specialize is constrained by their specialized architecture, i.e. *how* to specialize. There is a lack of research into how a target region should be specialized. This is exacerbated by the focus on irregular, general purpose applications with the focus on specialization to improve performance and energy efficiency.

High level synthesis tools such as LegUp [32], Bambu [134] and Vivado [52] require programmer annotations at the function level to identify the region to be specialized. Additional hints may be inferred by the usage of the function. Parallel invocation of the specialized unit can be derived from the use of the identified function in the `pthread` [126] (POSIX thread) programming model.

Integration Introducing heterogeneity in the system introduces additional challenges. In this thesis we study the challenge of data movement introduced by the integration of specialized units. Traditional methods of transferring data to specialized co-processors employ direct memory access (DMA). Initializing such transfers introduce delays. The performance obtained from the specialized unit must amortize the setup costs. Newer architectures such as the Intel HARP, Xilinx Zynq and Altera Cyclone, allow closer integration of reconfigurable accelerators. They add FPGAs on the same chip which have coherent access to the cache

memory hierarchy. Close integration allows for the use of "pull" based coherence protocols as opposed to "push" based DMA transfers.

1.3 Approach

This dissertation adopts generalized methods to address the challenges of application specific hardware specialization. In this section we introduce terminology and techniques used in this thesis. We discuss how the work presented in this dissertation leverages existing techniques and builds upon them.

We use program analysis driven workload characterization
Program analysis can be defined as –

> *. . . the process of automatically analyzing the behavior of computer programs regarding a property such as correctness, robustness, safety and liveness.* [179]

Software engineering disciplines have established varied program analysis techniques to address primarily issues related to program optimization or program correctness. We use program analysis to identify frequently executed code segments for further characterization. The workloads we study are programs which are representative of a group programs. They serve as benchmarks for computing systems and are crucial for performance engineering. Herein we describe our approach and rationale.

Program analysis can be *static* or *dynamic* in nature. Static program analysis is performed without executing the program, for example at compile time. Dynamic analysis is performed at runtime. We implement *efficient path profiling* [15, 13]. A path can be loosely defined as an acyclic sequence of basic blocks. A basic block is a sequence of instructions terminated by a branch. A more precise description is provided in section 3.2.1. In the work presented in chapter 3 we adopt a hybrid approach. We use *dynamic* path profiling to identify the "hot" paths in the program. We use *static* reconstruction of the hot paths for characterization. The characterization serves as a first order metric for assessing the amenability of the regions with respect to specialization.

The rationale behind our approach is the need for scalability. Prior works such as [151, 76] have adopted fine-grained instruction granularity profiling to characterize workloads. We find that these approaches impose a high runtime overhead. In contrast, our approach imposes a 30-40% overhead at runtime with no loss of control flow information. The low overhead and granularity of abstraction allows our approach to be flexible. The bane of dynamic analysis is input dependence. With dynamic analyses we can derive properties of programs observed for a particular input. Dynamic analysis cannot prove that a particular program satisfies a certain property [14]. However, using our approach we can mitigate the impact of input dependence by aggregating profiles collected for different inputs. We

study the program behavior across a set of representative inputs rather than a single one. The static reconstruction for characterization is fast, on the order of 10s of seconds for our largest workloads.

We target specialized units, i.e. "accelerators" With the breakdown of technology scaling, computer architects have resorted to the use of specialized units, i.e. accelerators to provide increased performance and energy efficiency. An accelerator is any unit tailored to optimize execution for a particular program pattern. For example, vector extensions (SIMD) to the x86 ISA serve to accelerate data parallel program segments. Thus accelerators can be broadly defined as any specialized circuitry which targets particular program characteristics. SIMD and floating point units can be classified as programmable accelerators. The research presented in chapters 3 and 3 focuses on reconfigurable accelerators. Reconfigurable accelerators can be Coarse Grain Reconfigurable Arrays [45], or Field Programmable Gate Arrays [28]. Reconfigurable accelerators that stem from academia such as DySER [64], CCA [39] and BERET [68] can also leverage our work.

We define program abstractions for accelerators Program abstractions are essential for scalable techniques be developed. For example, a function is an abstraction a programmer may use to group code which serves a singular purpose. Similarly, abstractions such as the control flow graph and program dependence graph [54] offer representations on which analyses can be performed. In this thesis (chapter 4) we describe a program abstraction, ("Braid", Section 4.4.2) to target for specialization. This abstraction builds upon program paths. It merges paths which start and end at the same basic block. Paths which contain features that are not supported on accelerators are left to execute on the host. We use a software based speculative execution model as the basis for the abstraction.

Speculative execution is a technique by which the microprocessor executes instructions past a branch *without* knowing the outcome of the branch. Effective hardware speculation is enabled by accurate branch predictors. Paths are composed of basic blocks spanning across several branches at a time. Executing a path at a time speculates on all the branches in the path. Merging paths to construct "Braids" speculates on branches which are never taken for the profiling input. Hardware speculation includes support for rolling back changes to program state if the wrong path is executed. We construct software "frames" that include instrumentation for checkpointing state. Our implementation of software speculation eliminates branches, reduces control flow complexity and provides software support for rollback in case of wrong path execution. More details are provided in section 4.5.

We propose coherence protocols for accelerators Coherence protocols allow shared memory multiprocessors to have a synchronized view of memory. Updates issued from a processor are propagated to all processors which access the same datum. Coherence

protocols enforce the contract established by the consistency model [162]. Data movement in accelerators have traditionally used non-coherent interfaces such as direct memory access (DMA). These require manual effort for the programmer to restructure the code so that data can be shipped to the accelerator for processing and then back again. This type of interface also has a high overhead; thus the accelerator must perform a significant amount of cost to amortize the cost of data transfer. This type of model is also referred to as a "push" based model. We design coherence protocols for accelerators, which allow for a "pull" based model. Accelerators are allowed to request for and cache data exclusively in our proposed protocol. We mitigate the overheads of data movement by using a specialized coherence protocol for accelerators.

We describe mechanisms for adaptive granularity caching Caching mechanisms in modern processors save data at a fixed granularity. Modern consumer microprocessor's cache data at the 64B or 32B granularity [44]. However, the design decisions for cache line sizes do not take into account the memory access behavior of modern workloads. Prior researchers [141, 150] have looked at caches which support block sizes of varied size. Our approach detailed in chapter 6 allows for variable sized cache blocks in multiples of 8B in size.

1.4 Dissertation Organisation

With the breakdown of technology scaling, computer architects have resorted to specialization as a means to provide the scaling in performance and energy benefits consumers have experienced since the invention of the microprocessor. The overarching theme of this dissertation addresses the challenges of application specific hardware specialization. The underlying motifs of the work presented in this dissertation are automation and generality. The tools and techniques are application agnostic and operate via automated compiler analyses or via transparent hardware mechanisms.

Chapter 2 discusses the current challenges faced by computer architects with the breakdown of transistor scaling. It puts the new challenge of harnessing specialized units in contrast to the dawn of the multi-core era in computing. Section 2.2 enumerates the challenges of what to specialize, how to specialize and integration issues. Section 2.3 lists the contributions made and finally, section 2.4 discusses the relationship of thesis content to prior peer reviewed published work.

Chapter 3 describes our work on extracting accelerator benchmarks from microprocessor benchmarks. We show the scalability and precision of analysis at the granularity of program paths. We present our characterization of workloads drawn from established microprocessor benchmark suites. In this chapter we present our approach to address the challenge of *what to specialize*. Section 3.2.1 provides a primer on path profiling and motivates its use

as an abstraction for characterization with the end goal of specialization. We study of 29 workloads drawn from prevalent microprocessor benchmark suites. Sections 3.3 and 3.4 present the data collected and discuss the implications on specialization.

Chapter 4 builds upon the path based characterization to demonstrate the feasibility of path based specialization in terms of performance and energy efficiency. Building upon paths, we describe a new abstraction for hardware specialization called "Braids". We evaluate and discuss the opportunities of leveraging Braids for hardware specialization. In this chapter we present our approach to address the challenge of *how to specialize*. Section 4.2 discusses related work showing how heuristic based approaches tailored for VLIW processors may fail when reused as specialization targets. Section 4.4.2 presents our methodology for the construction of "Braids". We describe the construction of speculative software frames in Section 4.5. We model the performance and energy implications of specialization for the abstractions described. We present our results in section 4.6.

Chapters 5 and 6 tackle the challenge of *integration*. We study issues with respect to data movement. We discuss coherence protocols tailored for specialized hardware. They allow for low overhead data movement as execution of the workload migrates across the host and the accelerators. Section 5.3 describes the hybrid coherence protocol. We develop a cycle accurate simulator to study the impact of using time stamp based coherence within the accelerator domain. Section 5.5 presents our results showing the reduced energy consumption for data movement in the workloads we study. We also study the utilization of cache lines based on workload behaviour. We look at the amount of data accessed by the processor while the line is cached. We describe an adaptive granularity cache memory hierarchy in section 6.3. We evaluate two dynamic predictors and present the results we observe across a diverse set of workloads including desktop applications such as Firefox. The results are presented in section 6.6.

Chapter 7 outlines the software developed to conduct the research presented herein. Finally, chapter 8 presents concluding thoughts and directions for future work.

Chapter 2

Background

In 2006, the industry shifted focus to shared memory multiprocessing as a means to scale computational throughput, marking the end of Dennard scaling. Shared memory multiprocessing was proposed in academia as early as 1996 [71] and well studied. Figure 2.1 shows the maximum possible speedup given what fraction of the program can be executed in parallel. The curves represent the parallel fraction and even the largest depicted in the graph (95% parallel) plateau after 256 cores. A more realistic 50–75% parallel fraction benefits very little overall with more than 16 cores. The shift to a parallel paradigm for consumer applications brought along with it a whole host of issues. Developers long accustomed to reasoning about sequential execution now have to deal with more complex multi-threaded code to extract performance from the hardware provided. In work by Blake et al. [23] available thread level parallelism in desktop applications was studied. Their exhaustive analysis across a wide range of applications such as gaming, office, web browsing and more showed little available thread level parallelism (TLP) that can be exploited for improved performance within a workload. For instance they found that for office, gaming and web browsing the average TLP was 1.2, 1.6 and 2 respectively. Only video processing applications showed the need for many cores with an average TLP of 7.4.

High Performance Computing (HPC) applications are often data parallel problems. In the effort to extract performance, GPUs have been repurposed to accelerate HPC workloads. This paradigm represents the extreme end of the TLP spectrum using many parallel yet simple processors that maintain a *Thermal Dissipation Power* close to a general purpose core. Their use however is via tailored programming models such as CUDA or OpenCL, thus requiring a rewrite of existing programs to take advantage of their processing power.

Figure 2.2 shows the stark contrast between Moore's law scaling and actual multi-core scaling as overall application speedup. While multi-core scaling has utilized the transistors afforded by Moore's law, it is infeasible to sustain with transistor feature sizes reaching single digits. At that nanometer scale, transistor sizes are comparable to a few atoms. Until late 2016, Intel followed the "Tick-Tock", two stage model of micro-architecture updates. A

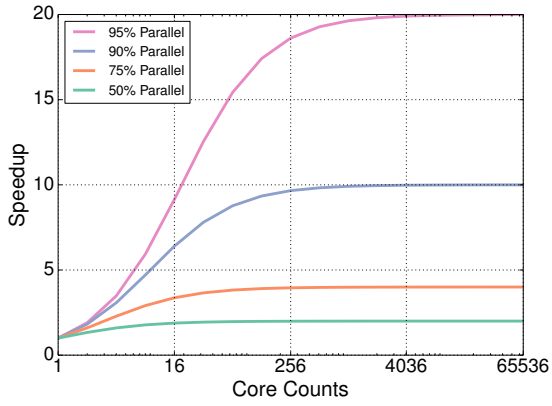


Figure 2.1: Amdahl's Projection [6]

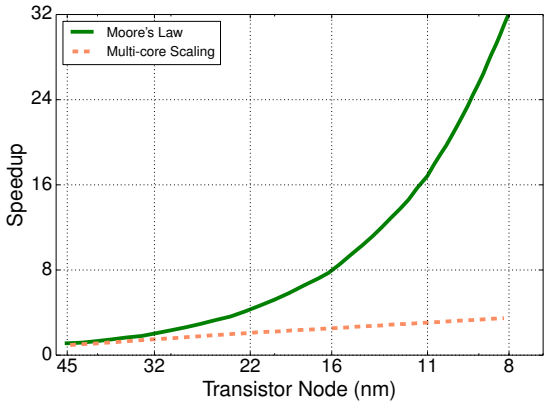


Figure 2.2: Multicore Scaling [49]

"Tick" in the model, represented a shift to a smaller process technology whereas a "Tock" represented a change in the micro-architecture design. Figure 2.3 shows the fabrication process used by Intel over the years and their projected usage. Reliability and yield at such small feature sizes has forced Intel to abandon the two stage "Tick-Tock" model in 2016. Going forward, Intel has announced the introduction of a second "Tock" which focuses on the optimization of the micro-architecture introduced previously.

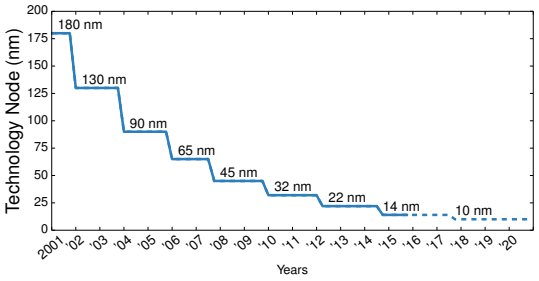


Figure 2.3: Intel Tick [7, 147]

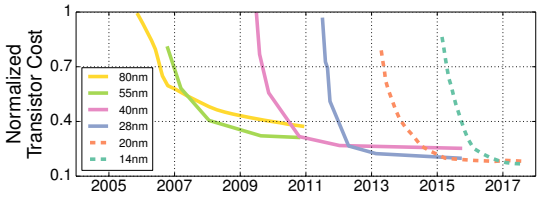


Figure 2.4: Cost/Transistor stops scaling [149]

The semiconductor industry has largely been affected by the increase in costs of research required for next generation semiconducting *fabs* (fabrication facilities). For example, FinFET technology allows for smaller feature sizes but requires more fabrication steps and supporting tools. This has forced many semiconductor companies to become "fab-less", i.e. they only provide the design and rely on other manufacturers to realize the design at a desired technology node. Larger wafer size can lower fabrication costs by increasing the number of dies per wafer and providing better yields. However, as mentioned previously, wafer sizes are not expected to change until 2020 due to the significant investment required to upgrade fab lines.

Figure 2.4 (data from NVidia), shows the cost of transistors for different technology nodes. It shows limited cost benefit after scaling past the 28nm technology node (curves normalized to 28nm). Smaller feature sizes will no longer provide an economic benefit for

fab-less semiconductor companies. This heralds the end of economic scaling and thus the breakdown of Moore's Law.

Scaling via specialization: With the imminent collapse of Moore's Law scaling, the semiconductor industry has begun efforts to compensate. Without either Dennard or Moore's scaling, there is a risk that the industry will stagnate. While a driving force for consolidation, i.e fewer companies can afford fabs, it also leads to increased research in technologies which offer more scaling at the micro-architecture level. It is expected that companies will increasingly adopt solutions which provide performance by the use of innovative hardware architectures. One such example is the success of the GPGPU computation. The massively parallel processing power of the GPU can only be harnessed by a specific programming model coupled with the target architecture. Furthermore, application and domain-specific accelerators become more attractive as it is able to provide the performance improvement no longer provided by device technology scaling, i.e increase in transistor density.

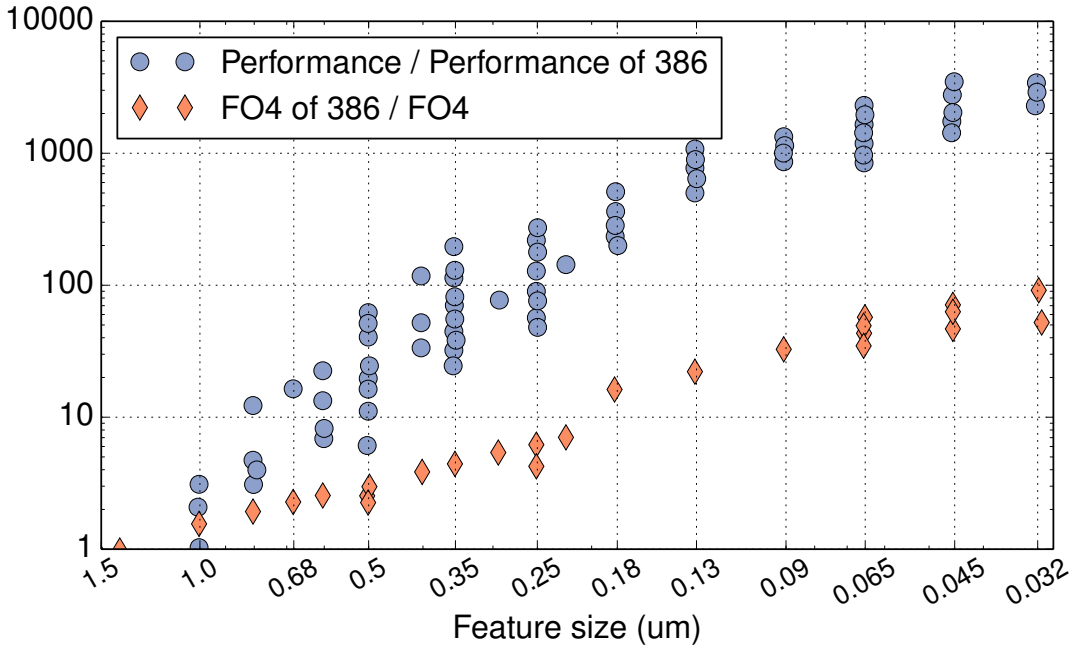


Figure 2.5: Performance increase more than technology scaling [44]

Figure 2.5 shows the normalized performance of microprocessors since Intel's 80386. Blue circles indicate normalized overall performance while the orange diamonds indicate the FO4 delay. The FO4 delay (process-dependent delay metric used in digital CMOS technologies) serves as a proxy for Dennard's transistor performance scaling. From the trends indicated on the graph, we see that the actual performance improvement outstrips the predicted performance from Dennard's scaling law. A significant amount of performance improvement is afforded by micro-architecture innovations, also known as architectural scaling. Computer architects have put the extra transistors afforded by Moore's law to use

in the design of subsystems such as superscalar and out-of-order (OOO) execution, cache memory hierarchies and specialized units such as vector extensions. These improvements target different characteristics of modern workloads such as the presence of instruction level parallelism (ILP) using superscalar and OOO execution. Data level parallelism is targeted using Single Instruction Multiple Data (SIMD) units, which introduce vector extensions to the ISA which the compiler or programmer can use while implementing their algorithms. These are fundamental advances which have provided performance increase over and above device scaling. In the future, computer architects may have to work with the assumption that no performance improvements will be from device scaling, and it is architectural scaling which carries the burden. As the blue squares show how processor performance scaled over time, the orange diamonds indicate how much of it came from device scaling. This graph is derived from [44]. Overall, the figure indicates the need for an order of magnitude improvement to be derived from architectural scaling due to the loss of device scaling.

2.1 Application Specific Hardware Specialization

Hardware accelerators are increasingly being adopted to mitigate the loss of device scaling. The term accelerator is loosely defined and alludes to *any* specialized architecture which provides a performance and/or energy efficiency benefit when compared to execution on a general purpose CPU. For specialized circuitry, often orders of magnitude greater performance can be obtained as indicated by research presented in [185, 70]. Compared to general-purpose processors, customized processors like DSPs deliver from 10× to 100× more energy efficiency, while dedicated application-specific accelerators (ASICs) are 1000× more energy efficient [185].

Customized architectures composed of CPUs, GPUs, and accelerators are already seen in mobile systems and are beginning to emerge in servers and desktops. Analysis of die photos from three generations of Apple's SoCs: A6 (iPhone 5), A7 (iPhone 5S), and A8 (iPhone 6), shows that *more than half of the die area* is dedicated to non-CPU, non-GPU blocks Most of these blocks are application-specific hardware accelerators. Customized hardware accelerators, implemented as application specific integrated circuits (ASICs) efficiently perform key kernel computations within larger applications. However, flexibility is sacrificed. After fabrication, an ASIC's custom datapath cannot be modified to meet new requirements. ASICs also have high non-recurring engineering (NRE) costs associated with manufacturing and long development cycles.

Heterogeneous systems combine general purpose processing cores with diverse characteristics. Architectures such as the ARM big.LITTLE target energy efficient execution based on workload characteristics. They target sequential sections of code with large OOO cores which can extract finer grain ILP. Many small cores are used for parallel portions of the workload as indicated by the programmer. Additionally, GPGPU accelerators have become

ubiquitous. Modern artificial intelligence methods have become increasingly reliant on neural architectures. Neural architectures are data parallel with a large amount of floating point computations. NVidia has made great strides in tailored GPGPU accelerators for neural networks.

A radical approach to the heterogeneous system model is the integration of reconfigurable FPGAs. IBM has developed CAPI (Coherent Accelerator Peripheral Interface) for the Power8 architecture. This allows for the addition of reconfigurable FPGA based accelerators over PCIe with ease. The CAPI model allows for shared virtual memory between the accelerator and the host without complicated DMA transfers. Intel has announced plans to release a Xeon server with an FPGA on chip (HARP). The Intel HARP project arises from the Altera's acquisition by Intel. It allows for even tighter integration of accelerators with the CPU. The HARP architecture utilizes QPI (Intel Quick Path Interconnect) to access the FPGA. The FPGA accelerator is also provided coherent access to memory. Similar SoCs have been previously developed by Altera and Xilinx which couple ARM cores with FPGAs on a coherent interconnect via the ACP (Accelerator Coherence Port).

While academia has explored coarse grain reconfigurable architectures (CGRAs) practical reconfigurable accelerators available today are primarily FPGAs. FPGAs allow engineers to design custom circuitry after manufacturing. FPGAs are configured using a hardware description language such as Verilog. They contain arrays of programmable logic blocks and interconnects to route data. Historically, FPGAs were primarily used in telecommunications and networking. In recent times, they are found everywhere from the automobiles to mobile phones. The primary benefit of a hardware accelerator is the elimination of core frontend costs incurred in a traditional OOO processor. Research indicates that the more than 40% of the power consumption in an OOO frontend is due to instruction fetch and control, considering pipeline registers as well this can be as high as 60% [70]. The energy consumed in the execution per operation is also generally lower than that of an OOO core. Furthermore there are many optimizations available such as the bitwidth reduction and fusion of operations when targeting an FPGA.

2.2 Challenges

With the flexibility afforded by reconfigurable FPGA accelerators come a host of challenges which need to be addressed to extract performance while retaining the energy benefits of specialized computation. In this section we enumerate and outline the challenges of harnessing hardware accelerators.

2.2.1 Challenge 1: What to specialize?

The foremost challenge is *what should developers target for specialization?* For the most part engineers have relied on domain experts to indicate what parts of an algorithm are

beneficial and amenable. This has been the de facto approach where a large non-recurring investment is made in terms of man-hours to understand and rewrite the algorithm to target a particular FPGA accelerator. Often such an effort will consist of multi-person teams over a project which lasts a few months. However, with the pressing need for more workloads to be accelerated, few companies can afford such a long turn around time on accelerator synthesis. An additional concern is the rate of evolution of the algorithm itself. This particular concern was discussed at length when Microsoft offloaded part of their machine learning for Bing search to FPGAs (Catapult [137]). Rapid evolution of the algorithm meant that the hardware design often lagged behind the software implementation. The inherent challenge is to eliminate the reliance on expert guidance while building accelerators. To address the question of what to specialize, two aspects must be taken into consideration.

a) Amenability – FPGA accelerators are not yet first class citizens in the SoC. There are many operations with well defined semantics on CPUs which are undefined on FPGAs. A simple example is the usage of exceptions. While software has well defined interfaces to halt and deal with exceptions, such support is not present on the FPGA. Furthermore, whether such a construct is feasible to implement on an FPGA is questionable. There are more examples such as dynamic memory allocation, runtime shared library invocation, etc which software developers take for granted but are difficult to mimic on an FPGA. Thus existing implementations of algorithms cannot be blindly used as templates to generate FPGA accelerators. The standard approach to understand what parts of the algorithm are amenable is to refer to an expert. Engineers often use statistical profilers (eg. *gprof*) to understand which parts of a program consume the largest amount of time. With this information they can then dissect the program to separate out the parts which are amenable from those that are not. For automated analyses, the key to assessing amenability is the choice of abstraction. Having a robust abstraction which represents the workload at a fine granularity lends scalability to automated analyses.

b) Profitability – The second and arguably more critical aspect of what to accelerate is to assess the profitability of offloading computation to a target accelerator. There are inherent overheads to any heterogeneous execution model when execution migrates from one substrate to another. For example, the overheads of kickstarting computation on an FPGA need to be understood in order to assess profitability. Often, benefits of fine grained acceleration can be nullified by associated overheads. The overheads themselves can be categorised as *system dependent* and *workload dependent*. System dependent overheads are those which are inherent to the offload model adopted by the SoC. For example, IBM CAPI simplifies the address translation and memory interface but latency of each access is in the order of 1000s of CPU cycles due to the PCIe interconnect. SoCs from Altera and Xilinx do not offer address translation but do have coherent access to the last level of cache in the

order of 20-50 cycles. Finally the key to understanding the workload overheads is the use of a proper abstraction. Given an abstraction, automated analyses can model the potential gain from offloading work to the accelerator.

2.2.2 Challenge 2: How to specialize?

It is imperative that we clearly distinguish the *how* from the *what*. It is easy to redefine *what* based on *how* a particular algorithm is accelerated. To demonstrate this potential pitfall, consider the following example. If a high level synthesis tool does not support dynamic memory allocation, it may discard any program region which has such operations. Thus the inability of an automated tool to target operations to run on an FPGA may determine what should be targeted at a coarser granularity. Once again the selection of a proper abstraction is paramount to achieving a clean partitioning of program regions to offload to a reconfigurable accelerator.

a) Compiler Analysis – There is a paucity of research on appropriate abstractions that allow for flexible representations of work to be offloaded to a target accelerator. Prior work such as Spatial Computation [29] and BERET [68] have repurposed Hyperblocks [114] and Superblocks [117] respectively. Both these abstractions were originally designed for VLIW processors. Often these heuristic based abstractions yield poor results. See Section 4.2.2 for more details. Meanwhile standard compiler research has made great progress. There are many robust building blocks available to the developer to work with. Yet, there are no techniques that target accelerators which take advantage of recent advances in compiler techniques and program analysis. Another potential application of compiler analyses is the detection of coarse grain parallelism. All the tools available rely on programmer annotations such as OpenMP pragmas or pthread invocations. There is a rich body of auto parallelization compiler analyses [57, 86, 103] which can be leveraged for the detection of coarse grain parallelism.

b) High Level Synthesis – This has been an open problem for more than 20 years. Xilinx supports high level synthesis, i.e C/C++ to Verilog, using compiler technology acquired from Autopilot. Their current generation product is built on the LLVM compiler infrastructure and supports programmer annotations to ascertain target functions to be synthesised in an FPGA SoC use case. There are also some notable open source projects available such as Bambu and LegUp. Bambu [134] is based on gcc whereas LegUp [32] is based on LLVM. These tools represent the cutting edge in terms of flexibility in high level synthesis from academia and industry. Yet they are unable to deal with simple workloads derived from benchmark suites such as SPEC2006. This is primarily due to the presence of "unacceleratable" features as determined by the tool. Altera has adopted an alternate view with the use of the OpenCL programming model. The OpenCL programming model

is originally intended for GPGPU applications. They allow for the specification of large amounts of data parallel work in the form of kernels. These kernels are allowed to perform only a restricted set of operations. This ensures that the kernels can be mapped onto the FPGA accelerator with ease. The data parallel and simple nature of the work offloaded makes it easy to precompute the data which needs to be fetched from memory. Having a parameterizable model, Altera is able to target different FPGA accelerator targets. While the OpenCL model generalizes well, it is unclear as to what benefits an FPGA provides over a GPGPU. The key segment where FPGAs excel, workloads with large heterogeneous parallelism, cannot be expressed in the OpenCL model.

2.2.3 Challenge 3: Integration

A third significant challenge for FPGA hardware accelerators is system integration. As alluded to previously in this section, vendors have built systems which have different tradeoffs. IBM has deployed CAPI with the Power8 architecture. This allows for PCIe based accelerators to be easily programmed using a unified shared memory model between the host and the FPGA. The system however incurs the latency of the PCIe bus for accesses which are routed to host memory. Therefore, while they are able to support larger FPGA chips with more onboard DRAM (1 GB) than on-chip FPGA solutions, they require careful programming to extract performance. The Altera Cyclone V SoC and Xilinx Zynq platforms couple ARM host cores with on-chip (SoC) FPGAs. They offer smaller FPGAs with lower onboard DRAM (64-256MB) but fast access to host memory. They do not however offer address translation hardware. The Intel HARP system couples Xeon cores with an Altera Arria 10 FPGA on a single chip. It performs hardware address translation and includes a 64K coherent local cache for the FPGA. Since each of these design choices has significant impact on the offload model, automated analyses can afford to quickly explore the design space in a short period of time. The following issues need to be considered when integrating an accelerator with the rest of the system.

a) Distributed Execution – With accelerated workloads on heterogeneous systems, the execution of the workload migrates back and forth between the accelerator and the host. The execution model of the accelerator can be synchronous or asynchronous. In the synchronous model the host is stalled until accelerator execution ends. The execution model is not set in stone, rather determined by the flexibility of the synthesis tool and structure of the workload. With HLS tools like LegUp, the accelerator invocation is synchronous whereas for Vivado the accelerator invocations may target either paradigm. Further complications arise in the presence of multiple accelerators which need to coordinate with each other.

b) Data Movement – It is unlikely that an accelerator can perform a large amount of useful work without a robust memory interface. Computation intensive accelerators such as

those which target cryptographic applications may have less performance oriented interfaces for data fetch. Others such as DySER [64] may eschew memory operations altogether in the accelerator. Instead these memory operations are executed from the CPU. However, with increasing focus on "Big Data", it is likely that more data-centric workloads will be of interest to consumers. Traditional data movement to peripheral devices including accelerators is orchestrated using DMA (Direct Memory Access). These interfaces are cumbersome and require effort from the programmers to use. They need to ensure enough data is gathered and shipped to the accelerator to perform useful work. Often optimizations such as "double buffering" are required to keep the accelerator busy. Recent research work has highlighted the drawbacks of DMA based approaches [95]. The use of pull based data fetch (automatically performed by coherence protocols) is often preferred to push based DMA interfaces. A further concern is the granularity of data movement. A pertinent example is the usage of 128B cache lines on GPGPU L1 caches. This is larger than the standard cache line size found on general purpose CPU cores (64B). Workloads may have distinct access patterns where the full data present in the cache line may not be useful to fetch. For such cases data movement at a finer granularity offer significant reduction in energy consumption.

2.3 Thesis Contributions

This thesis summarizes work done at a time when the semiconductor industry is undergoing a fundamental shift to cope with the breakdown of Dennard and Moore's Law scaling. With the breakdown of device scaling, it is the burden of architectural scaling to provide the performance and energy improvements expected with the passage of time. The novelty of this dissertation lies in the fact that it does not attempt to target a particular accelerator architecture. It addresses the overall challenges of defining what to accelerate without the constraints applied by prior works which consider how to accelerate. We borrow abstractions from program analysis and build upon them to suit the needs of hardware specialization. Finally, we describe hardware specialization for data movement. In this section, we enumerate the contributions and potential impact of the work.

Contributions

1. Profiling for Specialization: We advocate path profiling [15] as a low overhead and precise methodology to guide analyses. Path profiling offers an alternative to coarser grained sampling based profilers as well as finer grained instruction granularity profilers. We show that finer grained analyses at the path granularity yields different results from coarser grained analyses. This has implications with respect to the specialized region and/or the specialization methodology. We have released the LLVM based implementation as free and open source software.
   ***Impact*:** Adopting our methodology will allow researchers to study the characteristics

of large workloads without sacrificing precision. The tools released can also be used to provide rich profile information to guide compiler transformations. Additionally, Path profiles can be used as templates for the generation of micro-benchmarks which replicate the recurrent behaviour for stress testing architectures.
***Release*: `github.com/sfu-arch/epp`**

2. Accelerator Benchmarking: To enable computer architects to study recurring program behaviour at the path granularity, we have assembled and released a workload suite for accelerators. This suite is derived from well known existing microprocessor benchmarks. With our released benchmarks, computer architects can rest easy knowing that they are targeting regions which are a) representative of the workload they are derived from, b) allow comparison of different accelerator architectures and c) are specializable (contain no accelerator unfriendly program features).
***Impact*:** The workload suite is released in a convenient package to encourage adoption. It includes workloads of significant complexity previously not targeted by accelerator research.
***Release*: `github.com/sfu-arch/pdws`**

3. Program Abstraction for Accelerators: We define a new program abstraction for hardware accelerators based on the program paths. They abstraction allows for the program to be summarized as speculative, single-entry single-exit regions. Control flow is minimized and opportunities for optimizations are increased. The constructed regions can be tuned for available accelerator resources as well as features. We provide a reference implementation based on LLVM for researchers to use.
***Impact*:** We present an abstraction tailored particularly for accelerators. Using this abstraction eases target accelerator codegen while ensuring program coverage. Using this abstraction, we have shown high level synthesis tools (LegUp [32]) can work where they previously failed.
***Release*: `github.com/sfu-arch/needle`**

4. Software Speculation: We develop a framework for the construction of software speculation frames. These allow for the removal of control flow, i.e branches are converted into control flow assertions. With this framework, arbitrary control flow segments can be outlined for further analyses or target code generation. Simple deoptimization strategies ensure correct program execution if a control flow assertion fails. We release the implementation as free and open source software.
***Impact*:** The simplification of control flow specialized regions allows for techniques such as sequentially dependent macro operation fusion [154] and the simplification of hardware interfaces to memory for specialized program regions.
***Release*: `github.com/sfu-arch/needle`**

5. Coherence Protocols for Accelerators: We identify the issues of traditional DMA based approaches to data movement in the context of many specialized regions in modern workloads. We develop and evaluate a timestamp based coherence protocol for localized data movement amongst specialized units. We show significant energy savings as well as performance improvements for workloads which have fine grained data sharing. Additionally, our proposed coherence protocol allows for performance improvements while simplifying the programming model.
   ***Impact*:** Based on our observations, researchers have studied opportunities for pipelined execution of kernels [73], efficient synchronization [156] and selective caching on GPUs [1]. Our work has also influenced research in the design of SoC interfaces [153] and sandboxing of accelerators [129].
   ***Release*: `github.com/sfu-arch/fusion`**

6. Adaptive Granularity Caching: We develop mechanisms to support variable granularity caching and show that often workloads fetch data which is not used in the lifetime of a cache line. Combining support for variable granularity with accurate prediction of cache line granularity, we show performance and energy improvements across a wide range of workloads.
   ***Impact*:** With variable granularity caching enabled by our work, researchers have applied similar methods to DRAM caches [87, 88, 67] and GPGPU caches [8, 16]. Additionally, variable granularity caching enables cache compression by decoupling the compression from storage concerns. Coherence protocols [186, 38] and prefetching mechanisms [173, 69] also leverage our work.
   ***Release*: `github.com/sfu-arch/amoeba`**

2.4 Relationship to published work

This dissertation includes work published at four peer reviewed conferences. In this section we detail the venues where the individual works have been presented.

IISWC 2016 – What to specialize? [96] Path based characterization as means to assess what to specialize was previously published at the IEEE International Symposium on Workload Characterization with co-authors Nick Sumner and Arrvindh Shriraman. A workload suite derived from microprocessor benchmarks was also released.

HPCA 2017 – How to specialize? [97] Program abstractions for specialization and their evaluation has been accepted for publication at the 23^{rd} IEEE Symposium on High Performance Computer Architecture with co-authors Nick Sumner, Viji Srinivasan, Steven Margerm and Arrvindh Shriraman.

ISCA 2015 – Integration – Accelerator Coherence [94] The Fusion coherence protocol tailored to mitigate redundant data movement was previously published at the 42^{nd} International Symposium on Computer Architecture with co-authors Arrvindh Shriraman and Naveen Vedula.

MICRO 2012 – Integration – Adaptive Granularity Caching [99] The mechanisms for adaptive granularity caching with fetch size prediction was previously published at the 45^{th} IEEE/ACM International Symposium on Microarchitecture with co-authors Hongzhou Zhao, Arrvindh Shriraman, Eric Matthews, Sandhya Dwarkadas and Lesley Shannon.

Chapter 3

What to specialize – Extracting Accelerator Benchmarks from Microprocessor Benchmarks

This chapter presents a characterization of microprocessor workloads with the goal of discerning their acceleration potential. The characterization is performed at the granularity of program paths [15]. This is a new facet to existing dynamic and static approaches. We have shown how this can be done in a scalable manner via lightweight dynamic instrumentation and static reconstruction. Our results show that within a workload, paths have disparate characteristics. Summarized characteristics at coarser granularities blend these characteristics and may draw imprecise conclusions. We derive a workload suite to assist accelerator architects by isolating the dominant paths within a workload for easier analysis.

3.1 Introduction

The objective in accelerators is typically to design a fixed-function or programmable hardware that provides the necessary support for a given program behavior with the lowest possible overhead (area/power). In contrast, a general-purpose processor tries to maximize performance across all applications for a given cost. It is imperative that the computer architects have access to the specific code regions within existing target applications so that accelerators can be designed with confidence. It is imperative for designers to understand not just the statistical characteristics and microarchitectural behavior [48] of the specific applications but also the precise functionality and semantics of code when designing the custom hardware. By design, accelerators are expected to provide functionality and performance only for a narrow phase of the application.

However, many critical real world applications were developed for CPUs and do not have explicitly marked phases of the program on which computer architects and designers

can focus the accelerator design effort. The absence of such real world workloads makes it extremely challenging to reliably develop accelerators hoping that they will be used in existing or future applications. Unfortunately, there have not been good benchmark suites. Current accelerator-specific suites [142] are essentially important kernels from libraries in mature application domains, but real world workloads are significantly more complex. By design, different fixed-domain and fixed-function accelerator proposals tend to pick algorithms and kernels from a specific application domain (e.g., machine learning or databases). It is not clear how to compare other accelerator architectures that may target the same code region or what code regions within a workload different accelerators should even target.

Benchmarking is a key tool for assessing computer systems. A core benefit of benchmarks is to enable comparing design alternatives during research or development and evaluating power/performance tradeoffs. Inflection points in computing systems (e.g., multicore, cloud computing) have resulted in new benchmark suites (e.g., PARSEC [20], CLOUDSUITE [53]). The pitfall and limitation is that these benchmarks may not be representative of real-life applications and may be very different from the application(s) of interest. An alternative would be to use real-life applications of interest. Unfortunately, real-life applications are very often challenging to set up with the need for mature compiler, operating system and library stacks (typically not available with hardware accelerators). This introduces a chicken-and-egg problem: designing accelerators suitable for applications requires the applications to convey precisely what function needs to be accelerated in the first place. We take an alternative approach. Instead of developing a new benchmark suite for hardware acceleration, we highlight specific code regions within existing applications that accelerators should target.

Our objective is to explore acceleration opportunities in existing CPU-based applications and make it possible for architects and designers to rapidly explore behaviors to specialize and accelerate. We ensure that the accelerators developed for the demarcated regions within each application can be directly deployed within the original application. In order to achieve this we have employed precise analysis of the execution paths [15] and extracted the frequently executed path into a separate function within the original program. Converse to prior work that extracted only key performance characteristics, we extract the frequently executed code region paths that an accelerator should target. We have extracted the acceleratable program paths embedded amongst many other unimportant or unacceleratable code regions into functions that accelerator compilers can target or use for simulation studies (e.g., Pin instrumentation [111]). We demonstrate that extracting such frequently executed paths in many cases requires carefully navigating across the control flow and precisely characterizing the biases of the control flow.

The approach that we propose overcomes an important shortcoming of existing accelerator benchmarks – they only seek to retain the memory access patterns and control flow behavior similar to the workload they represent [89]. Similarity is typically characterized using statistics such as branch biases or cache miss rates, which may suffice for studying micro-

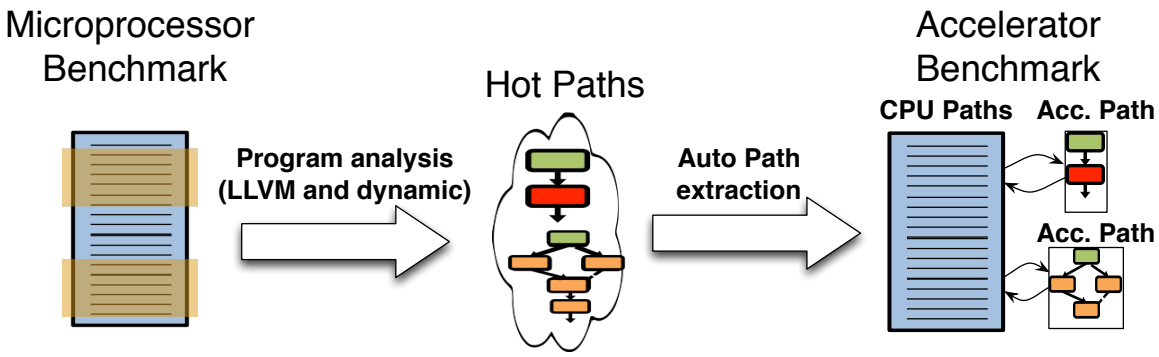


Figure 3.1: Using program analysis to demarcate and extract code paths [15] for accelerators within CPU programs.

architectural resource characteristics (e.g., branch prediction or cache architectures). While benchmarks have sufficed to study general-purpose microprocessor characteristics, they are too imprecise to indicate the specific code behavior that should be accelerated. We have extracted the specific code paths and ensure that our extracted paths i) replicate the functional semantics of the original application region ii) include the control flow of the original program, and iii) mimic the memory access behaviour of the original program. Figure 3.1 illustrates our overall approach.

3.2 Motivation & Methodology

Workloads often exhibit varied behavior internally. For instance, a program may have different phases of behavior representing initialization, computation, or cleanup. These phases perform different tasks and, as a result, exhibit different characteristics. However, analyzing an entire workload at once blends the characteristics of these different tasks together, obscuring the patterns in behavior of any one specific task. When exploring which behaviors in a workload to accelerate, these blended results may make it harder to tease out the characteristics of a particular program segment that capture desirable or undesirable behavior.

3.2.1 Acyclic Program Paths [15]

At a fine granularity of program segment, different acyclic paths in a program may exhibit different characteristics. A *path* is simply a sequence of instructions in a program. An acyclic path is a sequence of instructions that starts either at the beginning of a program or immediately after a back edge in a control flow graph and terminates at the end of a program or at the next back edge. Intuitively, acyclic paths divide the behavior of a program into loop free segments. For example, in Fig. 3.2, 1234 is an acyclic path that represents entering a loop starting at 2 but not revisiting 2. The acyclic path 234 represents a single

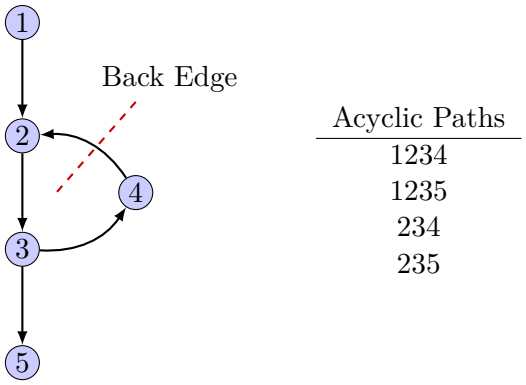


Figure 3.2: Acyclic paths in a control flow graph

iteration of the loop starting at 2, while the acyclic path 235 exits the loop. A function in a program comprises its constituent acyclic paths as well as any back edges that connect them. Thus, characterizing a workload's behavior at the function granularity will combine characteristics of all constituent paths within that function.

Some paths may be more beneficial to accelerate than others. For instance, when most of the computation in a workload happens along one path, it may be more fruitful to accelerate that path than an alternative. Indeed, we have found that the real world behavior of workloads can be highly biased toward some acyclic paths over others. Fig. 3.3 examines the relative coverage (in terms of dynamically executed instructions) of different paths for the hottest functions in workloads as determined by `gperftools` [63]. It presents the relative coverage of a workload provided by the five most frequently executed paths in the selected function vs the coverage of the workload provided by all other paths in the function combined. For most workloads, these five hottest acyclic paths are sufficient to cover the majority of workload's behavior. In many cases, the hottest path alone dominates the coverage. Accelerating those particular paths can thus be more beneficial than accelerating others. However, characterizing a workload at a coarser granularity, such as an entire function or loop body, will blend the characteristics across paths, once again potentially obscuring information that may help in making acceleration decisions. To overcome this problem, workloads can be characterized along acyclic paths in order to capture program behaviors within these fine grained program segments.

To examine the impact of characterizing paths of workloads, we first identify the most frequently executed acyclic paths within each workload. We then statically reconstruct each of these paths into independent functions and collect machine independent characteristics for each selected path in each workload. This section discusses the benchmarks that we used as well as our approach for selecting, reconstructing, and characterizing program paths.

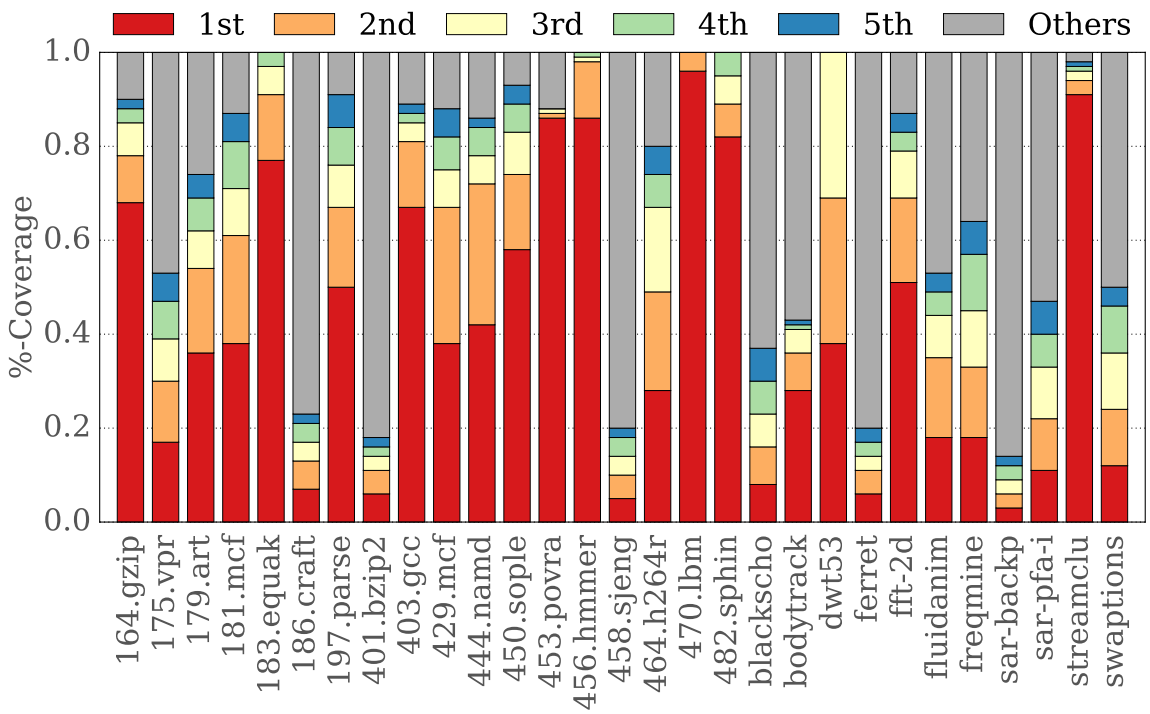


Figure 3.3: Path Bias

3.2.2 Selecting Paths to Characterize

Identifying frequently executed paths is an important part of many analyses for both architecture and for software. The classic approach to addressing this problem is to use the efficient path profiling technique developed by Ball and Larus [15]. This technique instruments a program to produce a dynamic profile as it runs. The instrumentation process first decomposes the control flow graph of a function into acyclic paths and assigns each path a unique integer id in the range from 0 to the total number of paths. Next, the program is instrumented so that the id for a path is computed as that path executes within the program, and the count or frequency of a path is incremented once the end of the path is reached. The end result of running an instrumented program is a count of how many times each acyclic path through a function is executed.

Efficient path profiling provides the foundation for our approach to identifying which paths inside a workload to characterize. For each workload in our benchmark suite, we select the 5 most frequently executed acyclic paths through the workload. Note, however, that the default efficient path profiling algorithm does not identify the frequencies of paths through an entire program, rather, it identifies the frequencies of paths through individual functions. Thus, we must adapt path profiling in order to profile acyclic paths at the program level.

In order to profile acyclic paths at the program level, we merge the control flow graphs of the entire program into a single function. We perform this by running an aggressive

inlining pass on the LLVM intermediate representation (*IR*) of a workload. This aggressive optimization performs function inlining at every possible call site within the IR. The impact of compiler optimizations on profiling is discussed in Section 3.2.3.

With inlining completed, efficient path profiling again enables us to identify the most frequently executed paths in the entire program. However, inlining introduces additional engineering burdens that must first be addressed. In particular, the number of acyclic paths through an entire program is larger than the number of paths through just one function within a program. As a result of inlining, the total number of paths may not be representable as a single integer during the profiling process. Column C1 in Table 3.1 shows the number of bits required to represent all the paths for a particular workload.

After performing path profiling on the fully inlined version of the program, we select the top five most frequently executed paths from each workload. Recursive function calls and calls through function pointers cannot necessarily be inlined. These constructs partition the program into disjoint functions after aggressive inlining. In these cases, we select the hottest of the remaining functions using `gperftools`.

3.2.3 Extracting identified paths

After identifying the most frequently executed paths inside each workload, we extracted each such path into its own function for easier, more isolated study. For each path, we created a new function containing the same sequence of instructions as in the original path. A branch instruction in the middle of the path can force program execution to deviate from the path once it has started. When this happens, the function returns early, and the original version of the program is executed. Thus, we call these exit guards. All incoming dependencies from live-ins inside the path are hoisted to arguments of the function, while all outgoing dependencies are returned through a struct when the function completes. All store instructions are recorded to an undo buffer that is replayed when the path exits early. After extracting each path into its own function, we once again run `-O2` optimizations to remove any unnecessary operations and simplify the path specific behavior. By extracting these paths into their own functions, we have created durable artifacts that may be reused by other analyses.

Impact of compiler optimizations on profiling: Compiler optimization applied prior to path identification and profiling can alter observed characteristics. Herein we focus on optimization applied in the optimizer or "middle-end" of a three pass compiler. These optimizations are generic in nature, i.e not tied to any particular source language or target architecture. We find that there is no correlation between the sizes of paths extracted with the level optimizations applied, i.e `O1, O2` and `O3`. The median size of paths profiled across applications remained the same. The instruction mix varied based on the optimizations applied. For example, some IR operations are only introduced by more aggressive optimiza-

tions such as `InstCombine`, LLVM's peephole optimizer. Loop specific optimizations such as unrolling, interchange, rotation and independent code motion all affect the instruction mix of hot paths. Furthermore, optimization passes are often scheduled multiple times (phase ordering) to take advantage of changes introduced by other passes. For example, Loop Independent Code Motion is scheduled 3× at `O2`. Phase ordering of compiler passes is fixed by standard compilers (GCC and LLVM) without regard to the input program. Optimal phase ordering is *undecidable* [167]. In our methodology, we use `O2` optimizations prior to profiling and identification. Prior work [40] has shown the impact of `O3` optimizations over `O2` is often indistinguishable from noise.

The methodology we adopt is target independent yet there are target specific optimizations or architectural features which may reduce the overheads of profiling and path identification; our current implementation imposes a 20–40% runtime overhead. For example, native support for 128 bit arithmetic will reduce the overhead of 128 bit emulation using 64 bit registers; required for path identification at runtime. Architectural extensions such as Intel Processor Trace [85] may also provide low overhead profiling alternatives.

3.2.4 Metrics & ISA-independence

We base our workload metrics upon prior work by Shao [151]. In particular, we examine the unique opcodes, memory address entropy, and the number of guards or unique branches. We also extend their metrics into analogues at the path granularity. This includes path predictability, an analogue of branch entropy, as well as the average number of read and write operations across extracted paths for a workload, which together provide an upper bound to the memory footprint. Finally, we add metrics that are more relevant for analyzing acyclic paths. This includes the number of live in and live out values to the path, the number of ϕ operations removed when extracting the path from the original control flow graph, and the total number of static instructions in the path. More details are provided in Section 3.3.2.

Our analysis operates on the LLVM Intermediate Representation (IR) used in the middle end. By utilizing LLVM IR as our representation for characterization, we are able to draw conclusions that better reflect the intrinsic semantics of the original program. Prior work has shown this to be highly desirable [151].

3.2.5 Characterizing at the Path Level

Static characteristics of the extracted paths are computed directly from their corresponding functions. Applying optimizations again after extracting each path into its own function produces characteristics that are more reflective of that particular path's behavior. Any computation used only on branches that exit from the path is removed, and the remaining computation is simplified to more accurately reflect the behavior of just the path of interest.

Dynamic characteristics of the path, namely the addresses of loads and stores to heap allocated memory, are computed by re-executing the entire workload with the path of interest outlined into its respective extracted function. The addresses of heap accesses are recorded using Pin for further characterization via, e.g., memory entropy. Stack accesses are ignored, as they reflect more architectural dependent characteristics rather than intrinsic behaviors of the workloads of interest.

3.2.6 Benchmarks

We include 29 workloads from SPEC2000, SPEC2006, PERFECT [17] and PARSEC [21]. [1] All benchmarks were compiled with the LLVM C and C++ compiler, Clang (version 3.8), in order to generate LLVM bitcode for instrumentation and analysis. We perform aggressive loop unrolling (4×) with an increased threshold (2×) and allow partial unrolling. Both before and after instrumentation, all workloads were optimized at the level of `-O2` with vectorization disabled. Executable versions of the extracted workloads were then compiled for X86-64 using LLVM 3.8.

3.3 Characterization

This section highlights the disparate behaviour of workloads along frequently executed acyclic paths. Our approach is in contrast to prior work [151, 181], which examines workloads as a whole or at a function granularity. We find that considering paths as the granularity for characterization yields insightful information for specialization. We describe our methodology in § 3.2.

3.3.1 Making a case for Path-based Acceleration

The key to an effective offload abstraction is that it must concisely capture varied dynamic phase behavior exhibited by a workload. For instance, a program may have phases of behavior representing initialization, computation, or cleanup that exhibit different execution characteristics. Analyzing an entire workload at once blends the characteristics of these different tasks together, obscuring the patterns in behavior that should be accelerated. When exploring which behaviors in a workload to accelerate, these blended results may make it harder to tease out the characteristics of a particular program segment that capture desirable or undesirable behavior. We show that paths executed by even a single program exhibit diverse characteristics and are a natural fit for specialization. A path is simply a sequence of dynamic instructions in a program. Intuitively, acyclic paths divide the behavior of a program into loop free segments. A function in a program comprises its constituent

[1] We drop benchmark programs where the selected function contains language features unsuitable for an accelerator: e.g.,setjmp and longjmp in 471.omnetpp and C++ exceptions 447.dealII in 483.xalancbmk.

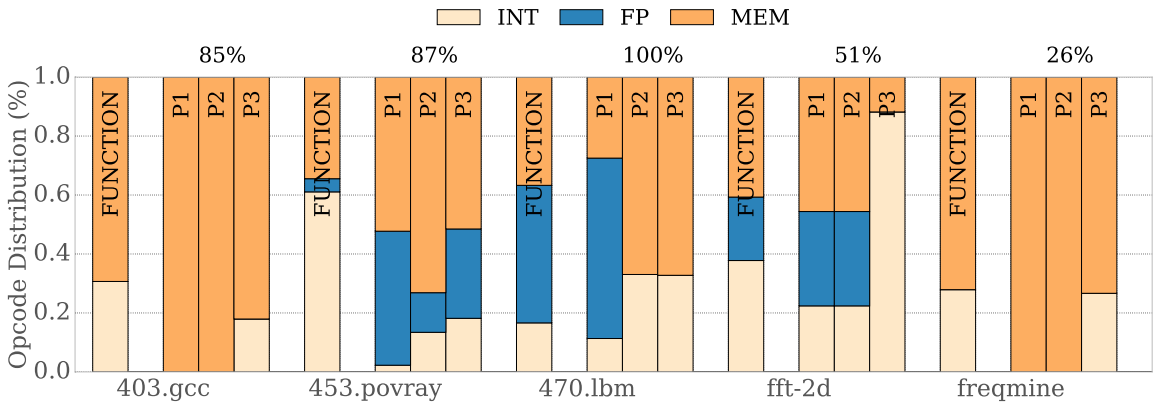


Figure 3.4: Benefits of Path-Based Execution. We have only shown a few workloads due to lack of space. Opcode histogram of paths within a function; % indicates exec coverage.

acyclic paths as well as any back edges that connect them. Thus, characterizing a workload's behavior at the function granularity will combine characteristics of all constituent paths, obscuring overall behavior. Furthermore, some paths may be more beneficial to accelerate than others. For instance, when most of the computation in a workload happens along one path, it may be more fruitful to accelerate that path. Indeed, we have found that the behavior of real workloads is highly biased with only a few hot paths.

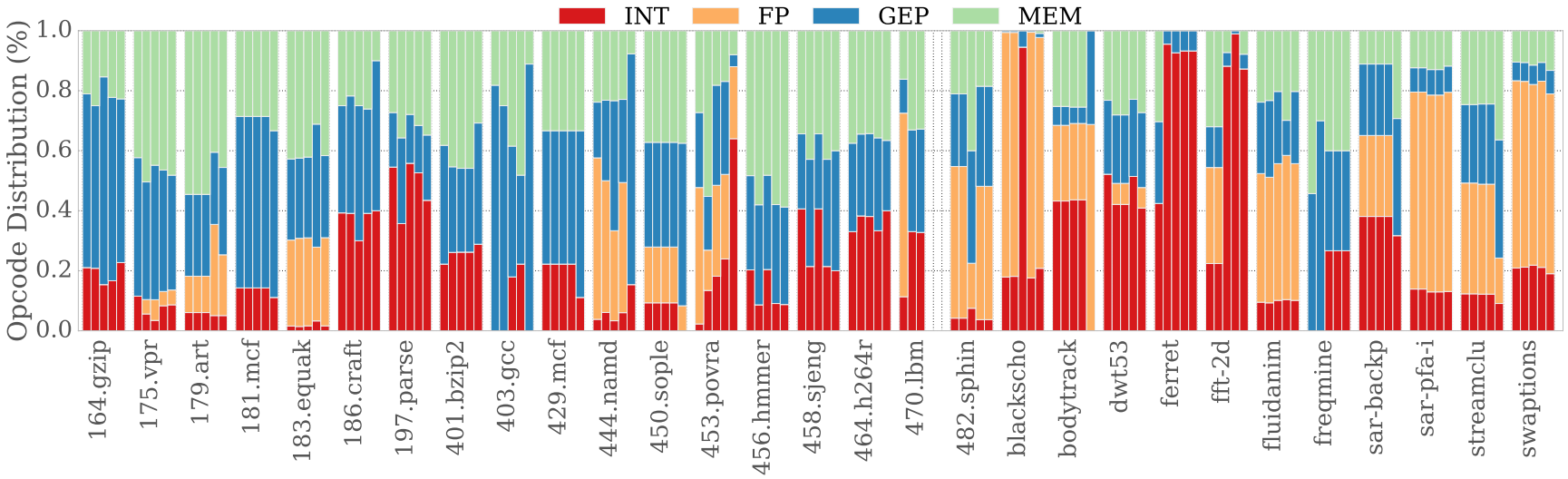


Figure 3.5: Opcode Distribution. The 5 bars for each workload represent the top-5 hot paths (L-R), GEP=pointer access.

The histogram in the Figure 3.4 demonstrates that the granularity of program analysis and offload abstraction fundamentally biases what to include on the hardware accelerator. The opcode distributions are shown for the top three hot paths in the most important function the program, and the % indicates the total execution coverage achieved by the paths. An example that motivates our approach is 470.lbm. The function breakdown shows an overall bias for FP (60% of dynamic ops). However, we see that the only one path executes floating point instructions. The remaining paths are dominated by MEM operations, and they have no FP operations at all! An accelerator designed to offload each path separately can be customized to support only the operations in that path. Another interesting observation is that while 453.povray is an INT-heavy function (60%+ operations), the hottest paths cover 87% of the dynamic execution yet consist less than 20% of INT operations. Thus, some colder path with INT ops skews the bias of the function overall. Finally, we highlight freqmine and gcc as cases where the overall function can be easily segregated into paths that access memory and paths that compute, which permits the synthesis of fully decoupled specialized accelerators. Another benefit of path-based regions is saving of wasted work. Typical program regions tend to have multiple execution paths due to control flow and the relative hotness of these paths is exhibited only by dynamic execution profiles. Current HLS tools use a static approach to acceleration region formation and conservatively offload multiple paths.

Table 3.1: Workload Characteristics

C1 : log2(NumPaths) **C2** : Exe. Paths (MAX_5) **C3** : Ins. **C4** Cov. **C5** : Guards **C6** : Phi Nodes Removed
C7 : Live Vals **C8** : Mem. Entropy ($GEOMEAN_5$) **C9** : Mem. RD **C10** : Mem. WR **C11** : Num Unique Opcodes

		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
Workload	Function	Bits	Exec	Size	Cov%	$\diamond$	ϕ	V$\uparrow\downarrow$	M.S	M.$\downarrow$	M.$\uparrow$	{Op}
164.gzip	longest_match	10.4	813	70	59	7	11	6,6	16.0	4.2	0	11
175.vpr	try_route	79.8	5394	332	2	25	10	9,6	14.6	26.8	6.2	13.8
179.art	match	19.8	6082	174	11	4	3	3,3	10.0	31.2	4	13.2
181.mcf	price_out_impl	10.8	402	21	1	2	2	3,2	8.3	2.2	0	7.8
183.equake	smvp	4.3	13	962	53	4	8	14,11	17.3	131.4	8.4	10.6
186.crafty	EvaluatePawns	62.7	37443	67	03	14	12	11,4	6.9	4.8	0	10.6
197.parser	table_pointer	9.5	250	115	51	12	2	7,2	10.8	8.8	2.6	12.8
401.bzip2	BZ2_compressBlock	69.5	72561	784	05	71	6	9,8	18.3	100.8	4	11
403.gcc	bitmap_operation	11.9	21	100	67	6	13	9,7	10.0	4.4	2	8
429.mcf	price_out_impl	9.7	141	24	1	2	2	3,2	8.9	3.0	0	8.8
444.namd	calc_pair_energy_fullelect	14.8	249	673	44	8	16	36,16	12.0	22.6	6.6	13.4
450.soplex	vSolveUrightNoNZ	9.3	389	94	13	4	3	11,4	12.8	11.0	3.6	11.8
453.povray	All_Sphere_Intersections	17.8	33377	331	86	10	10	15,12	6.0	16.6	5.8	17.6
456.hmmer	P7Viterbi	13.8	36	490	71	8	7	20,2	16	47.2	20.2	9.2
458.sjeng	gen	34.3	46971	95	05	13	4	3,3	5.4	6.8	0.8	10.6
464.h264ref	dct_luma_16x16	26.5	88	433	19	25	25	14,5	8.3	48.0	8	16
470.lbm	LBM_performStreamCollide	2.3	2	479	96	2	1	3,2	19.6	26.0	19	10.7
482.sphinx3	vector_gautbl_eval_logs3	5.9	9	154	4	4	4	13,8	14.0	12.0	1.2	11.6
blackscholes	BlkSchlsEqEuroNoDiv	22	34	314	08	32	52	9,4	2	0.4	0	20
bodytrack	InsideError	18.8	64516	233	16	16	6	12,5	4.7	16.0	6.4	15.2
dwt53	dwt53_row_transpose	5.9	12	122	37	4	1	9,2	19.0	9.8	5.4	12.6
ferret	image_segment	19.0	31136	485	04	32	54	8,7	17.3	7.2	4.8	12.6
fft-2d	fft	27.9	46	232	24	28	5	8,1	17.2	11.2	8	14.4
fluidanimate	ComputeForces	23.1	39838	143	13	12	4	18,3	14.0	13.8	1.2	14.8
freqmine	conditional_pattern_base	8.4	133	94	13	4	3	6,8	10.2	8.0	3.6	9.4
sar-backp	sar_backprojection	77.7	4616	127	01	13	9	12,7	8.4	4.2	3.8	22.2
sar-pfa-in	sar_interp1	40	173	509	07	54	29	17,3	6.5	23.4	7.6	22.4
streamc	pgain	11.4	74	249	41	16	3	11,6	13.6	27.4	0.6	13.6
swaptions	HJM_Swaption_Blocking	72.5	11663	462	12	30	18	9,3	11.5	24.0	8	24.8

3.3.2 Characteristics Summary

Table 3.1 presents key characteristics for the workloads we study. Column C1 indicates the number of bits required to encode all the static paths that a workload may execute. We see a large variance across workloads depending on their nature. Some of the more complex workloads we study are swaptions and 186.crafty with 73 and 63 bits required to enumerate all paths. Path explosion is described in more detail in § 3.2. Often, floating point workloads demonstrate less complex structure. Workloads such as 470.lbm, 183.equake and 482.sphinx3 all require fewer than 5 bits to enumerate all paths.In comparison to the potential number of paths in a workload, the actual number of unique paths executed is often low. Only 11 workloads have more than 1000 paths executed during program execution. 401.bzip2 has the largest number of paths executed with over 72K. The median number of paths executed is 250. Across the workloads we study, there exists path bias, i.e few paths executed far more frequently than others (see Figure 3.3).

Columns C3-C8 in Table 3.1 provide the maximum value of a particular characteristic across the five most frequent paths of a workload. This data combined with the normalized visualization presented in Figure 3.6 allows the reader to derive the absolute values for each of the 143 paths (across 29 workloads) presented. The observations are discussed in a workload centric manner in § 3.4. Herein, we discuss the path characteristics across workloads and their implications on accelerator synthesis.

Path Length and Opcode Mix: Column C3 in Table 3.1 shows the size of the largest path for each workload. The largest path overall was from 183.equake with 962 IR instructions. The median size of the largest path from each workload across the suite was 232 instructions. 7 out of 29 workloads have fewer than 100 operations, and four workloads have paths with more than 500 operations.

Prior work such as BERET [68], has sought to accelerate "Superblock" regions. Such characterization is often limited by predetermined hardware constraints of the accelerator. Our path based characterization yields different results as we do not have any preconceived notion of the specialization target. Figure 3.5 shows the distribution of Opcodes across the five frequent paths for each workload. We classify the opcodes as INT, FP, MEM and GEP.

GEP operations in the LLVM IR are a succinct representation of address generation logic. They define, in a platform independent manner, the operations required to generate a particular memory address prior to the access. Classifying GEPs separately allows us to quantify "work" required to fetch data independent of the actual compute on the data. Overall Figure 3.5 shows that across the frequent paths in a workload there may be significant differences in their opcode mix. GEP and MEM operations tend to account for a significant fraction of the work in the hot paths across workloads, on average 45% of the number of

operations. Only 49 of 143 paths have more compute (INT+FP) operations than memory (GEP+MEM).

In some floating point workloads, amongst the top 5 paths, there exist paths with no floating point operations at all. Workloads such as 444.namd, 470.lbm and blackscholes have at least one or more frequent paths devoid of floating point operations.

Conversely, four of the top five paths in 175.vpr have floating point operations (5% of total) on average. Similarly for dwt53, 7% of the operations across three of the top five paths are floating point operations (primary datatype was integer – `typedef int algPixel_t`).

Another interesting observation is the presence of paths with only GEP operations or GEP and MEM operations but no compute. 470.lbm is a workload with two paths that only compute GEP expressions. One of the paths is the macro definition `SWEEP_START`. The macro is defined shown in Listing 3.1.

Listing 3.1: Macro definition – 470.lbm

```
1 #define SWEEP_START(x1,y1,z1,x2,y2,z2) \
2     for(i = CALC_INDEX(x1, y1, z1, 0); \
3         i < CALC_INDEX(x2, y2, z2, 0); \
4             i += N_CELL_ENTRIES ) {
```

This particular case occurs in freqmine (2 paths) and 403.gcc as well. Column C11 in Table 3.1 shows the average number of unique IR instructions that are present in the top five paths. The number ranges from $\simeq$8 to 25 unique IR operations across the workloads. Within workloads the variability is low. The total number of opcodes present in the IR is 64 (LLVM 3.8).

Branches, Guards and ϕ: Column C4 in Table 3.1 shows the number of conditional branches converted to guard checks for exiting the middle of a path. Guards are discussed in more detail in § 3.2. The presented number is the maximum across the five frequent paths for each workload. The largest paths, 401.bzip2 and sar-pfa-interp1 have 71 and 54 guard checks respectively. All other workloads have 32 or fewer guards, and 13 workloads have $<$10 guards. The largest "guard density" (guards divided by size) we find is 22% for 183.crafty.

ϕ's in the LLVM IR are instructions that select incoming values based on the result of control flow operations. ϕ's have a direct impact on the complexity of specialization as observed in prior work [64, 127]. Reasoning about specialization along paths allows for ϕ simplification. The ϕ's can be resolved since the control flow is known a priori. This is proportional to the number of branches removed (conditional and unconditional). Workload behaviour determines the number of ϕ's required (and thus elided) at each branch.

The largest number of simplifications across 143 paths occur in ferret. Note that in this case there is no path correspondence with the other metrics such as the path size. The

maximum number of ϕ simplifications may occur along different paths. Over 143 paths, the average number of ϕ's simplified per path is 0.68 (geomean). However, it can be particularly high in some cases, with 6-8 ϕ's simplified in 5 of 143 paths (workloads – 164.gzip, 183.equake, 444.namd $\times$2, bodytrack). Note that ϕ used as induction variables are not included in this analysis.

Live Values: The live values of a path are the virtual register values that are a) used within the path (live in) and b) defined within the path and used outside (live out). Quantifying the live input and output values (Column C7) provides a notion of the overheads of data transfer into and out of the specialized unit. We present the maximum number of live values (input and output) per workload in Column C7. Memory state is treated separately and discussed in the following paragraph.

Across workloads and their top five paths, the average number of live values is 10. The maximum number of live values was observed in 444.namd (36/16 – in/out). Across benchmarks the least number of live values was observed for 181.mcf. 59 of 143 paths had fewer than 10 live values while only 1 path had more than 25 (444.namd).

Memory Access Characteristics: We present characteristics of the paths in Columns C8-C10 of Table 3.1 with respect to memory behaviour. Column C8 enumerates the maximum *memory address entropy* for the five most frequent paths per workload. This metric has been used previously [151, 182] to quantify the information content, i.e. the predictability of memory addresses. Entropy in information theory encodes the randomness of the variable. Herein, each unique location accessed is treated as a value for the variable. Lower numbers imply higher predictability. We analyze the memory address entropy for heap accesses only. Shao et al.[151] find that analyzing heap+stack addresses together provides less meaningful results. blackscholes has a pattern of reading from six arrays of same size, computing a value, and writing the computed value to a seventh array. The memory accesses are regular uniform strides and result in low memory address entropy (2). More details of the implementation are presented in § 3.5.

Columns C9 and C10 present the average number of memory read and memory write operations across the frequent paths. The range of average number of memory reads extends from 0.4 (blackscholes) to 131.4 (183.equake) across the frequent paths in our workloads. The blackscholes benchmark from PARSEC passes input as function arguments to the hot function, thus reducing memory reads. The range of memory writes extends from 1 to 20.2 operations (for workloads with non-zero memory writes on average). For applications with zero writes (52 of 143 paths), we find paths that return live values rather than issue stores to memory. Almost all paths are "consumer" in nature, where the reads outnumber writes. One path (sar-backprojection, rank 5) has more writes than reads. Only 11 of 143 paths

Table 3.2: Path Predictability

ID	Freq	Path	Probability
1	100	**A** B C	$100/(100+25) = 0.8$
2	25	**A** B D	$25/(100+25) = 0.2$
3	10	**F** G	$10/10 = 1.0$

have more than 16 writes to memory. Note that this does not distinguish aliasing memory locations. We only comment on the number of operations per path.

3.4 Path Characteristic Variability

In this section, we summarize our observations across workloads. Figure 3.6 presents key characteristics of the five most frequent paths across workloads. We present six features to contrast paths within a workload, four of which are derived statically. Prior work [151, 127] has indicated these features are key to understanding amenability to acceleration. We observe that different program paths exhibit different characteristics.

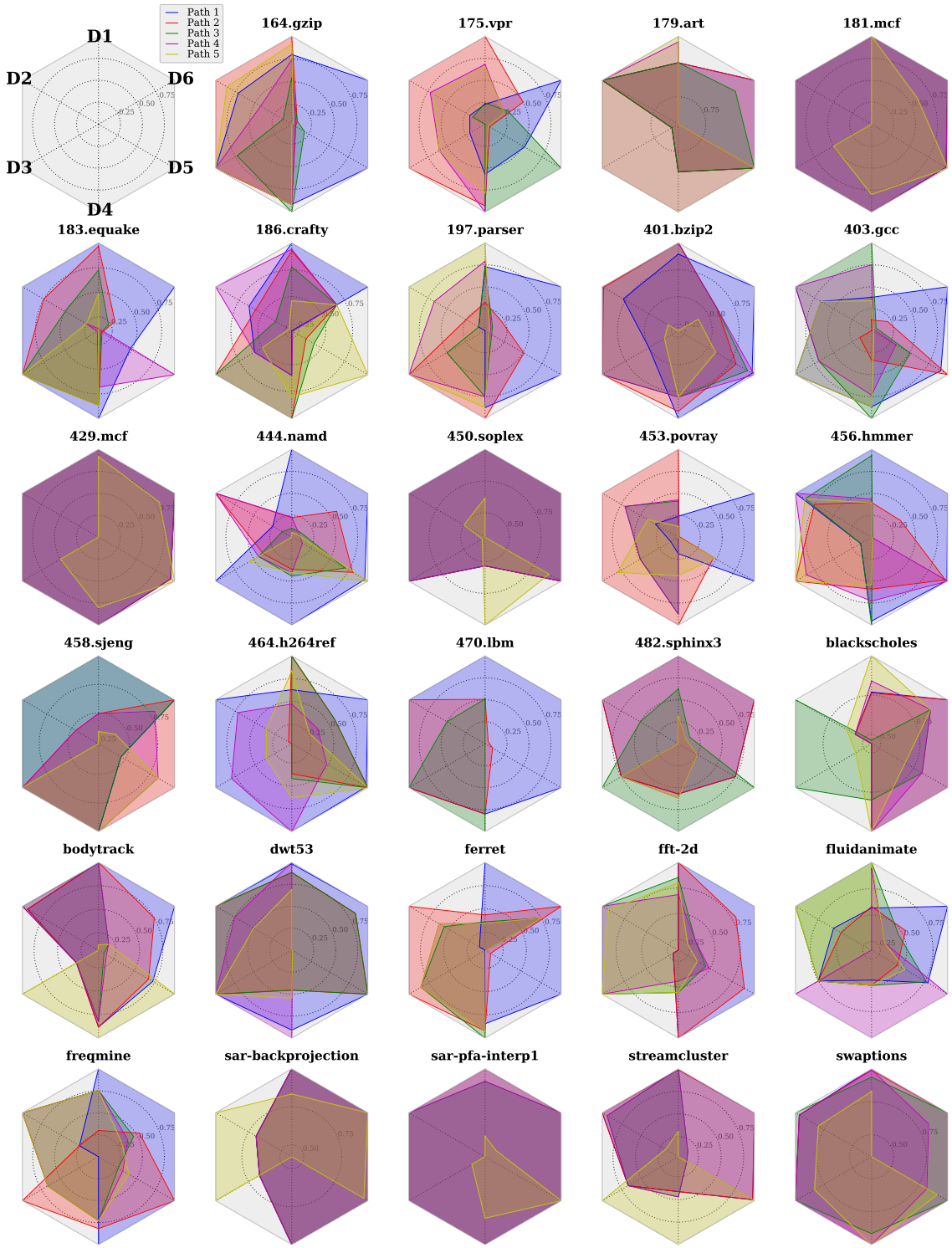
Description: Each radar chart represents a single workload. Each outlined overlay on the chart represents a frequent path (five in total). There are six dimensions on each radar chart. In counter clockwise order, they are i) **D1**: (Norm.) Number of instructions ii) **D2**: (Norm.) Number of guards, iii) **D3**: (Norm.) Number of ϕ's simplified, iv) **D4**: (Norm.) Total number of live values v) **D5**: Predictability, vi) **D6**: Coverage.

Of the six metrics enumerated, **D1**-**D4** have been discussed previously in § 3.3.2. The radar charts present normalized values. Absolute values per path can be derived from the max values presented in Table 3.1 (**C3**-**C7**). **D6**: Path predictability is a new metric we introduce in this section. Path predictability is the probability of following a known path given the starting basic block. Consider the contrived path profile in Table 3.2. The predictability of each path is calculated as the execution frequency of the path divided by the sum of the frequencies of all other paths which *begin at the same basic block*. Paths with IDs 1 and 2 start from basic block A. Based on their respective frequencies, the probability of executing 1 to completion is 0.8. Similarly, the probability of executing 2 to completion is 0.2. Larger numbers ($max = 1$) indicate amenable paths for specialization since they exit less often.

Discussion: Overall, we observe that paths within workloads have varied characteristics; i.e path outlines are clearly visible in Figure 3.6. In a few cases there is overall similarity amongst a subset of paths. Examples of such cases include *.mcf, 453.soplex, sar-*, and swaptions.

Along dimension **D1** (size) some workloads are linearly spaced out. Examples of such workloads are 183.equake, 186.crafty, blackscholes and freqmine. Workloads such as

Figure 3.6: **D1**: Total Ins, **D2**: Guards, **D3**: ϕ's Simplified, **D4**: Total Live Vals, **D5**: Path Predictability **D6**: Path Coverage



bodytrack and streamcluster have little to no variability in the size of the longest path. This is frequently observed where paths are spatially colocated.

D2 enumerates the number of guards introduced by eliminating conditional branches. Workloads such as 179.art, 456.hmmer, fft-2d and bodytrack show significant overlap along this axis. 450.soplex has 4 paths with the same number of guards (4). The same holds for four paths in streamcluster (paths #1-#4). It is interesting to contrast **D2** (guards) with **D1** (size). For example in 183.equake, four paths are linearly spaced out across **D1** and **D2**. 186.crafty has 3 paths of similar size but differing number of guards. In 179.art, paths #3 and #5 have similar number of guards but differ in size. One cause where such behaviour is observed is due to unbalanced if conditions.

In most workloads (21 of 29), the path with the largest number of instructions is also the path with the largest number of guard checks. This does not hold true for 186.crafty, 444.namd, 464.h264ref, blackscholes, ferret, fft-2d, freqmine and sar-backprojection.

Dimension **D3** enumerates the number of ϕ's simplified. 183.crafty and 197.parser have a similar number of ϕs removed while differing along axes **D2** and **D1**. dwt53 has the same number of ϕs simplified but differing number of branches (converted to guards) in each path. fluidanimate along **D1**, **D2** and **D3** have interesting characteristics. Path #4 has large size but significantly fewer guards and ϕ's simplified.

The sum of live input and output values is shown along dimension **D4**. It is interesting to compare the value along **D4** with **D1**, i.e whether the number of live in and live outs is proportional to the size of the path. For 21 of the 29 workloads the largest path also has the largest number of live values. The converse is true for streamcluster, fluidanimate, 450.soplex, ferret, bodytrack, fft-2d, 470.lbm and 464.h264ref. For each workload there is significant variability across paths with respect to live values. There are a few workloads where 3 or more paths have similar live values. Some examples are ferret, bodytrack, 444.namd and 164.gzip. In many workloads, accounting for the largest number of live values per path will waste 25% or more of the local scratchpad. This holds in 9 of 29 applications; one path has 25% more live values than others.

Path predictability **D5** is the measure of probability a path will execute to the end. Values close to one are desirable as they imply lower overheads for specialization. Overheads are incurred when partial execution on a specialized unit needs to be rolled back. The path also needs to be evaluated in software, restarting at the beginning. Five of the 29 workloads, 470.hmmer, 444.namd, 456.hmmer, fft-2d and freqmine, have frequent paths that are perfectly predictable for the given input data. In 17 out of 29 workloads, the largest path had near perfect predictability. Conversely, 8 out of 29 workloads had large paths with poor predictability ($< 50\%$).

Path coverage is shown along **D6**. It is representative of the amount of work each path does. It is computed as the frequency weighted size of each path. Often the largest path (**D1**)

does not have the highest coverage. Some examples are 164.gzip, 401.bzip2, blackscholes and fluidanimate.

Overall, there are interesting paths that stand out across workloads. Path #1 from freqmine is the largest path and has the highest number of live values and coverage, yet it is perfectly predictable for the given input. 444.namd has a few paths oriented along the **D2**-**D5**, i.e paths that have many guards yet are predictable. For some paths in bodytrack, streamcluster and 458.sjeng, increased ϕ simplification was observed along more predictable paths. Path #1 from 470.lbm has the maximum values along 5 of the 6 axes. While being the largest path, with the highest number of guards and ϕ simplifications and high coverage, it has fewer live values. Path #5 in 179.art has the maximum along all axes apart from **D5**, i.e it is a large (174 instructions) path with perfect predictability, few ϕ's simplified (3) and 6 live values. These characteristics make the path amenable for specialization. On analysis of the source for the particular path, we find lines 140–146 in `scanner.c`. This is a segment from the function `simtest2`. The code for the path is shown in Listing 3.2 (reformatted for typesetting).

Listing 3.2: Path from simtest2 – 179.art

```
1  Su = ((double)numf1s*su2-su*su)/
2    ((double)numf1s*((double)numf1s-1.0));
3  Su = sqrt(Su);
4  Sp = ((double)numf1s*sp2-sp*sp)/
5    ((double)numf1s*((double)numf1s-1.0));
6  Sp = sqrt(Sp);
7  numerator = (double) numf1s * sup - su * sp;
8  denom = sqrt((double) numf1s*su2 - su*su) *
9    sqrt((double) numf1s*sp2 - sp*sp);
10 r = (numerator+e)/(denom+e);
```

Changes in 181.mcf to 429.mcf in SPEC2000 to SPEC2006 are described [163] as ‘‘*Because there have been no significant errors or changes during the years 2000 - 2004, most of the source code of the CPU2000 benchmark 181.mcf was not changed in the transition to CPU2006 benchmark 429.mcf. However, several central type definitions were changed for the CPU2006 version by the author*”. For mcf, overall path characteristics remain the same between versions. Path #5 has increased coverage. On analysis of the source, we find a new condition that changes the memory reallocation criteria, which in turn make paths in 429.mcf more amenable to specialization (increased coverage). The condition is `if( net->n_trips <= MAX_NB_TRIPS_FOR_SMALL_NET )` in `implicit.c`.

470.lbm has a total of five dynamically executed paths. Of these three are represented on the chart as area filled polygons. The other two consist of a single basic block and have zeros for dimensions **D2**, **D3** and **D4**, i.e they have no guards, ϕ's simplified (artifact of

being a single block path) and no live values. They are perfectly predictable as there is only one basic block. They are not shown on the radar chart.

To summarize, we find that distinct paths across workloads have characteristics that are unique to the paths themselves. Analysis of characteristics at the function or coarser granularity blends the characteristics from many paths and adds noise. For the purpose of specialization, we advocate the adoption of a path granularity analysis.

3.5 Path Derived Workload Suite

The previous sections have established the significance of characterization at a path granularity and discussed at length the characterization of a large number of workloads. We have identified frequent acyclic paths across 29 workloads drawn from popular benchmark suites. Using our LLVM tool chain we have outlined the paths into independent functions free from control flow (apart from guard checks). These can now be easily analyzed using existing tools for dynamic analysis such as Intel Pin.

3.5.1 Memory Address Entropy Analysis

In this section we describe how our derived suite assists researchers perform precise analyses using existing tools. To compute the memory address entropy along a path, we extended an existing memory address tracing tool. A flag was added to enable / disable trace dump at runtime. We then added instrumentation to set and unset the flag at function invocation and return via the `IMG_AddInstrumentFunction(...)` interface. The code that targets the path is shown in Listing 3.3.

Listing 3.3: Pintool Modifications

```
1 string name =  PIN_UndecorateSymbolName(RTN_Name(rtn),
      UNDECORATION_NAME_ONLY);
2 if (name.find(string("__offload_func")) != string::npos) {
3     RTN_Open(rtn);
4     RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR) EnterROI, IARG_END
          );
5     RTN_InsertCall(rtn, IPOINT_AFTER, (AFUNPTR) ExitROI, IARG_END);
6     RTN_Close(rtn);
7 }
```

All the outlined paths have the following naming convention `__offload_func_XXX` where `XXX` is the identifier of the path computed by the Ball-Larus algorithm [15] (see Section 3.2 for more details). When control flow reaches the starting basic block of the outlined path, it optimistically chooses to invoke the *"path outlined as function"*. Providing a clean abstraction for the path allows us to write tools to target that particular region only. Should control

flow deviate from the path, the function returns false to indicate a side exit, and the original program code is executed after the original program state is restored.

3.6 Related Work

Benchmarks and Synthetic Workloads Several approaches have been proposed to construct synthetic benchmarks that are representative of the real workload for specific microarchitectural behavior (e.g., cache misses [118] or branch predictability [89]). These techniques typically measure the microarchitectural runtime behavior of the real workload and construct a set of synthetic code regions that place similar demand on hardware resources. It is not sufficient for an acceleratable region to simply demonstrate statistically similar behavior to be useful for computer architects. The benchmark must be functionally representative since accelerators by definition are functionally specialized for the targeted code. Prior work has largely focused on benchmarks that mimic a real workload's power consumption [77] memory locality behavior [161], cache hit/miss ratios or even branch behavior. Bell et al. [18] presented a framework for the automatically synthesizing benchmarks from executables/specax. They leveraged statistical simulation theory and generate C-code and assembly instructions that accurately model the workload attributes. Performance cloning [89] is another technique that seeks to more precisely capture the control flow and memory locality predictability of the original application. In contrast, our goal is to demarcate program regions in existing applications to indicate explicitly to simulators and binary analysis tools the code paths that are suitable for hardware acceleration. The demarcated paths within the original program precisely capture the functional behavior of the dynamic execution of the code paths.

Accelerator Studies: Current accelerator studies have developed compiler infrastructure for studying existing CPU workloads [64, 68, 128]. They identify specific code behaviors within the workload targeted by their hardware and have provided mechanisms to demarcate them in the application. Unfortunately, the compiler approaches are closely tied in with the hardware accelerator, and it is unclear whether the identified code regions can be used by researchers developing a different accelerator architecture. A key challenge with current accelerator studies is that it is not feasible to compare accelerator architectures directly since they may not even be commenting on the same code region. Our focus has been to approach the question of "acceleratability" from the application's perspective and demarcate code paths for which specialization can directly provide performance and power benefits to the application. This permits different accelerator microarchitectures to be directly comparable since they target a common code region. Machsuite [142] provided a set of kernels drawn from various algorithms. It is unclear yet whether accelerators should be targeted at fine-granularity regions such as kernels and also whether kernels can be representative of real programs that include frequent control and are not necessarily written

as a collection of kernel workloads. Our goal has been to identify the code paths within existing CPU workloads that accelerators should target.

3.7 Conclusion

We advocate analysis at the granularity of acyclic program paths when assessing the amenability for acceleration of a workload. This is a new facet to the existing dynamic and static approaches. We have shown how this can be done in a scalable manner via lightweight dynamic instrumentation and static reconstruction. We have built a robust LLVM based toolchain to automate the analysis and presented our results for 29 workloads drawn from SPEC2000, SPEC2006, PERFECT and PARSEC. We have analysed $\simeq 356K$ paths across workloads and presented data for 143 paths. Our results show that within a workload, paths have disparate characteristics. Summarized characteristics at coarser granularities blend these characteristics and may draw imprecise conclusions.

Chapter 4

How to specialize – Leveraging Program Analysis to Extract Accelerators from Whole Programs

The previous chapter analysed programs at the path granularity for suitability for specialization. This chapter evaluates the impact on performance and energy when frequent paths in the program are targeted for hardware specialization using a speculative execution model. While program paths represent dominant program behaviour for specialization, they may incur overheads due to naive speculative execution. We design an abstraction, called Braid, to address these challenges. We present our results for specializing programs using paths and Braids as well as their utility when applied to automated high level synthesis. Overall, we coarsen the granularity of offload to accelerators to achieve improvements in performance and energy efficiency.

4.1 Introduction

Often accelerators require an understanding of the specialized algorithm and program structure [139] to enable appropriate offload region formation. Since programs include complex control flow and have many possible execution paths, it is challenging to profile and compose an offload region for the accelerator. Real world examples of offloading a stable code region required that the API be redefined [137]. Recent works [152, 142, 128] have leveraged compiler intermediate representation (IR) to aid architectural simulation and enable comparison of different accelerator architectures. Such works still seem to largely leave unanswered the questions, "what code region in the original program should be specialized?" and "how to prepare it for offload?". Conventional profilers and analysis tools, e.g. gprof or trace analysis, are unsuitable for this task. Their scalability and accuracy is impeded by the structure of typical programs, which tend to have irregular control and dataflow. Compilers

for coarse-grained reconfigurable array (CGRA) like fabrics [64], while successful for simple inner loops, find it challenging to prepare effective offload regions with many flows of control. VLIW compiler research has studied the formation of scheduling regions larger than a basic block by exploiting hardware predication (e.g., hyperblocks). Unfortunately, defining high quality regions depend on heuristics [10] and predication hardware; in Section 4.2 we analyze the specific requirements of accelerators.

Our Insight and Proposal:

We demonstrate that an effective approach to building accelerators requires dynamic profiling for accurate early-stage exploration of the specialization tradeoffs between 1) targeting few code paths for efficiency and 2) coverage that seeks to offload a larger fraction of the application. We develop *Needle*, an LLVM framework that leverages dynamic program analysis profiles to identify "what paths to specialize" in a program, merge paths and prepare them for acceleration. We study existing region formation algorithms (see Section 4.2) and demonstrate the efficacy of Ball-Larus paths [15](BL-Path) for forming accelerator-friendly regions.

In **Step 1**, *Needle* analyzes programs to profile and construct two types of code regions for accelerators to target: BL-Paths and Braids. *BL-Paths* are single entry, single exit regions which represent a single flow of control. A control flow divergence leads to a jump back to the CPU and a reversion of externally visible program state.

Unfortunately, programs may execute a large number of paths (over 100K in the workloads we study) with no single path dominating execution. This may lead to accelerators frequently switching between different paths, imposing a high overhead. To achieve high coverage we introduce a new program abstraction, called *Braid*, that takes advantage of the observation that many frequently executed BL-Paths tend to have the same basic blocks. *Braids* merge overlapping BL-Paths and seek to achieve high coverage. While BL-Paths revert to the CPU on any control flow divergence, the intuition behind Braids is that the program exits from the accelerator to the CPU only when the control flow appears to break out of a hot region of code. These regions are single-entry, single exit but incorporate multiple flows of control. BL-Paths and Braids are also inherently acyclic, we employ path prediction to identify loop back edges and construct larger regions for accelerator offload.

In **Step 2**, *Needle* prepares the BL-Path and Braid abstractions to run on the hardware accelerator by generating *software frames* to handle control flow along the path and enable speculation on accelerators. This reduces the accelerator's reliance on the power-hungry OOO processor. Software frames support guarded execution on the accelerator [135]. *Needle* creates frames by hoisting instructions in a BL-Path above the branches in that BL-Path, fusing them to create coarse-grained atomic regions of offload. The branches are converted into asynchronous guards that determine whether speculation was successful. *Needle*'s frames permit all operations to be speculative, including memory operations. Software frames are

accelerator micro-architecture independent and do not depend on specific hardware features (e.g., store buffers [68, 146]). *Needle* regulates when the guards checks are inserted along the path to reduce the overheads of speculation failure while raising the number of hoisted operations to increase instruction parallelism.

4.2 Scope and Related Work

Needle is a profiling and compilation framework for sequential programs to target accelerators. A key impediment to implementing complexity effective hardware accelerators and precise code profiling is the control flow in sequential programs. Here, we study how *Needle* can help existing accelerators handle multiple flows of control in a program with software controlled speculation. We also discuss the challenges with existing compiler abstractions for often used for accelerators, superblocks and hyperblocks.

4.2.1 Hardware Accelerator Perspective

Spatial accelerators often use a dataflow-based approach, custom or reconfigurable hardware, and use a compiler to map computation to functional units. Prior work has shown that code regions with regular control flow and abundant data parallelism achieve high performance and efficiency [64, 58, 29, 132]. However, sequential code with limited data parallelism, nested control-flow, and irregular memory access patterns either compromise on performance [29, 58], or energy efficiency [56]. Additionally, fine grained offload regions require frequent interaction with the OOO processor, leading to further energy waste. Figure 4.1 discusses the design trade-offs in spatial accelerators. Prior approaches can be broadly classified into three designs: i) compound function units with minimal or no support for control flow. ii) non-speculative CGRAs that leverage predication to handle forward branches, and iii) speculative dataflow adopted by block architectures that can execute backward and forward branches.

The compound function unit approach fuses frequently used operations but terminates the fusion at branches, limiting offload granularity to basic blocks. Larger granularity offloads can be constructed by either leveraging an OOO processor's branch predictor [105] or using apriori profiling techniques with superblock construction. As observed by prior work [127], such architectures (e.g., BERET [68]) when integrated with an out-of-order processor, require frequent interactions with the processor and achieve low ILP. The non-speculative dataflow approach is prevalent amongst CGRAs that include support for predicating individual operations. This design converts control flow into dataflow dependencies through if-conversion and hyperblock formation. Many challenges remain including support for speculating on backward branches, conversion of nested ifs, occupation of hardware resources, and lengthening of critical path [10]. Dataflow architectures such as TRIPS [159] target whole programs and support forward and backward branches at the expense of increased hardware complexity.

		Compound Unit [39, 68, 139, 60]	**Non-Speculative CGRA** [64, 130, 157, 132]	**Speculative Dataflow** [159, 51]	**NEEDLE**
Target Code	*Granularity*	Basic blocks ~~③~~	Hyperblocks	Hyperblocks	BL-Path [15] or Braid
	Scope	Few ops ~~③~~	Inner loops	Full Program	BL-Path or Braid
Design	*Control flow*	None	Predication ~~①~~	Dataflow predication ~~④~~	Software speculation
	Branch prediction	Necessary for high performance.			Not required
	Granularity	Fine	Medium (hyperblock)		Coarse (paths)
	Accelerator⇔OOO	High ~~②~~	Medium (on block termination) ~~②~~		Low (on spec. failure)
	Speculative Ops.	None ~~①~~	None ~~①~~	Partial (No mem ops)	Full
	Rollback Granularity	Small	Medium (entire hyperblock)		Flexible (sub-path)
Compiler		Static	Static or Profile-driven		Profile-driven
	Profiling	Superblock	Path-trees [64]	None	BL-Path or Braid
	Code-gen	Superblock	Hyperblocks	Hyperblocks	BL-Path or Braid
① Full speculation support. ② High energy efficiency. ③ Coarse-grained offload. ④ Low hardware overhead					

Table 4.1: Comparison of sequential programs on spatial architectures

The TRIPS compiler relies on aggressive loop unrolling and flattening to reduce backwards branches and removes forward branches by forming hyperblocks. As a result of the increased hardware complexity, TRIPS exhibits only a 9% improvement in energy efficiency compared to an IBM Power4 superscalar processor at roughly the same performance [58].

Table 4.2: Control flow Characteristics

Branch⇒Mem. Avg. mem ops dependent on a branch		
1—10	10 Apps	hmmer,lbm, crafty, bodytrack, mcf, fluidanimate, ferret, sar-back, gcc
>10	8 apps	gzip, blackscholes, h264ref, swaptions, vpr ,sar-pfa-interp1, povray, sjeng .
Mem⇒Branch. Avg. mem ops a branch is dependent on.		
1—10	11 apps	art, parser, lbm, bodytrack, bzip2, freqmine, gcc, h264ref, mcf, blackscholes, mcf
>10	7 apps	crafty, gzip, vpr, sar-pfa, povray, swaptions, sjeng
Max. predication. #Bits required for hot path if-conversion		
>100	13 apps	povray,fluidaimate, bodytrack, ferret, hmmer,sar-pf, art,crafty,fft-2d,sar-back, sjeng, swaptions, bzip2,vpr.
Loops. Number of backward branches in hot function.		
>10	14 apps	streamcluster, art, gcc, ferret, blackscholes, mcf2k, mcf2k6, hmmer, bodytrack, crafty, povray, swaptions, bzip2, vpr

In the remainder of this section we summarize the challenges posed by control flow in real world programs. See Section 4.3 for workload specific statistics. Table 4.2 summarizes the number of predication bits required to if-convert the fully inlined hottest function. Nine workloads required > 100 bits of predication. Only four workloads required < 10 bits. Our predication statistics differ from prior work [130] because of aggressive inlining of call sequences. Prior work would need inter-procedural analysis prior to if-conversion to reveal this behavior. We studied the Hyperblock sizes for all the inner loops in our function assuming two bits for predication [64]. We find that Hyperblocks only attain $\simeq 2.2\times$ the basic block granularity. For four applications, sjeng, sar-pfa-interp1 and swaptions, hyperblocks increased block size by $6.3\times$. Overall, predication and Hyperblocks do not suffice to enlarge the offload region granularity and minimize interactions with the OOO processor. To understand whether speculation is required, we look at individual branches and classify them into two categories (see Table 4.2) `MEM-Branch` (`If`s that depend on

memory operations) and `Branch-MEM` (`If`s statements with memory operations dependent on the branch). Either case could introduce serialization and loss of ILP. We find that on average each branch includes > 10 memory ops per branch in 8 applications (including floating point intensive applications). In 18 workloads, the MEM-Branch ratio is > 1 i.e., the branch depends on at least one memory access. Both these statistics highlight the need for accelerators to implement a speculation framework including memory operations like an OOO processor. However, implementing hardware based speculation support is challenging in the absence of a notion of instruction or program order in dataflow accelerators. Thus we propose the use of *software based speculation*, where the compiler automatically inserts operations into the specialized region to support speculation (See Section 4.5 for details). Furthermore, workloads display varied characteristics and a unified hardware speculation strategy may be a poor fit. To summarize:

- Control dependencies limit the granularity of offload to accelerators and hence require frequent switchbacks to an OOO processor, reducing the energy benefits.
- Real programs have nested control flow, many backward branches and control interleaved with memory operations necessitating speculative execution support in accelerators.
- Finally, some current accelerators rely on a OOO processors to leverage speculation which makes it challenging to enlarge the offload granularity.
- Implementing speculation in accelerators is hindered by the need for a complex hardware mechanism to perform rollbacks which typically occur at fixed granularity.

Needle adopts a software based speculation approach to compile specialized regions for accelerators. *Needle* constructs atomic software frames from profiled hot regions. *Needle*'s LLVM framework extracts each hot region into a separate "frame", converting biased branches along the path into guards [135]. *Needle* generates the necessary rollback operations in software which enables workload tailored coarse-grained regions for offload. It improves energy efficiency and minimizes reliance on the OOO processor. *Needle* supports full speculation, including memory operations.

4.2.2 Compilers for VLIW processors

Needle's path-based offload region formation addresses a problem that at the high level seems similar to VLIW region formation strategies to handle control flow. Compilers for VLIW processors [5, 114, 113] pioneered the use of coarse-grained region formations by exploiting hardware predication. Hardware accelerators [64, 130] have primarily adopted "predication" to convert control flow dependencies into dataflow dependencies. Specialized regions offloaded to accelerators need to account for a large fraction of the dynamic instructions in order to

achieve energy efficiency[70]. Our observations show, heuristic based region construction such as Superblocks [117] and Hyperblocks [114] targeted towards VLIW processors may include infrequently executed operations. It is unclear whether effective, coarse-grained offload regions can be constructed by tuning the heuristics in a manner independent of the accelerator and program control flow.

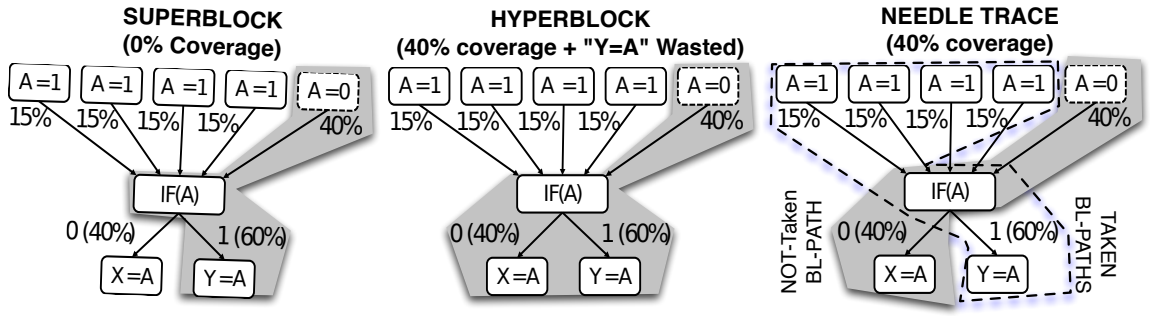


Figure 4.1: Superblock and Hyperblock construction for overlapped paths. % indicates the relative frequency.

Superblock and Hyperblock Construction Challenges Some existing accelerators have sought to target offload regions [68, 74] that are constructed from edge profiles of branches. The Superblock is an abstraction proposed by Hwu et. al [117] targeting Very Long Instruction Word (VLIW) and superscalar processors. They proposed a heuristic which uses edge profiles, frequencies of branch taken / not-taken edges, to construct a "macro-block" of frequently executed basic blocks. The constructed block has a single entry point but potentially multiple exit points. Side entrances to the Superblock, i.e branches which target blocks merged into the Superblock, are supported using "tail duplication" During Superblock construction, a local decision is made at each branch for which side of the branch to include. In the presence of overlapping paths, edge profiles may yield less than optimal results. Figure 4.1 illustrates such an example. The edge profile will lead to a Superblock that will *always* fail and trigger a side exit. Hyperblock construction may recognize the lack of bias and fold in both sides. However, this will lead to wasted blocks (since given `A=0` the Y=A is wasted). Using Ball-Larus path profiling [15] provides a precise characterization of executed program paths. *Needle* uses Ball-Larus paths (BL-Path) as a building block for offloaded regions. It is able to precisely identify the hottest path and construct the accelerator offload without waste.

Challenges in Achieving High Code Coverage. We find 5 benchmarks out of 29, 403.gcc, 181.mcf, 429.mcf, and swaptions that demonstrate "infeasible" Superblock construction for innermost loops. In these cases, the constructed Superblocks do not correspond to the actual paths taken by the program. Overlapping paths misleadingly cause

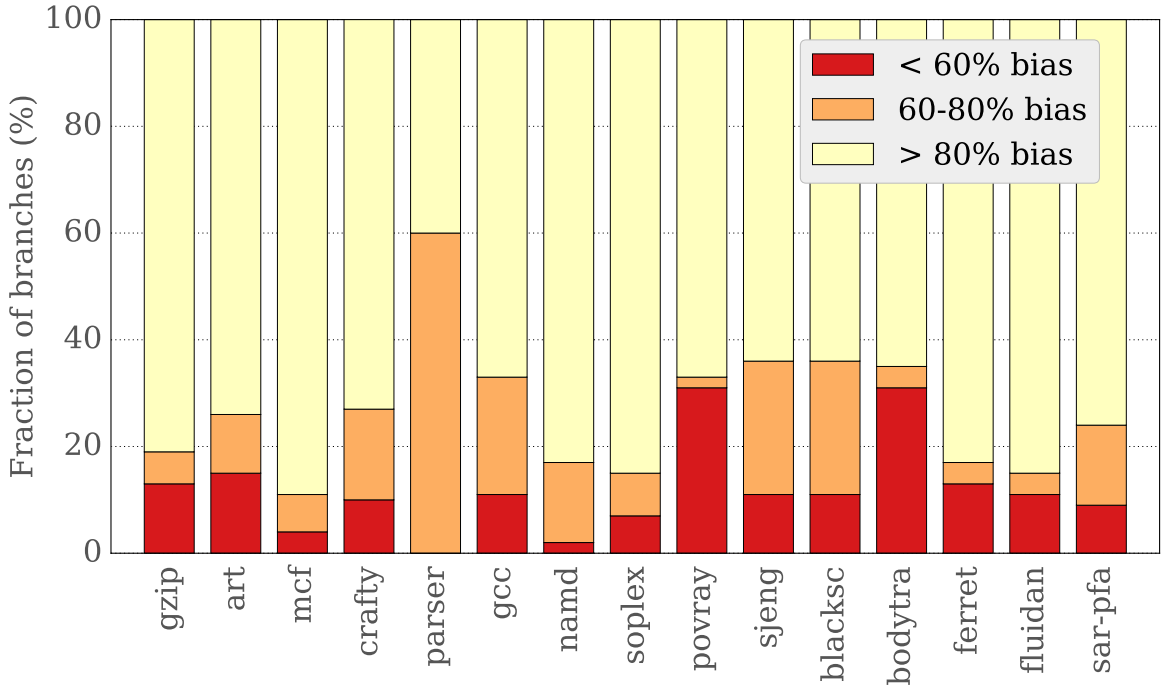


Figure 4.2: The distribution of biased branches in the application. Applications not shown in the plot have 99% of the branches with each branch > 80% bias.

individual block edges to become hot even though that particular sequence of hot blocks may never appear in program execution. Infeasible Superblocks degrade performance and provide no acceleration coverage.

When Superblocks and Hyperblocks are feasible, they still may not capture the hottest paths through the program. The local branch edge profiles may skew the ranks of hot basic block sequences, deprioritizing the offloading of hotter program paths. Ranking the paths in order of frequency, we find 6 workloads (453.povray, 458.sjeng, 181.mcf, bodytrack, swaptions and 401.bzip2) where the constructed Superblocks are not the hottest path. This implies that there exists some path for the same program region that is executed more frequently than the Superblock.

Challenges in Heuristic Tuning A key challenge for compilers seeking to leverage dynamic profiles is the tuning required for heuristic based approaches. To illustrate, we summarize the branch biases (i.e, how often a branch is taken) in the hottest function. The branch biases indicate to the compiler which successor basic block of a branch should be included (the taken or not-taken block). We find that in many workloads, 15 of 29, individual branch biases can vary significantly. Up to 24% of the branches have less than 80% bias (see Figure 4.2). In such cases, it is not clear how to tune the branch bias heuristic to achieve optimal coverage. Superblock and Hyperblock formation requires carefully tuned heuristics [9], multiple metrics including resource utilization and execution coverage need

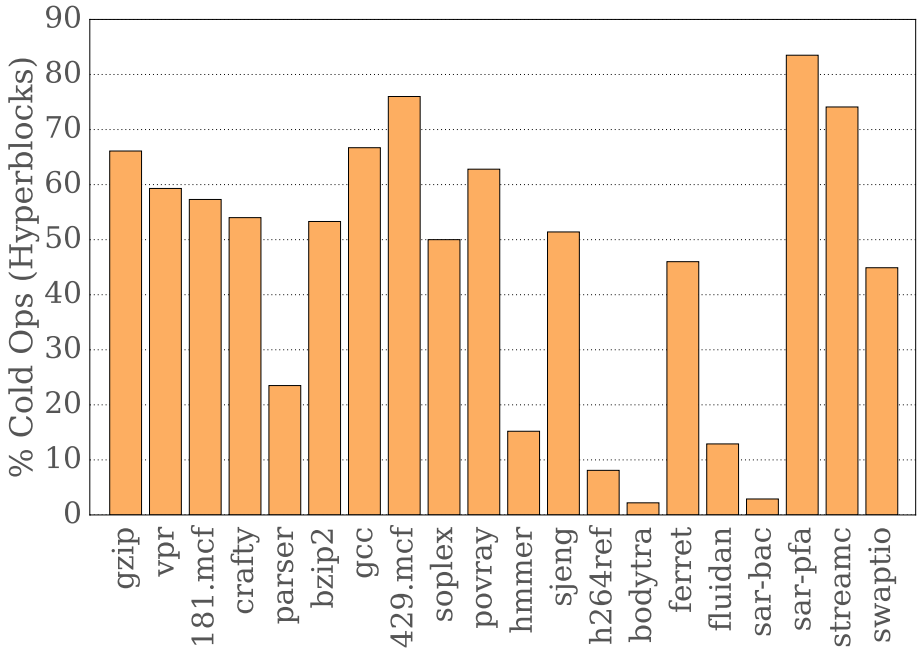


Figure 4.3: Fraction of "cold" ops included in Hyperblocks.

to be considered. More important, the heuristic must understand how the included blocks will interact with other included blocks and the runtime behavior of shared branches in the offloaded region. Figure 4.3 plots the number of operations that are part of the Hyperblock but are "cold", i.e infrequently executed. For a hardware accelerator, such operations tend to waste both 1) resources leading to area penalty for custom circuit and/or 2) energy in reconfigurable accelerators. The Hyperblock construction makes a local decision and thus may include wasted operations (See Figure 4.1 for an example). Without contextual program path information they may include blocks without including the other basic blocks in the path.

Dynamic Compilation for Accelerators Recent work has studied dynamic compilation to target a CGRA [176]. They however do not support control flow and map a single basic block to the CGRA at a time [176] . Control flow speculation is key to enabling coarse-grained regions that improve the effectiveness of a dynamic compiler. *Needle* focuses on identifying such coarse-grained regions in programs.

4.3 BL-Path Accelerators

In this section we revisit the notion of a Ball-Larus path (BL-Path) and contrast it against other region formation strategies (e.g., Superblock or Hyperblock). We identify the BL-Path characteristics that make them suited for accelerators. In particular, the BL-Path approach

will not encounter the same challenge as Superblocks in Figure 4.1 since it identifies not just the bias of the individual branch but the overall bias of the code path to reach the branch. This leads to accurate profiling of basic block hotness, formation of regions with guaranteed coverage of dynamic execution by *construction*, thus improving efficiency.

Ball-Larus path profiling [15] is used by *Needle* to obtain the initial set of acyclic candidate paths that summarize a program's dynamic behavior. The Ball-Larus method pre-processes the control flow graph of a routine to replace loop back edges with fake edges, one each from entry to back edge target and from back edge source to routine exit. Paths in the directed acyclic graph are enumerated bottom-up using dynamic programming, leading to unique ids for each acyclic path. Instrumentation is inserted to track which paths are executed at runtime. The dynamic profile of executed paths is collected, i.e for each unique path id we log the number of times it has been executed. Each unique id can be decoded to a sequence of basic blocks. We rank each uniquely executed path and select most suitable candidates for acceleration. *Needle* extracts the blocks in the selected path(s) into an offload function, adds support for software speculation and prepares it for hardware accelerator synthesis.

4.3.1 Path Ranking

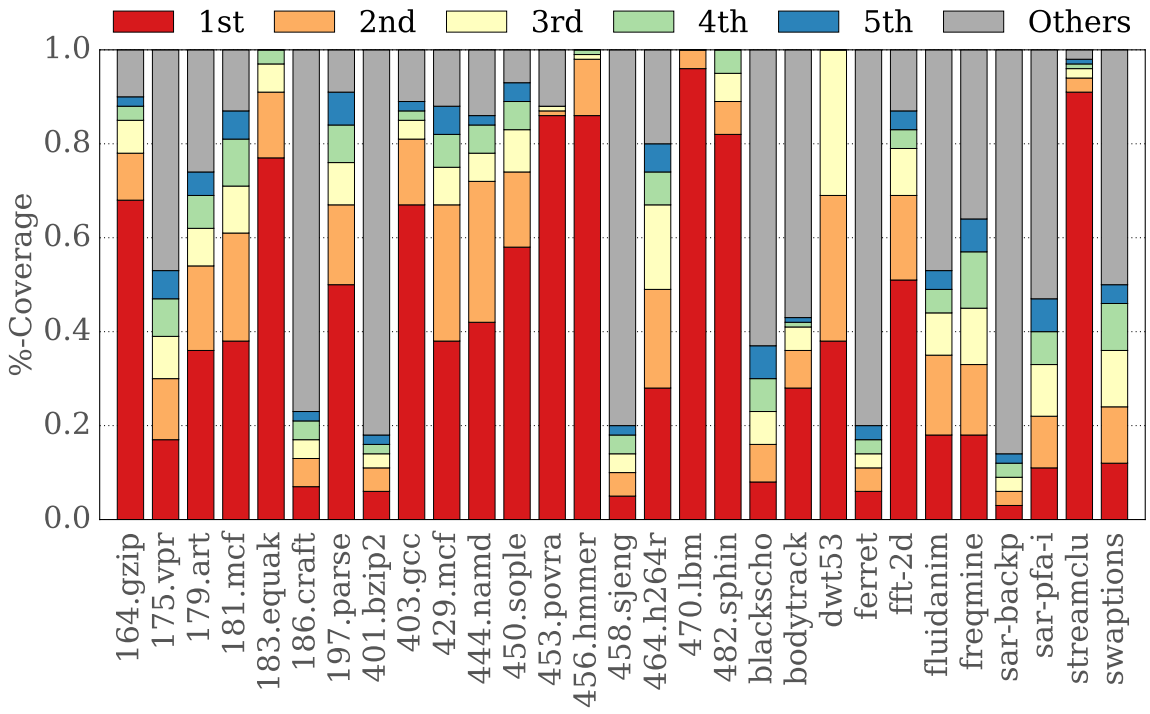


Figure 4.4: Path Coverage : Path weight (P_{wt}) by rank.

To rank the paths for hardware accelerators, we define a new metric, path weight (P_{wt}). It captures both the execution frequency of the path and number of operations. Eliding

instruction fetch is a primary source of energy efficiency in hardware accelerators [70]. Maximizing P_{wt} maximizes energy saved in the processor front-end. For the first order ranking of paths our weight metric assumes that all instructions carry the same weight, since instructions carry similar front-end energy costs in a processor. Latency of each instruction can be factored into the weight should the primary target be performance rather than energy efficiency. We also calculate Function Weight (F_{wt}), which accumulates all its constituent $P_{wt}s$. We present only the data for highest ranked function by weight for the sake of readability.

To understand the potential implications of selecting a frequency based metric, we profiled the time spent in the hottest ranked path using Linux's `pprof` (1500 samples/s) versus its parent function. We computed P_{wt}/F_{wt} as well as $P_{samples}/F_{samples}$ and compared the values as relative weights. In 12 of the 29 workloads we study, the sampling based profile indicated an average 10% increased weight; in 6 workloads we found a 15% decrease and in 4 workloads no change. The variability in sampling based profiling reaffirms our decision to use a frequency based metric that accounts for the dynamic power of the front-end.

4.3.2 BL-Path Properties

In the remainder of this section we present the characteristics of BL-Paths profiled across 29 workloads. Figure 4.4 shows the breakdown of dynamic instructions attributed to the top five paths amongst all paths in the highest ranked function. Table 4.3 presents the characteristics of the top five highest ranked BL-Paths.

Few BL-Paths Enable High Dynamic Execution Coverage Figure 4.4 shows the coverage (P_{wt}) of paths in our workloads. The stacks (bottom to top) represent the coverage of the highest to lower ranked paths. The average coverage (fraction of dynamic instructions) of the highest ranked BL-Path is 25%. In 18 of 29 applications the top path offers 20% or more coverage (See Figure 4.4). The median coverage using top five paths is 86%. Thus reasoning about paths allows us to understand the semantically different, yet frequent basic block sequences executed by a workload. For instance, as shown in Figure 4.1, it is desirable to precisely account for the frequency of the taken and not-taken sides based on which path is invoking the if-block.

BL-Paths Enable Coarse-Grained Offload (Table 4.3:C3) Table 4.3:C3 shows the average size (number of instructions) of the top five paths in the workloads. With a coarse-grained offload region, more computation is performed on the accelerator and fewer interactions with the host OOO processor. BL-Paths are acyclic; we investigate techniques to enlarge them further in Section 4.4.2. The median size across workloads is 65 operations. We have highlighted the applications that had a large number of branches in the path despite which the BL-Path was able to construct regions with 80+ operations (outliers are swaptions

– 438 and 458.sjeng – 50). The highlighted values in C4 indicate workloads in which the BL-Path traverses many branches. On 11 of the 29 workloads the highest ranked BL-Path spans across ≃13 branches. These sizes are larger than those observed with edge-profiled Superblocks in prior work [68]. An interesting workload is 401.bzip2, where the number of instructions in the top five paths vary significantly (29, 66, 371, 371, 194) with each path providing small coverage of the overall ($\sum_5 Cov. = 18$). Table 1:C7 indicates the number of memory operations that are part of the BL-Path and would be hoisted and become control independent when software frames are formed. The circled numbers highlight the workloads that benefit most from memory speculation.

Table 4.3: Path Characteristics

C1 : Exe. Paths **C2** : $\sum_5 Cov.$: Coverage of top 5paths **C3** : Ins. **C4** : Branch
C5 : Live Vals **C6** : Phi ops cancel **C7** : Mem.ops **C8**: # Overlapping paths

		C1	C2	C3	C4	C5	C6	C7	C8
	Name	Exec	$\sum_5 Cov.$	#Ins.	$\Diamond$	$\downarrow, \uparrow$	$\not\phi$	Mem	Ov.
SPECINT and SPECFP	164.gzip	80	90	33	4	7 , 5	4	4	6
	175.vpr	713	53	80	8	6 , 3	8	21	2
	179.art	1446	74	24	2	3 , 4	2	7	12
	181.mcf	48	87	30	2	5 , 3	2	7	2
	183.equake	7	100	88	1	9 , 5	1	32	1
	186.crafty	37K	23	49	7	8 , 3	7	4	31
	197.parser	10	91	33	3	6 , 2	3	6	2
	401.bzip2	54K	18	207	15	10, 6	15	29	15
	403.gcc	21	89	43	4	7 , 5	4	6	3
	429.mcf	41	88	21	2	4 , 2	2	6	2
	444.namd	57	86	90	2	18, 10	2	14	2
	450.soplex	67	93	33	2	7 , 3	2	7	3
	453.povray	375	88	137	8	7 , 4	8	17	21
	456.hmmer	61	100	105	6	12, 2	6	35	2
	458.sjeng	45K	20	50	9	3 , 3	9	8	43
	464.h264ref	43	80	49	4	11, 3	4	9	2
	470.lbm	2	100	232	2	3 , 2	2	45	2
	482.sphinx3	6	100	30	1	9 , 4	1	6	1
PARSEC and PERFECT	blackscholes	42	37	380	19	9 , 1	19	0	11
	bodytrack	732	43	68	4	10, 5	4	3	24
	dwt53	12	100	28	1	6 , 2	1	6	1
	ferret	556	20	98	9	7 , 6	9	2	10
	fft-2d	29	87	38	2	6 , 3	2	4	2
	fluidanimate	377	53	67	4	9 , 4	4	10	5
	freqmine	22	64	31	2	6 , 4	2	10	2
	sar-back.	539	14	85	9	7 , 5	9	6	3
	sar-pfa-interp1	53	47	146	14	14, 3	14	8	8
	streamcluster	42	98	35	3	6 , 4	3	6	2
	swaptions	11K	50	438	29	9 , 3	29	32	138

BL-Paths Have Overlapping Basic Blocks (Table 4.3:C8) A key concern with accelerator architectures is reusability or recurrence of acceleratable sections in the program. Prior work has evaluated this at the granularity of subgraphs containing a few operations [172, 39]. Here we present a methodical evaluation at the path granularity. Programs often execute a large number of paths in the same region. This often implies that many paths share common basic blocks. We quantify the overlap of basic blocks across the top five paths in each workload. Column C7 in Table 4.3 represents the average (geomean) block overlap. In 10 out of the 29 workloads we see that between 6–31 BL-Paths overlap (outlier: swaptions). In the other 19 workloads at least 2 paths overlap. BL-Paths enable precise accounting for common basic blocks.

Hardware overheads for BL-Path based accelerators (Table 4.3:C5 & C6) The number of live inputs and outputs determine the amount of data transferred to and from the accelerator. We summarize the results for the top five paths in Table 4.3:C5. These do not include the memory operations within the accelerator. Some workloads may have compute intensive regions with few live in and live out values (e.g., 470.lbm, 175.vpr, 183.equake, 444.namd). The workloads with coarse-granularity offload (C3 highlighted) have an average $\simeq$10 live ins and $\simeq$4 live outs). ϕ instructions in LLVM correspond to selection operator and incur significant hardware overhead [19]. When speculating on the control flow in a BL-Path (see Section 4.5), a frame is constructed, ϕs can be removed. It is interesting that in 10 out of 29 workloads, we remove multiple ϕs per branch. This implies speculation on just a few branches we can significantly reduce hardware.

4.4 BL-Path Expansion and Braids

In order to reduce execution migration between the host and the accelerator, we explore two approaches to increase the granularity of offload. *BL-Path Expansion* seeks to extend acceleration across back edges of loops, while *Braids* combine multiple paths for offloading.

4.4.1 BL-Path Target Expansion

BL-Paths are acyclic in nature. Sequencing paths across backward branches is essential to loop pipelining and extraction of data parallelism. Prior work [29, 116] has shown that outer loop pipelining is critical to finding parallelism in sequential programs. Acyclic regions superblocks and hyperblocks encounter the same challenge and typically attempt to grow in size via loop unrolling, branch target expansion and loop peeling [117]. *Needle* can construct offload regions by sequencing multiple BL-Paths using the dynamic execution profiles. We collected a path trace (sequence of path ids) during the profiling phase of the program. We then processed the trace and found that in many cases applications demonstrate a bias for back-to-back paths. We use the profile to guide which path to sequence next.

Table 4.4: Next Path Target Expansion

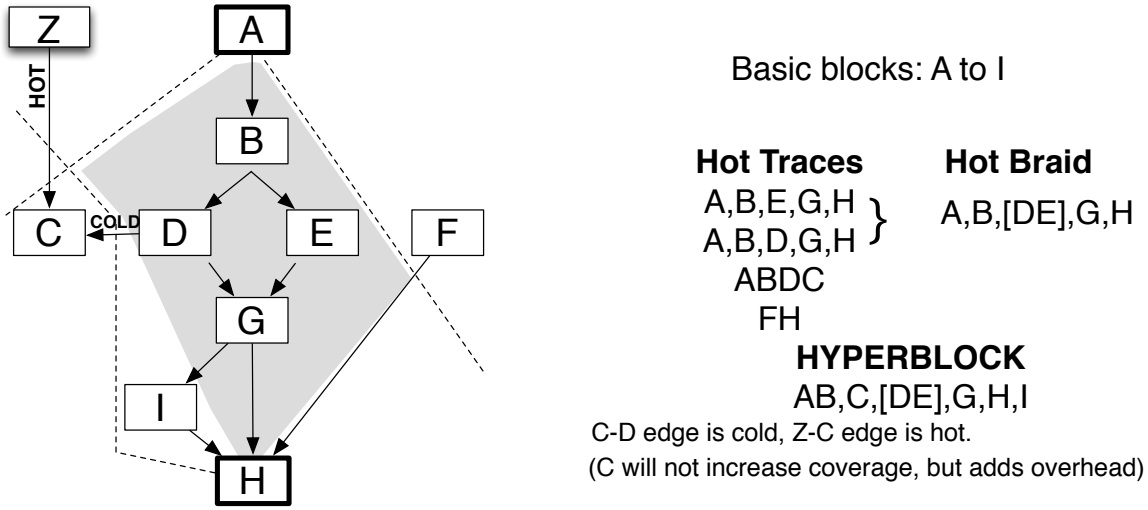
Path Seq. Bias	+Ops	Workloads
90-100%	68%	175.vpr 179.art 181.mcf 401.bzip2 403.gcc 429.mcf 444.namd 453.povray 456.hmmer 470.lbm 482.sphinx3 blackscholes dwt53 fft-2d streamcluster
70-90%	2×	183.equake 450.soplex 464.h264ref
<70%	73%	164.gzip 186.crafty 197.parser 458.sjeng bodytrack ferret fluidanimate freqmine sar-backprojection sar-pfa-interp1 swaptions

We summarize the data in Table 4.4. In 15 out of 29 workloads, a single path occurred in sequence more than 90% of the time. Of these 10 workloads repeated the same BL-Path. This enables us to enlarge the granularity of offload by a factor of 2×. The remaining 5 workloads (*.mcf, 401.bzip2, 403.gcc, blackscholes) where a different path follows in sequence we were able to expand the offload by a further 17%. Overall, the same path repeats in 17 out of 29 workloads, and the average offload unit can be increased in size by 72%.

4.4.2 Braids – Merging BL-Paths

Each BL-Path offload targets a specific sequence of basic blocks (which corresponds to a program path) with a single flow of control; any deviation requires accelerator rollback. Programs may have many paths which originate from the same basic block. It is challenging to determine exactly which path should be invoked. The penalty for invoking the wrong accelerator is rollback. A promising approach would be to merge paths to create a single offload unit. The key questions are "which paths to merge?" and "how to merge paths?". We present a new offload region abstraction, *Braids*, formed by merging BL-Paths thus achieving coverage equal to the cumulative coverage of the BL-Paths. We analyzed all the paths across our workloads and observed that in many cases of overlap, in particular the paths had a common start and end basic block. These paths diverged from the same point in the program and then re-converged. Consider Figure 4.5, the BL-Paths for this section are `ABDGH` and `ABEGH` (all hot paths start at block A and exit at block H). We construct a Braid by merging BL-Paths, and this requires the introduction of multiple flows of control within the region. The Braids are acyclic and thus introduce only forward branches. Braids include the basic blocks observed to have been executed and guarantees monotonic increase in coverage with each merged BL-Path. Braids are prevalent in program loops that have

multiple control flows within the loop body. Since Braids only merge BL-Paths that share the entry and exit block, live ins and live out values do not change. This permits the accelerator to transparently switch between the BL-Path or Braid configurations based on code coverage and area tradeoffs.



Braids offload multiple hot BL-Paths beginning and ending with the same basic blocks. In contrast, Hyperblocks also fold in cold blocks C and I.

Figure 4.5: Braid construction from BL-Paths

Relationship to Hyperblocks [114], Path-Trees [64] Hyperblocks are an extension to Superblocks where basic block successors to unbiased branches are merged for architectures that support predicated execution. Braids are a specific type of Hyperblocks that support multiple flows of control but always exit from the same block on completion. The heuristic based construction of Hyperblocks gives rise to multiple exits, which makes it challenging to bound the construction process. For example in Figure 4.5 the Hyperblock could include `C`. Additionally, Hyperblocks needs "tail duplication" for block `F` since it may merge paths that don't exit at F. Path trees are used by DySER [64]. In essence, they are Hyperblocks constructed from path profiles rather than edge profiles. They merge paths which originate from the same basic block and diverge. In Braids, the biased branches are converted to guards and enable speculation which is more energy efficient than predication when successful. While path trees originate from the same block, they may diverge to different basic blocks and have different live out sets based on the exiting blocks.

Braids improve accelerator code coverage (see Table 4.5:C3) We calculate the coverage-per-op ($\frac{C3}{C4}$) i.e., the fraction of dynamic execution covered by each operation in the Braid. This permits us to evaluate coverage by neutralizing the effect of a larger region size. Constructing Braids improved coverage-per-op for 17 applications (avg 0.85% of

Table 4.5: Braid Characteristics

C1 : Number of Braids **C2** : # paths merger to create a Braid
C3 : Code coverage **C4** : Ins. **C5** : Guards i.e., branches removed **C6** : IFs; branches introduced when merging paths **C7** : Live Vals

		C1	C2	C3	C4	C5	C6	C7
		#Braids	$\frac{\#Paths}{Braid}$	Cov%	#Ins.	◊	IFs	↓, ↑
SPECINT and SPECFP	164.gzip	48	1.5	80	39	3	3	8 , 5
	175.vpr	549	1.2	28	177	12	10	8 , 2
	179.art	84	2.3	36	21	1	0	2 , 1
	181.mcf	40	1.1	38	53	3	3	6 , 2
	183.equake	8	1.0	77	144	1	0	14 , 8
	186.crafty	388	2.0	6	28	5	0	6 , 3
	197.parser	7	1.4	49	56	1	0	5 , 2
	401.bzip2	3383	1.4	5	27	4	0	7 , 3
	403.gcc	9	1.8	73	50	1	6	6 , 7
	429.mcf	39	1.0	37	31	3	1	6 , 2
	444.namd	51	1.1	42	229	1	0	36 , 16
	450.soplex	47	1.3	57	30	2	0	5 , 3
	453.povray	8	11.8	85	54	1	1	2 , 1
	456.hmmer	47	1.1	85	61	2	0	16 , 1
	458.sjeng	296	1.7	27	2272	36	115	3 , 3
	464.h264ref	40	1.1	33	71	6	1	16 , 5
	470.lbm	2	1.4	100	511	1	1	3 , 1
	482.sphinx3	7	1.0	82	30	1	0	9 , 3
PARSEC and PERFECT	blackscholes	4	5.3	52	381	16	8	9 , 1
	bodytrack	19	6.0	27	45	4	0	12 , 2
	dwt53	13	1.0	37	23	1	0	9 , 1
	ferret	95	1.6	39	138	7	5	6 , 6
	fft-2d	23	1.2	51	39	1	0	8 , 1
	fluidanimate	74	1.3	25	117	8	8	4 , 4
	freqmine	21	1.1	17	43	4	0	6 , 2
	sar-backprojection	125	1.3	19	135	4	8	6 , 6
	sar-pfa-interp1	9	2.0	88	344	14	14	14 , 3
	streamcluster	31	1.2	91	47	4	0	5 , 2
	swaptions	85	3.0	38	1704	82	42	9 , 3

total dynamic execution per op). For 6 workloads the Braids improves coverage but also substantially increases the region size. For 444.namd, swaptions, 175.vpr, 470.lbm, 401.bzip, 186.crafty the BL-Path provided better coverage per op; Braid provided better coverage overall.

It might be beneficial to look beyond the hottest path and merge the lower-ranked paths to create hot Braids. This occurs in cases (eg.175.vpr, fluidanimate and sar-backprojection) where there is not much overlap between the hotter BL-Paths but there is a lot of overlap in

the lower-ranked BL-Paths making them amenable for merging. *Needle* provides a methodical framework to reason about this tradeoff.

Fewer guards than BL-Path ⇒ Fewer speculation failures When merging paths, Braids introduce multiple flows of control within the region. This effectively reduces the number of guards that would have otherwise been needed by the constituent paths individually. On 12 applications the Braids have 2× fewer guards than the hot BL-Path. The reduction in guards directly correlates with how much overlap is between the merged BL-Paths. Fewer guards mean that the offloaded region into the accelerator is less likely to fail (see Section 4.6 for the details). Outliers, 458.sjeng and swaptions increased the number of guards by 10× due to merging paths with minimal overlap, but each with path having many guards.

Braids enable memory speculation The control dependency enforced by the branch may limit memory level parallelism available in the dataflow graph; eliminating branches from the hot region will enable memory operations to speculative execute. We find that in 14 workloads the hottest braids have 0 memory operations dependent on a branch (in comparison to 11; see Branch-Mem in Table 4.2b). The number of workloads where the dependencies were greater than 10 was reduced to 4 workloads. In 6 workloads (179.art, 186.crafty, 197.parser, 401.bzip2, bodytrack, freqmine) the number of dependencies reduced to zero in the hottest Braid.

4.5 Execution Model

In *Needle*, candidate hot BL-Paths and Braids are converted into software frames for offload to the accelerator. They serve the same purpose as Traces within a Trace Scheduling compiler [110] or Superblocks [117] and hardware frames [146]. While Traces are multiple-entry, multiple-exit regions, Superblocks and hardware frames [146] are single-entry multi-exit regions with a single flow of control. *Needle* constructs single-entry single-exit regions but also support multiple flows of control. Software frames are atomic and coarse-grained, enabling effective speculative execution on accelerators. Software frames consist of three components (see Figure 4.6) the *frame*, a block of operations to run on the accelerator, the *guards* consisting of the control flow operations, and the *undo log* that captures values in locations modified by the frame to revert in case of speculation failure. All the branches within a frame (see ♢ `W>10` in Figure 4.6) are converted to guards. The compiler is permitted to move instructions within the frame. When multiple paths are merged to create a Braid (e.g., paths P and P' diverging at `p==0`), then the frame introduces multiple flows of control; we rely on non-speculative predication [64] being available in the accelerator. When a guard is triggered during a frame's execution, the externally visible state has to be reverted. No

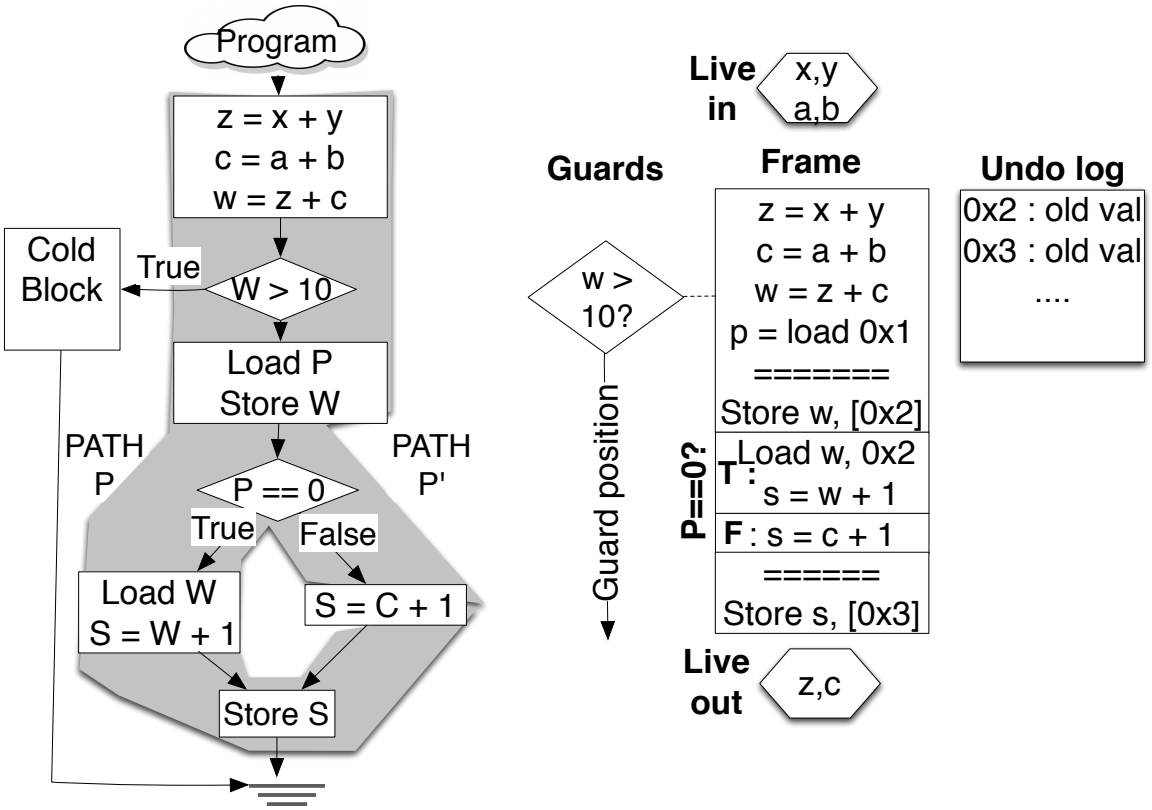


Figure 4.6: Frame construction from Braid.

architectural state is shared between the frame and OOO processor; live values and memory operations are the only form of communication to and from the frame. *Needle* implements the rollback using a software undo log populated by instrumenting stores.

When to invoke a BL-Path accelerator? As program execution approaches the entry basic block for the frame (see Figure 4.6) it has to determine whether to invoke the frame on the accelerator or to run on the host. Predicting that the accelerated path may actually fail due to a guard failure. This is challenging in BL-Paths since they may include multiple guards, and if any fail, the entire frame must be rolled back. This issue is not as critical to *Braids* as they merge paths and reduce guards. To resolve this we use an accelerator invocation history table that maintains information on program branch history prior to the accelerator path and determines whether the BL-Path accelerator should be invoked. In our suite, 9 applications always invoked the accelerator.

4.6 Evaluation

We have developed a cycle accurate simulator that models the host cores, the accelerator and data movement. The host OOO core pipeline is modelled in detail using macsim [81]. We

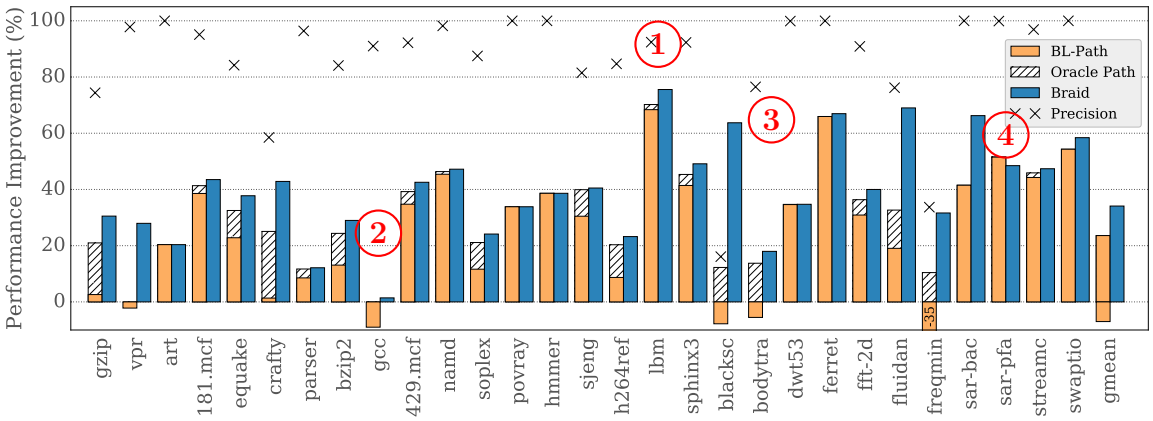


Figure 4.7: Performance Improvement

assume that the accelerator is uncore and transfers data to the OOO core via the L2 cache. We model a CGRA fabric similar to prior work [64, 131]. The CGRAs we model are capable of issuing memory operations and are cache coherent. We model the memory operations in detail. The OOO assumes a perfect branch predictor, but the accelerator simulation models guard failures and rollback overheads to obtain a conservative estimate of speedup.

Table 4.6: System parameters

Host Core	1 GHz, Embedded. 4-way OOO, 96 entry ROB, 6 ALU, 2 FPU, INT RF (64 entries), FP RF (64 entries)
L1	64K 4-way D-Cache, 2 cycles. LLC NUCA 8 banks, 20 cycles, MESI.
En.	Mcpat [102]; ARM 1Ghz Template.
Coarse-Grained Reconfigurable Array (CGRA)	
16×8 function units. 16 cycle reconfig. Energy Parameters (Dynamic) Network (12 pJ/switch+link), Function units (8 pJ/INT, 25pJ/FPU), 5pJ latch	

4.6.1 Performance

NEEDLE automatically identifies and offloads coarse-grained Braids to achieve a average performance improvement of 33% (max: 68%) across 29 applications.

We evaluate the performance of coarse-grained offload regions (BL-Paths and Braids) that have been *automatically curated* from large workloads using *Needle*. Figure 4.7 shows the improvement in performance (% reduction in cycle count) for the highest ranked BL-Path and the highest ranked Braid. For the BL-Path, we quantify the performance of a) an Oracle predictor and b) a branch pattern based predictor (see § 4.5). The precision of the predictor is displayed on the Y-scale for clarity and brevity.

With a dataflow oriented accelerator, performance benefits are primarily obtained by exploiting Instruction Level Parallelism (ILP) present in the workload. While an Out-of-Order processor can uncover more ILP than an In-Order processor, they are limited by available physical resources (eg. number of arithmetic units). Dataflow accelerators allocate a single functional unit for each operation in the dataflow graph. Thus the entire ILP of the workload can be exploited for performance. Other sources of performance include the elimination of fetch and decode for each instruction in the dataflow graph. The accelerator eliminates the need by configuring the spatial fabric prior to execution.

The performance of the BL-Path as offload is a tradeoff between the expected benefit of offload versus penalty of rollback on a guard failure. The penalty incurred in terms of performance includes the cycles spent in the accelerator as well the re-execution of the offload on the host. The performance benefit gleaned from mining more dataflow parallelism and eliding certain operations (e.g., bit manipulation) may be squandered by overly greedy invocation. Furthermore, many workloads have a small margin for error due to the constrained nature of their dataflow graphs. In this work, we explore the bounds of offload potential by assuming a) guard failure detection only at the end of the accelerator invocation and b) CPU re-execution of a failed BL-Path.

Overall, for offloading BL-Paths we see a mean performance improvement of $\simeq$24% across 24 applications. Five paths suffer from performance degradation, with an average of 7%. We discuss the results presented in Figure 4.7 with respect to their workload characteristics.

High Potential (Predictable and High ILP) ① Workloads with high ILP and coarse offload regions (e.g., 470.lbm, ferret, swaptions and sar-pfa-interp1) show significant performance improvement (up to 68%). While some workloads may be complex (e.g. swaptions with 11K paths), they demonstrate regular predictable behaviour (avg precision 98%) that translated to large gains as almost no work is wasted by wrong path rollback. Other applications such as 179.art, 197.parser are predictable, though they have lesser potential due to the nature of the computation being inherently sequential.

Access Execute Decoupling ① For 470.lbm, we also find the separation of memory access and computation paths. The inner loop path selected for acceleration is compute dominated. The effective decoupling of access and execute can yield better performance for the accelerated region. Recent research [36] has shown the utility of access execute decoupling for accelerator architectures. However, their work demonstrates automated prefetching for simple kernels [142] only. Using *Needle* based abstractions may lend to easier partitioning of access and execute phases as observed in 470.lbm.

Low margin for error ② 403.gcc has no ILP that the accelerator can take advantage of to improve performance. Thus, the Oracle predictor does not improve performance, and

the branch history predictor degrades performance with wasted rollback being executed on the CPU. 175.vpr suffers from a similar problem due to the offloaded region being only 7 operations in size; there is no performance benefit of the accelerator. Though it is highly predictable (97%), the rollbacks for the 3% of failed executions contribute to a net 2.2% degradation.

Pathological unpredictability ③ Due to a combination of data dependent loop branches and aggressive loop unrolling (4×) freqmine, bodytrack and blackscholes degrade performance, as the branch history patterns are insufficient. One possible approach to mitigate the issue would be to use loop fission to segregate unrolled iterations where the loop bounds are determined by data dependent values.

For Braids, we observe a mean 33% performance improvement. Note, there is low potential for degradation, as Braids have fewer guards than paths and include control flow in the offload via if-conversion. In all but one workload (sar-pfa-interp1), the highest ranked Braid provides equal or greater performance than a BL-Path with the Oracle predictor. In this workload, the BL-Path and Braid target different code regions with varying ILP.

Apples to Oranges ④ sar-pfa-interp1 is one of the 3 workloads where the highest ranked BL-Path is not part of the highest ranked Braid. This implies that the coverage of lower ranked BL-Paths contribute to an overall higher ranked Braid. In this case, the Braid has more than 2× the number of operations and provides a higher overall energy reduction (88% vs 79%) for the BL-Path. *Needle* provides a systematic approach to study offload granularity.

4.6.2 Energy Evaluation

Needle constructed Braids in a programmer independent, automated fashion which reduced the energy consumption by 20%. Figure 4.8 shows the net energy improvement for offloading Braids. While the performance improvements that can be obtained from coarse-grained offloading depend on the criticality of the dataflow graph, energy consumption can be reduced on a per operation basis due to the elimination of a processor front-end [70]. We present only Braids in this section due to their higher performance and simplicity. We present net reduction for the entire workload region (hottest function) in contrast to the Braid only to highlight the utility of our tool.

The coverage of the Braid is indicated on top of each bar. The reduction in energy consumption is commensurate with the coverage apart from a couple of application pairs. For 453.povray and 456.hmmer, the coverage is the same however the net energy reduction for 456.hmmer is lower. ferret has higher coverage than dwt53 yet has a lesser impact on energy reduction. Both pairs are due to the increased dataflow graph dependencies (2× in ferret ↔ dwt53) and (1.4× in 453.povray ↔ 456.hmmer).

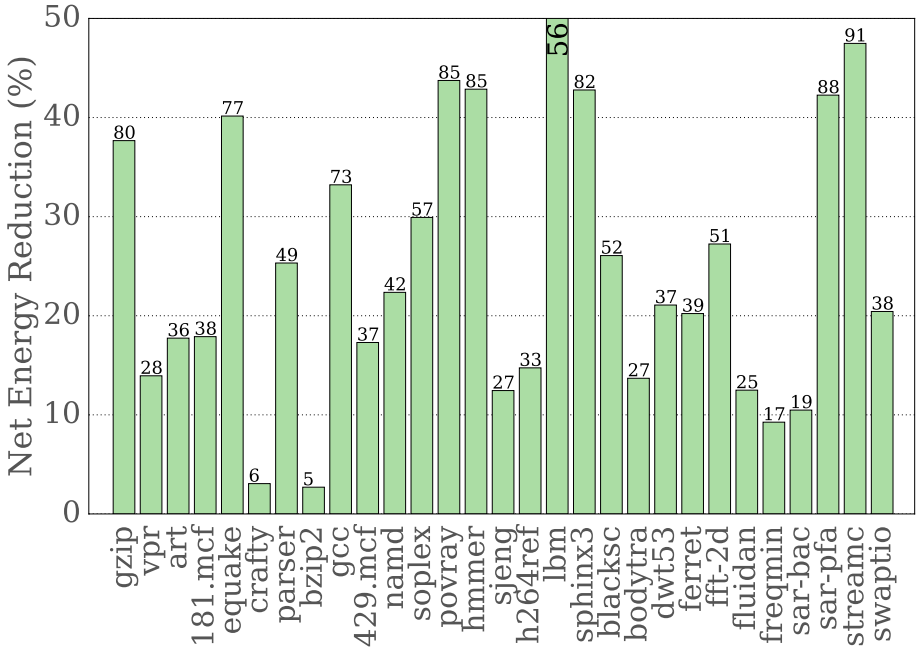


Figure 4.8: Net Energy Reduction for Braid

While 401.bzip2 comprises $> 3K$ Braids, we find a single Braid (5% coverage) reduces energy by 2.7% overall. Similarly, just one Braid for 186.crafty reduces energy by 2×. Both these Braids offer much larger energy reduction per coverage than other workloads. The other workloads that demonstrate similar behavior are fluidanimate, freqmine and sar-backprojection. In general, floating point workloads enjoy larger reductions in energy consumption due to reduced cost of floating point operations on the spatial fabric as well as simpler control flow structure.

HLS for *Needle* identified Braids *Needle* also enables high level synthesis tools [31] to target irregular workloads. We target a Altera Cyclone V SoC Processor/FPGA that combines dual ARM cores with a tightly coupled FPGA fabric. We synthesize both BL-Paths and Braids. Data can be moved between the ARM cores and the FPGA through the cache coherent AXI bus, and the memory maps permit us to share up to 1GB of coherence space between the core and the FPGA. Our backend RTL generator includes support for only a subset of the LLVM IR (v3.8). We use this infrastructure for functional testing of the hardware accelerators and area tradeoff analysis.

We synthesize RTL for 22 workloads from *Needle* identified hot Braids. We target an Altera Cyclone V SoC device and find that for all but four workloads, the Adaptive Logic Modules used is less than 20% (total ≃85K). For these workloads the average is 38%, max utilization is for 470.lbm (72%) where double precision floating operations are used. Modelsim simulations for power revealed that apart from three workloads all others consumed

5–60mW. The remaining three consumed 80mW, 175mW and 305mW for 444.namd, 470.lbm and swaptions respectively.

4.7 Conclusion

Needle is a automated tool chain that enables precise profiling, selection, and construction of "accelerator-friendly" regions. *Needle* is independent of accelerator architecture and released as free and open source software. We introduce a new program path abstraction, "Braids", that merges paths with many common basic blocks to help increase accelerator code coverage without impacting the hardware complexity and energy efficiency. Finally, we use Braids to enable energy efficient software speculation on accelerators. Overall, we enable offload of irregular workloads to accelerators and achieve a 34% improvement in performance and 20% reduction in energy. With *Needle*, we enable high level synthesis for irregular workloads where they previously failed.

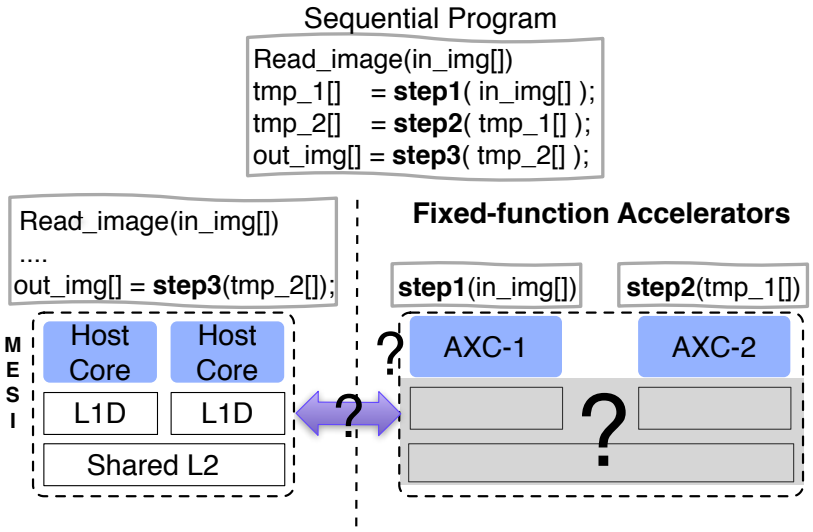
Chapter 5

Integration – Coherent Cache Hierarchies for Accelerators

With application specific hardware specialization, data movement can pose to be a challenging problem. In this chapter, we design a coherence protocol for hardware accelerators which optimizes data movement between multiple specialized units. We show that DMA transfers introduce overheads and traditional coherence protocols are unsuitable. We study multiple facets of the cache hierarchy including write optimizations and evaluate the tradeoffs between the pull-based model of the cache hierarchy and push-based model of the DMA.

5.1 Introduction

Current trends in microchip design indicate that to meet power budgets designers would have to power down an increasing fraction of the components on a chip [119]. Prior research in both industry and academia [133] have sought to address the power challenge through the use of hardware specialization (aka., accelerators) that target specific program regions. Prior work [171, 64, 66] has sought to extract accelerators from program regions such as functions and loops. We find that a key challenge to retaining the energy efficiency of accelerators is the data movement as the program moves around the chip seeking accelerators. This challenge is particularly relevant today as wire communication energy and cache access energy dominate compute energy in fixed function accelerators. Accelerators improve efficiency of the compute datapath to the level at which the overall energy consumption is primarily dominated by memory operations [66]. It is important to optimize data access energy to ensure that we retain the overall energy benefits from the use of accelerators. In this work, our quantitative analysis focuses on fixed-function accelerators [66] to highlight the energy overhead of data migration between accelerators. However, we believe that the overall qualitative conclusions should extend to other accelerator types. Prior work has recognized the need to minimize



Top: Image processing application that processes the input in 3 steps. An application representative of these steps would be "histogram" in our benchmark suite. Bottom: Multicore chip with fixed-function accelerators. AXC-1 and AXC-2 are accelerators collocated in a single tile. The ? indicate the design questions addressed in this work: the cache hierarchy for accelerators, efficient data migration between accelerators, and the tradeoffs when transferring data to/from the host.

Figure 5.1: Offloading Sequential Program to Accelerators

short data movement between producer and consumer operations within an accelerator [139]; here, we focus on data sharing and data movement between accelerators.

We highlight the design challenges with an image processing application in Figure 5.1. The application reads an image and passes it through different step functions (`step1(), step2(), step3()`). In the contrived example, `step1()` and `step2()` are accelerated by the accelerators AXC-1 and AXC-2 respectively while `step3()` continues to run in software on the host processor. The key questions are i) how are data elements `in_img[]` and `out_img[]` transferred between the host and the accelerator and ii) how do the accelerators, AXC-1 and AXC-2, exchange the intermediate data (`tmp_1()`).

Industry vendors, spurred by the need to reduce the latency overhead of host-accelerator communication, have developed coherent direct memory access (DMA) engines [62, 24, 83] that transfer data directly from the host's LLC into the accelerator's explicitly managed local storage (scratchpad). While this approach is suitable for computationally intensive accelerators with few memory operations it is inefficient when fixed-function accelerators offload functions that share data with each other. The DMA-based approach requires multiple data transfers between the accelerator's scratchpads and the host, expending significant cache and interconnect energy. The DMA overhead is particularly notable compared to the "un-accelerated" system in which the inter-function data reuse would be captured by the cache hierarchy of the core running the sequential program. Also accelerators that are not compute

dominated may experience performance overhead due to the DMA transfers on the critical path. Recent research [171, 187] has recognized the importance of collocating accelerators to minimize data transfer overhead. They integrate fixed-function accelerators at the host, and the L1 cache is shared between the host and the accelerators. Sharing the L1 cache helps minimize the overhead of data transfers between accelerators and enables participation in coherence. However, the shared L1 cache also introduces a challenge. We show that accelerators exhibit different memory level parallelism. To ensure that we retain the energy and performance benefits of the fixed function accelerators [66], the cache hierarchy needs to be optimized. We also show that the load-to-use latency and energy of the shared cache might minimize the benefits of fixed function accelerators. Additionally, the datapath of an in-core accelerator is limited by the bandwidth afforded by TLB, L1 cache, and register file ports of the core.

In this work we focus on fine-grain offloading of multiple functions from a sequential program. We find that cache and coherence protocol optimizations are even more important today since with fixed-function accelerators the data movement constitutes the dominant overhead. We propose, *FUSION*, a multi-level coherent cache hierarchy for accelerators. *FUSION* adopts a split organization in which fixed-function accelerators are grouped in a physical "tile" and implement a localized lightweight memory hierarchy with private L0 caches per accelerator (L0X) and a shared L1 (L1X) cache per accelerator-tile. The private L0X caches data and act like a scratchpad to capture the locality within an offloaded function, and ensure low load-to-use latency and energy for the memory operations. The shared L1X captures the inter-function temporal and spatial locality between functions offloaded to the accelerators. Typically sequential programs tend to include multiple functions suitable for acceleration like the image processing example and a lightweight multi-level hierarchy is needed to capture this locality while minimizing energy for the access to the cached data elements.

Coherence is maintained locally within the accelerator tile between the L0Xs and L1X using time-stamp based coherence protocol [155, 158, 121]. Implementing lightweight coherence between the private L0Xs and the shared L1X eliminates the need for "DMA-ing" (programmed explicit copying) data between accelerator cores. *FUSION* removes the ping-pong effect of moving data out of an accelerator scratchpad into the host's coherence space (at the shared L2) and then into another accelerator scratchpad. Overall *FUSION* saves bandwidth between the L2 and the accelerator tile and hence energy in the on-chip interconnect. In this work, we also propose *FUSION-Dx*, that further optimizes the inter-accelerator data sharing the accelerated functions similar to the example in Figure 5.1. In such cases *FUSION-Dx* leverages the accelerator tile's coherence protocol to proactively forward the dirty data from the producer accelerator's L0X to the consumer accelerator's L0X, thus saving write energy to the shared L1X and minimizing load-to-use latency for the shared data. Overall, we also discuss the energy benefits of the timestamp-based coherence

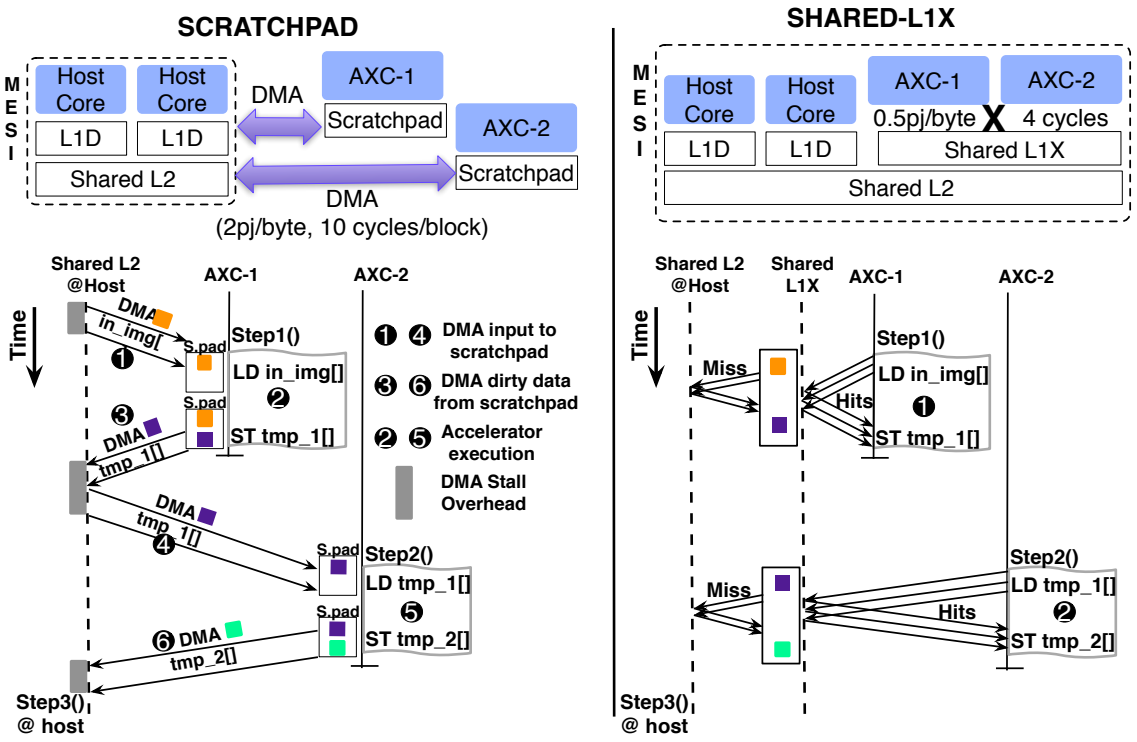


Figure 5.2: Left: *SCRATCH* Architecture. Per-accelerator scratchpads into which DMA transfers data. Switches to a different accelerator Right: *SHARED*. Shared L1 cache between the accelerators in a tile. The Shared L1 cache is kept coherent with the host multicore through MESI protocol. Host shared L2 maintains inclusion with the accelerators shared L1X.

which minimizes coherence messages, permits relocation of the TLBs to cache miss path, and enables proactive data movement optimizations.

FUSION is evaluated using a variety of applications drawn from the SD-VBS [170] and Machsuite [142] benchmark suites. *FUSION* eliminates the DMA transfers required to transfer data between accelerators and on average reduces energy by 2.5×. Like scratchpad based systems, *FUSION* exploits the temporal and spatial locality to minimize the accesses to the shared cache. Additionally, *FUSION-Dx* saves up to 17% of the dynamic link energy, for accelerators sharing data, by eliminating the shared L1X from the critical path.

The rest of the work is organized as follows: Section 5.2 describes the baseline architecture and the challenges in designing a cache hierarchy for accelerator. We also characterize the behavior of functions offloaded to accelerators. Section 5.3.1 provides an overview of the *FUSION* architecture and highlights the benefits compared to the scratchpad approach and shared cache approach, and Section 5.3.2 provides a detailed description of the architecture and discusses the details of the protocol and cache layout. Section 5.4 details the evaluation toolchain and finally in Section 5.5 we present the quantitative tradeoffs between the different cache organizations.

Lessons Learned

- **Coherency mechanisms help optimize data movement.** We find that localized cache coherence within the accelerator tile help optimize data sharing between accelerators while minimizing interaction with the host's LLC to save energy.
- **DMA transfers increase energy overhead.** We have analyzed the traffic caused by repeated DMA transfers and find that significant energy is expended in applications which share data between functions.
- **Need to eliminate request messages.** Caches operate in a pull-based mode and potentially expend significant energy in the network links that may offset the gains obtained from eliminating host-accelerator DMA transfers for compute intensive accelerators.
- **Write forwarding can reduce cache energy.** We find that accesses to the shared L1X can expend significant energy in accelerators and proactive transfer of data between accelerator caches directly can save significant energy.

5.2 Background and Motivation

In this section, we characterize the two baseline architectures currently explored by recent accelerator studies [66, 187]: *SCRATCH* and *SHARED*. Later in the section we present the characteristics of accelerators derived from sequential programs. We using the image

processing example shown in Figure 5.1 to highlight the overheads that may arise when running on *SCRATCH* and *SHARED* designs.

5.2.1 Baseline Architectures

Scratchpad per Accelerator (*SCRATCH*): Both ARM [62] and IBM [24] have enabled coherent access to the shared LLC from the accelerators. The coherence is limited to the DMA operations reading the most up-to-date data from the shared last level cache (LLC); writebacks from the scratchpad are managed using DMA. In Figure 5.2 the individual accelerators each include a scratchpad that connects with the shared L2 in the multicore through a coherent DMA controller. The scratchpad approach is well suited system which are either compute intensive (e.g., Cryptographic Unit [24]) or there is minimal interaction between accelerators (Cell SPE [164]). However, when functions or loops from sequential programs are offloaded to accelerators the shared data needs to be carefully managed as the program execution migrates between the different accelerators.

We illustrate the operation *SCRATCH* with the image processing example. To initialize the accelerator the host processor fills in the input data block at the shared L2 and DMAs the block to the accelerator (Steps ❶,❹). In the evaluation (see Section 5.5) we find that the proactive pushing of data into the scratchpad is one of the key benefits of DMA compared to a cache hierarchy which operates in a pull-based mode. We find that coherence request messages expend significant energy in links. The scratchpad size per accelerator introduces a tradeoff. A large scratchpad reduces the number of DMA operations and amortizes the cost of DMA but increases the load-to-use latency and load-to-use energy during accelerator execution. With the data accesses dominating the energy consumption in fixed-function accelerators [145], scratchpads tend to be small requiring repeated DMA. Another challenge is the interaction between multiple accelerators. As shown in Figure 5.2, when the program switches from `step1()` to `step2()`, DMA is needed to transfer the temporary data (`tmp_1[]`) from AXC-1's scratchpad to the shared L2 and then into AXC-2's scratchpad (❸ and ❹). Our analysis of data access patterns reveals that functions drawn from the same program tend to have significant data sharing (see Table 5.1 below).

***SHARED* between Accelerators:** Current work in academia [171, 64] and industry [70] are exploring the benefits of "at-the-core" accelerators that share a host core's L1 cache which enables the accelerators to efficiently transfer data between the host and the accelerators and between the accelerators. Unfortunately, the shared L1 needs to be sized to accommodate both software threads running on the host processor and the various accelerators. Since fixed-function accelerators expend minimal energy on the operation itself the load-to-use latency and energy of the shared L1 cache constitutes a significant overhead [145] in today's wire limited era [42]. In this work, we explore the benefit of

multi-level caches for accelerators and demonstrate how to efficiently maintain coherence in the hierarchy.

Figure 5.2: SHARED-L1X illustrates the *SHARED* architecture; we only show a single tile of accelerators. In *SHARED*, a tile of accelerators is connected to a multibanked L1 shared cache (Shared L1X) through a common switch. The accelerators incrementally load data into the shared cache as they run. As shown in steps ❶ and ❷, the accelerators are activated on a context switch once the basic register state is transferred. All accesses from the accelerators are issued to the shared L1X cache which appears as just another L1 agent to the coherence protocol and participates in the MESI operations. The host's shared L2 maintains inclusion with the L1X. Compared to *SCRATCH* all accesses from the accelerators are issued to the L1X which has a higher load-to-use latency and energy. Our *SHARED* architecture is similar to both Dyser [64, 143] in that accelerators share a common L1 cache that participates in coherence operations; in our model the host processor resides on a separate tile. Since caches tend to dominate the overall energy consumption in fixed function accelerators, earlier work [138] has recognized the need for customizing the cache size and organization for individual accelerators [145, 66]. However, there is a tradeoff between optimizing the shared L1X for data sharing between accelerators while supporting low load-to-use latency and energy per accelerator. Prior work [187] has managed the coherence between the host cache and the scratchpad using explicit instructions.

Fixed-Function accelerators from Sequential Applications: Table 5.1 lists the characteristics of the specific functions we accelerated from our benchmarks drawn from Machsuite [142] and SD-VBS [170]. We focus on the workloads in which multiple functions can be offloaded to accelerators and share data between the accelerated functions and the host processor. Please refer to Section 5.4 for a detailed description of our toolchain. The term "accelerator" is used in different contexts so we have listed the function names and their features to clarify to the reader the granularity; in this work we extract accelerators from loops and functions of sequential programs.

To choose the appropriate accelerators for this study we profiled the sequential programs on a Intel Core i5 processor. Table 5.1:%TIME lists the fraction of total time spent in the individual functions. In many of our applications (other than histogram) the critical path includes multiple functions and the functions are invoked repeatedly (possibly from different sites in the program). The breakdown of the operations within each accelerated function (%INT, %FP, %LD, %ST) is also presented in Table 5.1. The operation breakdown is obtained from the fixed-function hardware extracted from the dataflow graph of the accelerator (see Section 5.4). Given the dominance of interconnect energy ($\simeq$1pJ/mm/byte [42]) and cache energy [66] compared to operation energy (0.5pJ/integer add [12]) in fixed-function accelerators it is imperative to capture the spatial and temporal locality and thus reduce the energy for load and store operation. The sharing degree (%SHR) characterizes inter-

Table 5.1: Accelerator Characteristics

Function	% Time	%INT	%FP	%LD	%ST	MLP	%SHR
FFT							
step1	27.3	28	7.8	46.3	17.9	4.8	56.3
step2	8.8	52.1	0	29.9	18	4.0	99.5
step3	28.7	31.6	7.5	43.2	17.7	4.4	61.5
step4	8.5	49	0	31.8	19.2	2.9	50.8
step5	8.4	49	0	31.8	19.2	2.9	50.8
step6	18.4	20.3	3.3	53.8	22.6	4.3	19.4
Disparity							
padarray4	7.7	71	0	15.2	13.8	5.0	50
SAD	7.7	57.9	8.2	17.6	16.3	3.0	33.3
2D2D	15.4	62.8	0	24.9	12.3	3.5	49.7
finalSAD	30.8	22.8	0	71.3	5.9	5.7	47.9
findDisp.	23.1	32.7	32.3	30.7	4.3	2.2	31.4
Tracking							
imgBlur	14.3	52.8	15.1	24	8.1	2.0	58
imgResize	14.3	57.1	11.4	26.3	5.2	1.3	99.9
calcSobel	28.6	52.8	17.4	22.8	7.1	1.0	32.5
ADPCM							
coder	50	32.8	0.0	56.0	11.2	1.6	99.0
decoder	50	40.8	0.0	48.0	11.2	1.7	98.9
Susan							
bright	1.0	22.5	48.9	20.3	8.4	2.2	59.4
smooth	66.2	24.3	0.0	67.6	8.1	2.0	36.2
corn.	13.2	33.1	1.3	61.0	4.6	2.1	7.6
edges	20.6	32.6	1.6	60.3	5.5	1.9	12.3
Filter							
medfilt	74.4	48.2	0.0	49.1	2.7	1.6	14.2
edgefilt	25.5	41.3	23.9	28.1	6.7	4.1	23.1
Histogram							
rgb2hsl	48.2	22.1	51.8	20.7	5.4	3.5	8.3
histogram	3.6	40	0	53.3	6.7	1.0	100
equaliz.	3.6	36	0.1	59.9	4	1.0	66.2
hsl2rgb	15.7	26.3	40.8	22.1	10.8	3.1	75.0

See Section 5.4 for detailed description of our toolchain

accelerator communication. We define the sharing degree (%SHR) to be the fraction of cache blocks accessed by the accelerator that are also accessed by at least another accelerator. The %SHR here is reflective of the temporal locality in the original application between functions and manifests itself as data transfers between individual accelerators. In our applications, apart from the initialization functions (e.g., `rgb2hsl` in Histogram) in our accelerated functions the average %SHR is $\simeq 50\%$. Even in cases where only a small fraction of application time is spent (e.g., `paddaray4` in Disparity, `Equaliz.` in Histogram) the

%SHR degree can be significant (50%+). In such cases, if we do not ensure low overhead data transfers into the functions, the overheads could potentially dominate the overall energy consumption of the accelerator. The final columns, LT, presents the Lease Time (used in the ACC protocol, see Section 5.3.1) assigned to each cache block in the L0X per function per benchmark.

Summary

- When multiple functions from a sequential program are offloaded to accelerators the data sharing needs to be carefully managed.
- Fixed-function accelerators access both private and shared data and we need a multi-level cache hierarchy to effectively capture the locality of the accelerators and reduce data access energy.
- To effectively manage sharing between accelerators and maintain data across a multi-level cache hierarchy scattered across accelerators we need to employ low-overhead cache coherence that minimizes control messages.

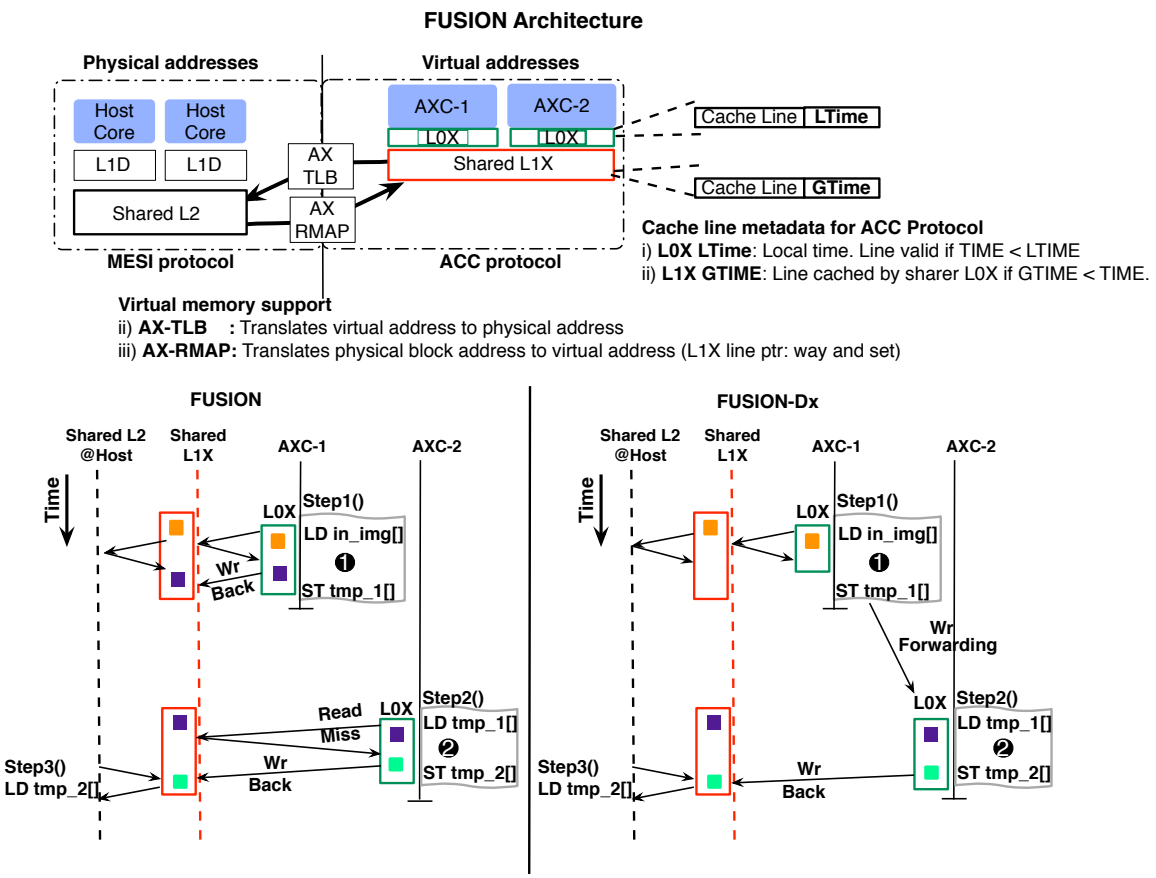
5.3 *FUSION*: A Coherent Accelerator Cache Hierarchy

5.3.1 Design Overview

FUSION is a multi-level coherent cache hierarchy for the accelerator tile that supports i) low load-to-use latency and low load-to-use energy for each fixed-function accelerator, ii) low overhead data sharing and data migration between accelerators within the tile, and iii) efficient data migration between the host and the accelerator. *FUSION* collocates fixed-function accelerators in a separate tile and implements a separate coherence protocol and cache hierarchy within the accelerator tile. The system can support multiple accelerator tiles, though in Figure 5.3 only a single accelerator tile is shown.

Figure 5.3 depicts the architectural details and we focus on the overall hierarchy herein to illustrate the tradeoffs between *FUSION* and existing designs, *SCRATCH* and *SHARED*. Private L0X caches, which can be sized independently, are provisioned for each accelerator within a tile to optimize load-to-use latency and energy [138]. All the private L0Xs are connected to a banked, shared L1X cache. The L0X supports write caching (unlike private caches in GPUs) to minimize write bandwidth and link energy. *FUSION* maintains coherence between the private L0X and shared L1X using the ACC (ACcelerator Coherence) protocol (details of ACC are presented in Section 5.3.2). The shared L1X is the ordering point and is a participant in the CPU's MESI protocol actions.

As shown in the timing diagram (Figure 5.3), like *SCRATCH*, *FUSION* caches the private data for the functions `step1()` and `step2()` in the L0X for energy efficiency.



FUSION: Baseline *FUSION* system that uses per-accelerator private caches (L0X) and a shared L1X to implement a multi-level hierarchy for accelerators. *FUSION-Dx*: Optimizes the accelerator coherence protocol (ACC) for direct Wr-forwarding between L0X of accelerators.

Figure 5.3: Top:*FUSION* Architecture. Bottom: Timeline for image processing example on *FUSION* and *FUSION-Dx*.

Like *SHARED*, *FUSION* enables the shared data (`tmp_1[]`) produced by `step1()` to be communicated efficiently to `step2()` without requiring intervention of the host processor (unlike *SCRATCH* which uses DMA). Finally, AXC-2 writes back the `tmp_2[]` which the host processor incrementally fetches as it runs `step3()`. Thus *FUSION* eliminates DMA transfers from the critical path, allowing fixed-function accelerators to incrementally fetch data when needed. More importantly, the extraneous DMA operations *between* accelerators are also eliminated.

FUSION-Dx further optimizes the data migration between accelerators to save energy. While *FUSION* eliminates the DMA required when execution switches to a different accelerator, (e.g. `step2()`) it requires subsequent read misses to transfer data from the shared L1X to the consumer L0X. We find that the control messages for requests fetching intermediate data (e.g. `tmp_1[]`) expends significant energy (see Section 5.5.2: Lesson 4) compared

to *SCRATCH*. *FUSION-Dx* (see Figure 5.3, bottom right) optimizes producer-consumer sharing found between the accelerators by proactively pushing the data from the producer's (AXC-1) cache into the consumer's (AXC-2) cache and eliminates cold misses. Overall, *FUSION-Dx* eliminates the writeback from AXC-1's L0X to the L1X, AXC-2's read miss and the L1X access.

5.3.2 *FUSION* Architecture

In this section, we describe the architecture of *FUSION* and illustrate how individual accelerator and host memory operations, as well as their interaction, are handled. As depicted in Figure 5.3 the architecture is segmented into separate host and accelerator tiles. Only the operations within a single accelerator tile are described for brevity. The accelerator tile operates on virtual addresses and maintains coherence between L0Xs and the shared L1X using a time-stamp based coherence [155, 158, 121] protocol.

Virtual Memory: In *FUSION*, the accelerators operate with virtual addresses while the host processor operates with physical addresses. This design eliminates TLB's from the critical path of accelerator memory operations and minimizes the energy consumption per memory access. Figure 5.3 (top) shows the point of virtual-to-physical translation in *FUSION*; we address the issue of synonyms in the Appendix. Process id (PID) tags are added to the L0Xs and L1Xs to ensure that accelerators executing functions from different processes can co-exist on the same tile. The private L0Xs and shared L1Xs are indexed using virtual addresses on AXC memory operations. Since ACC is a self-invalidation protocol there are no internally forwarded coherence requests that need to look up the L0X caches. We add a TLB (see AX-TLB in Figure 5.3, top) on the miss path of the shared L1X when transiting from the accelerator tile to the host tile; the translation is needed to index into the shared L2 and participate in MESI actions.

Since the host uses physical addresses, a reverse translation (physical to virtual address) is needed to handle forwarded requests from the shared L2 to the L1X. A naive solution is to include the virtual address for the memory access in the coherence control message. Unfortunately, this doubles the size of the control message for all host memory requests since it is not known beforehand which ones may need to be forwarded to the accelerator tile. In the current wire energy dominated era, this is not an energy efficient solution. Instead, we chose to expend area and dedicate a separate accelerator reverse map (AX-RMAP) per accelerator tile. The AX-RMAP maintains the physical address of the lines in the shared L1X; it is indexed using physical block address and stores a pointer to the shared L1X line. Since the shared L2's directory acts as a filter (sharer list indicates if an accelerator tile has cached the line), only few requests are forwarded to the accelerator and require and AX-RMAP look up. Section 5.5.6 evaluates the proposed address translation scheme for the accelerator tiles.

Accelerator Coherence (ACC) Protocol: ACC is a time-stamp based self-invalidation protocol similar to protocols proposed for multiprocessors and GPUs [155, 158, 121]. ACC adds two important write optimizations for accelerators compared to the earlier approaches: Write caching and Write forwarding. We describe the baseline writeback caching here and implement write forwarding as part of the *FUSION-Dx* system described below. ACC is a self-invalidation protocol and is a strict 2-hop protocol that requires no extra coherence messages over the baseline scratchpad systems. The reduced need for coherence makes it attractive for the purpose of accelerators from an energy perspective. Note that the primary purpose of ACC is to *enable data migration* between accelerators without host intervention (DMA) as the program context migrates and *not concurrent sharing* between different accelerators. The ACC protocol supports sequential consistency semantics for accelerator execution.

Figure 5.3, *FUSION* Architecture (top), shows the components of ACC. Only the accelerator cores within a single tile need to have a synchronized time-stamp register since ACC implements coherence only within a tile. A small time-stamp field (32 bits) is added to each cache line in the private L0X and shared L1X caches, as shown in Figure 5.3 (top). The local timestamp value (LTIME) in the L1 cache line indicates the lease time, essentially the time until which the particular cache line is valid. An L0X cache line with a local time-stamp *less* than the current system time is invalid. The global time-stamp (GTIME) in the L1X indicates a time by when all L0X caches will have self-invalidated the particular cache line.

Figure 5.4 (left) shows how ACC handles accelerator load and store misses. When AXC-1 issues a load request to the shared L1X, it requests a read-only epoch for the address (A) ending at time T=10 – ❶. The shared L1X receives AXC-1's load request (including the epoch request), records it and forwards it to the host-side along with a pointer to the L1X location (way and set). When transiting into the host tile, the AXC's load request is translated to a physical address. When the request crosses over to the host tile, it appears as a MESI load request from an L1 to the host's L2.

We have implemented a directory based 3-hop MESI protocol that takes the requisite actions to supply the requested data to the L1X. The data response includes a pointer to the L1X location so that on transitioning into the accelerator tile (which uses virtual addresses), the data response can update the he appropriate L1X entry. The shared L1X then replies to AXC-1 with the data and time-stamp of T=10 – ❷. The time-stamp indicates to AXC-1 that it cannot use this location beyond time T=10 – ❸. Subsequently, AXC-1 requests a write-epoch that expires at T=15 – ❹. To satisfy this write request the L1X implicitly locks the line and updates the L1X time-stamp to T=15. Subsequent readers or writers detect the locked line and simply stall at the L1X until the write lease expires and writeback completes. AXC-1 triggers a self downgrades and issues a writeback to the L1X – ❺ at T=15. When AXC-2 issues a read request – ❻ – the L1X finds the global time-stamp (T=20) to be

less than the current time (T=25) and checks if the writeback has completed and waits if necessary. Once the writeback is complete, the L0X responds with a read lease.

A key implementation decision is how to implement self downgrade. This requires checking for dirty lines in the cache. We implement the downgrade checks without sweeping the entire cache by using the time-stamps as a filter. Each L0X cache set includes a writeback time-stamp (the earliest write lease in the set); each accelerator includes a writeback time-stamp for the entire L0X cache (the shortest lease time-stamp amongst all the sets). These timestamps are used to filter the checks for the dirty lines. The writeback timestamps are updated whenever the dirty bits in the cache are updated.

Overall, accelerators can acquire both read and write epoch's on the cache line and the shared L1X distinguishes such cases; subsequent accesses stall on write epochs, while reads are permitted in conjunction with other read epochs. The epoch requests are fixed based on the expected latency of the accelerator invocation since it experiences minimal non-determinism (only due to memory hierarchy).

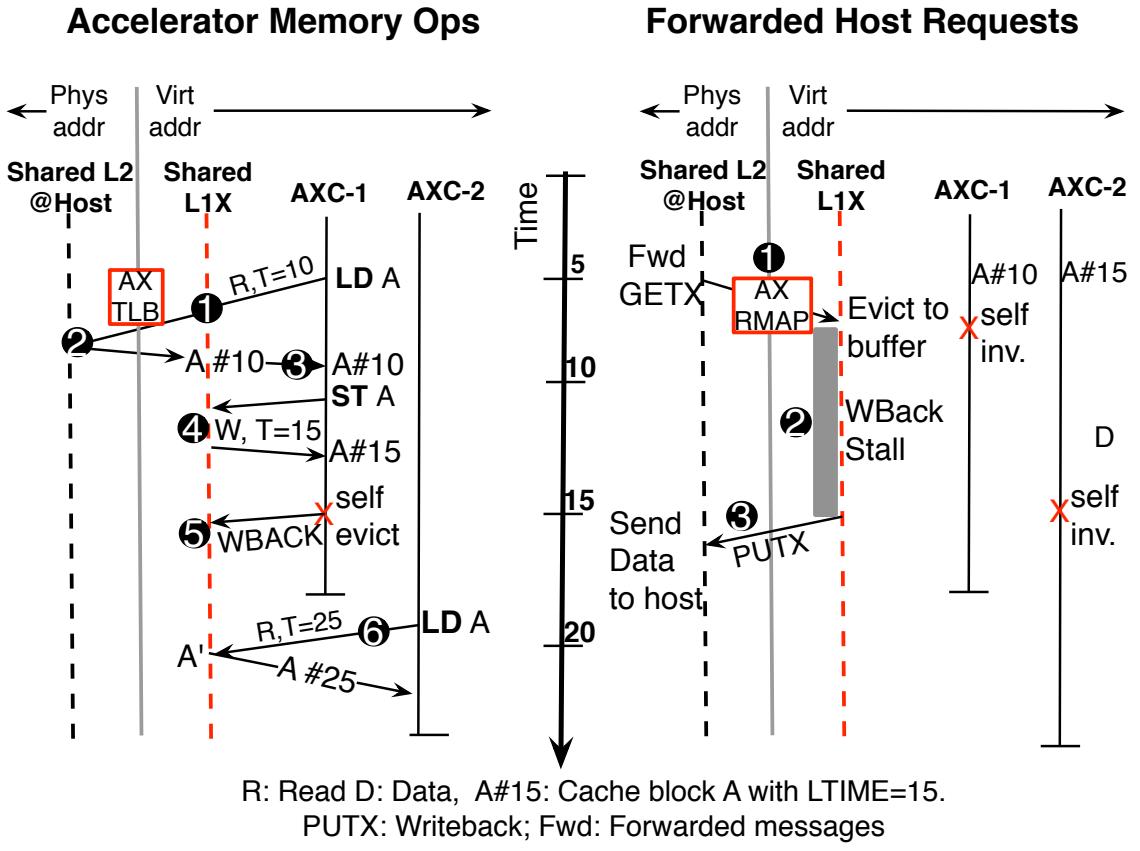


Figure 5.4: Left: ACC protocol servicing requests from accelerator and interaction with MESI. Right: ACC Protocol servicing forwarded requests from MESI.

Integrating ACC with MESI: Herein we describe how ACC participates in MESI operations. Exclusivity is maintained between the host processor tile and accelerator tile. The shared L1X always caches a block in exclusive state irrespective of the accelerator operation (read or write). When participating in the MESI protocol the shared L1X states map to a 3-state MEI protocol (M: modified, E: exclusive and clean, I:invalid). The ACC protocol responds in the same manner to all forwarded host coherence requests; i.e. relinquish ownership when the GTIME time-stamp expires and send an eviction notice to the shared L2. The shared L2 has perfect information on whether the accelerator tile is caching the block and eliminates any extraneous coherence messages that a cache would need to deal with as a result of silent drops from S→I. When a block is in I state in the accelerator tile, the L1X will not receive any messages from the shared L2 for that block.

Figure 5.4 (right) illustrates a forwarded request from the host processor. The host processor performs a store operation which results in a forwarded request to the accelerator tile. The forwarded request translates the physical address to the shared L1X pointer via the Accelerator Reverse Map (AX-RMAP) – ❶. A key benefit of the time-stamp based approach of the ACC protocol is that shared L1X will filter out the MESI *Fwd* messages and not forward them to the L0X. In the illustration, the forwarded request is received at time T=5 while the block is cached in the private L0X until time T=15. The *Fwd* message triggers an eviction from the shared L1X to a writeback buffer – ❷ – but the response eviction notice is stalled until time T=15. At T=15, a PUTX (eviction notice) is sent back to the shared L2 – ❸. The shared L1X uses the GTIME (see Figure 5.3, top) to ascertain when it is safe to respond to MESI protocol action and does so without involving the private L0Xs. In our workloads, we observed between up to ≃800 forwarded requests from the CPU to the accelerator tile for the whole workload. (*TRACK*:817, *ADPCM,DISP.*:≃500, others < 50).

***FUSION-Dx* – Extending ACC to support Forwarding:** A key overhead present in the cache based coherence model (pull-based) versus the scratchpad based DMA model (push based) is the extraneous control messages issued per cache miss. While the control message overhead is minimal in a multicore [155] or GPGPU [158] context, for fixed-function accelerators, they add notable overhead to the overall energy consumption. One particular source of inefficiency is the store-load forwarding shown in Figure 5.5:*FUSION*. AXC-1 writes to A – ❶ – but does not complete processing until later – ❸. In the meantime, AXC-2 wishes to read the data but has to stall until AXC-1 self-evicts the line. There are two inefficiencies in the contrived example: i) the coherence messages (write back, read request and data response) needed over the L0X-L1X link wire and ii) the stalled read on AXC-2. With accelerators exploiting all available operation parallelism and reducing compute energy consumption, a significant challenge in our workloads is the energy cost of the coherence messages. *FUSION-Dx* optimizes by providing a mechanism to directly forward data from AXC-1 to AXC-2. For MESI protocols such "proactive" forwarding

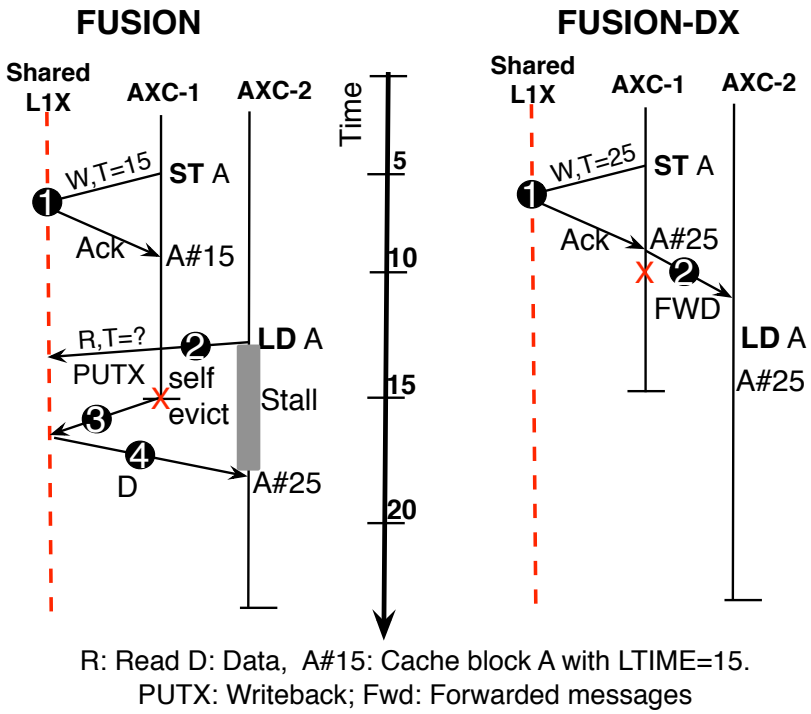


Figure 5.5: Left: *FUSION* without write forwarding. Right: *FUSION-Dx*. ACC protocol with write forwarding.

requires significant complexity and involvement of the coherence directory [91]. With the ACC protocol, forwarding simply involves self-eviction and forwarding the data with the already requested lease lifetime (see Figure 5.5:*FUSION-Dx*). Forwarding without informing the shared L1X is feasible with ACC since the L1X only tracks the lease epoch and is not concerned with the owner of the lease. The only challenge is to identify the stores that may benefit for forwarding and the producer-consumer cores. In this work, where the simulation infrastructure is trace driven (see Section 5.4), we post process the trace to identify the stores to be forwarded from the producer to the consumer accelerator.

5.4 Toolchain and Benchmarks

We have developed a detailed cycle accurate simulator that models the host cores, fixed-function accelerators and memory system faithfully. The host OOO core pipeline is modelled in detail using macsim [81] and the memory hierarchy using GEMS [115]. Table 5.1 characterizes the accelerators that we extracted; we assume that all accelerators derived from an application are collocated on the same accelerator tile.

Modelling accelerator cores: To identify and model the fixed-function datapath of the accelerator we adopt a technique similar to Aladdin [66]. The applications are

profiled using `gprof` which identifies the critical functions and the function call hierarchy. Based on the `gprof` profile, we identify top level functions for acceleration and ensure that accelerated functions are free of external library calls such as `malloc`. A dynamic trace of these functions is used to generate a constrained dynamic data dependence graph that includes program order constraints (control and memory dependencies). To model the fixed function accelerator we traverse the activity of the constrained data dependence graph on a cycle-by-cycle, generating any requisite memory operations in a cycle and stalling the appropriate operations as necessary based on the availability of allocated resources. We assume an aggressive non-blocking interface to memory.

Table 5.2: System parameters

Host Core	2 GHz, 4-way OOO, 96 entry ROB, 6 ALU, 2 FPU, INT RF (64 entries), FP RF (64 entries) 32 entry load queue, 32 entry store queue
L1	64K 4-way D-Cache, 3 cycles
LLC	4M shared 16 way, 8 tile NUCA, ring, avg. 20 cycles. Directory MESI coherence
Main Memory	4ch,open-page, 16GB 32 entry cmd queue, 200 cycle latency
Accelerator Cache Hierarchy	
Scratchpad	4 or 8KB RAM (ITRS HP)
Shared-L1X	64KB or 256KB, 16 banks. 8 way. (ITRS, HP)
Private L0X	4 or 8KB Cache (ITRS HP)
# of AXCs	2 (FILT) — 6 (FFT)
Link Energy Parameters	
	Accelerator-L1X (0.4pJ/byte), L1X-Host L2 (6pJ/byte)

Systems compared: We evaluate the following systems:
i) Oracle-*SCRATCH*: For the *SCRATCH* system we model individual scratchpads per-accelerator and generate DMA code to move data into and out of the accelerator. We assume a particularly aggressive oracle DMA implementation, and auto generate the DMA operations based on the memory accesses we observe in the dynamic trace of the application. We *only* DMA into the accelerator scratchpad read data and DMA out dirty data. All the benchmarks have working set sizes larger than the scratchpad (4096 bytes) and thus are segmented into "windows" of execution with DMA operations required for each window. We faithfully model the complete state machine of the DMA controller and assume that it

Table 5.3: Accelerator Execution Metrics

Function	KCyc.	LT	%En.	Function	KCyc.	LT	%En.
FFT (Cache/Compute Energy = 0.8)							
step1	25.3	500	34	step4	9.9	700	6
step2	7.1	700	4	step5	9.9	700	6
step3	23.5	200	35	step6	17.8	500	15
Disparity (1.6)							
padarray4	11.2	500	5	finalSAD	25.9	500	23
SAD	27.7	500	25	finalDisp	71.4	500	33
2D2D	34.2	500	14				
Tracking (0.5)							
imgBlur	9587	700	48	calcSobel	7358	720	34
imgResize	3837	770	18				
Histogram (2.7)							
rgb2hsl	38007	500	47	equaliz.	3250	500	1
histogram	3244	500	2	hsl2rgb	69671	500	50
ADPCM (9.7)							
coder	3453	1400	55	decoder	3364	1400	45
Filter (4.9)							
medfilt	48403	400	49	edgefilt	5663	400	51
Susan (3.1)							
bright	18.6	1000	1	corn	6328	1200	5
smooth	61496	1700	86	edges	18858	1700	8

KCyc.: Execution time (K Cycles), LT: lease time assigned to blocks, %En.: % of total accelerator energy. Cache / Core energy ratio shown in brackets beside benchmark name.

resides at the host's LLC i.e., no overhead for issuing DMA requests.
ii) *SHARED*: This system models a single shared L1 cache for all the accelerators in a tile. We collocate the functions from the same application in a tile and ensure that there is no inter-tile communication between offloaded functions from the same application.
iii) Finally, *FUSION* and *FUSION-Dx* include the full cache hierarchy with private L0Xs and shared L1X. GEMS was modified to add support for interfacing with the fixed-function accelerator cores and to model the ACC and MESI protocols along with their interaction in detail.

Energy Model: To model the energy of fixed function accelerators we use an activity count based power model from Aladdin [66]. We assume a 45nm ITRS HP technology. Cache energy is modeled using CACTI [125]; Table 5.2 lists the transistor types we assume for each cache. The L0X tag accesses include a 32 bit time stamp field check which is accounted for as an 15% energy overhead. In the benchmarks studied, we find that provisioning for 24 bits accounts for 98% of accelerator invocations; (*HIST.*, *FILT.* and *SUSAN* have functions

which run longer) and using 3 additional bits accounts for all invocations. We estimate link energy based on published figures/fusion [42] (1pj/mm/byte) and calculate wire length based on the area of the components where $Wire\ Length\ =\ 2 \times \Sigma_{i=1}^{n} \sqrt{Component\_Area_i}$, for each i $\in$ dataflow path). Table 5.2 lists our simulation parameters and Table 5.3 lists the % of energy spent in each accelerator and the ratio of energy spent in caches relative to compute.

5.5 Evaluation

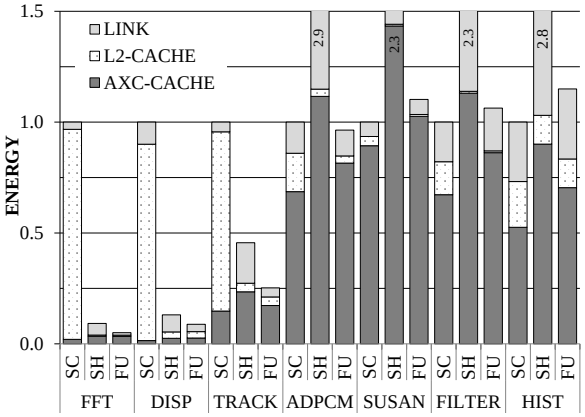
In this section, we evaluate the proposed *FUSION* architecture and present our results as a set of "Lessons Learned" in the design and implementation of a lightweight coherent cache hierarchy for accelerators. The *FUSION* architecture is compared to *SCRATCH* and *SHARED* designs and we present results for overall performance and energy before discussing the evaluation of i) Write-Back vs Write-Through at the L0X ii) accelerator cache sizes iii) address translation iv) the *FUSION-Dx* design, optimized for producer consumer data sharing.

5.5.1 Performance

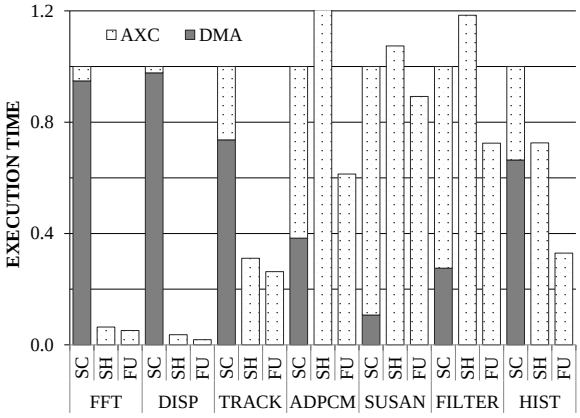
Lesson 1: Small shared cache (L1X) improves performance. The *SHARED* system, as described in [187], employs a shared 64KB cache to filter out accesses to the L2. Figure 5.6b, shows the cycle time of each system normalized to the SCRATCH system. *FFT, DISP., TRACK., HIST.* spend a significant amount of time (82%) in DMA transfers and the *SHARED* system outperforms the *SCRATCH* system (average 5.71×). For *ADPCM, SUSAN* and *FILT.*, where the DMA cycle time is less than 40% of the total for the *SCRATCH* system, the *SHARED* system degrades performance by 14%. For these 3 benchmarks the working set size is less than 30kB. The high spatial locality is captured in the *SCRATCH* system and offers low latency access to the data. Thus the higher penalty per access to the shared L1X in the *SHARED* design causes a performance degradation.

Lesson 2: Small private caches (L0X) are needed to optimize for load-to-use latency. The *FUSION* system builds upon the *SHARED* system with the addition of coherent private caches which are of the same size as the scratchpad in *SCRATCH* system. The *FUSION* system is able to capture the spatial locality for *SUSAN, FILT.* and *ADPCM* which is the cause of degradation in the *SHARED* system. The *FUSION* system improves performance over *SCRATCH* by 2.8×.

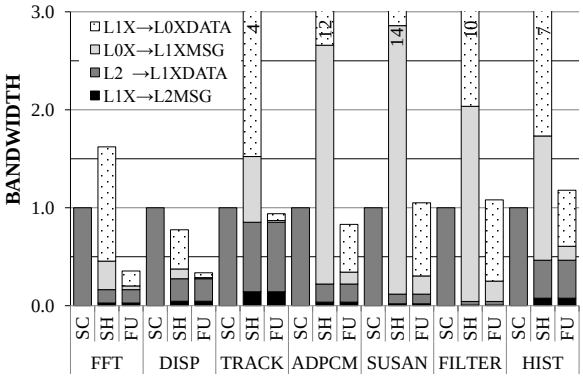
With *FUSION* we eschew the traditional "load-before-use model" of *SCRATCH* based accelerators. The *SCRATCH* ensures deterministic latency for memory accesses while *FUSION* may incur cache misses which need to fetch data from the shared last level cache



(a) Energy



(b) Execution Time



(c) Bandwidth Breakdown

	Energy	WSet(kB)	# DMA	DMA #lines	DMA(kB)	DMA / WSet
FFT	✓	15	640	79550	2486	165.0
DISP.	✓	163	963	130164	4067	25.0
TRACK.	✓	371	524	67450	2107	5.7
ADPCM	✓	28	7508	7508	234	8.3
SUSAN	–	21	3466	3466	108	5.1
FILT.	–	21	15034	15034	470	22.0
HIST.	–	1191	95613	95613	2988	2.5

(d) DMA Breakdown

Figure 5.6: Design tradeoffs in the accelerator cache hierarchy. X-Axis SC: *SCRATCH*, SH: *SHARED*, FU: *FUSION*. Y-Axis: All plots/fusion, lower is better and values are normalized to *SCRATCH* system. Note for the *SHARED* design, the L1X→L0XDATA represents response from shared L1X to AXC and L0X→L1XMSG represents requests from AXC to the shared L1X. For the *SCRATCH* design, there is only one link for data from L2 to the local scratchpad.

between the host CPU and accelerator tile. Furthermore, it may even miss in the shared cache which means a long latency operation which loads data from DRAM. However, we find that this is not a first order concern for the workloads we study. We observe that there are negligible misses incurred by requests issued from the accelertor tile at the shared last level cache. In *FUSION*, we accelerate routines which share data with other routines executed on the host CPU (see Table 5.1). This often implies that the data is present in the shared cache when the accelerator is invoked. In this work we do not model cache contention due to other processes. Current generation server chips include mechanisms to partition and prioritize allocation in the shared last level cache, eg. Intel Cache Allocation Technology [178]. Such mechanisms can be used to alleviate contention and ensure that the accelerator execution is not stalled on memory accesses which miss in the shared cache.

5.5.2 Energy

The energy breakdown of the benchmarks are presented in Figure 5.6a. We observe that the energy tradeoffs of pull-based cache architectures are different from that of push-based DMA execution models. The results in this subsection indicate that when optimizing for energy, a single architectural paradigms does not fit all applications.

For the *SCRATCH* system, *FFT* and *DISP.* are dominated by the L2 access energy due to repeated inter-AXC DMA transfers (963 and 640 respectively, see Table 5.6d). The large ratio of data transferred via DMA, column DMA(kB), compared to working set size, column WSet(kB), is an indicator of such pathological behaviour (165 for *FFT*). The *SHARED* system caches the AXC shared working set, eliminates spurious L2 accesses and reduces energy consumption by 10.6× and 7.6× for *FFT* and *DISP.*

Lesson 3: Small private caches (L0X) also improve energy. The *FUSION* system further reduces energy by introducing a 4K L0X which is 1.5× more energy efficient than even a heavily banked L1X and filters out 83% and 80% of the accesses (effect seen in Figure 5.6c) to the L1X for *FFT* and *DISP.* respectively significantly reducing the L0X-L1X link energy. *TRACK.* also spends a large fraction of energy in L2 accesses due to a large working set (371kB). Function *imgResize*, shares 99% of its data access (173 kB), triggering inter-AXC DMA transfers. The *SHARED* system and the *FUSION* system do not incur this overhead.

Lesson 4: The L1X filters accesses to the L2 but L0X→L1X coherence message overhead is significant. *FILT.* has a large ratio of DMA data transferred with respect to working set size and *SHARED* and *FUSION* designs save energy by eliminating L2 accesses (filtered by shared L1X). This can be seen as the diminished L2 stack of Figure 5.6a.

However these gains are lost to repeated thrashing behaviour of the L0X as the benchmark iterates over each pixel in the image. This increases coherence request messages between L0X

and L1X (see Figure 5.6c), expending significant energy. Similar behaviour is also observed in *SUSAN* and *HIST*. *HIST.* incurs additional penalty of coherence request messages (L1X→L2) for the *SHARED* and *FUSION* designs; large working set (1191kB) does not fit in L1X. The *FUSION* design mitigates some of the degradation observed in the *SHARED* system (see Figure 5.6c), but not enough to provide an overall energy benefit for *HIST*.

ADPCM sees a modest improvement of 4% as most of what is gained from the reduction in L2 accesses is lost in repeated L1X accesses. Overall *FUSION* reduces energy consumption by 2.4×, however for *HIST, SUSAN* and *FILT.*, *FUSION* increases energy consumption by 10% (improves performance by 67%). The *SHARED* system performs poorly in general due to the higher penalty of link energy for a) messages from AXC→L1X (see Figure 5.6c) b) data from L1X→AXC and c) access energy for the shared L1X cache.

5.5.3 Writeback vs Write-Through at L0X

Lesson 5: Write-through caches are expensive in terms of energy. Recent work [158, 72] have studied the effect of writes in data parallel accelerators and highlights their "bursty" behavior in GPGPU applications. For such applications, the contention introduced by writeback operations may evict freshly read data. We find that in fixed-function accelerators, write caching at the L0X is an important requirement to exploit the inherent data locality of the offloaded functions. Write-through adds energy overhead due to data transfers on the L0X→L1X link and L1X data access energy. In Table 5.4 we list the bandwidth consumption of both write-through and writeback models. We find that fixed-function accelerators that offload functions from existing programs reflect the locality behavior of the original program although the memory level parallelism may increase.

Table 5.4: Bandwidth in Flits (8bytes/flit)

	Write-Through	Writeback	% Dirty Blocks
FFT	230232	6642	39.5
DISP.	142656	40896	37.7
TRACK.	2266764	19428	45.3
ADPCM	3188262	14100	46.9
SUSAN	13973187	81600	35.1
FILT.	5728776	12096	51.3
HIST.	18575484	194316	46.4

5.5.4 *FUSION-Dx*: Write Forwarding

Lesson 6: Protocol extensions can exploit inter-AXC producer consumer relationships. Optimized hardware realizations of large functions are often split into smaller

	# FWD Blocks	AXC Cache	AXC Link
FFT	4309	6.4%	16.9%
TRACK.	4582	1.5%	5.7%

Table 5.5: Inter-AXC forwarded blocks and percentage reduction in energy consumption per component

blocks[66]. Using a shared cache introduces writebacks for data which is written by an accelerator and read by the next, effectively creating a producer-consumer relationship. *FUSION-Dx* optimizes for such relationships as described in Section 5.3.2. For each block forwarded from an L0X, the *FUSION-Dx* system saves energy spent by the *FUSION* system in 1 WriteBack to L1X, 1 Read from L1X and 1 request from L0X→L1X, while incurring the significantly lower cost of a L0X→L0X transfer (0.1pJ/byte). Table 5.5 enumerates the total number of blocks forwarded between AXC's of *FFT* and *TRACK.* and the corresponding savings in energy for the AXC component.

5.5.5 Larger AXC caches

Lesson 7: Larger may not be better. We experimented with a larger AXC cache configuration (AXC-Large) where the L0X was 8KB (2×) in size while the L1X was 256KB (4×) in size. The working set sizes of the workloads were such that only 1 benchmark (*DISP.*) fit into the Large-L1X (163kB footprint, see Table 5.6d) amongst the ones which did not fit into the Small-L1X (*TRACK., HIST.* and *DISP.*). For *ADPCM, SUSAN* and *FILT.* (working set sizes smaller than 30kB), the severe degradation seen in Figure 5.7 is due to the higher L1X access energy (2× as much as L1X-Small). There were negligible drops in miss rate for *DISP.* and *TRACK.* (less than 10 blocks) at the L0X. *DISP.* experienced 22% drop in L1X misses, which translated to a 3% reduction in cycle time (mostly obviated by the increased L1X access latency; 2 cycles more than L1X-Small).

5.5.6 Address Translation

Bench	AX-TLB	AX-RMAP
FFT	514	41
DISP.	4243	589
TRACK.	3237	831
ADPCM	1447	548
SUSAN	671	6
FILT.	668	18
HIST.	60K	20

Table 5.6: Virtual memory table look up count

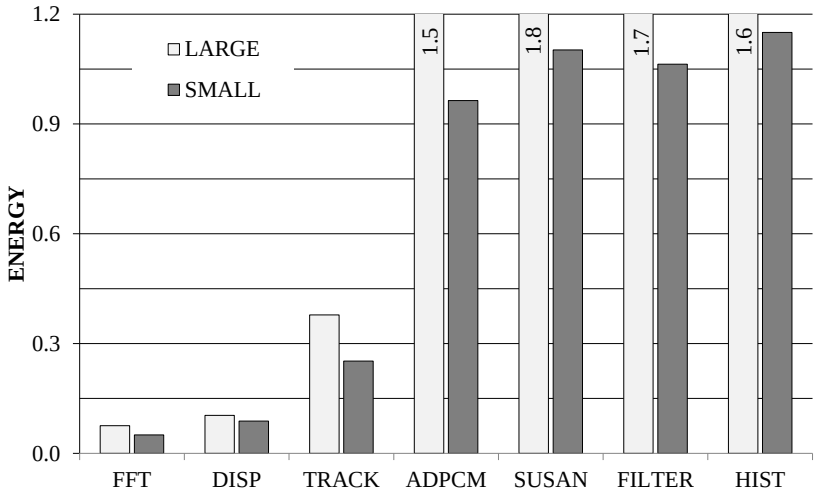


Figure 5.7: Comparing the benefits of LARGE (L0X:8KB,L1X:256KB) vs SMALL (L0X:4KB,L1X:64KB)

Lesson 8: Address translation overheads need to be mitigated. The address translation energy is an important consideration in coherent cache hierarchies [92]. *FUSION* places the TLB off the critical path and on the shared L1X's miss path where the request has to transition into the physical address space. Here we illustrate the benefits by listing the number of look ups in the TLB for the baseline (64KB shared L1X). *FUSION* also needs reverse map look ups on forwarded requests from the host's shared L2. The host's directory tracks the accelerator tile in the sharer list and only forwards requests to lines cached in the accelerator tile. Table 5.6 lists the number of look ups to the AX-RMAP and AX-TLB. Overall, we expend less than 1% of the energy on the AX-RMAP and AX-TLB; workloads that overflow the shared L1X and generate more misses, could possibly expend more energy.

5.6 Related Work

System-On-Chips: Chip designers have recognized the need for reducing the overhead of communication between the accelerator and the host processor. Current ARM multicores include an AXI bus [62] that snoops the shared L2 on the multicore; similarly IBM's Power processor [24] includes a Powerbus that ensures the most-up-to-date data is read from the processor. Both systems are similar to the *SCRATCH* configuration we study. The host processor and DMA is required to move in and out of the scratchpad and between the accelerators. The DMA overheads are minimal when accelerators are computationally intensive and read few data elements (e.g., cryptographic units) or have minimal interaction with each other (e.g., XML and cryptographic accelerators in PowerEN [27]). Past work [79, 104] have also studied the benefits of integrating network cards with the last level cache of the chip. The type of accelerators we study in this work are functions extracted from

a sequential program that have plenty of read-write sharing. In such cases, involving the DMA controller for moving shared data expends energy in the memory hierarchy and adds latency to the critical path of the accelerator. The *FUSION* designs we study in this work implicitly move data between the accelerators directly and minimizes the overheads of the locality lost as a result of the execution migration between the different accelerators.

In-core Accelerators: Recent research from academia [171, 64, 139] has extracted accelerators from sequential programs and have proposed to integrate these accelerators at- the-core and leverage the host processor's L1 cache. Integrating tightly with the L1 cache enables the accelerators to maintain coherence. Unfortunately, the load-to-use latency of the L1 cache shared between the accelerators may introduce a performance overhead. A pertinent example is the *demosaic* benchmark [139] which does not see as much of a performance gains as the other workloads due to the abundance of loads and stores. The *SHARED* configuration is a representative design and our analysis reveals that using a 64K shared cache to supply the accelerators results in a 2.7× energy overhead compared to using a cache hierarchy for *HIST., FILT., SUSAN* and *ADPCM* while saving 83% of energy for *FFT, DISP.* and *TRACK*. [50] proposes an interesting design where the accelerator-cache interface is configurable. The accelerator may choose to share the L1 data cache of the core or use it's own private data cache. The coprocessor dominated architecture proposal [187], ensures that accelerators with higher memory traffic are placed closer to the shared L1 data cache of the host to reduce energy consumption. However typical L1 caches are designed to meet the strict cycle-time constraints of the host processor and supplying data to accelerators with much higher memory level parallelism (up to 6×) is challenging. *FUSION* manages the cache hierarchy between un-core accelerators using a lightweight hardware coherence protocol and we have also explored the benefits of actively forwarding data between the accelerators.

Coherence Forwarding: Past work in academia[177, 160] have extensively studied the benefits of proactively forwarding and streaming data in multiprocessors to reduce cache misses and improve performance. Current multicores[61] have also included coherence states to help with forwarding data between caches directly. Herein we have studied the benefits of proactively pushing data to save energy. While past work studied the addition of forwarding support to lazy coherence protocols [91] we have shown the benefit of adding forwarding to a time-stamp based protocol [155, 158].

Un-core Accelerators: Vuletić et al. [174] propose a hardware memory management unit (WMU) to allow accelerators to operate in the virtual address space of the invoking process. DASX (data structure accelerator) [98] leverages LLC TLBs present in modern multicores to issue accesses to memory and maintains coherence at kernel boundaries. Recent

research [33, 180] incorporates application specific streaming frameworks independent of the host processor's cache hierarchy. *FUSION* allows accelerators to issue virtual memory addresses while removing address translation from the critical path; it also supports a flat coherence model.

5.7 Summary

The design tradeoffs for a coherent cache hierarchy for fixed-function accelerators have been evaluated. With the increasing energy cost of interconnects and caches relative to compute, it is imperative to optimize data movement to retain the energy benefits of accelerators. We develop *FUSION*, a lightweight multi-level cache hierarchy for accelerators and study the tradeoffs compared to a scratchpad-based architecture. *FUSION* leverages proposed time-stamp based coherence [155, 158, 121] to maintain coherency efficiently amongst the accelerator caches as well as integrating them with the MESI protocol. We find that i) small L0 private caches are essential to retain the energy benefit of accelerators and ii) shared L1 caches help optimize data movement between the functions offloaded from the same program and minimize host-accelerator data transfers. A comprehensive toolchain was developed for modelling fixed-function accelerators in a cycle accurate manner and used to study the tradeoffs compared to optimized DMA code. We study multiple facets of the cache hierarchy including write optimizations and evaluate the tradeoffs between the pull-based model of the cache hierarchy and push-based model of the DMA.

Chapter 6

Integration – Adaptive Granularity Caching

In this chapter, we describe mechanisms to support variable sized blocks in the cache hierarchy. With such a mechanism in place, unused data can be eliminated from the cache hierarchy. This reduces energy consumption for data transfers as well as showing improved performance due to greater effective cache capacity.

6.1 Introduction

A cache block is the fundamental unit of space allocation and data transfer in the memory hierarchy. Typically, a block is an aligned fixed granularity of contiguous words (1 word = 8bytes). Current processors fix the block granularity largely based on the average spatial locality across workloads, while taking tag overhead into consideration. Unfortunately, many applications (see Section 6.2 for details) exhibit low— moderate spatial locality and most of the words in a cache block are left untouched during the block's lifespan. Even for applications with good spatial behavior, the short lifespan of a block caused by cache geometry limitations can cause low cache utilization. Technology trends make it imperative that caching efficiency improves to reduce wastage of interconnect bandwidth. Recent reports from industry [3] show that on-chip networks can contribute up to 28% of total chip power. In the future an L2 — L1 transfer can cost up to 2.8× more energy than the L2 data access [80, 102]. Unused words waste $\simeq$ 11% (4%—21% in commercial workloads) of the cache hierarchy energy.

Figure 6.1 organizes past research on cache block granularity along the three main parameters influenced by cache block granularity: miss rate, bandwidth usage, and cache space utilization. Sector caches have been used to [150, 24] minimize bandwidth by fetching only sub-blocks but miss opportunities for spatial prefetching. Prefetching [93, 136] may help reduce the miss rate for utilized sectors, but on applications with low—moderate or variable

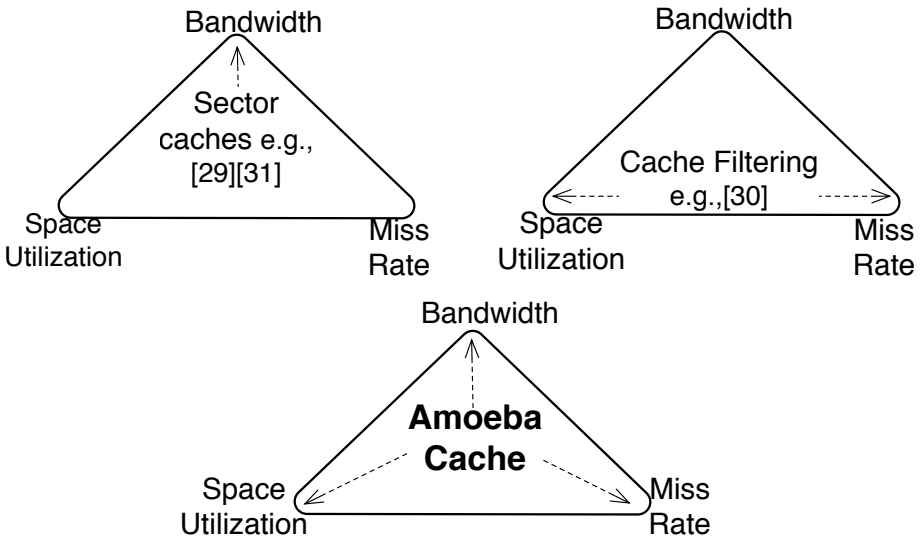


Figure 6.1: Cache designs optimizing different memory hierarchy parameters. Arrows indicate the parameters that are targeted and improved compared to a conventional cache.

spatial locality, unused sectors due to misprediction, or unused regions within sectors, still pollute the cache and consume bandwidth. Line distillation [141] filters out unused words from the cache at evictions using a separate word-granularity cache. Other approaches identify dead cache blocks and replace or eliminate them eagerly [101, 100, 78, 106]. While these approaches improve utilization and potentially miss rate, they continue to consume bandwidth and interconnect energy for the unutilized words. Word-organized cache blocks also dramatically increase cache associativity and lookup overheads, which impacts their scalability.

Determining a fixed optimal point for the cache line granularity at hardware design time is a challenge. Small cache lines tend to fetch fewer unused words, but impose significant performance penalties by missing opportunities for spatial prefetching in applications with high spatial locality. Small line sizes also introduce high tag overhead, increase lookup energy, and increase miss processing overhead (e.g., control messages). Larger cache line sizes minimize tag overhead and effectively prefetch neighboring words but introduce the negative effect of unused words that increase network bandwidth. Prior approaches have proposed the use of multiple caches with different block sizes [169, 59]. These approaches require word granularity caches that increase lookup energy, impose high tag overhead (e.g., 50% in [169]), and reduce cache efficiency when there is good spatial locality.

We propose a novel cache architecture, *Amoeba-Cache*, to improve memory hierarchy efficiency by supporting fine-grain (per-miss) dynamic adjustment of cache block size and the # of blocks per set. To enable variable granularity blocks within the same cache, the tags maintained per set need to grow and shrink as the # of blocks/set vary. *Amoeba-Cache*

eliminates the conventional tag array and collocates the tags with the cache blocks in the data array. This enables us to segment and partition a cache set in different ways: For example, in a configuration comparable to a traditional 4-way 64K cache with 256 sets (256 bytes per set), we can hold eight 32-byte cache blocks, thirty-two 8-byte blocks, or any other collection of cache blocks of varying granularity. Different sets may hold blocks of different granularity, providing maximum flexibility across address regions of varying spatial locality. The *Amoeba-Cache* effectively filters out unused words in a conventional block and prevents them from being inserted into the cache, allowing the resulting free space to be used to hold tags or data of other useful blocks. The *Amoeba-Cache* can adapt to the available spatial locality; when there is low spatial locality, it will hold many blocks of small granularity and when there is good spatial locality, it can adapt and segment the cache into a few big blocks.

Compared to a fixed granularity cache, *Amoeba-Cache* improves cache utilization by 90% - 99% for most applications, saves miss rate by up to 73% (omnetpp) at the L1 level and up to 88% (twolf) at the LLC level, and reduces miss bandwidth by up to 84% (omnetpp) at the L1 and 92% (twolf) at the LLC. We compare against other approaches such as Sector Cache and Line distillation and show that *Amoeba-Cache* can optimize miss rate and bandwidth better across many applications, with lower hardware overhead. Our synthesis of the cache controller hit path shows that *Amoeba-Cache* can be implemented with low energy impact and 0.7% area overhead for a latency- critical 64K L1.

This chapter is organized as follows: Section 6.2 provides quantitative evidence for the acuteness of the spatial locality problem. Section 6.3 details the internals of the *Amoeba-Cache* organization and Section 6.4 analyzes the physical implementation overhead. Section 6.5 deals with wider chip-level issues (i.e., inclusion and coherence). Section 6.6 — Section 6.10 evaluate the *Amoeba-Cache*, commenting on the optimal block granularity, impact on overall on-chip energy, and performance improvement. Section 6.11 outlines related work.

6.2 Motivation for Adaptive Blocks

In traditional caches, the cache block defines the fundamental unit of data movement and space allocation in caches. The blocks in the data array are uniformly sized to simplify the insertion/removal of blocks, simplify cache refill requests, and support low complexity tag organization. Unfortunately, conventional caches are inflexible (fixed block granularity and fixed # of blocks) and caching efficiency is poor for applications that lack high spatial locality. Cache blocks influence multiple system metrics including bandwidth, miss rate, and cache utilization. The block granularity plays a key role in exploiting spatial locality by effectively prefetching neighboring words all at once. However, the neighboring words could go unused due to the low lifespan of a cache block. The unused words occupy interconnect

bandwidth and pollute the cache, which increases the # of misses. We evaluate the influence of a fixed granularity block below.

6.2.1 Cache Utilization

In the absence of spatial locality, multi-word cache blocks (typically 64 bytes on existing processors) tend to increase cache pollution and fill the cache with words unlikely to be used. To quantify this pollution, we segment the cache line into words (8 bytes) and track the words touched before the block is evicted. We define utilization as the average # of words touched in a cache block before it is evicted. We study a comprehensive collection of workloads from a variety of domains: 6 from PARSEC [21], 7 from SPEC2006, 2 from SPEC2000, 3 Java workloads from DaCapo [22], 3 commercial workloads (Apache, SpecJBB2005, and TPC-C [107]), and the Firefox web browser. Subsets within benchmark suites were chosen based on demonstrated miss rates on the fixed granularity cache (i.e., whose working sets did not fit in the cache size evaluated) and with a spread and diversity in cache utilization. We classify the benchmarks into 3 groups based on the utilization they exhibit: Low (<33%), Moderate (33%—66%), and High (66%+) utilization (see Table 6.1).

Table 6.1: Benchmark Groups

Group	Utilization %	Benchmarks
Low	0 — 33%	art, soplex, twolf, mcf, canneal, lbm, omnetpp
Moderate	34 — 66%	astar, h2, jbb, apache, x264, firefox, tpc-c, freqmine, fluidanimate
High	67 — 100%	tradesoap, facesim, eclipse, cactus, milc, ferret

Figure 6.2 shows the histogram of words touched at the time of eviction in a cache line of a 64K, 4-way cache (64-byte block, 8 words per block) across the different benchmarks. Seven applications have less than 33% utilization and 12 of them are dominated (>50%) by 1-2 word accesses. In applications with good spatial locality (cactus, ferret, tradesoap, milc, eclipse) more than 50% of the evicted blocks have 7-8 words touched. Despite similar average utilization for applications such as astar and h2 (39%), their distributions are dissimilar; ≃70% of the blocks in astar have 1-2 words accessed at the time of eviction, whereas ≃50% of the blocks in h2 have 1-2 words accessed per block. Utilization for a single application also changes over time; for example, ferret's average utilization, measured as the average fraction of words used in evicted cache lines over 50 million instruction windows, varies from 50% to 95% with a periodicity of roughly 400 million instructions.

6.2.2 Effect of Block Granularity on Miss Rate and Bandwidth

Cache miss rate directly correlates with performance, while under current and future wire-limited technologies [3], bandwidth directly correlates with dynamic energy. Figure 6.3

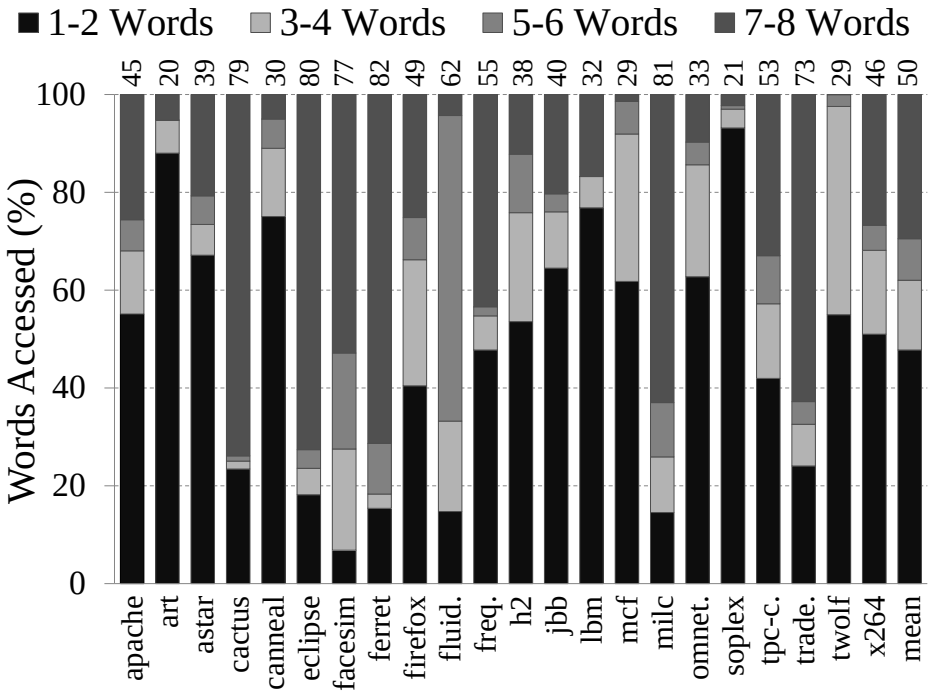


Figure 6.2: Distribution of words touched in a cache block. Avg. utilization is on top. (Config: 64K, 4 way, 64-byte block.)

shows the influence of block granularity on miss rate and bandwidth for a 64K L1 cache and a 1M L2 cache keeping the number of ways constant. For the 64K L1, the plots/amoeba highlight the pitfalls of simply decreasing the block size to accommodate the Low group of applications; miss rate increases by 2× for the High group when the block size is changed from 64B to 32B; it increases by 30% for the Moderate group. A smaller block size decreases bandwidth proportionately but increases miss rate. With a 1M L2 cache, the lifetime of the cache lines increases significantly, improving overall utilization. Increasing the block size from 64→256 halves the miss rate for all application groups. The bandwidth is increased by 2× for the Low and Moderate.

Since miss rate and bandwidth have different optimal block granularities, we use the following metric: $\frac{1}{MissRate \times Bandwidth}$ to determine a fixed block granularity suited to an application that takes both criteria into account. Table 6.2 shows the block size that maximizes the metric for each application. It can be seen that different applications have different block granularity requirements. For example, the metric is maximized for apache at 128 bytes and for firefox (similar utilization) at 32 bytes. Furthermore, the optimal block sizes vary with the cache size as the cache lifespan changes. This highlights the challenge of picking a single block size at design time especially when the working set does not fit in the cache.

6.2.3 Need for adaptive cache blocks

Our observations motivate the need for adaptive cache line granularities that match the spatial locality of the data access patterns in an application. In summary:

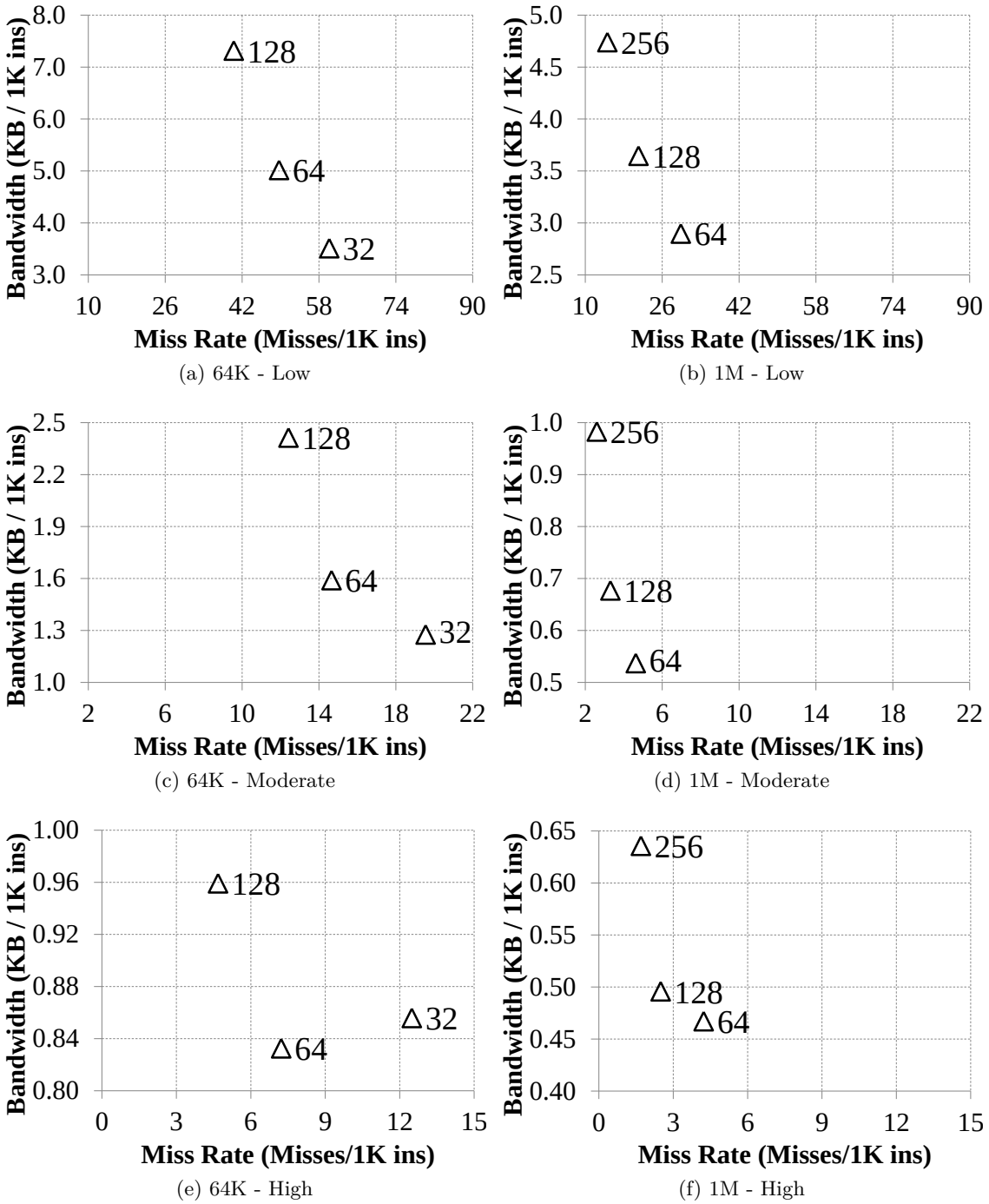
- Smaller cache lines improve utilization but tend to increase miss rate and potentially traffic for applications with good spatial locality, affecting the overall performance.
- Large cache lines pollute the cache space and interconnect with unused words for applications with poor spatial locality, significantly decreasing the caching efficiency.
- Many applications waste a significant fraction of the cache space. Spatial locality varies not only across applications but also within each application, for different data structures as well as different phases of access over time.

Table 6.2: Optimal block size. Metric: $\frac{1}{\mathbf{Miss-rate} \times \mathbf{Bandwidth}}$

64K, 4-way	
Block	Benchmarks
32B	cactus, eclipse, facesim, ferret, firefox, fluidanimate,freqmine, milc, tpc-c, tradesoap
64B	art
128B	apache, astar, canneal, h2, jbb, lbm, mcf, omnetpp, soplex, twolf, x264
1M, 8-way	
Block	Benchmarks
64B	apache, astar, cactus, eclipse, facesim, ferret, firefox, freqmine, h2, lbm, milc, omnetpp, tradesoap, x264
128B	art
256B	canneal, fluidanimate, jbb, mcf, soplex, tpc-c, twolf

6.3 Amoeba-Cache: Architecture

The *Amoeba-Cache* architecture enables the memory hierarchy to fetch and allocate space for a range of words (a variable granularity cache block) based on the spatial locality of the application. For example, consider a 64K cache (256 sets) that allocates 256 bytes per set. These 256 bytes can adapt to support, for example, eight 32-bytes blocks, thirty-two 8-byte blocks, or four 32-byte blocks and sixteen 8-byte blocks, based on the set of contiguous words likely to be accessed. The key challenge to supporting variable granularity blocks is how to grow and shrink the # of tags as the # of blocks per set vary with block granularity? *Amoeba-Cache* adopts a solution inspired by software data structures, where programs hold meta-data and actual data entries in the same address space. To achieve maximum flexibility, *Amoeba-Cache* completely eliminates the tag array and collocates the tags with the actual data blocks (see Figure 6.4). We use a bitmap (`T? Bitmap`) to indicate which words in the



(a) 64K - Low

(b) 1M - Low

(c) 64K - Moderate

(d) 1M - Moderate

(e) 64K - High

(f) 1M - High

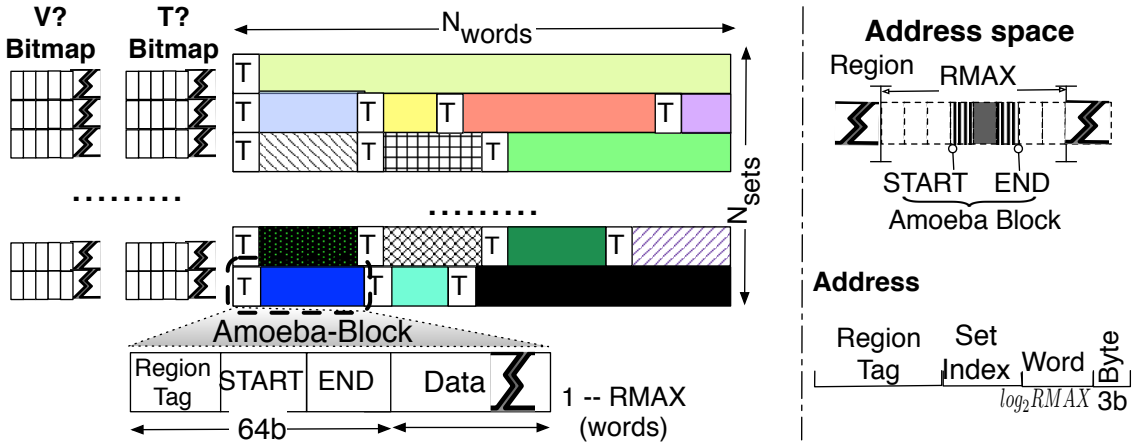
Figure 6.3: Bandwidth vs. Miss Rate. (a),(c),(e): 64K, 4-way L1. (b),(d),(f): 1M, 8-way LLC. Markers on the plot indicate cache block size. Note the different scales for different groups.

data array represent tags. We also decouple the conventional valid/invalid bits (typically associated with the tags) and organize them into a separate array (V? : Valid bitmap) to simplify block replacement and insertion. V? and T? bitmaps both require 1 bit for very

word (64bits) in the data array (total overhead of 3%). *Amoeba-Cache* tags are represented as a range (`Start` and `End` address) to support variable granularity blocks. We next discuss the overall architecture.

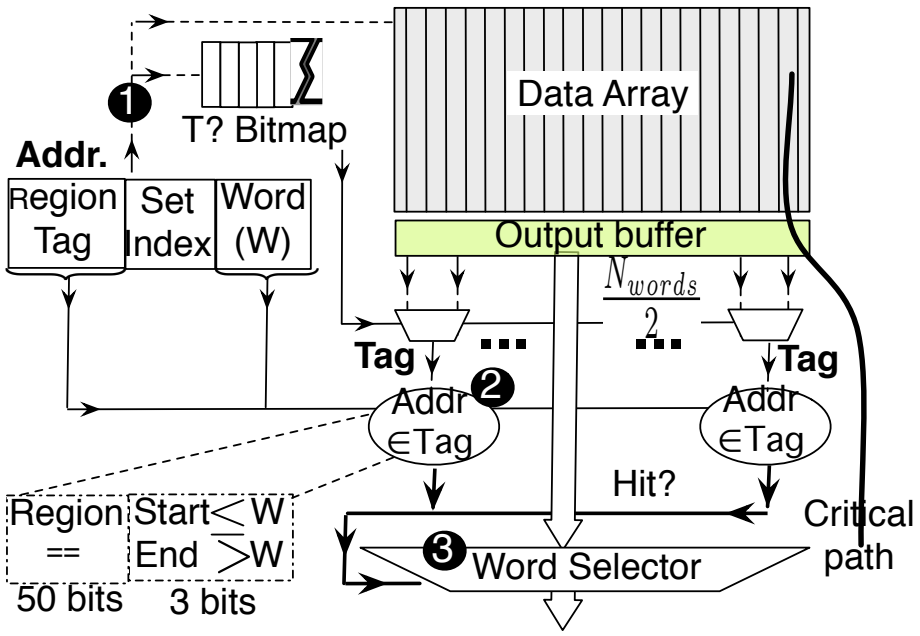
6.3.1 Amoeba Blocks and Set-Indexing

The *Amoeba-Cache* data array holds a collection of varied granularity *Amoeba-Block*s that do not overlap. Each *Amoeba-Block* is a 4 tuple consisting of `<Region Tag, Start, End, Data- Block>` (Figure 6.4). A `Region` is an aligned block of memory of size `RMAX` bytes. The boundaries of any *Amoeba-Block* block (`Start` and `End`) always will lie within the regions' boundaries. The minimum granularity of the data in an *Amoeba-Block* is 1 word and the maximum is `RMAX` words. We can encode `Start` and `End` in $log_2(RMAX)$ bits. The set indexing function masks the lower $log_2(RMAX)$ bits to ensure that all *Amoeba-Block*s (every memory word) from a region index to the same set. The Region Tag and Set-Index are identical for every word in the *Amoeba-Block*. Retaining the notion of *sets* enables fast lookups and helps elude challenges such as synonyms (same memory word mapping to different sets). When comparing against a conventional cache, we set `RMAX` to 8 words (64 bytes), ensuring that the set indexing function is identical to that in the conventional cache to allow for a fair evaluation.



Left: Cache Layout. Size: `$N_sets*N_words*8 bytes$`. T? (Tag map) 1 bit/word. Is word a tag? Yes(1). V? (Valid map): 1 bit/word. Is word allocated? Yes(1). Total overhead: 3%. **Right:** Address space layout: Region: Aligned regions of size RMAX words (×8 bytes). *Amoeba-Block* range cannot exceed Region boundaries. Cache Indexing: Lower $log_2 RMAX$ bits masked and set index is derived from remaining bits; all words (and all *Amoeba-Block*s) within a region index to the same set.

Figure 6.4: Amoeba-Cache Architecture.



Lookup steps. ❶ Mask lower $log_2 RMAX$ bits and index into data array. ❷ Identify tags using T? bitmap and initiate tag checks ($\in$) operator (Region tag comparator ==? and $\leq$ range comparator). ❸ Word selection within *Amoeba-Block*.

Figure 6.5: Lookup Logic

6.3.2 Data Lookup

Figure 6.5 describes the steps of the lookup. ❶ The lower $log_2(RMAX)$ bits are masked from the address and the set index is derived from the remaining bits. ❷ In parallel with the data read out, the `T? bitmap` activates the words in the data array corresponding to the tags for comparison. Given the minimum size of a *Amoeba-Block* is two words (1 word for the tag metadata, 1 word for the data), adjacent words cannot be tags. We need $\frac{N_{words}}{2}$ 2-1 multiplexers that route one of the adjacent words to the comparator ($\in$ operator). The comparator generates the hit signal for the word selector. The $\in$ operator consists of two comparators: a) an aligned `Region tag` comparator, a conventional == (64 - $log_2 N_{sets}$ - $log_2 RMAX$ bits wide, e.g., 50 bits) that checks if the *Amoeba-Block* belongs to the same region and b) a $Start <= W < END$ range comparator ($log_2 RMAX$ bits wide; e.g., 3 bits) that checks if the *Amoeba-Block* holds the required word. Finally, in ❸, the tag match activates and selects the appropriate word.

6.3.3 Amoeba Block Insertion

On a miss for the desired word, a spatial granularity predictor is invoked (see Section 6.5.1), which specifies the range of the *Amoeba-Block* to refill. To determine a position in the set to slot the incoming block we can leverage the V? (Valid bits) bitmap. The V? bitmap

has 1 bit/word in the cache; a "1" bit indicates the word has been allocated (valid data or a tag). To find space for an incoming data block we perform a substring search on the V? bitmap of the cache set for contiguous 0s (empty words). For example, to insert an *Amoeba-Block* of five words (four words for data and one word for tag), we perform a substring search for *00000* in the V? bitmap / set (e.g., 32 bits wide for a 64K cache). If we cannot find the space, we keep triggering the replacement algorithm until we create the requisite contiguous space. Following this, the *Amoeba-Block* tuple (Tag and Data block) is inserted, and the corresponding bits in the T? and V? array are set. The 0s substring search can be accomplished with a lookup table; many current processors already include a substring instruction [55, PCMISTRI].

6.3.4 Replacement: Pseudo LRU

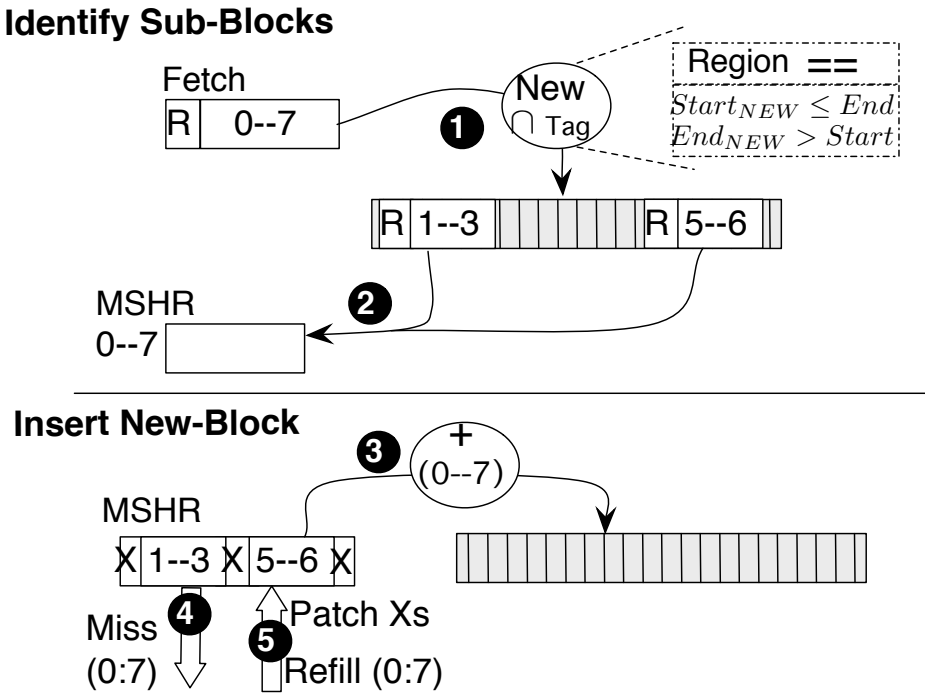
To reclaim the space from an *Amoeba-Block* the tag bits T? (tag) and V? (valid) bits corresponding to the block are unset. The key issue is identifying the *Amoeba-Block* to replace. Classical pseudo-LRU algorithms [90, 90] keep the metadata for the replacement algorithm separate from the tags to reduce port contention. To be compatible with pseudo-LRU and other algorithms such as DIP [140] that work with a fixed # of ways, we can logically partition a set in *Amoeba-Cache* into N_{ways}. For instance, if we partition a 32 word cache set into 4 logical ways, any access to an *Amoeba-Block* tag found in words 0 —7 of the set is treated as an access to logical way 0. Finding a replacement candidate involves identifying the selected replacement way and then picking (possibly randomly) a candidate *Amoeba-Block* . More refined replacement algorithms that require per-tag metadata can harvest the space in the tag-word of the *Amoeba-Block* which is 64 bits wide (for alignment purposes) while physical addresses rarely extend beyond 48 bits.

6.3.5 Partial Misses

With variable granularity data blocks, a challenging although rare case (5 in every 1K accesses) that occurs is a *partial miss*. It is observed primarily when the spatial locality changes. Figure 6.6 shows an example. Initially, the set contains two blocks from a region R, one *Amoeba-Block* caches words 1 –3 (Region:R, START:1 END:3) and the other holds words 5 –6 (Region:R START:5 END:6). The CPU reads word 4, which misses, and the spatial predictor requests an *Amoeba-Block* with range START:0 and END:7. The cache has *Amoeba-Block*s that hold subparts of the incoming *Amoeba-Block*, and some words (0, 4, and 7) need to be fetched.

Amoeba-Cache removes the overlapping sub-blocks and allocates a new *Amoeba-Block*. This is a multiple step process: ❶ On a miss, the cache identifies the overlapping sub-blocks in the cache using the tags read out during lookup. $\cap \neq NULL$ is true if $START_{new} < END_{Tag}$ and $END_{new} > START_{Tag}$ (New = incoming block and Tag = *Amoeba-Block* in

set). ❷ The data blocks that overlap with the miss range are evicted and moved one-at-a-time to the MSHR entry. ❸ Space is then allocated for the new block, i.e., it is treated like a new insertion. ❹ A miss request is issued for the entire block (START:0 — END:7) even if only some words (e.g., 0, 4, and 7) may be needed. This ensures request processing is simple and only a single refill request is sent. ❺ Finally, the incoming data block is patched into the MSHR; only the words not obtained from the L1 are copied (since the lower level could be stale).

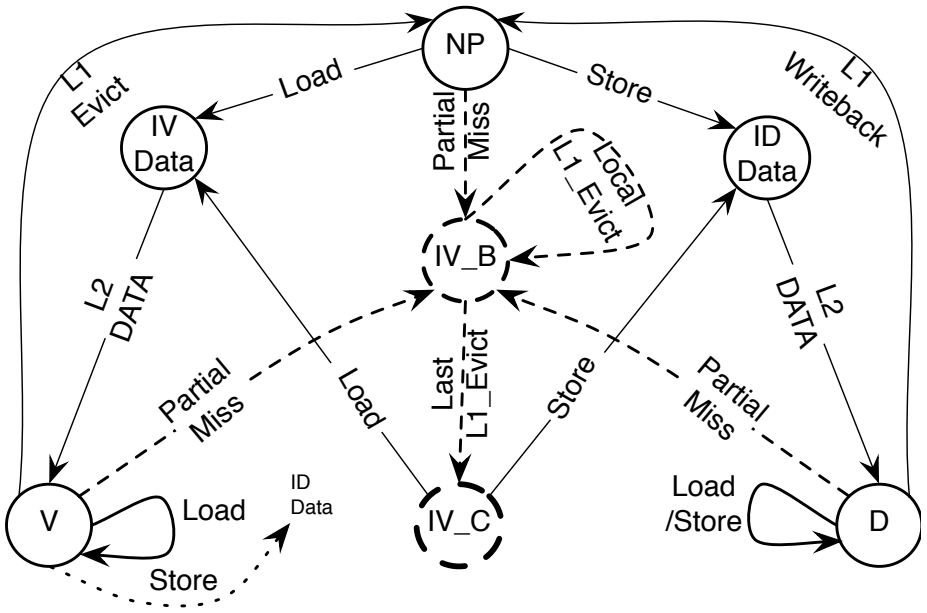


❶ Identify blocks overlapping with `New` block. ❷ Evict overlapping blocks to MSHR. ❸ Allocate space for new block (treat it like a new insertion). ❹ Issue refill request to lower level for entire block. ❺ Patch only newer words as lower-level data could be stale.

Figure 6.6: Partial Miss Handling. Upper: Identify relevant sub-blocks. Useful for other cache controller events as well, e.g., recalls. Lower: Refill of words and insertion.

6.4 Hardware Complexity

We analyze the complexity of *Amoeba-Cache* along the following directions: we quantify the additions needed to the cache controller, we analyze the latency, area, and energy penalty, and finally, we study the challenges specifically introduced by large caches.



Cache controller states

State	Description
NP	*Amoeba-Block* not present in the cache.
V	All words corresponding to *Amoeba-Block* present and valid (read-only)
D	Valid and atleast one word in *Amoeba-Block* is dirty (read-write)
IV_B	Partial miss being processed (blocking state)
IV_Data	Load miss; waiting for data from L2
ID_Data	Store miss; waiting for data. Set dirty bit.
IV_C	Partial miss cleanup from cache completed (treat as full miss)
Amoeba-specific Cache Events	
Partial miss: Process partial miss.	
Local_L1_Evict: Remove overlapping *Amoeba-Block* to MSHR.	
Last_L1_Evict: Last *Amoeba-Block* moved to MSHR. Convert to full miss and process load or store.	

Bold and Broken-lines: *Amoeba-Cache* additions.

Figure 6.7: Amoeba Cache Controller (L1 level).

6.4.1 Cache Controller

The variable granularity *Amoeba-Block* blocks need specific consideration in the cache controller. We focus on the L1 controller here, and in particular, partial misses. The cache controller manages operations at the aligned RMAX granularity. The controller permits only

one in-flight cache operation per RMAX region. In-flight cache operations ensure no address overlap with stable *Amoeba-Block*s in order to eliminate complex race conditions. Figure 6.7 shows the L1 cache controller state machine. We add two states to the default protocol, `IV_B` and `IV_C`, to handle partial misses. `IV_B` is a blocking state that blocks other cache operations to RMAX region until all relevant *Amoeba-Block*s to a partial miss are evicted (e.g., 0–3 and 5–7 blocks in Figure 6.6). `IV_C` indicates partial miss completion. This enables the controller to treat the access as a full miss and issue the refill request. The other stable states (I, V, D) and transient states (`IV_Data` and `ID_Data`) are present in a conventional protocol as well. `Partial-miss` triggers the clean-up operations (`1` and `2` in Figure 6.6). `Local_L1_Evict` is a looping event that keeps retriggering for each *Amoeba-Block* involved in the partial miss; `Last_L1_Evict` is triggered when the last *Amoeba-Block* involved in the partial miss is evicted to the MSHR. A key difference between the L1 and lower-level protocols is that the Load/Store event in the lower-level protocol may need to access data from multiple *Amoeba-Block*s. In such cases, similar to the partial-miss event, we read out each block independently before supplying the data (more details in Section 6.5.2).

6.4.2 Area, Latency, and Energy Overhead

The extra metadata required by *Amoeba-Cache* are the T? (1 tag bit per word) and V? (1 valid bit per word) bitmaps. Table 6.3 shows the quantitative overhead compared to the data storage. Both the T? and V? bitmap arrays are directly proportional to the size of the cache and require a constant storage overhead (3% in total). The T? bitmap is read in parallel with the data array and does not affect the critical path; T? adds 2%—3.5% (depending on cache size) to the overall cache access energy. V? is referred only on misses when inserting a new block.

Table 6.3: *Amoeba-Cache* Hardware Complexity.

Cache configuration			
	64K (256by/set)	1MB (512by/set)	4MB (1024by/set)
Data RAM parameters			
Delay	0.36ns	2ns	2.5 ns
Energy	100pJ	230pJ	280pJ
Amoeba-Cache components (CACTI model)			
T?/V? map	1KB	16KB	64KB
Latency	0.019ns (5%)	0.12ns (6%)	0.2ns (6%)
Energy	2pJ (2%)	8pJ (3.4%)	10pJ (3.5%)
LRU	$\frac{1}{8}$KB	2KB	8KB
Lookup Overhead (VHDL model)			
Area	0.7%	0.1%	
Latency	0.02ns	0.035ns	0.04ns

% indicates overhead compared to data array of cache. 64K cache operates in *Fast mode*; 1MB and 4MB operate in *Normal mode*. We use 32nm ITRS HP transistors for 64K and 32nm ITRS LOP transistors for 1MB and 4MB.

We synthesized[1] the cache lookup logic using Synopsys and quantify the area, latency, and energy penalty. *Amoeba-Cache* is compatible with *Fast* and *Normal* cache access modes [125, -access-mode config], both of which read the entire set from the data array in parallel with the way selection to achieve lower latency. *Fast* mode transfers the entire set to the edge of the H-tree, while *Normal* mode, only transmits the selected way over the H-tree. For synthesis, we used the Synopsys design compiler (Vision Z-2007.03-SP5).

Figure 6.5 shows *Amoeba-Cache*ś lookup hardware on the critical path; we compare it against a fixed-granularity cache's lookup logic (mainly the comparators). The area overhead of the *Amoeba-Cache* includes registering an entire line that has been read out, the tag operation logic, and the word selector. The components on the critical path once the data is read out are the 2-way multiplexers, the $\in$ comparators, and priority encoder that selects the word; the T? bitmap is accessed in parallel and off the critical path. *Amoeba-Cache* is made feasible under today's wire-limited technology where the cache latency and energy is dominated by the bit/word lines, decoder, and H-tree [125]. *Amoeba-Cache*'s comparators, which operate on the entire cache set, are 6$\times$ the area of a fixed cache's comparators. The data array occupies 99% of the overall cache area. The critical path is dominated by the wide word selector since the comparators all operate in parallel. The lookup logic adds 60% to the conventional cache's comparator time. The overall critical path is dominated by the data array access and *Amoeba-Cache*'s lookup circuit adds 0.02ns to the access latency and $\simeq$ 1pJ to the energy of a 64K cache, and 0.035ns to the latency and $\simeq$2pJ to the energy of a 1MB cache. Finally, *Amoeba-Cache* amortizes the energy penalty of the peripheral components (H-tree, Wordline, and decoder) over a single RAM.

Amoeba-Cache's overhead needs careful consideration when implemented at the L1 cache level. We have two options for handling the latency overhead a) if the L1 cache is the critical stage in the pipeline, we can throttle the CPU clock by the latency overhead to ensure that the additional logic fits within the pipeline stage. This ensures that the number of pipeline stages for a memory access does not change with respect to a conventional cache, although all instructions bear the overhead of the reduced CPU clock. b) we can add an extra pipeline stage to the L1 hit path, adding a 1 cycle overhead to all memory accesses but ensuring no change in CPU frequency. We quantify the performance impact of both approaches in Section 6.6.

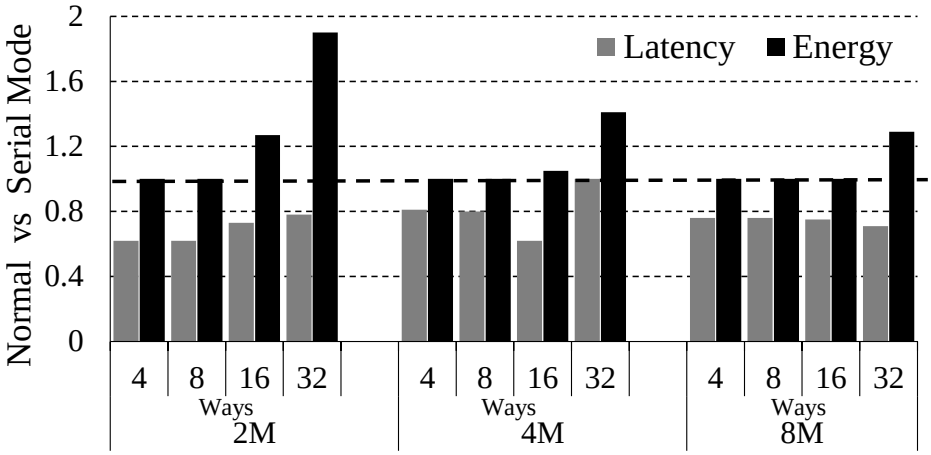
6.4.3 Tag-only Operations

Conventional caches support tag-only operations to reduce data port contention. While the *Amoeba-Cache* merges tags and data, like many commercial processors it decouples the replacement metadata and valid bits from the tags, accessing the tags only on cache

[1]We do not have access to an industry-grade 32nm library, so we synthesized at a higher 180nm node size and scaled the results to 32 nm (latency and energy scaled proportional to Vdd (taken from [44]) and Vdd^2 respectively).

lookup. Lookups can be either CPU side or network side (coherence invalidation and Wback/Forwarding). CPU-side lookups and writebacks ($\simeq$ 95% of cache operations) both need data and hence *Amoeba-Cache* in the common case does not introduce extra overhead. *Amoeba-Cache* does read out the entire data array unlike serial-mode caches (we discuss this issue in the next section). Invalidation checks and snoops can be more energy expensive with *Amoeba-Cache* compared to a conventional cache. Fortunately, coherence snoops are not common in many applications (e.g., 1/100 cache operations in SpecJBB) as a coherence directory and an inclusive LLC filter them out.

6.4.4 Tradeoff with Large Caches



Baseline: Serial. ≤ 1 Normal is better. 32nm, ITRS LOP.

Figure 6.8: Serial vs Normal mode cache.

Large caches with many words per set ($\equiv$ highly associative conventional cache) need careful consideration. Typically, highly associative caches tend to serialize tag and data access with only the relevant cache block read out on a hit and no data access on a miss. We first analyze the tradeoff between reading the entire set (normal mode), which is compatible with *Amoeba-Cache* and only the relevant block (serial mode). We vary the cache size from 2M—8M and associativity from 4(256B/set) — 32 (2048B/set). Under current technology constraints (Figure 6.8), only at very high associativity does serial mode demonstrate a notable energy benefit. Large caches are dominated by H-tree energy consumption and reading out the entire set at each sub-bank imposes an energy penalty when bitlines and wordlines dominate (2KB+ # of words/set).

Amoeba-Cache can be tuned to minimize the hardware overhead for large caches. With many words/set the cache utilization improves due to longer block lifetimes making it feasible to support *Amoeba-Block*s with a larger minimum granularity (> 1 word). If we increase

Table 6.4: % of direct accesses with fast tags

	64K(256by/set)		1MB(512by/set)		2MB(1024 by/set)	
# Tags/set	2	4	4	8	8	16
Overhead	1KB	2KB	2KB	16KB	16KB	32KB
Benchmarks						
Low	30%	45%	42%	64%	55%	74%
Moderate	24%	62%	46%	70%	63%	85%
High	35%	79%	67%	95%	75%	96%

minimum granularity to two or four words, only every third or fifth word could be a tag, meaning the # of comparators and multiplexers reduce to $\frac{N_{words/set}}{3}$ or $\frac{N_{words/set}}{5}$. When the minimum granularity is equal to max granularity (RMAX), we obtain a fixed granularity cache with $N_{words/set}/RMAX$ ways. Cache organizations that collocate all the tags together at the head of the data array enable tag-only operations and serial *Amoeba-Block* accesses that need to activate only a portion of the data array. However, the set may need to be compacted at each insertion. Recently, Loh and Hill [108] explored such an organization for supporting tags in multi-gigabyte caches.

Finally, the use of *Fast Tags* help reduce the tag lookups in the data array. Fast tags use a separate traditional tag array-like structure to cache the tags of the recently-used blocks and provide a pointer directly to the *Amoeba-Block*. The # of *Fast Tags* needed per set is proportional to the # of blocks in each set, which varies with the spatial locality in the application and the # of bytes per set (more details in Section 6.6.1). We studied 3 different cache configurations (64K 256B/set, 1M 512B/set, and 2M 1024B/set) while varying the number of fast tags per set (see Table 6.4). With 8 tags/set (16KB overhead), we can filter 64—95% of the accesses in a 1MB cache and 55— 75% of the accesses in a 2MB cache.

6.5 Chip-Level Issues

6.5.1 Spatial Patterns Prediction

Amoeba-Cache needs a spatial block predictor, which informs refill requests about the range of the block to fetch. *Amoeba-Cache* can exploit any spatial locality predictor and there have been many efforts in the compiler and architecture community [37, 93, 136, 34]. We adopt a table-driven approach consisting of a set of access bitmaps; each entry is `RMAX` (maximum granularity of an *Amoeba-Block*) bits wide and represents whether the word was touched during the lifetime of the recently evicted cache block. On a miss, the predictor will search for an entry (indexed by either the miss PC or region address) and choose the range of words to be fetched on a miss on either side (left and right) of the critical word. The PC-based indexing also uses the critical word index for improved accuracy. The

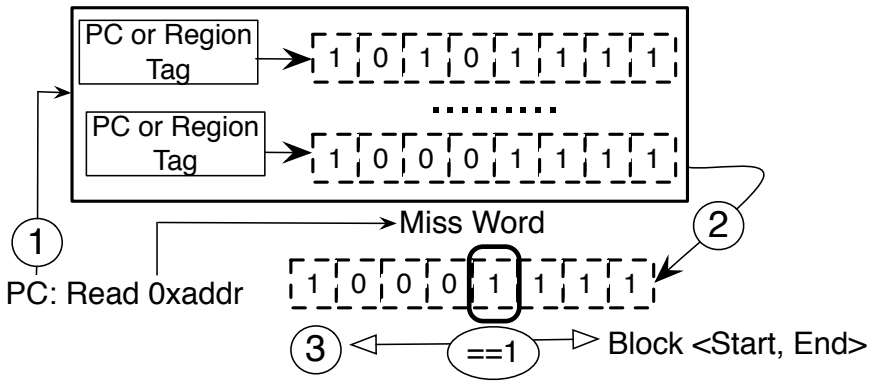


Figure 6.9: Spatial Predictor invoked on a *Amoeba-Cache* miss

predictor optimizes for spatial prefetching and will overfetch (bring in potentially untouched words), if they are interspersed amongst contiguous chunks of touched words. We can also bypass the prediction when there is low confidence in the prediction accuracy. For example, for streaming applications without repeated misses to a region, we can bring in a fixed granularity block based on the overall global behavior of the application. We evaluate tradeoffs in the design of the spatial predictor in Section 6.6.2.

6.5.2 Multi-level Caches

We discuss the design of inclusive cache hierarchies including multiple *Amoeba-Cache*s; we illustrate using a 2-level hierarchy. Inclusion means that the L2 cache contains a superset of the data words in the L1 cache; however, the two levels may include different granularity blocks. For example, the Sun Niagara T2 uses 16 byte L1 blocks and 64 byte L2 blocks. *Amoeba-Cache* permits non-aligned blocks of variable granularity at the L1 and the L2, and needs to deal with two issues: a) L2 recalls that may invalidate multiple L1 blocks and b) L1 refills that may need data from multiple blocks at the L2. For both cases, we need to identify all the relevant *Amoeba-Block*s that overlap with either the recall or the refill request. This situation is similar to a Nigara's L2 eviction which may need to recall 4 L1 blocks. *Amoeba-Cache*'s logic ensures that all *Amoeba-Block*s from a region map to a single set at any level (using the same RMAX for both L1 and L2). This ensures that L2 recalls or L1 refills index into only a single set. To process multiple blocks for a single cache operation, we use the step-by-step process outlined in Section 6.3.5 (❶ and ❷ in Figure 6.6). Finally, the L1-L2 interconnect needs 3 virtual networks, two of which, the L2→L1 data virtual network and the L1→L2 writeback virtual network, can have packets of variable granularity; each packet is broken down into a variable number of smaller physical flits.

6.5.3 Cache Coherence

There are three main challenges that variable cache line granularity introduces when interacting with the coherence protocol: 1) How is the coherence directory maintained? 2) How to support variable granularity read sharing? and 3) What is the granularity of write invalidations? The key insight that ensures compatibility with a conventional fixed-granularity coherence protocol is that a *Amoeba-Block* always lies within an aligned RMAX byte region (see Section 6.3). To ensure correctness, it is sufficient to maintain the coherence granularity and directory information at a fixed granularity $\leq$ RMAX granularity. Multiple cores can simultaneously cache any variable granularity *Amoeba-Block* from the same region in *S*hared state; all such cores are marked as sharers in the directory entry. A core that desires exclusive ownership of an *Amoeba-Block* in the region uses the directory entry to invalidate every *Amoeba-Block* corresponding to the fixed coherence granularity. All *Amoeba-Block*s relevant to an invalidation will be found in the same set in the private cache (see set indexing in Section 6.3). The coherence granularity could potentially be $<$ RMAX so that false sharing is not introduced in the quest for higher cache utilization (larger RMAX). The core claiming the ownership on a write will itself fetch only the desired granularity *Amoeba-Block*, saving bandwidth.

6.6 Evaluation

Framework We evaluate the *Amoeba-Cache* architecture with the Wisconsin GEMS simulation infrastructure [115]; we use the in-order processor timing model. We have replaced the SIMICS functional simulator with the faster Pin [111] instrumentation framework to enable longer simulation runs. We perform timing simulation for 1 billion instructions. We warm up the cache using 20 million accesses from the trace. We model the cache controller in detail including the transient states needed for the multi-step cache operations and all the associated port and queue contention. We use a Full— LRU replacement policy, evicting *Amoeba-Block*s in LRU order until sufficient space is freed up for the block to be brought in. This helps decouple our observations from the replacement policy, enabling a fairer comparison with other approaches (Section 6.9). Our workloads are a mix of applications whose working sets stress our caches and includes SPEC- CPU benchmarks, Dacapo Java benchmarks [22], commercial workloads (SpecJBB2005, TPC-C, and Apache), and the Firefox web browser. Table 6.1 classifies the application categories: Low, Moderate, and High, based on the spatial locality. When presenting averages of ratios or improvements, we use the geometric mean.

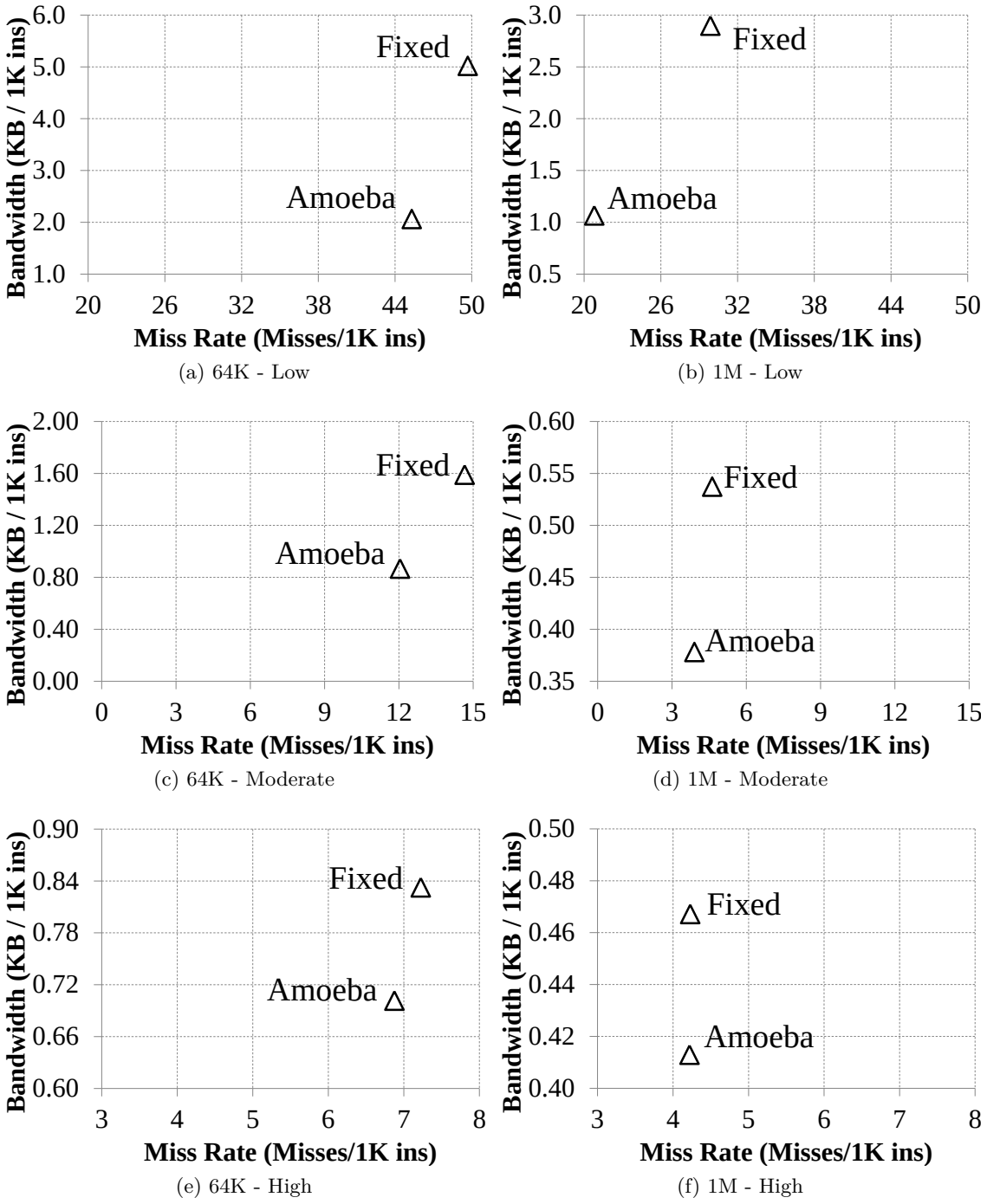
6.6.1 Improved Memory Hierarchy Efficiency

Result 1: *Amoeba-Cache increases cache capacity by harvesting space from unused words and can achieve an 18% reduction in both L1 and L2 miss rate.*
Result 2: *Amoeba-Cache adaptively sizes the cache block granularity and reduces L1↔L2 bandwidth by 46% and L2↔Memory bandwidth by 38%.*

In this section, we compare the bandwidth and miss rate properties of *Amoeba-Cache* against a conventional cache. We evaluate two types of caches: a **Fixed** cache, which represents a conventional set-associative cache, and the *Amoeba-Cache*. In order to isolate the benefits of *Amoeba-Cache* from the potentially changing accuracy of the spatial predictor across different cache geometries, we use utilization at the next eviction as the spatial prediction, determined from a prior run on a fixed granularity cache. This also ensures that the spatial granularity predictions can be replayed across multiple simulation runs. To ensure equivalent data storage space, we set the *Amoeba-Cache* size to the sum of the tag array and the data array in a conventional cache. At the L1 level (64K), the net capacity of the *Amoeba-Cache* is 64K + 8*4*256 bytes and at the L2 level (1M) configuration, it is 1M + 8*8*2048 bytes. The L1 cache has 256 sets and the L2 cache has 2048 sets.

Figure 6.10 plots/amoeba the miss rate and the traffic characteristics of the *Amoeba-Cache*. Since *Amoeba-Cache* can hold blocks varying from 8B to 64B, each set can hold more blocks by utilizing the space from untouched words. *Amoeba-Cache* reduces the 64K L1 miss rate by 23%(stdev:24) for the Low group, and by 21%(stdev:16) for the moderate group; even applications with high spatial locality experience a 7%(stdev:8) improvement in miss rate. There is a 46%(stdev:20) reduction on average in L1↔L2 bandwidth. At the 1M L2 level, *Amoeba-Cache* improves the moderate group's miss rate by 8%(stdev:10) and bandwidth by 23%(stdev:12). Applications with moderate utilization make better use of the space harvested from unused words by *Amoeba-Cache*. Many low utilization applications tend to be streaming and providing extra cache space does not help lower miss rate. However, by not fetching unused words, *Amoeba-Cache* achieves a significant reduction (38%(stdev:24) on average) in off-chip L2↔Memory bandwidth; even High utilization applications see a 17%(stdev:15) reduction in bandwidth. Utilization and miss rate are not, however, always directly correlated (more details in Section 6.8).

With *Amoeba-Cache* the # of blocks/set varies based on the granularity of the blocks being fetched, which in turn depends on the spatial locality in the application. Table 6.5 shows the avg.# of blocks/set. In applications with low spatial locality, *Amoeba-Cache* adjusts the block size and adapts to store many smaller blocks. The 64K L1 *Amoeba-Cache* stores 10 blocks per set for mcf and 12 blocks per set for art, effectively increasing associativity without introducing fixed hardware overheads. At the L2, when the working set starts to fit in the L2 cache, the set is partitioned into fewer blocks. Applications like eclipse and omnetpp hold only 3—5 blocks per set on average (lower than conventional associativity)



(a) 64K - Low (b) 1M - Low

(c) 64K - Moderate (d) 1M - Moderate

(e) 64K - High (f) 1M - High

Figure 6.10: Fixed vs. Amoeba (Bandwidth and Miss Rate). Note the different scale for different application groups.

due to their low miss rates (see Table 6.6). With streaming applications (e.g., canneal), *Amoeba-Cache* increases the # of blocks/set to >15 on average. Finally, some applications like apache store between 6—7 blocks/set with a 64K cache with varied block sizes (see

Table 6.5: Avg. # of *Amoeba-Block* / Set

# Blocks/Set	64K Cache, 288 B/set
4—5	ferret, cactus, firefox, eclipse, facesim, freqmine, milc, astar
6—7	tpc-c, tradesoap, soplex, apache, fluidanimate
8—9	h2, canneal, omnetpp, twolf, x264, lbm, jbb
10—12	mcf, art
	1M Cache, 576 B/set
3—5	eclipse, omnetpp
8—9	cactus, firefox, tradesoap, freqmine, h2, x264, tpc-c
10—11	facesim, soplex, astar, milc, apache, ferret
12—13	twolf, art, jbb, lbm , fluidanimate
15—18	canneal, mcf

Figure 6.11): approximately 50% of the blocks store 1-2 words and 30% of the blocks store 8 words at the L1. As the size of the cache increases and thereby the lifetime of the blocks, the *Amoeba-Cache* adapts to store larger size blocks as can be seen in Figure 6.11.

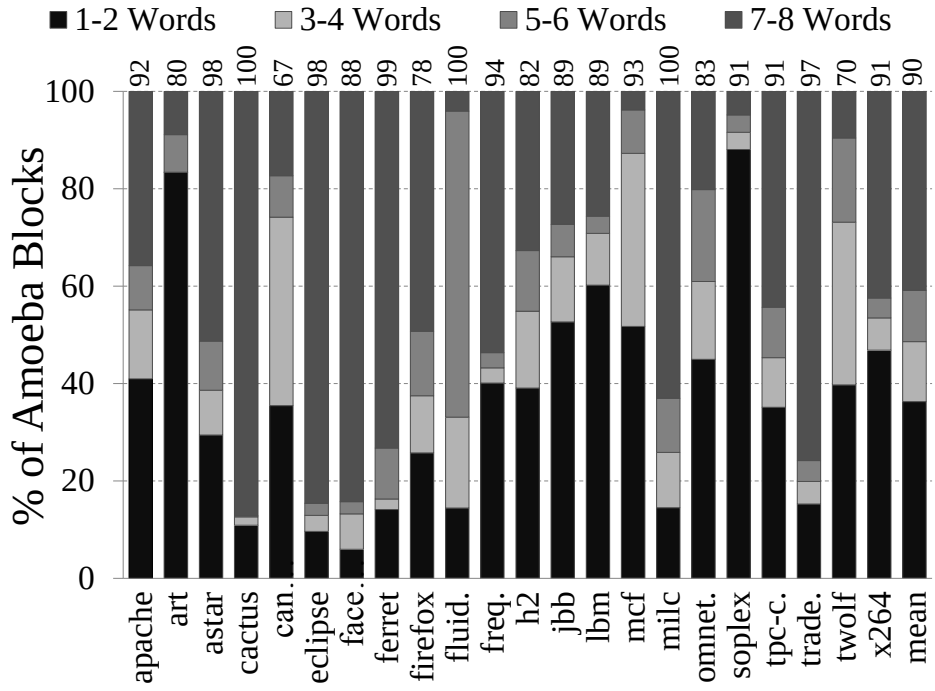
Utilization is improved greatly across all applications (90%+ in many cases). Figure 6.11 shows the distribution of cache block granularities in *Amoeba-Cache*. The *Amoeba-Block* distribution matches the word access distribution presented in Section 6.2). With the 1M cache, the larger cache size improves block lifespan and thereby utilization, with a significant drop in the % of 1—2 word blocks. However, in many applications (tpc-c, apache, firefox, twolf, lbm, mcf), up to 20% of the blocks are 3–6 words wide, indicating the benefits of adaptivity and the challenges faced by Fixed.

6.6.2 Overall Performance and Energy

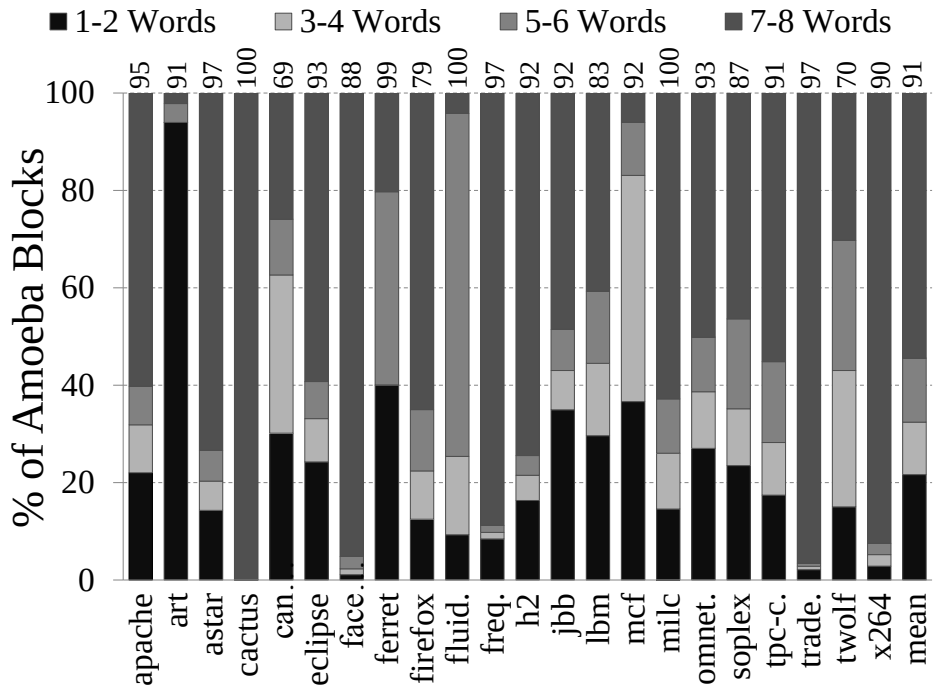
Result 3:*Amoeba-Cache improves overall cache efficiency and boosts performance by 10% on commercial applications*[2]*, saving up to 11% of the energy of the on-chip memory hierarchy. Off-chip L2↔memory energy sees a mean reduction of 41% across all workloads (86% for art and 93% for twolf) .*

We model a two-level cache hierarchy in which the L1 is a 64K cache with 256 sets (3 cycles load-to-use) and the L2 is 1M, 8192 sets (20 cycles). We assume a fixed memory latency of 300 cycles. We conservatively assume that the L1 access is the critical pipeline stage and throttle CPU clock by 4% (we evaluate an alternative approach in the next section). We calculate the total dynamic energy of the *Amoeba-Cache* using the energy #s determined in Section 6.4 through a combination of synthesis and CACTI [125]. We use 4 fast tags per set at the L1 and 8 fast tags per set at the L2. We include the penalty for all the extra metadata in *Amoeba-Cache*. We derive the energy for a single L1—L2 transfer ((6.8pJ per byte) from [3, 125].The interconnect uses full-swing wires at 32nm, 0.6V.

[2]"Commercial" applications includes Apache, SpecJBB and TPC-C.



(a) 64K L1 cache



(b) 1M L2 cache

Figure 6.11: Distribution of cache line granularities in the 64K L1 and 1M L2 Amoeba-Cache. Avg. utilization is on top.

Figure 6.12 plots/amoeba the overall improvement in performance and reduction in on-chip memory hierarchy energy (L1 and L2 caches, and L1↔L2 interconnect). On applications

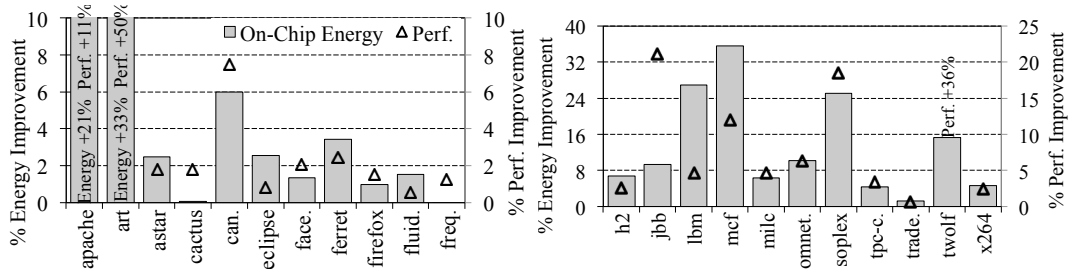


Figure 6.12: % improvement in performance and % reduction in on-chip memory hierarchy energy. Higher is better. Y-axis terminated to illustrate bars clearly. Baseline: Fixed, 64K L1, 1M L2.

that have good spatial locality (e.g., tradesoap, milc, facesim, eclipse, and cactus), *Amoeba-Cache* has minimal impact on miss rate, but provides significant bandwidth benefit. This results in on-chip energy reduction: milc's L1↔L2 bandwidth reduces by 15% and its on-chip energy reduces by 5%. Applications that suffer from cache pollution under Fixed (apache, jbb, twolf, soplex, and art) see gains in performance and energy. Apache's performance improves by 11% and on-chip energy reduces by 21%, while SpecJBB's performance improves by 21% and energy reduces by 9%. Art gains approximately 50% in performance. Streaming applications like mcf access blocks with both low and high utilization. Keeping out the unused words in the under-utilized blocks prevents the well-utilized cache blocks from being evicted; mcf's performance improves by 12% and on-chip energy by 36%.

Extra cache pipeline stage. An alternative strategy to accommodate *Amoeba-Cache*'s overheads is to add an extra pipeline stage to the cache access which increases hit latency by 1 cycle. The cpu clock frequency entails no extra penalty compared to a conventional cache. We find that for applications in the moderate and low spatial locality group (for 8 applications) *Amoeba-Cache* continues to provide a performance benefit between 6—50%. milc and canneal suffer minimal impact, with a 0.4% improvement and 3% slowdown respectively. Applications in the high spatial locality group (12 applications) suffer an average 15% slowdown (maximum 22%) due to the increase in L1 access latency. In these applications, 43% of the instructions (on average) are memory accesses and a 33% increase in L1 hit latency imposes a high penalty. All applications continue to retain the energy benefit. The cache hierarchy energy is dominated by the interconnects and *Amoeba-Cache* provides notable bandwidth reduction. While these results may change for an out-of-order, multiple-issue processor, the evaluation suggests that *Amoeba-Cache* if implemented with the extra pipeline stage is more suited for lower levels in the memory hierarchy other than the L1.

Off-chip L2↔Memory energy The L2's higher cache capacity makes it less susceptible to pollution and provides less opportunity for improving miss rate. In such cases, *Amoeba-*

Cache keeps out unused words and reduces off-chip bandwidth and thereby off-chip energy. We assume that the off-chip DRAM can provide adaptive granularity transfers for *Amoeba-Cache*'s L2 refills as in [183]. We use a DRAM model presented in a recent study [42] and model 0.5nJ per word transferred off-chip. The low spatial locality applications see a dramatic reduction in off-chip energy. For example, twolf sees a 93% reduction. On commercial workloads the off-chip energy decreases by 31% and 24% respectively. Even for applications with high cache utilization, off-chip energy decreases by 15%.

6.7 Spatial Predictor Tradeoffs

We evaluate the effectiveness of spatial pattern prediction in this section. In our table-based approach, a pattern history table records spatial patterns from evicted blocks and is accessed using a prediction index. The table-driven approach requires careful consideration of the following: prediction index, predictor table size, and training period. We quantify the effects by comparing the predictor against a baseline fixed-granularity cache. We use a 64K cache since it induces enough misses and evictions to highlight the predictor tradeoffs clearly.

6.7.1 Predictor Indexing

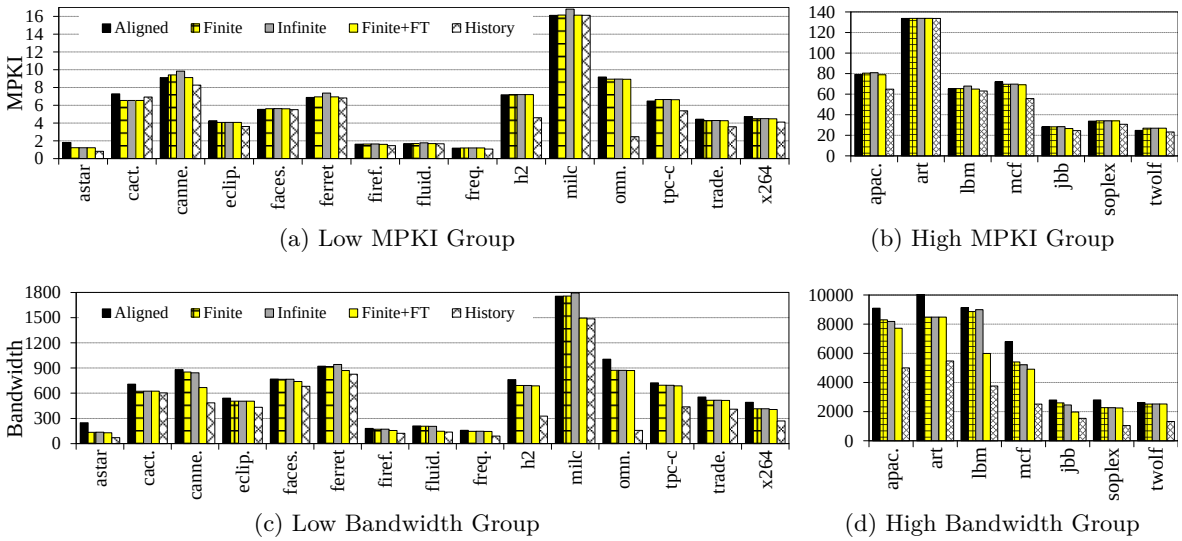
A critical choice with the history-based prediction is the selection of the predictor index. We explored two types of predictor indexing a) a PC-based approach [34] based on the intuition that fields in a data structure are accessed by specific PCs and tend to exhibit similar spatial behavior. The tag includes the PC and the critical word index: $((PC >> 3) << 3) + \frac{(addr\%64)}{8}$. and b) a Region-based (`REGION`) approach that is based on the intuition that similar data objects tend to be allocated in contiguous regions of the address space and tend to exhibit similar spatial behavior. We compared the miss rate and bandwidth properties of both the `PC` (256 entries, fully associative) and `REGION` (1024 entries, 4KB region size) predictors. The size of the predictors was selected as the sweet spot in behavior for each predictor type. For all applications apart from cactus (a high spatial locality application), `REGION`-based prediction tends to overfetch and waste bandwidth as compared to `PC`-based prediction, which has 27% less bandwidth consumption on average across all applications. For 17 out of 22 applications, `REGION`-based prediction shows 17% better MPKI on average (max: 49% for cactus). For 5 applications (apache, art, mcf, lbm, and omnetpp), we find that `PC` demonstrates better accuracy when predicting the spatial behavior of cache blocks than `REGION` and demonstrates a 24% improvement in MPKI (max: 68% for omnetpp).

6.7.2 Predictor Table

We studied the organization and size of the pattern table using the `REGION` predictor. We evaluated the following parameters a) region size, which directly correlates with the coverage

of a fixed-size table, and b) the size of the predictor table, which influences how many unique region patterns can be tracked, and c) the # of bits required to represent the spatial pattern.

Large region sizes effectively reduce the # of regions in the working set and require a smaller predictor table. However, a larger region is likely to have more blocks that exhibit varied spatial behavior and may pollute the pattern entry. We find that going from 1KB (4096 entries) to 4KB (1024 entries) regions, the 4KB region granularity decreased miss rate by 0.3% and increased bandwidth by 0.4% even though both tables/amoeba provide the same working set coverage (4MB). Fixing the region size at 4KB, we studied the benefits of an unbounded table. Compared to a 1024 entry table (`FINITE` in Figure 6.13), the unbounded table increases miss rate by 1% and decreases bandwidth by 0.3% . A 1024 entry predictor table (4KB region granularity per-entry) suffices for most applications. Organizing the 1024 entries as a 128-set×8-way table suffices for eliminating associativity related conflicts (<0.8% evictions due to lack of ways).



(a) Low MPKI Group

(b) High MPKI Group

(c) Low Bandwidth Group

(d) High Bandwidth Group

`ALIGNED`: fixed-granularity cache (64B blocks). `FINITE`: *Amoeba-Cache* with a `REGION` predictor (1024 entry predictor table and 4K region size). `INFINITE`: *Amoeba-Cache* with an unbounded predictor table (`REGION` predictor).`FINITE+FT` is `FINITE` augmented with hints for predicting a default granularity on compulsory misses (first touches). `HISTORY`: *Amoeba-Cache* uses spatial pattern hints based on utilization at the next eviction, collected from a prior run.

Figure 6.13: Spatial Predictor Performance Comparison

Focusing on the # of bits required to represent the pattern table, we evaluated the use of 4-bit saturation counters (instead of 1-bit bitmaps). The saturation counters seek to avoid pattern pollution when blocks with varied spatial behavior reside in the same region. Interestingly, we find that it is more beneficial to use 1-bit bitmaps for the majority of the applications (12 out of 22); the hysteresis introduced by the counters increases training period. To summarize, we find that a `REGION` predictor with region size 4KB and 1024

entries can predict the spatial pattern in a majority of the applications. CACTI indicates that the predictor table can be indexed in 0.025ns and requires 2.3pJ per miss indexing.

6.7.3 Spatial Pattern Training

A widely-used approach to training the predictor is to harvest the word usage information on an eviction. Unfortunately, evictions may not be frequent, which means the predictor's training period tends to be long, during which the cache performs less efficiently and/or that the application's phase has changed in the meantime. Particularly at the time of first touch (compulsory miss to a location), we need to infer the global spatial access patterns. We compare the finite region predictor (`FINITE` in Figure 6.13) that only predicts using eviction history, against a `FINITE+FT`: this adds the optimization of inferring the default pattern from a prior run when there is no predictor information. `FINITE+FT` demonstrates an avg. 1% (max: 6% for jbb) reduction in miss rate compared to `FINITE` and comes within 16% the miss rate of `HISTORY`. In terms of bandwidth `FINITE+FT` can save 8% of the bandwidth (up to 32% for lbm) compared to `FINITE`. The percentage of first-touch accesses is shown in Table 6.6.

Table 6.6: *Amoeba-Cache* Performance. Absolute #s.

	MPKI		BW bytes/1K		CPI	Predictor Stats	
	L1 MPKI	L2 MPKI	L1⟷L2 #Bytes/1K	L2⟷Mem #Bytes/1K	Cycles/Ins.	First Touch % Misses	Evict Win. # ins./Evict
apache	64.9	19.6	5,000	2,067	8.3	0.4	17
art	133.7	53.0	5,475	1,425	16.0	0.0	9
astar	0.9	0.3	70	35	1.9	18.0	1,600
cactus	6.9	4.4	604	456	3.5	7.5	162
canne.	8.3	5.0	486	357	3.2	5.8	128
eclip.	3.6	<0.1	433	<1	1.8	0.1	198
faces.	5.5	4.7	683	632	3.0	41.2	190
ferre.	6.8	1.4	827	83	2.1	1.3	156
firef.	1.5	1.0	123	95	2.1	11.1	727
fluid.	1.7	1.4	138	127	1.9	39.2	629
freqm.	1.1	0.6	89	65	2.3	17.7	994
h2	4.6	0.4	328	46	1.8	1.7	154
jbb	24.6	9.6	1,542	830	5.0	10.2	42
lbm	63.1	42.2	3,755	3,438	13.6	6.7	18
mcf	55.8	40.7	2,519	2,073	13.2	0.0	19
milc	16.1	16.0	1,486	1,476	6.0	2.4	66
omnet.	2.5	<0.1	158	<1	1.9	0.0	458
sople.	30.7	4.0	1,045	292	3.1	0.9	35
tpcc	5.4	0.5	438	36	2.0	0.4	200
trade.	3.6	<0.1	410	6	1.8	0.6	194
twolf	23.3	0.6	1,326	45	2.2	0.0	49
x264	4.1	1.8	270	190	2.2	12.4	274

MPKI : Misses / 1K instructions. BW: # words / 1K instructions
CPI: Clock cycles per instruction.
Predictor First touch: Compulsory misses. % of accesses that use default granularity.
Evict window: # of instructions between evictions.
Higher value indicates predictor training takes longer.

6.7.4 Predictor Summary

- For the majority of the applications (17/22) the address-region predictor with region size 4KB works well. However, five applications (apache, lbm, mcf, art, omnetpp) show better performance with PC-based indexing. For best efficiency, the predictor should adapt indexing to each application.
- Updating the predictor only on evictions leads to long training periods, which causes loss in caching efficiency. We need to develop mechanisms to infer the default pattern based on global behavior demonstrated by resident cache lines.
- The online predictor reduces MPKI by 7% and bandwidth by 26% on average relative to the conventional approach. However, it still has a 14% higher MPKI and 38% higher bandwidth relative to the `HISTORY`-based predictor, indicating room for improvement in prediction.
- The 1024-entry (4K region size) predictor table imposes $\simeq$0.12% energy overhead on the overall cache hierarchy energy since it is referenced only on misses.

6.8 *Amoeba-Cache* Adaptivity

We demonstrate that *Amoeba-Cache* can adapt better than a conventional cache to the variations in spatial locality.

Tuning RMAX for High Spatial Locality A challenge often faced by conventional caches is the desire to widen the cache block (to achieve spatial prefetching) without wasting space and bandwidth in low spatial locality applications. We study 3 specific applications: milc and tradesoap have good spatial locality and soplex has poor spatial locality. With a conventional 1M cache, when we widen the block size from 64 to 128 bytes, milc and tradesoap experience a 37% and 39% reduction in miss rate. However, soplex's miss rate increases by 2× and bandwidth by 3.1×.

With *Amoeba-Cache* we do not have to make this uneasy choice as it permits *Amoeba-Block*s with granularity 1—RMAX words (RMAX: maximum block size). When we increase RMAX from 64 bytes to 128 bytes, miss rate reduces by 37% for milc and 24% for tradesoap, while simultaneously lowering bandwidth by 7%. Unlike the conventional cache, *Amoeba-Cache* is able to adapt to poor spatial locality: soplex experiences only a 2% increase in bandwidth and 40% increase in miss rate.

Predicting Strided Accesses Many applications (e.g., firefox and canneal) exhibit strided access patterns, which touch a few words in a block before accessing another block. Strided accesses patterns introduce intra-block holes (untouched words). For instance,

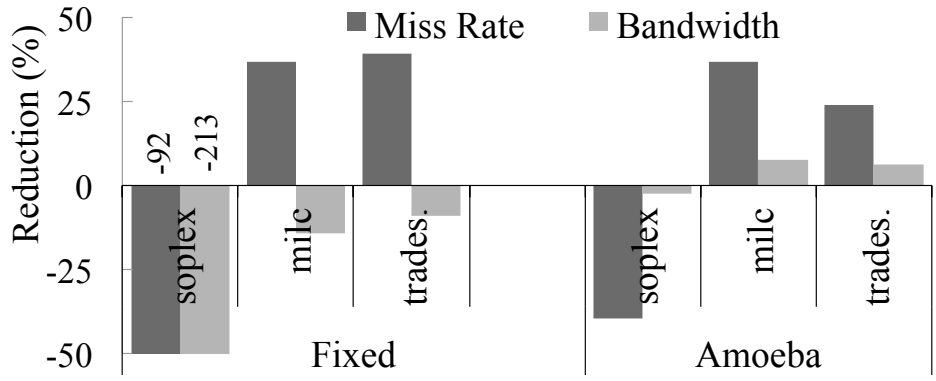


Figure 6.14: Effect of increase in block size from 64 to 128 bytes in a 1 MB cache

canneal accesses ≃10K distinct fixed granularity cache blocks with a specific access pattern, **[--x--x--]** (**x** indicates i^{th} word has been touched). Any predictor that uses the access pattern history has two choices when an access misses on word 3 or 6 a) A miss oriented policy (Policy-Miss) may refill the entire range of words 3–6 and eliminate the secondary miss but bring in untouched words 4–5, increasing bandwidth, and b) a bandwidth focused choice (Policy- BW) that refills only the requested word but will incur more misses. Table 6.7 compares the two contrasting policies for *Amoeba-Cache* (relative to a Fixed granularity baseline cache). Policy-BW saves 9% bandwidth compared to Policy-Miss but suffer 25-30% higher miss rate.

Table 6.7: Predictor Policy Comparison

	canneal		firefox	
	Miss Rate	BW	Miss Rate	BW
Policy-Miss	10.31%	81.2%	11.18%	47.1%
Policy-BW	–20.08%	88.09%	–13.44%	56.82%
Spatial Patterns	**[--x--x--]** **[x--x----]**		**[-x--x---]** **[x---x---]**	

–: indicates Miss or BW higher than Fixed.

6.9 *Amoeba-Cache* vs other approaches

We compare *Amoeba-Cache* against four approaches.

- **Fixed-2x**: Our baseline is a fixed granularity cache 2× the capacity of the other designs (64B block).
- **Sector** is the conventional sector cache design (as in IBM Power7 [24]): 64B block and a small sector size (16bytes or 2 words). This design targets optimal bandwidth.

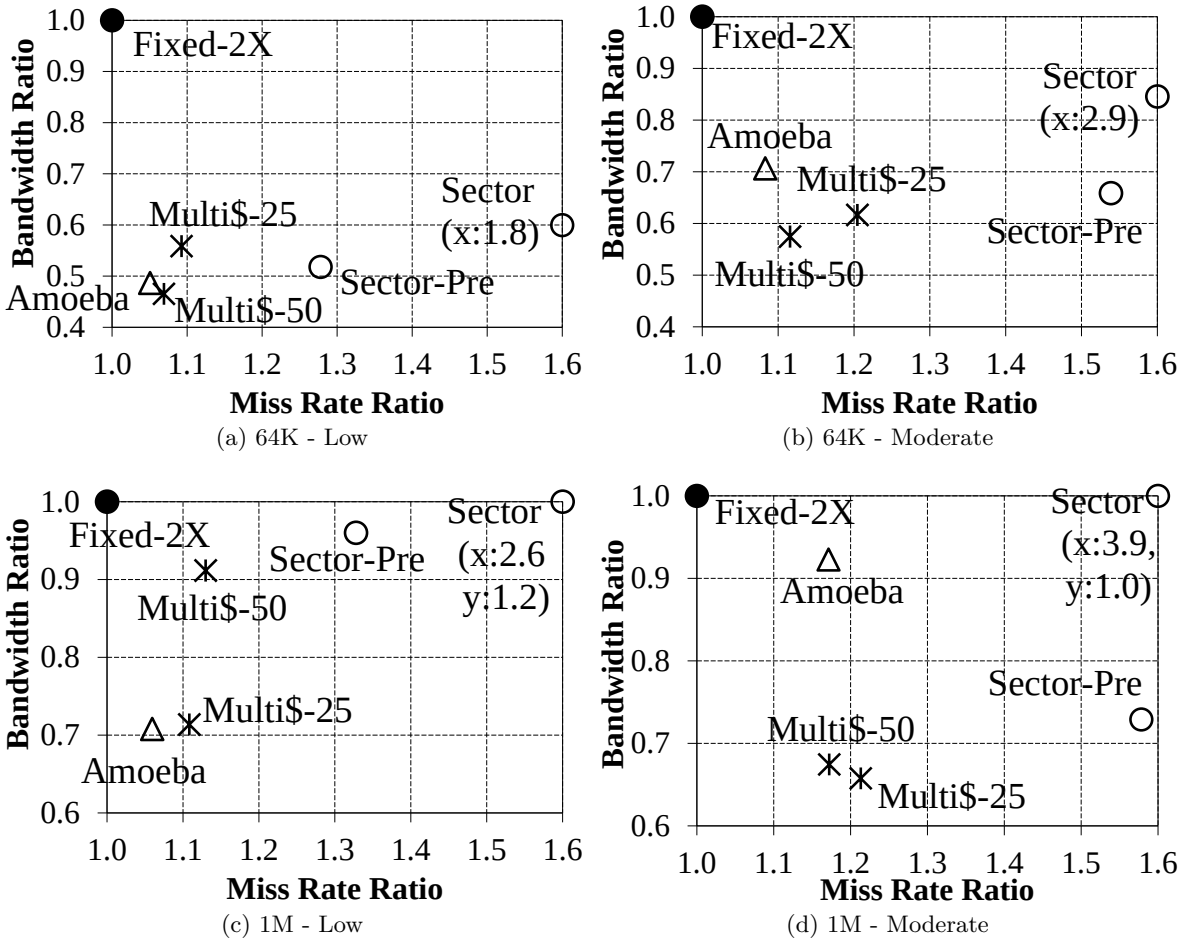
On any access, the cache fetches only the requisite sector, even though it allocates space for the entire line.

- **Sector-Pre** adds prefetching to Sector. This optimized design prefetches multiple sectors based on a spatial granularity predictor to improve the miss rate [93, 136].
- **Multi$** combines features from line distillation [141] and spatial-temporal caches [59]. It is an aggressive design that partitions a cache into two: a line organized cache (LOC) and a word-organized cache (WOC). At insertion time, Multi$ uses the spatial granularity hints to direct the cache to either refill words (into the WOC) or the entire line. We investigate two design points: 50% of the cache as a WOC (Multi$-50) and 25% of cache as a WOC (Multi$-25).

Sector, Sector-Pre, Multi$, and *Amoeba-Cache* all use the same spatial predictor hints. On a demand miss, they prefetch all the sub-blocks needed together. Prefetching only changes the timing of an access; it does not change the miss bandwidth and cannot remove the misses caused by cache pollution.

Energy and Storage The sector approaches impose low hardware complexity and energy penalty. To filter out unused words, the sector granularity has to be close to word granularity; we explored 2words/sector which leads to a storage penalty of $\simeq$64KB for a 1MB cache. In Multi$, the WOC increases associativity to the number of words/block. Multi$-25 partitions a 64K 4-way cache into a 48K 3-way LOC and a 16K 8-way WOC, increasing associativity to 11. For a 1M cache, Multi$-50 increases associativity to 36. Compared to Fixed, Multi$-50 imposes over 3$\times$ increase in lookup latency, 5$\times$ increase in lookup energy, and $\simeq$4$\times$ increase in tag storage. *Amoeba-Cache* provides a more scalable approach to using adaptive cache lines since it varies the storage dedicated to the tags based on the spatial locality in the application.

Miss Rate and Bandwidth Figure 6.15 summarizes the comparison; we focus on the moderate and low utilization groups of applications. On the high utilization group, all designs other than Sector have comparable miss rates. *Amoeba-Cache* improves miss rate to within 5%—6% of the Fixed-2$\times$ for the low group and within 8%—17% for the moderate group. Compared to the Fixed-2$\times$, *Amoeba-Cache* also lowers bandwidth by 40% (64K cache) and 20% (1M cache). Compared to Sector-Pre (with prefetching), *Amoeba-Cache* is able to adapt better with flexible granularity and achieves lower miss rate (up to 30% @ 64K and 35% @ 1M). Multi$'s benefits are proportional to the fraction of the cache organized as a WOC; Multi$-50 (18-way@64K and 36-way@1M) is needed to match the miss rate of *Amoeba-Cache*. Finally, in the moderate group, many applications exhibit strided access. Compared to Multi-$'s WOC, which fetches individual words, *Amoeba-Cache*



(a) 64K - Low (b) 64K - Moderate

(c) 1M - Low (d) 1M - Moderate

Figure 6.15: Relative miss rate and bandwidth for different caches. Baseline (1,1) is the Fixed-2x design. Labels: • Fixed-2x, ○ Sector approaches. ∗: Multi$, △ Amoeba. (a),(b) 64K cache (c),(d) 1M cache. Note the different Y-axis scale for each group.

increases bandwidth since it chooses to fetch the contiguous chunk in order to lower miss rate.

6.10 Multicore Shared Cache

We evaluate a shared cache implemented with the *Amoeba-Cache* design. By dynamically varying the cache block size and keeping out unused words, the *Amoeba-Cache* effectively minimizes the footprint of an application. Minimizing the footprint helps multiple applications effectively share the cache space. We experimented with a 1M shared *Amoeba-Cache* in a 4 core system. Table 6.8 shows the application mixes; we chose a mix of applications across all groups. We tabulate the change in miss rate per thread and the overall change in bandwidth for *Amoeba-Cache* with respect to a fixed granularity cache running the same mix. Minimizing the overall footprint enables a reduction in the miss rate of each application

in the mix. The commercial workloads (SpecJBB and TPC-C) are able to make use of the space available and achieve a significant reduction in miss rate (avg: 18%). Only two applications suffered a small increase in miss rate (x264 Mix#2: 2% and ferret Mix#3: 4%) due to contention. The overall L2 miss bandwidth significantly improves, showing 16%—39% reduction across all workload mixes. We believe that the Amoeba-based shared cache can effectively enable the shared cache to support more cores and increase overall throughput. We leave the design space exploration of an Amoeba-based coherent cache for future work.

Table 6.8: Multiprogrammed Workloads on 1M Shared *Amoeba-Cache*% reduction in miss rate and bandwidth. Baseline: Fixed 1M.

	Miss	Miss	Miss	Miss	BW
Mix	T1	T2	T3	T4	(All)
jbb×2, tpc-c×2	12.38%	12.38%	22.29%	22.37%	39.07%
firefox×2, x264×2	3.82%	3.61%	–2.44%	0.43%	15.71%
cactus, fluid., omnet., sopl.	1.01%	1.86%	22.38%	0.59%	18.62%
canneal, astar, ferret, milc	4.85%	2.75%	19.39%	–4.07%	17.77%

– indicates Miss or BW higher than Fixed. T1—T4, threads in the mix; in the order of applications in the mix

6.11 Related Work

Burger et al. [30] defined cache efficiency as the fraction of blocks that store data that is likely to be used. We use the term cache utilization to identify touched versus untouched words residing in the cache. Past works [34, 136, 141] have also observed low cache utilization at specific levels of the cache. Some works [100, 101, 78, 106] have sought to improve cache utilization by eliminating cache blocks that are no longer likely to be used (referred to as dead blocks). These techniques do not address the problem of intra-block waste (i.e., untouched words).

Sector caches [144, 150] associate a single tag with a group of contiguous cache lines, allowing cache sizes to grow without paying the penalty of additional tag overhead. Sector caches use bandwidth efficiently by transferring only the needed cache lines within a sector. Conventional sector caches [144] may result in worse utilization due to the space occupied by invalid cache lines within a sector. Decoupled sector caches [150] help reduce the number of invalid cache lines per sector by increasing the number of tags per sector. Compared to the Amoeba cache, the tag space is a constant overhead, and limits the # of invalid sectors that can be eliminated. Pujara et al. [136] consider a word granularity sector cache, and use a predictor to try and bring in only the used words. Our results (see Figure 6.15) show that smaller granularity sectors significantly increase misses, and optimizations that prefetch [136] can pollute the cache and interconnect with unused words.

Line distillation [141] applies filtering at the word granularity to eliminate unused words in a cache block at eviction. This approach requires part of the cache to be organized as a word-organized cache, which increases tag overhead, complicates lookup, and bounds performance improvements. Most importantly, line distillation does not address the bandwidth penalty of unused words. This inefficiency is increasingly important to address under current and future technology dominated by interconnects [80, 125]. Veidenbaum et al. [169] propose that the entire cache be word organized and propose an online algorithm to prefetch words. Unfortunately, a static word-organized cache has a built-in tag overhead of 50% and requires energy-intensive associative searches.

Amoeba-Cache adopts a more proactive approach that enables continuous dynamic block granularity adaptation to the available spatial locality. When there is high spatial locality, the *Amoeba-Cache* will automatically store a few big cache blocks (most space dedicated for data); with low spatial locality, it will adapt to storing many small cache blocks (extra space allocated for tags). Recently, Yoon et al. have proposed an adaptive granularity DRAM architecture [183]. This provides the support necessary for supporting variable granularity off-chip requests from an *Amoeba-Cache*-based LLC. Some research [47, 38] has also focused on reducing false sharing in coherent caches by splitting/merging cache blocks to avoid invalidations. They would benefit from the *Amoeba-Cache* design, which manages block granularity in hardware.

There has a been a significant amount of work at the compiler and software runtime level (e.g. [37]) to restructure data for improved spatial efficiency. There have also been efforts from the architecture community to predict spatial locality [136, 175, 93, 184], which we can leverage to predict *Amoeba-Block* ranges. Finally, cache compression is an orthogonal body of work that does not eliminate unused words but seeks to minimize the overall memory footprint [2].

6.12 Conclusion

We propose a cache design, Amoeba-Cache, that can dynamically hold a variable number of cache blocks of different granularities. The Amoeba-Cache employs a novel organization that completely eliminates the tag array and collocates the tags with the cache block in the data array. This permits the Amoeba-Cache to trade the space budgeted for the cache blocks for tags and support a variable number of tags (and blocks). For applications that have low spatial locality, *Amoeba-Cache* can reduce cache pollution, improve the overall miss rate, and reduce bandwidth wasted in the interconnects. When applications have moderate to high spatial locality, *Amoeba-Cache* coarsens the block size and ensures good performance. Finally, for applications that are streaming (e.g., lbm), *Amoeba-Cache* can save significant energy by eliminating unused words from being transmitted over the interconnects.

Chapter 7

Software Release

This chapter details the release of software developed to conduct the research presented herein. All code is distributed via repositories on `github` under the Simon Fraser University Architecture Group organization. All code is released under the *Community Research and Academic Programming License (CRAPL)* [120] unless otherwise specified.

7.1 Path Profiling

This is an implementation of *Efficient Path Profiling* [15]. It also incoporates the *Optimal Event Counting* [13] to reduce the overhead of instrumentation. The optimization merges multiple instrumentation points to yield the same net effect. The implementation incoporates 128 bit counters (64 bit in 32-bit mode), ensuring scalability of the tool up to 128 branches in the control flow graph of the profiled functions. We implement another optimization called "segmentation" where the acyclic paths are terminated at loop entry as well as loop exit. The original Ball-Larus definition terminated paths only on backedges and function exits. This ensures better interoperability with the rest of the tool chain while also reducing complexity at negligible loss of information.

Language: C++
Infrastructure: LLVM
Release: `github.com/sfu-arch/epp`
License: `University of Illinois/NCSA Open Source License`

7.2 Path Derived Workload Suite

A workload suite which outlines the "hot" paths in each workload. Paths are identified using Ball-Larus path profiling [15]. Original workloads are derived from SPEC2000, SPEC2006 [124], PARSEC 3.0 [21] and PERFECT [17]. 29 workloads were selected from

the four suites. Each workload is compiled into a single *bitcode* file. All dependencies are statically linked in. The function consuming the largest amount of time on a real machine is identified using `gprof`. All called functions from the identified functions are inlined into their parents in a bottom-up recursive manner. The top five highest ranked paths by *coverage* (fraction of dynamic instructions executed) are outlined and presented as separate bitcode modules.

Language: LLVM Intermediate Representation
Release: `github.com/sfu-arch/pdws`
License: Workload specific license derived from their original open source license.

7.3 Needle

The tool serves three goals. They are:

1. **Construct Abstraction:** The tool constructs "Braids" (see section 4.4.2) from the dynamic path profile of an application. Braids are constructed from by merging paths. The create a single-entry, single-exit region with assertions to trigger rollback in case an infrequent path is executed.

2. **Outlining:** Once the abstraction is constructed the original CFG is instrumented. All basic blocks which are part of the abstraction are merged and outlined into a new function. Live in values for the abstraction (defined outside the region and used inside) are passed in as function arguments whereas live out values (defined inside and used outside) are returned in a packed struct. Returned values are selected at merge points in the CFG for the original vs the outlined abstraction version.

3. **Software Speculation:** Software speculative frames are created by the compiler tool chain as offload targets. They may span many branches to encompass a large number of basic blocks. Side exits due to control flow assertion fail, may be taken for some inputs. The compiler instruments writes to memory to checkpoint state. Each write is preceded by a read to the same location which logs the value along with the address. When a control flow assertion fails, the logged values are restored and execution resumes from the original control flow graph.

Language: C++
Infrastructure: LLVM
Release: `github.com/sfu-arch/needle`

7.4 Fusion Simulator

Simulates the FUSION protocol, a hybrid coherence protocol tailored for accelerator architectures. The simulator modifies MacSim [81] to add support for the Ruby [115] memory system. MacSim is a trace driven simulator where traces are collected from Intel Pin [111]. The simulator provided implements a hybrid MESI+Temporal [158] coherence protocol for specialized functions.
Language: C++
Infrastructure: Pin, MacSim, GEMS (Ruby)
Release: `github.com/sfu-arch/fusion`

7.5 Amoeba Simulator

Simulates the Amoeba Cache architecture. It implements variable granularity caching with online predictors. The implementation uses a MESI four hop protocol as base. The Ruby infrastructure is extended to support variable granularity caching. The simulator is driven by memory access traces derived from Intel Pin.
Language: C++
Infrastructure: Pin, GEMS (Ruby)
Release: `github.com/sfu-arch/amoeba`

Chapter 8

Future Work and Conclusion

In this chapter we review the contributions of this dissertation and provide insights for future work. We also discuss how recent advances in compiler engineering can enable energy efficient Von-Neumman execution models 8.1.1 and simplify memory access interfaces for specialized units 8.1.2. We discuss the potential of software specialized regions where JIT-like optimizations are performed apriori using profile guided information 8.1.3. Generating independent micro-benchmarks from the path profiles may be useful in synthesizing stress tests or performance analysis of existing systems. Section 8.1.4 discusses the challenges and potential for such an approach. Finally, we summarize the contributions made by this thesis.

8.1 Concurrent and Future Work

8.1.1 Macro Instructions from Sequentially Dependent Operations

In chapter 4 we demonstrated the utility of building specialized units for BL-Paths and Braids. While beneficial, we find there is little overlap in frequently executed BL-Paths and Braids across workloads. This leads to concerns about reusability and cost of reconfiguration in a scenario where multiple workloads are executed on the same machine. This is stark contrast to the general purpose processor. Furthermore, the use of spatial architectures for specialization introduces inefficiencies. The lack of temporal reuse of each functional unit on the spatial fabric and energy cost of data migration can lead to increased overall power consumption.

To address these shortcomings, we study the construction of macro operation chains. Sequentially dependent operations are "chained" together to form macro-instructions. Within each macro instruction, data values are bypassed. Using a lane based execution model and static chain instruction scheduling enables reuse of functional units. Parallelism present in the dataflow graph of the specialized region is expressed at the macro-instruction granularity. Using multiple chain execution lanes allows for the extraction of chain level parallelism. The chain abstraction for macro-instructions is enabled by software speculatioan. This allows us

to construct coarse enough regions across basic blocks to make it profitable. Research which investigates the chain based abstraction for efficient execution using the Von-Neumman model has been published at the 49^{th} IEEE/ACM International Symposium on Microarchitecture with co-authors Amirali Sharifian, Apala Guha and Arrvindh Shriraman.

8.1.2 Eliminating the Load-Store Queue for Specialized Units

Specialized architectures often rely on the Load-Store Unit of the processor to disambiguate conflicting memory addresses at runtime [127, 68] others may serialize accesses [171, 32]. Compiler based *alias analyses* can reason about abstract memory locations in order to determine whether a pair of memory operations can dynamically issue addresses that conflict. Using this information, a compiler targeting a spatial fabric can introduce dependency edges where necessary while eliding edges where it can be proved that no conflicts can exist. In cases where it cannot be proved that a conflict does not exist a dependency edge can be inserted in the dataflow graph. This ensures correct execution at the expense of performance. While the techniques are generalized, software speculation and simplified control flow allow for more robust alias analyses.

8.1.3 Software specialization based on dynamic profiling

Just-In-Time (JIT) compilers often perform simple optimizations to frequently executed code traces prior to execution. These techniques have been integrated in modern JIT compilers such as Java Virtual Machines from IBM [165]. A priori path profiling can determine the hot code segments in a program. We hypothesize that software based instrumentation for runtime path profiling may be too heavy weight for JIT compiled or even interpreted languages. Tracing JIT's such as PyPy (for Python) or TraceMonkey (Javascript) use low overhead techniques which track looping traces (sequences of basic blocks). Our optimized implementation of path profiling adds 20-40% runtime overhead. Additionally, dynamically typed languages executed on tracing JITs include added checks to ensure type matches before executing an optimized trace. Such "versioning" may lead to multiple variants of the same path; paths are determined by control flow only. Due to these reasons, we believe runtime path profiling for dynamically compiled languages has limited scope to improve performance.

We conducted an experiment to determine the effectiveness of standard compiler optimizations of frequently executed paths and "Braids". Specialized paths consist of control flow assertions only. They do not contain branches. Braids include only those branches which lead to blocks identified in the profiling phase as "hot". The simplification of control flow allows for deeper compiler optimizations above those applied already. In practice, compiler optimization heuristics are limited to simple control flow scenarios as it becomes harder to reason about multiple conditions and their flows of control. Techniques such

as predication [5] convert control flow to data flow so that they can be reasoned about within a single heuristic. These techniques have been known to produce better instruction schedules for VLIW processors. Figure 8.1 shows our observations on the impact of compiler optimizations once control flow has been simplified or eliminated altogether.

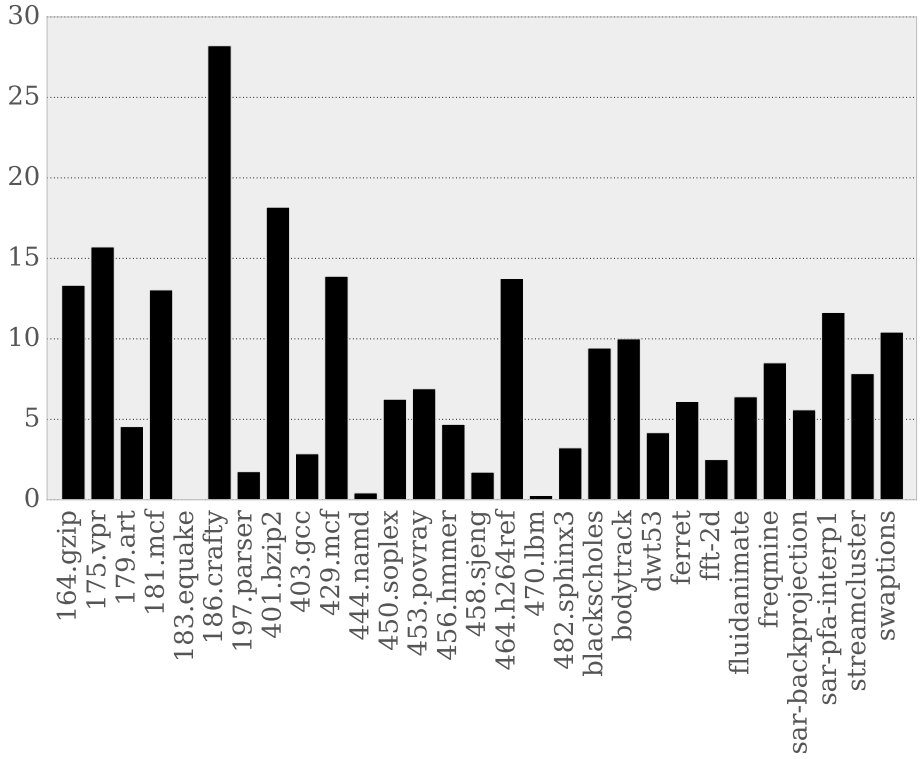


Figure 8.1: % Reduction in IR Instructions

Using the tool chain for software speculation described in section 4.5, we construct frames which contain the hottest braid. We then apply standard compiler optimizations (O2) available in LLVM. The original control flow graph was optimized with O2 prior to profiling and analyses. The impact of compiler optimizations is quantified by the removal of IR instructions. Overall, we find that many benchmarks see significant reduction in the number of operations. The median percent reduction in ops for the braids was 6% with a maximum of 28% (186.crafty). We only study Braids as we find that the cost of rollback for paths is prohibitive. Braids reduce the possibility of side exits as then incorporate all the frequently executed basic blocks in the dynamic profile of the workloads.

Speculatively executing the optimized version of a hot region on a general purpose processor may lead to overall program speedup. However, the cost of restoring program state when a side exit is taken can be prohibitive. Fortunately, we can borrow from existing work in transactional memory. Modern microprocessors from Intel (post Haswell) and IBM (Power8 onwards) incorporate hardware support for transactional memory. Using these can

provide low cost rollback mechanisms. Thus for certain programs, we hypothesize that an overall speedup can be achieved via software specialization of frequently executed regions.

8.1.4 Micro-Workload Generation

To understand the behaviour of complex workloads, often computer architects reconstruct a synthetic micro benchmark. The goal is to isolate the behaviour of the workload to be able to study it in greater detail. Often more computationally intensive tools are brought to bear on the micro-benchmark which previously was not possible. This task has usually been performed manually. We hypothesize that automated construction of functionally identical micro-benchmarks can help architecture studies. We have implemented a value profiling framework at the path and Braid granularity. Coupled with a harness which repeatedly invokes a path with the profiled values will create a synthetic micro-benchmark which replicates the functional behaviour dominant in the program. A key challenging issue is the replication of memory access behaviour. There are a few options to consider in the replication of memory system behaviour. One can profile the values which each memory instruction reads or writes in the original program to construct a *shadow map*, a data structure which will serve as a look up table instead of accessing memory. However, this may require runtime checks for operations which cannot be identified as accessing a unique location statically using alias analysis. Thus there may be significant perturbation in the overall behaviour of the benchmark depending on the number of memory operations. Alternatively, the proposed harness can allocate memory which the operations can access at runtime. The memory needs to be initialized to the values profiled at runtime. The memory state needs to be represented in an abstract manner with the pointers to relative offsets. Memory operations within the region may need to be rewritten to point to new locations allocated by the harness. System features like Address Space Layout Randomization may further complicate the issues with replaying memory behaviour.

8.2 Summary of contributions

This dissertation describes generalized methods for application specific hardware specialization. We leverage program paths as a useful abstraction to reason about specialization. We have built an LLVM based tool-chain to profile applications with low overhead. From the profile we further extract and characterize paths and assess their amenability for specialization. We have released a workload suite (SPEC-AX) derived from well known microprocessor benchmarks. They highlight the hot paths within the program so that computer architects can easily target the dominant behaviour that manifests in the program for specialization.

We build further transformation passes which merge paths based on their control flow to create “Braids”. This allows for fine-grained control of application specific specialization with strong guarantees on the amount of work offloaded.

We develop hardware mechanisms which mitigate integration issues with respect to data movement. The Fusion coherence protocol eliminates redundant data movement within the accelerator domain. A time-stamp based protocol is used in the accelerator domain to reduce the overhead of message which allowing for fine-grained data sharing with ease.

Finally, the Amoeba cache architecture uses adaptive granularity cache blocks to eliminate waste. Only data predicted to be used by the workload is fetched into the cache increasing utilization, thus improving hit rate and lowering energy consumption.

Bibliography

[1] Neha Agarwal, David Nellans, Eiman Ebrahimi, Thomas F Wenisch, John Danskin, and Stephen W Keckler. Selective gpu caches to eliminate cpu-gpu hw cache coherence. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 494–506. IEEE, 2016.

[2] Alaa R Alameldeen. *Using compression to improve chip multiprocessor performance*. PhD thesis, UNIVERSITY OF WISCONSIN-MADISON, 2006.

[3] David Albonesi, Kodi A, and Stojanovic V. Proceedings of the nsf workshop on emerging technologies for interconnects(WETI). *NSF workshop on emerging technologies for interconnects(WETI)*, 2012.

[4] Alan Allan. The international technology roadmap for semiconductors 2.0. 2015.

[5] J R Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. Rice University, ACM, January 1983.

[6] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[7] Anandtech. Tick-tock on the rocks. `http://www.anandtech.com/show/9447/intel-10nm-and-kaby-lake`.

[8] Akhil Arunkumar, Shin-Ying Lee, and Carole-Jean Wu. Id-cache: Instruction and memory divergence based cache management for gpus. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'16)*.

[9] David I August, Daniel A Connors, Scott A Mahlke, John W Sias, Kevin M Crozier, Ben-Chung Cheng, Patrick R Eaton, Qudus B Olaniran, and Wen-mei W Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[10] David I August, Wen-mei W Hwu, and Scott A Mahlke. A framework for balancing control flow and predication. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, 1997.

[11] Rajeev Balasubramonian, Naveen Muralimanohar, Karthik Ramani, and Venkatanand Venkatachalapathy. Microarchitectural wire management for performance and power in partitioned architectures. In *11th International Symposium on High-Performance Computer Architecture*, pages 28–39. IEEE, 2005.

[12] James Balfour. EFFICIENT EMBEDDED COMPUTING. `http://cva.stanford.edu/publications/2010/jbalfour-thesis.pdf`.

[13] Thomas Ball. Efficiently counting program events with support for on-line queries. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1399–1410, 1994.

[14] Thomas Ball. The concept of dynamic analysis. In *Software Engineering FSE'99*, pages 216–234. Springer, 1999.

[15] Thomas Ball and James R. Larus. Efficient Path Profiling. In *Proceedings of the 1996 Annual IEEE/ACM International Symposium on Microarchitecture*, 1996.

[16] Nagesh Bangalore Lakshminarayana. Efficient graph algorithm execution on data-parallel architectures. 2014.

[17] Kevin Barker, Thomas Benson, Dan Campbell, David Ediger, Roberto Gioiosa, Adolfy Hoisie, Darren Kerbyson, Joseph Manzano, Andres Marquez, Leon Song, Nathan Tallent, and Antonino Tumeo. *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*. Pacific Northwest National Laboratory and Georgia Tech Research Institute, December 2013. `http://hpc.pnnl.gov/projects/PERFECT/`.

[18] Robert H. Bell and Lizy K. John. Efficient power analysis using synthetic testcases. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'05)*, pages 110–118, October 2005.

[19] J Benson, R Cofell, C Frericks, Chen-Han Ho, V Govindaraju, T Nowatzki, and K Sankaralingam. Design, integration and implementation of the DySER hardware accelerator into OpenSPARC. pages 1–12, 2012.

[20] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th Parallel Architectures and Compilation Techniques*, 2008.

[21] Christian Bienia and Kai Li. Benchmarking modern multiprocessors. 2011.

[22] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[23] Geoffrey Blake, Ronald G Dreslinski, Trevor Mudge, and Krisztián Flautner. Evolution of thread-level parallelism in desktop applications. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 302–313. ACM, 2010.

[24] B. Blaner, B. Abali, B.M. Bass, S. Chari, R. Kalla, S. Kunkel, K. Lauricella, R. Leavens, J.J. Reilly, and P.A. Sandon. Ibm power7+ processor on-chip accelerators for cryptography and active memory expansion. *IBM Journal of Research and Development*, 57(6):3:1–3:16, Nov 2013.

[25] M. Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, Winter 2007.

[26] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[27] Jeffrey Brown, Sandra Woodward, Brian Bass, and Charlie Johnson. IBM Power Edge of Network Processor: A Wire-Speed System on a Chip. *IEEE Micro*, 31(2):76–85, 2011.

[28] Stephen D Brown, Robert J Francis, Jonathan Rose, and Zvonko G Vranesic. *Field-programmable gate arrays*, volume 180. Springer Science & Business Media, 2012.

[29] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In *ACM SIGARCH Computer Architecture News*, volume 32, pages 14–26. ACM, 2004.

[30] Doug Burger, James R Goodman, and Alain Kagi. *The declining effectiveness of dynamic caching for general-purpose microprocessors*. University of Wisconsin-Madison. Computer Sciences Department, 1995.

[31] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[32] Andrew Canis, Jongsok Choi, Blair Fort, Ruolong Lian, Qijing Huang, Nazanin Calagar, Marcel Gort, Jia Jun Qin, Mark Aldham, Tomasz Czajkowski, Stephen Brown, and Jason Anderson. From software to accelerators with LegUp high-level synthesis. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–9. IEEE.

[33] Sai Rahul Chalamalasetti, Kevin Lim, Mitch Wright, Alvin AuYoung, Parthasarathy Ranganathan, and Martin Margala. An FPGA memcached appliance. `http://dl.acm.org/citation.cfm?id=2435264.2435306`, 2013.

[34] Chi F Chen, S-H Yang, Babak Falsafi, and Andreas Moshovos. Accurate and complexity-effective spatial pattern prediction. In *Software, IEE Proceedings-*, pages 276–287. IEEE, 2004.

[35] Fred Chen, Matthew Spencer, Rhesa Nathanael, Chengcheng Wang, Hossein Fariborzi, Abhinav Gupta, Hei Kam, Vincent Pott, Jaeseok Jeon, Tsu-Jae King Liu, et al. Demonstration of integrated mico-electro-mechanical switch circuits for vlsi applications. Institute of Electrical and Electronics Engineers, 2010.

[36] Tao Chen and G Edward Suh. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[37] Trishul M Chilimbi, Mark D Hill, and James R Larus. Cache-conscious structure layout. In *ACM SIGPLAN Notices*, volume 34, pages 1–12. ACM, 1999.

[38] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V Adve, Vikram S Adve, Nicholas P Carter, and Ching-Tsun Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 155–166. IEEE, 2011.

[39] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, 2004.

[40] Charlie Curtsinger and Emery D Berger. Stabilizer: statistically sound performance evaluation. *ACM SIGARCH Computer Architecture News*, 41(1):219–228, 2013.

[41] Hamed F Dadgour and Kaustav Banerjee. Design and analysis of hybrid nems-cmos circuits for ultra low-power applications. In *2007 44th ACM/IEEE Design Automation Conference*, pages 306–311. IEEE, 2007.

[42] Bill Dally. Power, programmability, and granularity: The challenges of exascale computing. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 878–878. IEEE, 2011.

[43] WJ Dally and B Towles. Route packets, not wires. In *Proceedings of the Design Automation Conference (DAC)*, pages 18–22.

[44] Andrew Danowitz, Kyle Kelley, James Mao, John P Stevenson, and Mark Horowitz. Cpu db: recording microprocessor history. *Communications of the ACM*, 55(4):55–63, 2012.

[45] Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts. Coarse-grained reconfigurable array architectures. In *Handbook of signal processing systems*, pages 553–592. Springer, 2013.

[46] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[47] Czarek Dubnicki and Thomas J LeBlanc. Adjustable block size coherent caches. *ACM SIGARCH Computer Architecture News*, 20(2):170–180, 1992.

[48] Lieven Eeckhout, Robert H Bell, Bastiaan Stougie, Koen De Bosschere, and Lizy K John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 350–361. IEEE, 2004.

[49] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[50] Amin Farmahini-Farahani, Nam Sung Kim, and Katherine Morrow. Energy-efficient reconfigurable cache architectures for accelerator-enabled embedded systems. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 211–220. IEEE, 2014.

[51] Muhammad Umar Farooq, Lizy John, and Margarida F Jacome. Compiler Controlled Speculation for Power Aware ILP Extraction in Dataflow Architectures. In *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*. University of Texas at Austin, Springer-Verlag, December 2008.

[52] Tom Feist. Vivado design suite. *White Paper*, 5, 2012.

[53] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[54] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[55] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. *Copenhagen University College of Engineering*, 2011.

[56] C Frericks, R Cofell, and K Sankaralingam. Performance evaluation of a DySER FPGA prototype system spanning the compiler, microarchitecture, and hardware implementation. *... Software (ISPASS)*, 2015.

[57] Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor. Kremlin: Rethinking and rebooting gprof for the multicore age. In *PLDI '11: Proceedings of the Conference on Programming Language Design and Implementation*, 2011.

[58] Mark Gebhart, Bertrand A Maher, Katherine E Coons, Jeff Diamond, Paul Gratz, Mario Marino, Nitya Ranganathan, Behnam Robatmili, Aaron Smith, James Burrill, Stephen W Keckler, Doug Burger, and Kathryn S McKinley. An evaluation of the TRIPS computer system. In *PROC of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[59] Antonio González, Carlos Aliagas, and Mateo Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 217–226. ACM, 2014.

[60] Cecilia González-Álvarez, Jennifer B Sartor, Carlos Álvarez, Daniel Jiménez-González, and Lieven Eeckhout. Automatic design of domain-specific instructions for low-power processors. *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–8, 2015.

[61] James Goodman. Source Snooping Cache Coherence Protocols. *Science*, 2009.

[62] Goodridge. The effect and technique of system coherence in arm multicore technology. 2008.

[63] Google. Google performance tools. `https://github.com/gperftools/gperftools`.

[64] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):0038–51, 2012.

[65] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982.

[66] Yakun Sophia Shao Brandon Reagen Gu and Yeon Wei David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures.

[67] Nagendra Gulur, Mahesh Mehendale, R Manikantan, and R Govindarajan. Bi-modal dram cache: Improving hit rate, hit latency and bandwidth. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 38–50. IEEE, 2014.

[68] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[69] Diana Guttman and Mahmut Taylan Kandemir. Performance and energy evaluation of data prefetching on intel xeon phi. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 288–297. IEEE, 2015.

[70] 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 *Proceedings of the 37th Annual International Symposium of Computer Architecture*, 2010.

[71] L Hammond, BA Hubbert, M Siu, MK Prabhu, M Chen, and K Olukotun. The stanford hydra microprocessor. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, pages 67–77, 1996.

[72] Joel Hestness, Stephen W. Keckler, and David A. Wood. A comparative analysis of microarchitecture effects on cpu and gpu memory system behavior. In *IEEE International Symposium on Workload Characterization*, 2014.

[73] Joel Hestness, Stephen W Keckler, and David A Wood. Gpu computing pipeline inefficiencies and optimization opportunities in heterogeneous cpu-gpu processors. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 87–97. IEEE, 2015.

[74] Justin Holewinski, Ragavendar Ramamurthi, Mahesh Ravishankar, Naznin Fauzia, Louis-NoÃńl Pouchet, Atanas Rountev, and P. Sadayappan. Dynamic trace-based analysis of vectorization potential of applications. In *PLDI*, pages 371–382. ACM, 2012.

[75] Yatin Hoskote, Sriram Vangal, Arvind Singh, Nitin Borkar, and Shekhar Borkar. A 5-ghz mesh interconnect for a teraflops processor. *IEEE Micro*, 27(5):51–61, 2007.

[76] Kenneth Hoste, Aashish Phansalkar, Lieven Eeckhout, Andy Georges, Lizy K John, and Koen De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 114–122. ACM, 2006.

[77] Cheng-Ta Hsieh and M. Pedram. Microprocessor power estimation using profile-driven program synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1998.

[78] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 209–220. IEEE, 2002.

[79] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[80] Christopher J Hughes, Changkyu Kim, Yen-Kuang Chen, et al. Performance and energy implications of many-core caches for throughput computing. *IEEE micro*, 30(6):25–35, 2010.

[81] Nagesh B. Lakshminarayana Hyesoon Kim, Jaekyu Lee. Macsim : Simulator for heterogeneous architecture - https://code.google.com/p/macsim/. `https://code.google.com/p/macsim/`.

[82] IEEE. Ieee jun-ichi nishizawa medal. `http://ethw.org/IEEE_Jun-ichi_Nishizawa_Medal`.

[83] Intel. Xeon chip with integrated fpga. 2014.

[84] Adrian M Ionescu and Heike Riel. Tunnel field-effect transistors as energy-efficient electronic switches. *Nature*, 479(7373):329–337, 2011.

[85] Intel James R. Intel processor tracing. `https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`, 2014.

[86] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Kismet: parallel speedup estimates for serial programs. In *ACM SIGPLAN Notices*, volume 46, pages 519–536. ACM, 2011.

[87] Djordje Jevdjic, Gabriel H Loh, Cansu Kaynak, and Babak Falsafi. Unison cache: A scalable and effective die-stacked dram cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–37. IEEE, 2014.

[88] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. Die-stacked dram caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache. *ACM SIGARCH Computer Architecture News*, 41(3):404–415, 2013.

[89] A Joshi, L Eeckhout, R H Bell, and L John. *Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks*. October 2006.

[90] K Kamil, Miquel Moreto, Francisco J Cazorla, Mateo Valero, et al. Adapting cache partitioning algorithms to pseudo-lru replacement policies. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[91] S. Kaxiras and G. Keramidas. Sarc coherence: Scaling directory cache coherence in performance and power. *Micro, IEEE*, 30(5):54–65, Sept 2010.

[92] Stefanos Kaxiras and Alberto Ros. A new perspective for efficient virtual-cache coherence. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 1–12, April 2013.

[93] Sanjeev Kumar and Christopher Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 357–368. IEEE Computer Society, 1998.

[94] Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula. Fusion : Design Tradeoffs in Coherence Hierarchies for Accelerators. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA 2015, New York, NY, USA, jun 2015. ACM.

[95] Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula. Fusion: Design tradeoffs in coherent cache hierarchies for accelerators. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 733–745, New York, NY, USA, 2015. ACM.

[96] Snehasish Kumar, Nick Sumner, and Arrvindh Shriraman. SPEC-AX : Extracting Accelerator Benchmarks from Microprocessor Benchmarks. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*, IISWC 2016, September 2016.

[97] Snehasish Kumar, Nick Sumner, Vijayalakshmi Srinivasan, Steve Margerm, and Arrvindh Shriraman. Needle : Leveraging program analysis to extract accelerators from whole programs. In *Proceedings of the 23rd ACM International Conference on High Performance Computer Architecture*, HPCA 2017, feb 2017.

[98] Snehasish Kumar, Naveen Vedula, Arrvindh Shriraman, and Vijayalakshmi Srinivasan. DASX : Hardware accelerator for software data structures. In *Proceedings of the 29th ACM International Conference on Supercomputing*, ICS 2015, june 2015.

[99] Snehasish Kumar, Hongzhou Zhao, Arrvindh Shriraman, Eric Matthews, Sandhya Dwarkadas, and Lesley Shannon. Amoeba-Cache : Adaptive Blocks for Eliminating Waste in the Memory Hierarchy. In *Proceedings of the 45th Annual IEEE/ACM*

International Symposium on Microarchitecture, MICRO 2012, Washington, DC, USA, dec 2012. IEEE Computer Society.

[100] An-Chow Lai and Babak Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 139–148. IEEE, 2000.

[101] Alvin R Lebeck and David A Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 48–59. ACM, 1995.

[102] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[103] Zhen Li, Ali Jannesari, and Felix Wolf. Discovery of potential parallelism in sequential programs. In *2013 42nd International Conference on Parallel Processing*, pages 1004–1013. IEEE, 2013.

[104] Kevin Lim, David Meisner, Ali G Saidi, Parthasarathy Ranganathan, and Thomas F Wenisch. Thin servers with smart pipes: designing SoC accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[105] Feng Liu, Heejin Ahn, Stephen R Beard, Taewook Oh, and David I August. Dynaspam: dynamic spatial architecture mapping using out of order instruction schedules. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 541–553. IEEE, 2015.

[106] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 222–233. IEEE Computer Society, 2008.

[107] Diego R Llanos and Belén Palop. Tpcc-uva: an open-source tpc-c implementation for parallel and distributed systems. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 8–pp. IEEE, 2006.

[108] Gabriel H Loh and Mark D Hill. Efficiently enabling conventional block sizes for very large die-stacked dram caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 454–464. ACM, 2011.

[109] Robert Love. *Linux Kernel Development (Novell Press)*. Novell Press, 2005.

[110] P Geoffrey Lowney, Stefan M Freudenberger, Thomas J Karzes, W D Lichtenstein, Robert P Nix, John S O'Donnell, and John Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1-2), May 1993.

[111] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[112] Nir Magen, Avinoam Kolodny, Uri Weiser, and Nachum Shamir. Interconnect-power dissipation in a microprocessor. In *Proceedings of the 2004 international workshop on System level interconnect prediction*, pages 7–13. ACM, 2004.

[113] Scott A Mahlke, Richard E Hank, James E McCormick, David I August, and Wen-mei W Hwu. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[114] Scott A Mahlke, David C Lin, William Y Chen, Richard E Hank, and Roger A Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual IEEE/ACM International Symposium on Microarchitecture*, 1992.

[115] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005.

[116] Daniel S McFarlin, Charles Tucker, and Craig Zilles. Discerning the dominant out-of-order performance advantage: is it speculation or dynamism? In *Proceedings of the eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[117] Wen mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *THE JOURNAL OF SUPERCOMPUTING*, 7:229–248, 1993.

[118] Jiayuan Meng, Xingfu Wu, Vitali Morozov, Venkatram Vishwanath, Kalyan Kumaran, and Valerie Taylor. SKOPE. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, 2014.

[119] R Merritt. ARM CTO: Power surge could create "dark silicon". *EE Times, Oct*, 2009.

[120] Matt Might. The community research and academic programming license. `http://matt.might.net/articles/crapl/`.

[121] S. L. Min and J. L. Baer. Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps. *IEEE Trans. Parallel Distrib. Syst.*, 3(1), 1992.

[122] Gordon Moore. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, 1965.

[123] Gordon E Moore. No exponential is forever: but "forever" can be delayed! In *Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC. 2003 IEEE International*, pages 20–23. IEEE, 2003.

[124] Matthias S. Muller, John Baron, William C. Brantley, Huiyu Feng, Daniel Hackenberg, Robert Henschel, Gabriele Jost, Daniel Molka, Chris Parrott, Joe Robichaux, Pavel Shelepugin, Matthijs Waveren, Brian Whitney, and Kalyan Kumaran. Spec omp2012

– an application benchmark suite for parallel systems using openmp. In *OpenMP in a Heterogeneous World*, volume 7312 of *Lecture Notes in Computer Science*, pages 223–236. Springer Berlin Heidelberg, 2012.

[125] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[126] Bradford Nicbols, Dick Buttlar, and PJ FARRELL. Pthread programming, 1996.

[127] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. Exploring the potential of heterogeneous von neumann/dataflow execution models. IEEE Computer Architecture Letters, 2015.

[128] Tony Nowatzki, Venkatraman Govindaraju, and Karu Sankaralingam. A Graph-Based Program Representation for Analyzing Hardware Specialization Approaches. *IEEE Computer Architecture Letters*, 2015.

[129] Lena E Olson, Jason Power, Mark D Hill, and David A Wood. Border control: sandboxing accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 470–481. ACM, 2015.

[130] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. Triggered instructions: a control paradigm for spatially-programmed architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 1–12, April 2013.

[131] Hyunchul Park, Yongjun Park, and Scott Mahlke. Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[132] Yongjun Park, Hyunchul Park, and Scott Mahlke. *CGRA express: accelerating execution using dynamic operation fusion.* accelerating execution using dynamic operation fusion. ACM, New York, New York, USA, October 2009.

[133] Editor Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. In *DARPA IPTO*, September 2008.

[134] Christian Pilato and Fabrizio Ferrandi. Bambu: A free framework for the high-level synthesis of complex applications. *University Booth of DATE*, 2012.

[135] D N Pnevmatikatos and G S Sohi. Guarded execution and branch prediction in dynamic ILP processors. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994.

[136] Prateek Pujara and Aneesh Aggarwal. Increasing the cache efficiency by eliminating noise. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 145–154. IEEE, 2006.

[137] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth, Gopal Jan, Gray Michael, Haselman Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi, and Xiao Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[138] 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 *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[139] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, C Kozyrakis, and M Horowitz. Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing. *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 1–12, April 2013.

[140] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 381–391. ACM, 2007.

[141] Moinuddin K Qureshi, M Aater Suleman, and Yale N Patt. Line distillation: Increasing cache capacity by filtering unused words in cache lines. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 250–259. IEEE, 2007.

[142] Brandon Reagen, Robert Adolf, Sophia Yakun Shao, Gu-Yeon Wei, and David Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2014.

[143] Scott Ricketts. Efficient Cache-Coherent Migration for Heterogeneous Coprocessors in Dark Silicon Limited Technology.

[144] Jeffrey B Rothman and Alan Jay Smith. The pool of subsectors cache design. In *Proceedings of the 13th international conference on Supercomputing*, pages 31–42. ACM, 1999.

[145] Jack Sampson, Ganesh Venkatesh, Nathan Goulding-Hotta, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor. Efficient complex operators for irregular codes. In *Proceedings of the 17th High Performance Computer Architecture*, 2011.

[146] Steven S Lumetta Sanjay J Patel. rePLay: A Hardware Framework for Dynamic Program Optimization. *IEEE Transactions on Computers archive. Volume 50*, 1999.

[147] SeekingAlpha. Brian krzanich on q2 2015 - earnings call transcript. `http://seekingalpha.com/article/3329035`.

[148] Charles L. Seitz. Let's route packets instead of wires. In *Proceedings of the Sixth MIT Conference on Advanced Research in VLSI*, AUSCRYPT '90, pages 133–138, Cambridge, MA, USA, 1990. MIT Press.

[149] SemiWiki. Are 28nm transistors the cheapest .. forever? `https://www.semiwiki.com/forum/content/2768-28nm-transistors-cheapest-forever.html`.

[150] André Seznec. Decoupled sectored caches: conciliating low tag implementation cost. In *ACM SIGARCH Computer Architecture News*, volume 22, pages 384–393. IEEE Computer Society Press, 1994.

[151] Yakun Sophia Shao and David Brooks. ISA-independent workload characterization and its implications for specialized architectures. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 245–255. IEEE, April 2013.

[152] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David M Brooks. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*, pages 97–108, 2014.

[153] Yakun Sophia Shao, Sam Likun Xi Vijayalakshmi Srinivasan, and Gu-Yeon Wei David Brooks. Co-designing accelerators and soc interfaces using gem5-aladdin. In *International Symposium on Microarchitecture (MICRO)*, 2016.

[154] Amirali Sharifian, Snehasish Kumar, Apala Guha, and Arrvindh Shriraman. Chainsaw: Von-neumann accelerators to leverage fused instruction chains. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 454–464. ACM, 2016.

[155] Keun Sup Shim *et al.* Library Cache Coherence. Csail technical report mit-csail-tr-2011-027, May 2011.

[156] Matthew D Sinclair, Johnathan Alsop, and Sarita V Adve. Efficient gpu synchronization without scopes: saying no to complex consistency models. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 647–659. ACM, 2015.

[157] H Singh, Ming-Hau Lee, Guangming Lu, F J Kurdahi, N Bagherzadeh, and E M Chaves Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, May 2000.

[158] Inderpreet Singh, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M Aamodt. Cache coherence for gpu architectures. In *HPCA*, pages 578–590, 2013.

[159] Aaron Smith, Jon Gibson, Bertrand A Maher, Nicholas Nethercote, Bill Yoder, Doug Burger, Kathryn S McKinley, and James H Burrill. Compiling for EDGE Architectures. *CGO*, pages 185–195, 2006.

[160] Stephen Somogyi, Thomas F Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[161] E. S. Sorenson and J. K. Flanagan. Evaluating synthetic trace models using locality surfaces. In *Workload Characterization, 2002. WWC-5. 2002 IEEE International Workshop on*, pages 23–33, Nov 2002.

[162] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[163] SPEC. 429.mcf - SPEC CPU2006 Benchmark Description. `https://www.spec.org/cpu2006/Docs/429.mcf.html`.

[164] D Stasiak, R Chaudhry, D Cox, S Posluszny, J Warnock, S Weitzel, D Wendel, and M Wang. Cell processor low-power design methodology. In *Micro, ieee*, Nov-Dec. 2005.

[165] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the ibm java just-in-time compiler. *IBM systems Journal*, 39(1):175–193, 2000.

[166] Michael B Taylor. Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1131–1136. ACM, 2012.

[167] Sid-Ahmed-Ali Touati and Denis Barthou. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd conference on Computing frontiers*, pages 147–156. ACM, 2006.

[168] SMP Variable. A multi-core cpu architecture for low power and high performance. *Whitepaper-http://www. nvidia. com*, 2011.

[169] Alexander V Veidenbaum, Weiyu Tang, Rajesh Gupta, Alexandru Nicolau, and Xiaomei Ji. Adapting cache line size to application behavior. In *Proceedings of the 13th international conference on Supercomputing*, pages 145–154. ACM, 1999.

[170] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. *SD-VBS: The San Diego Vision Benchmark Suite*. IEEE, October 2009.

[171] 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 *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[172] Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. Qscores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 163–174. ACM, 2011.

[173] Stavros Volos, Javier Picorel, Babak Falsafi, and Boris Grot. Bump: Bulk memory access prediction and streaming. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 545–557. IEEE Computer Society, 2014.

[174] Miljan Vuletic, Paolo Ienne, Christopher Claus, and Walter Stechele. Multithreaded virtual-memory-enabled reconfigurable hardware accelerators. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 197–204. IEEE, 2006.

[175] Matthew A Watkins, Sally A McKee, and Lambert Schaelicke. Revisiting cache block superloading. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 339–354. Springer, 2009.

[176] Matthew A Watkins, Tony Nowatzki, and Anthony Carno. Software transparent dynamic binary translation for coarse-grain reconfigurable architectures. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 138–150. IEEE, 2016.

[177] Thomas F Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Temporal Streaming of Shared Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[178] Intel Whitepaper. Improving real time performance by utilizing cache allocation technology. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf`, 2015.

[179] Wikipedia. Program analysis. `https://en.wikipedia.org/wiki/Program_analysis`.

[180] Lisa Wu, Raymond J Barker, Martha A Kim, and Kenneth A Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[181] Lisa Wu and Martha A Kim. Acceleration targets: A study of popular benchmark suites. In *The First Dark Silicon Workshop, DaSi*, 2012.

[182] Luke Yen, Stark C Draper, and Mark D Hill. Notary: Hardware techniques to enhance signatures. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 234–245. IEEE, 2008.

[183] Doe Hyun Yoon, Min Kyu Jeong, and Mattan Erez. Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput. In *ACM SIGARCH Computer Architecture News*, number 3, pages 295–306. ACM, 2011.

[184] Doe Hyun Yoon, Min Kyu Jeong, Michael Sullivan, and Mattan Erez. The dynamic granularity memory system. In *ACM SIGARCH Computer Architecture News*, number 3, pages 548–559. IEEE Computer Society, 2012.

[185] Ning Zhang and Bob Brodersen. The cost of flexibility in systems on a chip design for signal processing applications. *University of California, Berkeley, Tech. Rep*, 2002.

[186] Hongzhou Zhao, Arrvindh Shriraman, Snehasish Kumar, and Sandhya Dwarkadas. Protozoa: Adaptive granularity cache coherence. In *ACM SIGARCH Computer Architecture News*, number 3, pages 547–558. ACM, 2013.

[187] Qiaoshi Zheng, Nathan Goulding-Hotta, Scott Ricketts, Steven Swanson, Michael Bedford Taylor, and Jack Sampson. Exploring energy scalability in coprocessor-dominated architectures for dark silicon. *ACM Trans. Embed. Comput. Syst.*, 13(4s):130:1–130:24, April 2014.