ABSTRACT

KRISHNA, ANIL. Performance Modeling and Characterization of Multicore Computing Systems. (Under the direction of Dr. Yan Solihin.)

The microprocessor industry and the architecture research community finds itself in interesting times. The opportunities enabled by the continued scaling in transistor device density are being challenged by pin-limited off-chip bandwidths and increasing transistor power density. These challenges have driven chip design to first embrace multi-core designs, and then embrace special purpose cores, or, heterogeneous general purpose cores, or both. Designing multi-core chips raise several interesting research questions - some of which we address in this dissertation.

The exponential growth in transistor density suggests a similarly exponential growth in the number of on-chip computational contexts. However, bandwidth to off-chip memory is pin-limited, and, therefore, does not grow at the same exponential rate as does transistor density. This leads to a situation referred to as the bandwidth wall problem. The first goal of this dissertation is to quantify the extent to which the bandwidth wall problem affects multi-core scaling, and the extent to which various bandwidth conservation techniques are able to mitigate this problem. We develop a simple but powerful analytical model to predict the number of on-chip cores that a multi-core chip can support in the presence of the bandwidth wall. This model confirms that the bandwidth wall can severely limit core scaling if additional bandwidth conservation techniques are not employed. Further analysis with this model identifies bandwidth conservation techniques that can sustain multi-core scaling for the next several technology generations.

Continued increase in the on-chip core counts makes such processor chips attractive for running parallel, multi-threaded, workloads. Multi-threaded workloads can share a non-trivial fraction of the instructions and data between threads. The second goal of this dissertation is to study the impact of such sharing on chip design. We propose a methodology that quantifies the reduction in on-chip cache miss rate that is solely attributable to the presence of data sharing. We incorporate the impact of data sharing in contemporary multi-threaded benchmarks into an analytical model that projects multi-core chip performance. We find that the optimal design point for a multi-core chip is substantially different when the impact of data sharing is considered.

There remains a continued abundance of transistors per chip, however the transistor power density is growing at each technology generation. This trend is forcing the industry to pay particular attention to the power-performance tradeoff and is encouraging hardware specialization. Instead of integrating more and more identical general purpose cores on a chip, specialized computation engines called Hardware Accelerators are being integrated on a chip. A key chip-design
challenge in this space is balancing the general purpose computation capability with hardware acceleration of selective functions, while supporting a convenient programming model. A third goal of this dissertation is to evaluate the architectural considerations, design choices and performance potential of hardware acceleration. To this end, we perform an in-depth study of IBM’s PowerEN processor, one of the first multi-core chips to integrate programmable hardware accelerators alongside general purpose cores. We find that hardware acceleration has the potential to improve throughput by orders of magnitude for some applications. We also find that a coherent shared memory architecture is an important part of enabling accelerators to be easily accessed from user level code running on the general purpose cores.

Recent research makes a case for adding heterogeneity across the general purpose cores in a multi-core chip. Heterogeneity can help improve the overall chip performance as well as its energy efficiency. However, evaluating the corresponding chip design space is a big challenge. The number of unique designs in the design space grow with the number of cores, and the type and granularity of the heterogeneity. Detailed cycle-accurate simulation is often far too slow to be able to exhaustively evaluate the design choices. Added to that is the complication of evaluating the performance of each design across the many application schedules. Even a single mix of applications can lead to a combinatorially scaling set of static application-to-core mappings as the number of non-identical cores scales linearly. A fourth goal of this dissertation is to develop a fast, analytical modeling framework, ReSHAPE, that estimates chip level performance for a heterogeneous multi-core chip, while allowing shared cache configurations and respecting the impact of limited memory bandwidth on chip performance. Being an analytical model at its core, ReSHAPE achieves orders of magnitude speedup over instruction-driven, timing simulation. We validate ReSHAPE’s accuracy against a timing approximate full-system simulator. We use ReSHAPE to study several interesting chip level optimization questions. For example, we find that as the number of cores on a chip increases, the potential benefits from heterogeneity can increase provided there is a good application-to-core mapper. Increasing the granularity of heterogeneity (the number of unique core and cache sizes), however, does not benefit beyond a point.
Performance Modeling and Characterization of Multicore Computing Systems

by

Anil Krishna

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina
2013

APPROVED BY:

__________________________  ___________________________
Dr. Eric Rotenberg  Dr. James M. Tuck

__________________________  ___________________________
Dr. Edward F Gehringer  Dr. Yan Solihin
Chair of Advisory Committee
DEDICATION

To Dr. Ramamurthy Kuppachi - his generosity, practicality and wit forever inspire
Anil Krishna was born in a small town called Bhilai, in central India, in 1978, where he completed his early schooling. He earned his bachelor’s degree in Electronics and Communications Engineering from Indian Institute of Technology (IIT), Guwahati, India in 1999. His pursuit of higher education took him to Purdue University, West Lafayette, Indiana, where he completed his master’s degree in Electrical and Computer Engineering in 2001. He then started working for IBM’s System Architecture and Performance team in Research Triangle Park, North Carolina, focusing his efforts on modeling and design guidance for the PowerPC-based processors. In 2005, while being still employed full time at IBM, Anil started working towards a PhD degree in the Department of Electrical and Computer Engineering at North Carolina State University, under the advisorship of Professor Yan Solihin. Starting late 2012 Anil works for Qualcomm’s CPU Research team in Raleigh, North Carolina.
I am indebted to my advisor, Dr. Yan Solihin, for his astute guidance and critique as I learned from him the process of conducting research. This work has benefitted from his insistence on clear hypotheses, well constructed experiments, insightful analysis, and a top-down approach to research. I thank Dr. Solihin for always giving me ample space and time to balance my full-time job and family life with my research.

I thank my committee members, Dr. Eric Rotenberg, Dr. James Tuck and Dr. Edward Gehringer, for their valuable feedback that gave shape and scope to this work. I particularly hold dear all that I learnt from Dr. Rotenberg during the several courses I took with him - directly, via course material, but also indirectly. His devotion to quality, attention to detail, and enthusiasm for teaching have influenced me and my work.

I thank my comrade in arms, Ahmad Samih, for the energy and confidence (and the sugary black tea) he brought to all our brainstorming sessions. His help with the full system simulator was crucial to this work.

I thank my fellow graduate students at the ARPERS and the CESR Lab for all their help. I am especially grateful to Sandeep Navada for his company while we were preparing for the PhD Qualifiers, as well as for his valuable advice during one particularly tough phase during the PhD.

I am thankful for the support and encouragement from my colleagues and managers at work, both at IBM and at Qualcomm. Brian Rogers, Timothy Heil, Gordon Bell, Steve VanderWiel, Nicholas Lindberg, Farnaz Toussi, Trey Cain, Rami Al Sheikh and Ken Vu were excellent mentors and research collaborators.

The pursuit of a PhD is largely a quiet and lonely endeavor. When it comes intertwined with the demands of a full-time job, there is little time left for much else. The toll this takes on family life, and the guilt that engenders in you, can only be overcome by unquestioning love, sacrifice and faith. I could not have done it without the sincere goodwill of my family and friends.

I remain eternally grateful to my parents, Venkata Rama Kishore and Subbalakshmi Kishore; it was their lifelong love, sacrifice and encouragement that gave me the emotional and intellectual ammunition to attempt this work. I thank my mother-in-law, Lakshmi Vadali, whose quiet courage and easy laughter were inspirational. My brother, Anant, and sister-in-law, Latha, were always steady in their support. My daughter, Meera, contributed in her own way - in her company all of life’s stresses dissolved. And finally, this journey of 8 years would have been impossible without the love, patience and support that my wife, Kavita Vadali, graced me with. Her companionship made this happen and makes it all worthwhile.
# TABLE OF CONTENTS

## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td></td>
</tr>
</tbody>
</table>

## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>viii</td>
<td></td>
</tr>
</tbody>
</table>

## Chapter 1 Introduction

### 1.1 The Memory Bandwidth Wall and Multi-core Scaling

### 1.2 Multi-core design in the presence of Data Sharing

### 1.3 Architecture and Performance of the Hardware Accelerators in IBM’s PowerEN Processor

### 1.4 ReSHAPE - Resource Sharing and Heterogeneity-aware Analytical Performance Estimation tool

### 1.5 Organization of the Dissertation

## Chapter 2 The Memory Bandwidth Wall and CMP Scaling

### 2.1 Assumptions, Scope, and Limitations of Study

### 2.2 Modeling CMP Memory Traffic

#### 2.2.1 The Power Law of Cache Misses

#### 2.2.2 Memory Traffic Model for CMP Systems

### 2.3 Bandwidth Wall Impact on CMP Scaling

#### 2.3.1 Bandwidth Wall Impact on CMP Scaling

#### 2.3.2 CMP Scaling and Die Area Allocation

### 2.4 Analyzing Memory Traffic Reduction Techniques

#### 2.4.1 Indirect Techniques

#### 2.4.2 Direct Techniques

#### 2.4.3 Dual Techniques

#### 2.4.4 Discussion and Recommendations

### 2.5 Related Work

### 2.6 Acknowledgments

## Chapter 3 Data Sharing in Multi-Threaded Applications and its Impact on Chip Design

### 3.1 Measuring the Impact of Data Sharing

#### 3.1.1 Why is measuring the impact of sharing difficult?

#### 3.1.2 How do current approaches measure this impact?

#### 3.1.3 Why are current approaches insufficient?

#### 3.1.4 Our approach

#### 3.1.5 Methodology

#### 3.1.6 Results

### 3.2 Impact of Sharing in a fixed-area CMP

#### 3.2.1 Simulated Machine

#### 3.2.2 Results

### 3.3 Analytical Model for Chip Performance
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>CMP system variables used in our model</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Summary of memory traffic reduction techniques</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>PARSEC and NAS benchmarks grouped by their data sharing trends</td>
<td>44</td>
</tr>
<tr>
<td>3.2</td>
<td>Parameters used in the IPC model</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Overview of PowerEN Capabilities</td>
<td>59</td>
</tr>
<tr>
<td>4.2</td>
<td>Example patterns from the complex pattern sets</td>
<td>69</td>
</tr>
<tr>
<td>4.3</td>
<td>Workload file sizes and compression ratios</td>
<td>78</td>
</tr>
<tr>
<td>4.4</td>
<td>Throughput Ratios HW vs. SW</td>
<td>80</td>
</tr>
<tr>
<td>5.1</td>
<td>Workload Characteristics inputs to ReSHAPE</td>
<td>84</td>
</tr>
<tr>
<td>5.2</td>
<td>Example configuration input to ReSHAPE</td>
<td>85</td>
</tr>
<tr>
<td>5.3</td>
<td>Workload mixes used in this work</td>
<td>93</td>
</tr>
<tr>
<td>5.4</td>
<td>Configurations used for validation of ReSHAPE</td>
<td>94</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2.1 Normalized cache miss rate as a function of cache size. ............... 12
Figure 2.2 Memory traffic as the number of CMP cores varies in the next technology generation. .................................................. 15
Figure 2.3 Die area allocation for cores and the number of supportable cores assuming constant memory traffic requirements. ......................... 16
Figure 2.4 Increase in number of on-chip cores enabled by cache compression (32 CEAs). ............................................................... 18
Figure 2.5 Increase in number of on-chip cores enabled by DRAM caches. ........ 19
Figure 2.6 Increase in number of on-chip cores enabled by 3D-stacked caches. ..... 20
Figure 2.7 Increase in number of on-chip cores enabled by filtering unused data from the cache. ......................................................... 21
Figure 2.8 Increase in number of on-chip cores enabled by smaller cores. .......... 23
Figure 2.9 Increase in number of on-chip cores enabled by link compression. .... 24
Figure 2.10 Increase in number of on-chip cores enabled by a sectored cache. .... 25
Figure 2.11 Increase in number of on-chip cores enabled by smaller cache lines. ... 26
Figure 2.12 Increase in number of on-chip cores enabled by cache+link compression. 27
Figure 2.13 Impact of data sharing on traffic. ............................................ 29
Figure 2.14 Data sharing behavior in PARSEC. ............................................ 29
Figure 2.15 Core-scaling with various techniques for four future technology generations. 30
Figure 2.16 Core-scaling with combinations of various techniques for four future technology generations. ............................................. 32
Figure 2.17 Core scaling with select techniques for a high and low $\alpha$. .......... 34
Figure 3.1 Miss rate profiles for ferret(a), x264(b) and fluidanimate(c) - x-axis is L2 cache size in bytes, y-axis is the cache’s data miss rate .............. 36
Figure 3.2 Impact of data sharing on miss rates across projection techniques, cache sizes and thread counts for vips with private caches(a), vips with shared cache(b), canneal with private caches(c), canneal with shared cache(d), bodytrack with private caches(e), and, bodytrack with shared cache(f). .... 40
Figure 3.3 Impact of data sharing on the average miss rates for PARSEC benchmarks with a shared cache configuration(a) and a private cache configuration(b). 42
Figure 3.4 Absolute miss rates (bars) and relative miss rates (line, 2nd y-axis) with and without sharing for freqmine(a) and canneal(b). ................. 43
Figure 3.5 Data sharing impact for application groups 1 and 2: measured impact and a canonical trend line ............................................. 44
Figure 3.6 Optimizing the core-to-cache ratio under No-Sharing (a), Sharing Style 1 (b), and Sharing Style 2 (c). Limited b/w = 30 GBps ............... 48
Figure 3.7 Optimizing core size under No-Sharing (a), Sharing Style 1 (b) and Sharing Style 2 (c) .......................................................... 50
Figure 4.1 PowerEN Chip Layout ............................................................. 55
Chapter 1

Introduction

Transistor density continues to scale according to Moore’s Law, even as the power density scaling has slowed down. This shift has fundamentally shaped the trends in microprocessor chip design over the last decade and will continue to do so into the next decade. With most of the efficiently accessible Instruction Level Parallelism already mined, processor designs over the last decade explored thread and task level parallelism in order to better utilize the exponentially growing supply of on-chip transistors. This led, first, to the advent of multi-threaded cores and, later, to multi-core chips. Towards the latter half of the decade, these same forces begat newer themes in chip integration - specialization and heterogeneity. Multi-core chips are moving away from simply being a collection of homogeneous cores to becoming an intricate ecosystem of heterogeneous computations units which can be selectively and dynamically enabled. On-chip heterogeneity may be evident at design time (for example, via the integration of hardware accelerators, special purpose programmable processors, general purpose programmable processors with different microarchitectures), or manifest itself at run time (for example, operation of cores at different voltage and frequency settings, powering off regions of the chip that are not being effectively utilized), or both. These themes open up a rich vista of microprocessor design choices and challenges, and may well be the overarching framework for research and innovation in the next decade.

First, memory bandwidth is growing far slower than the demand enabled by the continued increase in the number of cores per chip. Architects need to understand the severity of this problem and need guidance in selecting the best combination of microarchitectural techniques in order to overcome this bandwidth wall.

Second, large multi-cores enable highly multi-threaded workloads to run on a single chip, forcing chip designers to consider the impact of instruction and data sharing on chip design.

Third, on-chip hardware accelerators promise area and energy efficient step towards hardware specialization. However, for their benefit to be readily accessible to general purpose pro-
grams and cores, chip architecture and microarchitecture enhancements must be investigated.

Finally, heterogeneity among general purpose cores on a multi-core, while promising, opens up a very large space of design choices; the number of core and workload combinations grows factorially with the number of cores. Detailed cycle accurate simulation is impractical to tackle this exploration; and robust analytical modeling frameworks have not been developed to tackle this problem.

In this dissertation we address these challenges.

1.1 The Memory Bandwidth Wall and Multi-core Scaling

In recent years, a variety of concerns – power and thermal issues, instruction-level parallelism (ILP) limits, and design complexity, among others – have driven a shift in focus away from uniprocessor systems to multi-core designs. Multi-cores, also known as Chip MultiProcessors (CMPs), help alleviate many of these uniprocessor scaling barriers by providing the potential for throughput and performance gains with an increasing number of on-chip processor cores at each process technology generation. Ideally, designers would like to extract performance and throughput gains from CMPs proportional to the increase in the number of cores at each generation. However, one of the major obstacles to this goal is limited bandwidth to off-chip memory, since each additional core generates additional cache misses which must be serviced by the memory subsystem.

In general, doubling the number of cores and the amount of cache in a CMP to utilize the growing transistor counts results in a corresponding doubling of off-chip memory traffic. This implies that the rate at which memory requests must be serviced also needs to double to maintain a balanced design. If the provided off-chip memory bandwidth cannot sustain the rate at which memory requests are generated, then the extra queuing delay for memory requests will force the performance of the cores to decline until the rate of memory requests matches the available off-chip bandwidth. At that point, adding more cores to the chip no longer yields any additional throughput or performance.

Due to factors such as pin-limitations, power constraints, and packaging costs, memory bandwidth scaling typically lags significantly behind transistor density scaling described by Moore’s Law. The ITRS Roadmap [43] predicts that pin counts will increase by about 5% per year whereas the number of on-chip cores is expected to double every 18 months. The net result is that the rate at which memory traffic is generated by an increasing number of cores is growing faster than the rate at which it can be serviced. Similar to the memory wall and power wall problems faced by previous generations of systems, today’s designers also face a bandwidth wall; total system performance and throughput are increasingly limited by the amount available off-chip bandwidth.
This work tries to answer two fundamental questions: (1) to what extent does the bandwidth wall problem restrict future multicore scaling, and (2) to what extent are various bandwidth conservation techniques able to mitigate the bandwidth wall problem. To answer these questions, we propose a simple, yet powerful analytical model that projects how the generated memory traffic is affected by the workload characteristics, number of cores on a chip, the total cache size, and the allocation of die area for cores vs. caches. Then, we extend the model to incorporate the impact of various bandwidth conservation techniques on the memory traffic generated by the cores, and we evaluate the effectiveness of each technique in allowing multicore scaling in future systems.

Using the model, we arrive at several interesting findings:

- Without bandwidth conservation techniques, the bandwidth wall problem is multiplying at an astounding rate. For example, if the starting configuration has eight cores and 50% die area allocated to cores and 50% to caches, in four technology generations the allocation for caches must grow to 90% (vs. 10% for cores) so that the caches become large enough to reduce the total memory traffic per core such that the total traffic for all cores remains unchanged. Consequently, the number of cores would increase from 8 to only 24 (3× increase), versus 128 cores (16× increase) under “proportional” scaling.

- Bandwidth conservation techniques vary greatly in their ability to reduce the memory traffic requirement. For example, using DRAM caches allows the number of cores to increase to 47 in four technology generations, whereas leveraging smaller sized cores to gain more chip area for caches has a much more limited benefit to core scaling.

- Techniques which have a direct effect on the memory traffic itself offer significantly more benefit than techniques that reduce memory traffic indirectly by reducing cache miss rates. For example, in four technology generations, link compression can enable 38 cores while cache compression can enable only 30.

- When several bandwidth conservation techniques are combined, some of them allow super-proportional scaling. For example, using the combined techniques of 3D-stacked cache, DRAM caches, cache+link compression, and ideally-sized cache lines, in four technology generations, we can increase the number of cores on a chip to 183 (71% of the die area) while still maintaining the bandwidth envelope. Considering this, it is likely that the bandwidth wall problem can be delayed by several technology generations as long as multiple bandwidth conservation techniques are combined and applied effectively.
1.2 Multi-core design in the presence of Data Sharing

Technology scaling trends [13, 14, 42, 43] suggest that computer chips will continue to incorporate more and more cores onto a single chip over the next decade, even as the rate of this growth will be tempered by the Power Wall [31] and the Bandwidth Wall [82]. As detailed simulation of the full chip, with potentially tens or hundreds of cores, becomes prohibitively cumbersome [37], analytical modeling of chip performance is gaining traction [2, 31, 48, 74, 82, 94] as an alternative approach to design optimization. Analytical models predict chip performance by capturing key technology trends, design constraints, workload characteristics and microarchitectural variables into a mathematical framework. Tractable, accurate and insightful analytical models will become crucial in evaluating many-core chip designs of the future.

Software uses one of two fundamental approaches to capitalize on the growth in the number of cores on a chip. The first approach, multiprogramming, is to run several different applications simultaneously (perhaps in virtualized partitions) on the different hardware thread contexts [63]. The second approach, multithreading, is to spread a single application across multiple threads and run these threads on different hardware thread contexts (e.g., Apache-Hadoop [6]). Analytical performance models of a computer system must carefully consider, characterize and capture the relevant characteristics of the target workloads.

Multithreading has been traditionally used by parallel programs in scientific computing. However, multithreading is becoming more relevant as software outside the scientific computing realm attempts to take advantage of parallel hardware on a single chip [6, 11, 60, 62, 87]. The key difference between multithreaded applications and multiprogrammed workloads, in the context of chip performance, is data sharing. Multithreading permits hardware threads on a chip to share data and instructions, whereas multiprogrammed workloads run on independent virtual address spaces and do not share data with each other. Data sharing allows a thread’s cached data to be used by other threads without the latter suffering additional cache misses, thereby improving cache performance and reducing the off-chip bandwidth demand.

However, most analytical models of chip performance either ignore multithreaded applications, focusing instead on multiprogrammed workloads [94], or, oversimplify their behavior [2, 82]. Studies which do diligently study multi-threading [99] are restricted to Chip Multi-Processors (CMPS) with a small number of cores because they rely on detailed timing simulation or measurement on real hardware, both of which are infeasible techniques when designing many-core chips with potentially hundreds of cores. There is a need to address this deficiency in current analytical performance models, and answer the following questions. What is the right way to quantify the impact of data sharing on cache miss rates? What is the impact of data sharing in contemporary workloads? How can this impact be incorporated into an analytical...
performance model? Is the impact of data sharing significant enough to affect chip level design decisions? We investigate these questions in this work.

We start by studying what makes quantifying the impact of data sharing inherently difficult. We also analyze why, and by how much, prior approaches that attempt this quantification fall short.

We find that some prior approaches assume that an application’s miss profile (cache miss rate as a function of cache size) does not change much going from single-threaded to multi-threaded versions of the application [82]. Others rely on a single parameter – the average number of threads that share a cache block – to isolate the impact of data sharing [2, 74, 82]. We find that these approaches lead to an incorrect, and typically pessimistic, quantification of the significance of data sharing.

We then perform a sensitivity study involving 20 benchmarks from the PARSEC 2.1 [11] and NAS [8] suites, in order to understand the nature of data sharing in contemporary multi-threaded applications. For each benchmark, we investigate the impact of data-sharing on the last level cache (LLC) miss rates under varying thread counts (1, 2, 4, 8, 16 and 32), varying L2 cache configurations (private per-thread cache vs. shared cache) and varying L2 cache sizes (ranging from 256KB to 32MB). We use the Simics full-system simulator [61], modeled with 1 through 32 cores, running the appropriately threaded versions of the benchmarks. We find that both the miss rates and the impact of data sharing on miss rates, display a wide range of behaviors across the applications and simulated designs. In addition, we confirm that prior techniques consistently underestimate the positive impact of data sharing on miss rate reduction, sometimes by a small amount and sometimes significantly.

We then extend an existing analytical performance model [94], developed for multiprogrammed workloads, to be applicable to multithreaded workloads with data sharing. The purpose of the analytical model is to uncover qualitative trends, and not to make specific quantitative claims; and to that extent we believe the model to be simple, yet insightful.

We conduct a case study in which we apply the sharing-aware analytical performance model to optimize the design of a hypothetical, future, many-core chip (a 400mm² chip in the 22nm technology generation which can fit up to 256 simple cores). In particular, we focus on how the optimal answer to two key chip design questions changes with and without data sharing considered. The first design question we investigate is the optimal core-to-cache ratio – what is the optimal ratio in which to split the chip’s die area between core resources and cache resources? The second design question we investigate is the optimal core size – should we use smaller cores or larger cores?

We find that data sharing allows the optimal fraction of the chip area dedicated to cores to be increased from 33% to up to 49%. This, in turn, improves chip throughput by up to 58%. These absolute numbers are tied to the assumptions we make in our case study, which,
while realistic, must still be interpreted within that context. Qualitatively, however, these numbers indicate that data sharing can have a significant impact on future many-core chip design. We also find that data sharing tends to favor the use of fewer, larger, higher performing cores, as opposed to many, smaller, lower performing cores that workloads without data sharing favor. This counter-intuitive preference is even more pronounced when the pin-limited, off-chip memory bandwidth becomes a scarcer resource.

1.3 Architecture and Performance of the Hardware Accelerators in IBM’s PowerEN Processor

The continuing divergence between the scaling of transistor feature size and power density is encouraging hardware specialization and heterogeneity [35, 52] in addition to adding more cores on a single processor chip. Domain specific software can exploit heterogeneity further by embracing domain specific hardware accelerators [36]. Hardware acceleration has been identified as a key approach to continue to scaling performance in the future [15, 28]. However, the computational advantages of hardware acceleration must be balanced against the communication and data movement overheads they impose [73].

One area where such hardware acceleration is particularly applicable is in processors that operate at the edge of a datacenter, handling both general purpose computations as well as offloading of network function. As networking continues to mature, more and more function is being embedded in the network itself. Edge-of-network applications represent a growing class of workloads with unique characteristics that blur the boundaries between server and network applications. Such applications include cyber security, network intrusion detection, database acceleration, online transaction processing request checking and normalization, service oriented architecture acceleration, and TCP/FTP/HTTP offload. These functions are typically highly parallel in nature, work with streaming data, and stress cryptography, compression and pattern scanning algorithms, as well as network function. Hardware acceleration presents an opportunity to greatly enhance both the performance and energy efficiency of edge-of-network functions compared to handling them via traditional, general-purpose computing.

There has been substantial work in industry over the last decade to address this computation domain by companies such as Cavium [20], NetLogic [16], Tilera [90] and others. Such SOCs attack network applications by integrating tens of simple, efficient cores with a bevy of domain-specific accelerators. This highlights the fact that edge-of-network processing is becoming a key differentiator in data center performance, and is emerging as one of the first platforms for true co-design of general purpose and specialized hardware. In this work we study and describe the architecture of IBM’s PowerEN processor [17, 29, 38, 46, 75], focusing on how the architecture enables hardware acceleration.
We find that many unique architecture and implementation features come together in PowerEN to make it an exciting step towards heterogeneous special-purpose chip design. While accelerators provide orders of magnitude improvement in power and performance over software implementations of key kernels, general purpose cores are needed to both handle tasks that are not acceleratable, and to co-ordinate accelerator operations. Given the throughput oriented nature of the tasks at the edge of the network, the chip supports many general purpose cores, each with many hardware threads. In order to maintain low overhead communication between the cores and the accelerators the architecture provides a new instruction that can be accessed from user mode code (without needing a switch to system mode). Similarly, in order to maintain low overhead data movement between cores and accelerators, the accelerators share the virtual address space with each program context, have hardware address translation support and participate in the cache coherence protocol. The implementation of the chip supports various power reduction and power management techniques. We describe these architectural enhancements in PowerEN, and take an in-depth look at the architecture and performance capabilities of each hardware accelerator on chip.

1.4 ReSHAPE - Resource Sharing and Heterogeneity-aware Analytical Performance Estimation tool

Design of individual processor cores and cache hierarchies is typically guided by detailed timing and functional simulators. This process leads to near optimal design decisions at the core level. As the number of cores on a chip start to increase, the design space exploration (DSE) process is forced to become more abstract. Relying on detailed simulation for DSE in many core chip designs is already a cumbersome process, even if the cores are homogeneous. This is partly because the number of cores to be simulated is scaling; the bigger challenge is that with core scaling the workload mixes to be evaluated increase combinatorially. Adding heterogeneity into the mix makes this exercise nearly intractable. The simulator needs to be able to not only evaluate all the heterogeneous core combinations that the design constraints allow, but also evaluate different scheduling permutations of each of the workload mix combinations.

Additionally, heterogeneity comes in various forms. Several circuit level techniques are either already being applied or are actively being researched [45] as a first step towards enabling heterogeneity. These include support for per-core dynamic voltage and frequency scaling, employing asynchronous clocking domains between cores and large caches, and power gating or clock gating of cores, caches or sub-structures. Full-blown heterogeneity in the general purpose core microarchitectures [30] as well as supporting function specific hardware, such as Graphics Processing Units and Accelerators [46], is another approach to power efficient computation.

This is leading to heuristics-driven chip design followed by heuristics-driven application
scheduling, leading to overall sub-optimal performance and power efficiency. While DSE of heterogeneous many-cores is a difficult challenge, we believe that it is a challenge that will only increase in importance. Further, it is a challenge where a small investment at design or schedule time can provide significant payback in terms of performance and power efficiency.

As a step towards addressing this challenge, in this work we develop the Resource Sharing and Heterogeneity aware Analytical Performance Estimation tool, ReSHAPE. This tool can then be used to project coarse but accurate performance estimates for a given design and workload schedule. ReSHAPE is a C++ based model that combines aspects of detailed simulation with aspects of analytical modeling. ReSHAPE relies on inputs from detailed simulations that capture key characteristics of each application-microarchitecture pair of interest. ReSHAPE then estimates the performance of a given multi-core chip design and workload schedule by iteratively applying the analytical model until the solution converges. Due to its iterative nature, ReSHAPE is able to avoid the need for closed form solutions, a need which often forces analytical models to oversimplify application behavior (such as using the Power Law to model an application’s cache sensitivity). Compared to detailed simulations ReSHAPE is 3 to 4 orders of magnitude faster, allowing very fast DSE. It is able to model heterogeneity in cores, heterogeneity in cache configurations, as well as the impact of shared caches and shared memory bandwidth.

We validate ReSHAPE’s accuracy by simulating a wide range of multiprogrammed workload mixes running across several multi-core configurations, under both ReSHAPE and a detailed simulator. We then compare key performance metrics projected by ReSHAPE against those measured by detailed simulation. We found ReSHAPE to be reasonably accurate - sufficient to identify trends and narrow down design choices before more detailed analysis using traditional approaches such as simulations. For example, in a chip configuration with 4 cores and a shared 2MB L2 cache, we found that the average error in IPC across 12 workload mixes composed out of Spec2006 applications was 6% (maximum error of 14%). The average error in the L2 cache miss rate was 12.8% (maximum error of 30%).

Encouraged by ReSHAPE’s accuracy we apply it to a few interesting chip level design optimization questions.

In the first use case we use ReSHAPE to study various aspects of heterogeneity. We study whether heterogeneity always helps improve performance compared to a homogeneous design. We find that, within the constraints of our assumptions, in a 4-core design, a homogeneous design often outperforms a heterogeneous design, regardless of whether the heterogeneity is in the cores, the caches or both. When we scale to a chip with 9 cores, the benefit from heterogeneity starts to become more obvious; however, it still requires a good application-to-core mapper to deliver any benefits. We find that heterogeneity in cores is more important than heterogeneity in cache sizes, with heterogeneity in both cores and caches being the best option.
We also find that integrating 2 core types and 2 cache types provides sufficient advantage from heterogeneity in this case; going to 3 core types and 3 cache types does not provide any further benefit.

In a second use case we apply ReSHAPE to a post-design optimization problem. We assume a 4-core homogeneous multi-core chip with a shared L2 cache, where each core’s frequency (and voltage) can be individually configured within a certain range. We identify the optimal DVFS settings for each workload mix that we study. We find that the optimal setting is different depending on the workload mix and the metric being optimized (Weighted Speedup, Energy Delay product and Performance per Watt).

While ReSHAPE builds on existing chip-level performance models, we believe it advances the field in three important ways. First, by not attempting to be closed form solver, ReSHAPE is able to directly utilize statistics from detailed simulation infrastructure without needing to abstract the statistics into parameterized distributions. Second, ReSHAPE can simulate a much richer variety of chip configurations (heterogeneous core microarchitectures, varied core frequencies, memory bandwidth constraints, configurable number of cache levels, sizes and sharing configurations) than existing many-core analytical models can. Third, by converging towards the performance estimate iteratively, ReSHAPE is able to close the feedback loop between IPC and memory bandwidth; a larger IPC leads to greater demand on the memory bandwidth, which, in turn, slows down the core, thereby decreasing IPC. We believe this work is a promising step in the direction of addressing the design space challenges and opportunities that a heterogeneous many-core chips bring.

1.5 Organization of the Dissertation

The rest of the report is organized as follows. Chapter 2 quantifies the impact of the bandwidth wall on future CMP scaling, and studies the ability of various schemes proposed in the literature to mitigate this wall. Chapter 3 studies the impact that data-sharing in typical multi-threaded workloads has on cache miss rates and, incorporates these findings into a system-level analytical performance model in order to identify whether data-sharing can impact chip design. Chapter 4 studies the architecture and performance of IBM’s PowerEN processor, one of the first heterogeneous multi-core to implement a coherent shared memory architecture that eases communication with on-chip hardware accelerators. Chapter 5 develops a new modeling infrastructure, ReSHAPE, that tackles the problem of design space explosion brought about by continued core scaling and heterogeneity in multi-core chips. Finally, Chapter 6 reflects on the studies that gave shape to this dissertation and analyzes the import of this work.
Chapter 2

The Memory Bandwidth Wall and CMP Scaling

This chapter is organized as follows: Section 2.1 discusses the major assumptions used by our model, Section 2.2 describes how we model CMP memory traffic requirements, Section 2.3 describes how the model is extended to predict the memory traffic requirements of different CMP configurations, Section 2.4 shows evaluation results and insights into the effects of a variety of bandwidth conservation techniques, and Section 2.5 overviews related work.

2.1 Assumptions, Scope, and Limitations of Study

To make our analytical model tractable, we make several simplifying assumptions. First, we restrict our study to CMPs with uniform cores. A heterogeneous CMP has the potential of being more area efficient overall, and this allows caches to be larger and generates less memory traffic from cache misses and write backs. However, the design space for heterogeneous CMPs is too large for us to include in our model.

We also assume single-threaded cores. The consequence of this assumption is that our study tends to underestimate the severity of the bandwidth wall problem compared to a system with multithreaded cores. This is because multiple threads running on a multi-threaded core tend to keep the core less idle, and hence it is likely to generate more memory traffic per unit time than a simpler core.

In addition, we assume that we always have threads or applications that can run on all cores in the CMP, regardless of the number of cores on a chip. We also assume that characteristics of workload do not change across technology generations. While it is possible that future workloads will be more bandwidth efficient (i.e. they generate fewer cache misses), past trends point to the contrary, as the working set of the average workload has been increasing due to the...
increasing complexity of software systems. This means that the future severity of the bandwidth wall problem may be worse than what we project in our study.

In addition to these assumptions, there are configurations that we use as the base, and apply throughout the study, unless mentioned otherwise. For example, we assume a configuration in which each core has a private L2 cache and threads that run on different cores do not share data with each other, but we relax this assumption when we discuss the impact of data sharing across threads. Another example is that we assume the size of cores, in terms of number of transistors, remains unchanged across technology generations, but we relax this assumption when we discuss the impact of using smaller cores in future generations.

For the scope of this study, we do not evaluate the power implications of various CMP configurations, and only qualitatively discuss the implementation complexity of the bandwidth conservation techniques we study.

Finally, the projection obtained by our analytical model is the memory traffic generated for different configurations for a constant amount of computation work, rather than for a constant unit of time (i.e., bandwidth). Both metrics are equal if the rate at which memory traffic is generated does not change. However, many factors affect the latter metric (bandwidth), including the features of the core, cache configuration, as well as queuing delay at the off-chip interconnect. Predicting the actual bandwidth demand complicates our model with no additional insights in the bandwidth saturated scenario that we assume. Hence, we focus on predicting the total memory traffic generated instead.

2.2 Modeling CMP Memory Traffic

At the basic level, our analytical model must relate memory traffic to the workload characteristics, the number of cores on a chip, the total cache size, and the allocation of die area for cores vs. caches. To construct such a model, we can start from a simpler model that relates the cache miss rate with the cache size in a single core system.

2.2.1 The Power Law of Cache Misses

In a single core system, it has long been observed that the way the cache size affects the miss rate follows the power law [33]. Mathematically, the power law states that if \( m_0 \) is the miss rate of a workload for a baseline cache size \( C_0 \), the miss rate \( m \) for a new cache size \( C \) can be expressed as:

\[
m = m_0 \cdot \left( \frac{C}{C_0} \right)^{-\alpha},
\]

where \( \alpha \) is a measure of how sensitive the workload is to changes in cache size. Hartstein
et al. in [33] validated the power law on a range of real-world benchmarks, and found that $\alpha$ ranges from 0.3 to 0.7, with an average of 0.5 (hence the $\sqrt{2}$ rule).

Figure 2.1 plots the cache miss rate of each application we evaluate normalized to the smallest cache size as a function of cache size for a single level of cache. Note that both axes are shown in logarithmic scale; the miss rate curve will follow a straight line if it obeys the power law. The figure shows that these applications tend to conform to the power law of cache miss rate quite closely. Also, note the four bold lines in the figure correspond to the power law fit for all commercial applications, for all SPEC 2006 benchmarks, for the commercial application with the smallest $\alpha$ (OLTP-2), and for the commercial application with the largest $\alpha$ (OLTP-4). The larger the $\alpha$ value, the steeper the negative slope of the line. The curve-fitted average $\alpha$ for the commercial workloads is 0.48, close to 0.5 in Hartstein et al.’s study [33], while the minimum and maximum $\alpha$ for the individual commercial applications are 0.36 and 0.62, respectively. The smallest $\alpha$ (SPEC 2006) has a value of 0.25. Note that individual SPEC2006 applications exhibit more discrete working set sizes (i.e. once the cache is large enough for the working set, the miss rate declines to a constant value), and hence they fit less well with the power law. However, together their average fits the power law well.

![Figure 2.1: Normalized cache miss rate as a function of cache size.](image)

### 2.2.2 Memory Traffic Model for CMP Systems

For our CMP memory traffic model, we extend the power law model to include the effect of additional memory traffic due to writebacks, and multiple cores on a chip. We find that for a
given application, the number of writebacks tends to be an application-specific constant fraction of its number of cache misses, across different cache sizes. If such a fraction is denoted as $r_{wb}$, then the total memory traffic becomes $M = m \times (1 + r_{wb})$, where $m$ is the miss rate. Substituting it into Equation 2.2, and symmetrically using $M_0$ to represent the base memory traffic with the $C_0$ sized cache, the $(1 + r_{wb})$ term appears in both the numerator and denominator, which cancel out each other, leaving us with:

$$M = M_0 \cdot \left( \frac{C}{C_0} \right)^{-\alpha}$$ (2.2)

which states that the power law holds for the memory traffic as well.

To take into account multiple cores on a chip, we first assume that the chip’s die area is divided into some number of Core Equivalent Areas (CEAs). Note that while the general idea behind a CEA is common across this thesis, the specific amount of cache memory a single CEA represents may vary across the chapters; these differences will be clarified before the first use of the term CEA in each chapter. One CEA is equal to the area occupied by one processor core and its L1 caches. Additionally, we assume that on-chip components other than cores and caches occupy a constant fraction of the die area regardless of the process technology generation.

Table 2.1: CMP system variables used in our model

<table>
<thead>
<tr>
<th>CEA</th>
<th>Core Equivalent Area (die area for 1 core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td># of CEAs for cores(equivalent to # cores)</td>
</tr>
<tr>
<td>C</td>
<td># of CEAs for on-chip cache</td>
</tr>
<tr>
<td>N</td>
<td>= P+C, total chip die area in terms of CEAs</td>
</tr>
<tr>
<td>S</td>
<td>= C/P, amount of on-chip cache per core</td>
</tr>
</tbody>
</table>

Table 2.1 shows the major parameters of our model which capture the fact that the die area in a CMP is allocated between cores and caches. Assuming that threads do not share data (Section 2.1), all cores in a CMP generate cache miss and write back traffic independently. Thus, the total memory traffic of the entire multicore chip is equal to the number of cores multiplied by the memory traffic generated by each core:

$$M = P \cdot M_0 \cdot \left( \frac{S}{S_0} \right)^{-\alpha}$$ (2.3)

where $S = C/P$. In order to compare the memory traffic from different CMP configurations, suppose that we have a baseline configuration with $P_1$ cores and $S_1$ amount of cache per core, and a new configuration with $P_2$ cores and $S_2$ cache per core. Dividing the expressions from Equation 2.3 for both configurations, we obtain:
\[
\frac{M_2}{M_1} = \frac{P_2 \cdot M_0 \cdot \left(\frac{S_2}{S_0}\right)^{-\alpha}}{P_1 \cdot M_0 \cdot \left(\frac{S_1}{S_0}\right)^{-\alpha}} = \frac{P_2}{P_1} \left(\frac{S_2}{S_1}\right)^{-\alpha} \quad (2.4)
\]

\[
M_2 = \left(\frac{P_2}{P_1}\right) \cdot \left(\frac{S_2}{S_1}\right)^{-\alpha} \cdot M_1 \quad (2.5)
\]

That is, the memory traffic of two different configurations depend on two terms: \(\frac{P_2}{P_1}\), which accounts for the difference in the number of cores, and \(\left(\frac{S_2}{S_1}\right)^{-\alpha}\), which depends on the change in the cache space that is allocated to each core. For example, suppose that a workload has \(\alpha = 0.5\), and a baseline CMP configuration with 8 cores and 1 CEA of cache per core, for a total of 16 CEAs. If we reallocate 4 CEAs from caches to make room for four extra cores, then \(P_2 = 8 + 4 = 12\) and \(S_2 = \frac{8+4}{8+4} = \frac{1}{3}\). Using Equation 2.5, we find that the new configuration yields memory traffic of 2.6\(\times\) more than the baseline configuration, of which 1.5\(\times\) is due to the increase in the number of cores, and 1.73\(\times\) is due to the increase in the memory traffic per core because of the reduced cache size per core.

2.3 Bandwidth Wall Impact on CMP Scaling

The previous section described an analytical model for memory traffic as a function of die allocation to cores versus cache in the same processor generation. In this section, we will look at how the memory traffic requirement changes in future technology generations.

2.3.1 Bandwidth Wall Impact on CMP Scaling

To investigate the impact of the bandwidth wall on CMP scaling, we start with a baseline CMP configuration in the current process technology generation, with parameters chosen to be similar to that of Sun Niagara2 [65]. In this baseline, we assume a balanced configuration with 8 cores and 8 CEAs allocated for on-chip L2 cache (roughly corresponding to 4MB in capacity). The parameters corresponding to this baseline configuration are \(N_1 = 16\), \(P_1 = 8\), \(C_1 = 8\), and \(S_1 = 1\). We will also assume that \(\alpha = 0.5\), representing the characteristic of an average commercial workload (Figure 2.1). We also assume that in future generations, we wish to keep the memory traffic the same as in the baseline CMP configuration.

Now, let us suppose that twice as many transistors are available in the following technology generation, and so we now have 32 CEAs of available area (assuming each core uses the same number of transistors as before). We can allocate \(P_2\) cores, leaving \(C_2 = 32 - P_2\) CEAs for cache, resulting in a cache-to-core ratio of \(S_2 = \frac{32 - P_2}{P_2}\). Substituting \(P_2\) values ranging from 1 to 28 into Equation 2.5, we can determine how memory traffic demand changes as a function of the number of cores on a chip, normalized to the baseline system (Figure 2.2).
The figure shows that as we increase the number of processor cores (and reduce the area allocated for caches), the memory traffic relative to the baseline configuration grows super-linearly. If we wish to double the number of cores from the baseline to 16, then the memory traffic increases by a factor of 2. Unfortunately, off-chip pin bandwidth will not likely keep up as pin count only grows a projected 5% per year [43]. If we assume that memory traffic should remain constant in the next technology generation, then the new CMP configuration can only support 11 cores (the intersection of the two curves in Figure 2.2), which is equivalent to an increase of only 37.5%. Even when we assume that the bandwidth envelope can grow by an optimistic 50% in the next generation, the number of cores can only increase by 62.5% to 13. Of course, it is always possible to build more cores on a chip if one is willing to sacrifice a balanced design. Unfortunately, adding more cores beyond the bandwidth envelope will force total chip performance to decline until the rate of memory requests matches the available off-chip bandwidth. Even worse is the fact that the area taken up by excess cores could have been allocated for more productive use such as for extra cache capacity.

2.3.2 CMP Scaling and Die Area Allocation

One useful angle for chip designers to look into the bandwidth wall problem is to determine the fraction of chip area that should be allocated to cores versus cache to achieve a balanced CMP design in a future technology generation. To arrive at this answer, suppose that memory bandwidth grows by a factor of $B$ in each technology generation, allowing the aggregate memory
traffic of a next-generation CMP to grow by a factor of $B$ relative to a current design, i.e., $M_2 = B \cdot M_1$. Incorporating this into Equation 2.5, we can express:

$$\left( \frac{P_2}{P_1} \right) \cdot \left( \frac{S_2}{S_1} \right)^{-\alpha} = \frac{M_2}{M_1} = B \quad (2.6)$$

Next, we rework Equation 2.6 by expressing $S_2$ in terms of $N_2$, which is known given our fixed chip area constraints, and $P_2$, which is what we want to find, i.e., $S_2 = \frac{N_2 - P_2}{P_2}$. Substituting this expression for $S_2$ in Equation 2.6, we obtain the following expression:

$$\left( \frac{P_2}{P_1} \right) \cdot \left( \frac{N_2 - P_2}{P_2} \right)^{-\alpha} = B \quad (2.7)$$

Solving for $P_2$ numerically, we obtain the results shown in Figure 2.3.

![Figure 2.3: Die area allocation for cores and the number of supportable cores assuming constant memory traffic requirements.](image)

Figure 2.3 reveals that if we keep the memory traffic constant across process generations, the fraction of the die area allocatable to cores declines rapidly. At $16 \times$ scaling (four process generations from now), only 10% of the die area can be allocated for cores, corresponding to 24 cores (versus $16 \times 8 = 128$ cores under proportional scaling). The die area allocation for cores decreases even more at future process generations.


2.4 Analyzing Memory Traffic Reduction Techniques

In Sections 2.2 and 2.3, our model has shown that the bandwidth wall, left unaddressed, will quickly dampen CMP scaling and force die area allocation heavily tilted toward caches. Considering the severity of the problem, we believe that future large-scale CMPs must take the bandwidth wall problem as a primary concern. The goal of this section is to evaluate the effectiveness of various bandwidth conservation techniques at addressing the bandwidth wall problem.

Recall how memory traffic is impacted by the number of cores and the amount of cache per core (Equation 2.5). There are two main ways to maintain a nearly-constant memory traffic requirement as we increase the number of on-chip cores proportionally to the growth in transistor density. The first way is to \textit{indirectly} reduce the memory traffic requirements by increasing the effective on-chip cache capacity, which reduces cache misses. The second way is to \textit{directly} reduce the memory traffic requirements by either reducing the number of bytes that are brought on-chip, or by increasing the raw amount of available off-chip bandwidth. Based on this insight, we divide the discussion of various bandwidth conservation techniques into three categories: those in the \textit{indirect} category, \textit{direct} category, and \textit{dual} category for techniques that produce both indirect and direct effects.

Note that the scope of our study is to evaluate existing or promising techniques that have been proposed in literature, and compare them in their effectiveness in reducing memory traffic requirements. Fully analyzing whether or when the different techniques become feasible for commercial implementation is beyond the scope of this study.

To compare the benefits of various techniques to CMP scaling, we will use the same baseline CMP as in Section 2.3.1, (8 cores and 8-CEA caches) and evaluate how many cores can be supported in the next generation when the number of total CEAs doubles to 32. Under proportional scaling, we would like to be able to support 16 cores, but the bandwidth wall limits it to only 11 (Figure 2.2).

2.4.1 Indirect Techniques

Indirect techniques reduce memory traffic requirements by increasing the effective cache capacity per core. In our model, the impact is captured by a multiplicative factor $F$, which we refer to as the \textit{effectiveness factor}:

$$M_2 = \left( \frac{P_2}{P_1} \right) \cdot \left( \frac{F \cdot S_2}{S_1} \right)^{-\alpha} \cdot M_1$$

(2.8)

For some indirect techniques, the memory traffic equation may not conform to the simple form in Equation 2.8. In these cases, we develop the appropriate traffic model, as necessary.
Note that the impact of increased effective cache capacity per core (i.e. $F$) on memory traffic is damped by the $-\alpha$ power. The smaller $\alpha$ is for a workload, the larger the damping becomes. For example, if $\alpha = 0.9$, to reduce memory traffic by half, the cache size per core needs to be increased by a factor of $2.16 \times$. However, if $\alpha = 0.5$, to reduce memory traffic by half, the cache size per core needs to be increased by a factor of $4 \times$.

**Cache Compression.** Cache compression (e.g. [2]) exploits repeated patterns in data values to store them using fewer bits. A hardware compression engine is used to compress data before it is placed in an on-chip cache and to decompress it when the processor needs it. Prior work on cache compression has shown that a compression scheme with reasonable area and latency overheads can achieve compression ratios (i.e. effectiveness factors) of 1.4x to 2.1x for a set of commercial workloads, 1.7x to 2.4x for the SPEC2k integer benchmarks, and 1.0x to 1.3x for the SPEC2k floating point benchmarks [2, 3, 4].

Figure 2.4 shows the implications of this observation on the number of cores that can be supported in a next-generation CMP with 32 total CEAs, if the memory traffic requirement is kept constant relative to a baseline system with 8 cores and 8-CEA caches. Recall from Section 2.3 that without cache compression, our new CMP can support only 11 cores under a constant memory traffic requirement. The figure shows the number of cores possible using a cache compression scheme with various effectiveness factors (i.e. compression ratio) within the ranges observed in [2, 3, 4, 89], as well as an upper-limit case.

![Figure 2.4: Increase in number of on-chip cores enabled by cache compression (32 CEAs).](image)

With cache compression support with a compression ratio of 1.3x, 1.7x, 2.0x, 2.5x, and 3.0x,
the number of supportable cores grows to 11, 12, 13, 14, and 14 respectively. As CMP systems scale, this savings potentially can represent a larger raw number of additional cores that can be placed on chip. However, unless the compression ratios reach the upper end of achievable range, the ability of cache compression in helping CMP scaling is relatively modest. Note that our analysis does not factor the power, complexity, and die area overheads needed to implement the compression engine.

**DRAM Cache.** Another way to increase the effective cache capacity is to implement caches using denser memory such as DRAM [12, 97]. The increase in density from replacing SRAM caches with DRAM caches is estimated to be between $8\times$ [12] and $16\times$ [97]. Figure 2.5 shows the impact of changing the on-chip L2 cache from an SRAM to a DRAM technology on the number of supportable cores in the next-generation CMP under a constant memory traffic requirement.

![Figure 2.5: Increase in number of on-chip cores enabled by DRAM caches.](image)

The bar labeled “SRAM L2” represents the baseline case in which the L2 caches are implemented in SRAM. The remaining bars show the number of cores that can be supported without increasing the memory traffic demand if DRAM technology with the above mentioned density assumptions is used to implement the L2 cache. From this figure, we can see that the desired proportional scaling of 16 cores is possible even assuming a conservative density increase of $4\times$ in the DRAM implementation. With $8\times$ and $16\times$ density improvement, the situation improves further and allows super-proportional scaling to 18 and 21 cores respectively. While there are other implementation aspects to consider, such as the refresh capacity needed for DRAM or possible access latency increases, DRAM caches provide a very effective means of filtering the
extra memory traffic generated by a larger number of cores, and they alleviate the bandwidth wall problem by slightly more than one technology generation.

**3D-Stacked Cache.** Another potential solution to achieve a large on-chip cache capacity is 3D chip stacking, where an additional die consisting of cache storage is stacked vertically on top of the traditional CMP die. We restrict our analysis to the case where there are only 2 dies stacked and the processors only occupy one die. The cache occupies the entire extra die, and, potentially, a share of the processor die. We also assume that the cache that shares the die with the processors is a traditional SRAM cache. For the extra (cache-only) die, we investigate the impact of an SRAM cache as well as DRAM cache with various density improvements over SRAM. Our assumptions for this analysis are based on prior work on 3D caches and die-stacking [12, 80].

To incorporate 3D stacking into our model, we note that the extra cache-only die increases the area for cache allocation by \( N \) CEAs. The new total CEAs for cache are \( N + (N - P) \). If the cache-only-layer of the stacked CMP is implemented using \( D \times \)-denser DRAM cells, then the effective cache CEAs on the cache-only layer is \( D \cdot N \), and the total cache CEAs for the CMP are \( D \cdot N + (N - P) \). Substituting the \( S_2 \) term in the memory traffic model of Equation 2.5 by \( \frac{D \cdot N + (N - P)}{P_2} \), we get:

\[
M_2 = \left( \frac{P_2}{P_1} \right) \cdot \left( \frac{\frac{D \cdot N + (N - P)}{P_2}}{S_1} \right)^{-\alpha} \cdot M_1
\]  

Figure 2.6 shows the impact of adding a 3D-stacked die for cache storage on the number of supportable cores in the next generation CMP under a constant memory traffic requirement.
The bar labeled “No 3D Cache” corresponds to the next-generation CMP without an additional die for caches, which can support only 11 cores. The remaining bars show how the caches in the additional die are implemented: in SRAM or DRAM with \(8\times\) or \(16\times\) density improvement over SRAM. From this figure, we can see that adding a die layer of SRAM caches allows 14 cores to be implemented, and 25 and 32 cores when DRAM caches are used with \(8\times\) or \(16\times\), respectively. The result shows that the density improvement of DRAM caches, coupled with an entire die allocated for them, result in a highly effective way to mitigate the bandwidth wall, allowing super-proportional scaling.

**Unused Data Filtering.** Previous research [25, 81] has shown that a substantial amount of cached data (roughly 40%) is unused by the processor because of the mispredicted spatial locality inherent in typical cache line sizes. Unused data filtering consists of techniques that improve cache efficiency by retaining only predicted-useful words in a cache line and discarding predicted-unused words. Words may be defined as predicted-useful in many ways. One simple heuristic is that a word that has been used at least once stands a chance of being used again. While unused data filtering might not be designed primarily as a memory traffic reduction technique, its impact on memory traffic is significant enough to merit its inclusion in our model. There are also related studies [25, 54, 79] which reduce memory traffic directly by selectively reading in predicted-useful words of a cache line, and they will be discussed in Section 2.4.2 under the broad category of Sectored Caches.

![Graph showing the increase in number of on-chip cores enabled by filtering unused data from the cache.](image)

**Figure 2.7:** Increase in number of on-chip cores enabled by filtering unused data from the cache.

Using Equation 2.8 and effectiveness factors for unused-data-filtering techniques from pre-
vious research [81] we can evaluate the sensitivity of CMP scaling to this class of techniques. Figure 2.7 shows that under optimistic assumptions in which 80% of cached data goes unused, corresponding to a $5 \times$ effective increase in cache capacity, a proportional scaling to 16 cores can be achieved. Under our more realistic assumptions in which 40% of cached data goes unused, the techniques provides a much more modest benefit of one additional core in the next technology generation.

**Smaller Cores.** One approach for allowing aggressive core scaling is to use simpler, smaller cores. One reason behind this approach is that more complex processors waste bandwidth due to speculative techniques geared towards increasing the throughput of each individual core. With less speculation, smaller cores can be more area, power, and bandwidth efficient. This efficiency allows a greater number of cores to be placed on-chip and improves the overall throughput of the chip, even as the throughput of each core is lower compared to a more complex core. However, there is a limitation to this approach. There is a certain design point which is most efficient in terms of performance per unit area and bandwidth per unit area. Any further reduction in the size of the core will yield no overall throughput benefit.

In the discussion of all the other memory traffic reduction techniques, we have assumed an area- and bandwidth-efficient base core design, such that simplifying the core further is not a good design choice. However, for sake of completeness and to draw further insights, in this section we assume that going to a smaller core leads to a more efficient design. In addition, we assume that the memory traffic requirements (not bandwidth) of the smaller core remain the same as the larger core. However, by going to smaller cores, the die area occupied by the same number of cores declines. This frees up die area that can be reallocated to increase cache capacity and reduce memory traffic. As defined in Table 2.1, the amount of cache per core, $S$, is equal to $\frac{N-P}{P}$. Let the smaller core be a fraction, $f_{sm}$, of the larger base core. Then the new effective amount of cache per core, $S'$, can be expressed as:

$$S' = \frac{N - f_{sm} \cdot P}{P}$$

(2.10)

Replacing $S_2$ in Equation 2.5 by the expansion for $S'_2$ as suggested by Equation 2.10 we get:

$$M_2 = \left( \frac{P_2}{P_1} \right) \cdot \left( \frac{N_2 - f_{sm} \cdot P_2}{P_2} \frac{S_2}{S_1} \right)^{-\alpha} \cdot M_1$$

(2.11)

Using Equation 2.11 and solving for $P_2$ for different values of $f_{sm}$, we can evaluate the sensitivity of CMP scaling to the size of the core. Figure 2.8 shows the number of smaller on-chip cores the next-generation chip can support. We use chip size estimates from prior work [50, 53] which shows that simpler smaller cores can be up to $80 \times$ smaller.

Note that even with significantly smaller cores, the system does not scale very well even across one generation. The reason is that even when the core is infinitesimally small and the
cache occupies the entire chip, the amount of cache per core only increases by $2\times$, whereas for proportional core scaling the cache needs to grow by $4\times$.

Therefore, the main reason going to a simpler core design allows a more aggressive core scaling is because simpler cores are also slower in generating memory traffic, hence they naturally fit within a lower bandwidth envelope. For example, if a simpler core is twice as slow and twice as small as a more complex one, we can fit twice as many simple cores while keeping the same bandwidth envelope. However, in such a case, the scaling is obtained directly at the cost of lower per-core performance. In addition, in practice, there is a limit to this approach, since with increasingly smaller cores, the interconnection between cores (routers, links, buses, etc.) becomes increasingly larger and more complex.

2.4.2 Direct Techniques

We have discussed that by increasing the effective cache capacity per core, the memory traffic demand from each core can be reduced. However, the benefit of such techniques on memory traffic reduction is damped by the power of $-\alpha$ in Equation 2.5. A more promising approach is to increase the actual or effective bandwidth term, $\frac{M_2}{M_1}$, directly. To directly increase the available bandwidth, the industry has relied on (1) increasing the frequency of the off-chip memory interface, or (2) increasing the number of channels to off-chip memory. For example, Sun’s Niagara2 went with Fully Buffered memory DIMMs compared to the DDR2 memory in Niagara1, increasing the peak memory bandwidth from about 25GBps to 42GBps [65]. IBM’s Power6 chip doubled the number of memory controllers and memory channels on chip, and went to a faster clocked DDR2 memory compared to the Power5 chip [56] (800MHz vs.
533MHz). Unfortunately, increasing the actual bandwidth is eventually limited by the growth in pin counts and in the power and cooling requirements from a higher DRAM frequency.

**Link Compression.** One technique to increase the effective memory bandwidth is link compression, which reduces the amount of data communicated from/to off-chip memory by receiving/sending it in a compressed form. Previous research has shown that with relatively simple compression schemes, such as those which exploit value locality, memory bandwidth demand can be reduced by about 50% for commercial workloads, and up to 70% for integer and media workloads [89]. In other words, link compression, can result in 2x to 3x increase in effective bandwidth. Though link compression may be applied only once, its potential impact and its orthogonality to other techniques make this promising. Figure 2.9 shows the number of supportable cores in a next-generation CMP assuming various compression ratios. The figure shows that proportional scaling is achievable, while a super-proportional scaling is a possibility.

![Figure 2.9: Increase in number of on-chip cores enabled by link compression.](image)

**Sectored Caches.** As discussed in the section on Unused Data Filtering, studies have found that many words in a cache line go unused. Here, we consider techniques that exploit this finding by fetching on chip only words that are likely to be referenced [25, 54, 79]. We evaluate their potential benefit assuming that a cache line is broken into sectors, and when a cache line is requested, only sectors that will be referenced by the processor are fetched and fill the cache line. Because unfilled sectors still occupy cache space, a sectored cache only reduces memory traffic but does not increase the effective cache capacity.

Figure 2.10 shows the number of supportable cores in a next generation CMP, assuming various fractions of data that are unused by the processor, as in Unused Data Filtering in
Section 2.4.1. The figure shows that Sectored Caches have more potential to improve core scaling compared to Unused Data Filtering, especially when the amount of unused data is high.

2.4.3 Dual Techniques

The previous sections explored two classes of techniques to overcome the bandwidth wall: those that directly increase the effective bandwidth at the chip interface, and those that indirectly reduce off-chip traffic through increasing the effective cache capacity. This section considers mechanisms that simultaneously achieve both of these benefits.

Small Cache Lines. Historically, larger cache lines have been used to improve application performance by banking on the existence of spatial locality for all words in a cache line. However, if not all words in a line are accessed, unused words waste bandwidth in two ways: they consume off-chip bandwidth during their transfer from the off-chip memory, and they reduce the cache capacity available to useful words while they are cached on chip. Therefore, while using smaller line sizes can increase the number of cache misses, it can reduce memory traffic both directly and indirectly, since only useful words are brought in and stored in the cache.

Prior work [25, 81] has studied the amount of useless traffic for various line sizes and concluded that, on average, 40% of the 8-byte words in a 64-byte cache line are never accessed. This represents a 40% decrease in effective cache capacity and, potentially, a corresponding 40% decrease in memory bandwidth effectiveness. Let \( f_w \) denote the average fraction of unused cache space. The new memory traffic demand from using smaller cache lines is described by

Figure 2.10: Increase in number of on-chip cores enabled by a sectored cache.
Equation 2.12.

\[ M_2 = \left( \frac{P_2}{P_1} \right) \cdot \left( \frac{S_2 \cdot \left( \frac{1}{1-f_w} \right)}{S_1} \right)^{-\alpha} \cdot \frac{M_1}{\left( \frac{1}{1-f_w} \right)} \]  

\( \frac{1}{1-f_w} \) represents the factor by which the cache space increases and the memory traffic decreases. Figure 2.11 shows the number of supportable cores in a next-generation CMP for various fractions of unused data, with an assumption that unreferenced words consume neither bus bandwidth nor cache capacity, corresponding to the case in which word-sized cache lines are used. We point out the realistic case, which shows that a 40% reduction in memory traffic enables proportional scaling (16 cores in a 32-CEA). Note also that our base cache line size of 64 bytes is on the small end. Some current systems use larger L2 cache line size, in which case we would expect the effectiveness of this technique to be greater.

**Figure 2.11**: Increase in number of on-chip cores enabled by smaller cache lines.

**Cache + Link Compression.** If link and cache compression are applied together, i.e. compressed data transmitted over the off-chip link is also stored compressed in the L2 cache, memory traffic can be reduced directly and indirectly. In our model, such a combined compression is expressed by replacing the \( \frac{1}{1-f_w} \) terms in Equation 2.12 with the achievable compression ratio. Figure 2.12 shows the number of supportable cores in a next generation CMP with various compression ratios. The dual benefit of direct and indirect reduction of memory traffic improves core scaling substantially. For example, even a moderate compression ratio of 2.0 is sufficient to allow a super-proportional scaling to 18 cores, while higher compression ratios
allow even higher scaling.

**Data Sharing.** So far we have assumed that threads that run in different cores do not share data. We recognize that parallel scientific workloads as well as multi-threaded commercial workloads may have substantial data sharing between different threads [9, 19, 91]. In this section, we relax the assumption of no data sharing, and evaluate how data sharing in multi-threaded workloads affect CMP core scaling. However, as with independent threads, we assume *problem scaling*, i.e. each additional thread brings its own working set, except for the data shared by multiple threads.

Data sharing affects our model in two ways. First, a data line that is shared by multiple threads is fetched from off-chip memory into the on-chip cache by only one thread. Secondly, how much cache capacity is occupied by a shared data block depends on the cache configuration. If we assume each core has a private L2 cache, then the block will be replicated at different caches by all cores which access it. However, if we assume the cores share the L2 cache (which may be physically distributed), the block only occupies one cache line. In order to estimate the full potential for CMP scaling due to data sharing, we assume a shared cache configuration. Thus, data sharing brings both a direct and indirect reduction in memory traffic. We also restrict our analysis to the scenario where data is either private to a thread, or is shared by all threads on the chip. This simplification, in addition to the shared cache assumption, overstates the benefit provided by data sharing in most cases, thus giving an upper bound to its effectiveness.

From the point of view of modeling, the effect of data sharing impacts the memory traffic as
if there are fewer independent (non data sharing) cores which are generating memory requests because the first core that accesses a block suffers a cache miss, but other cores’ requests result in cache hits. The effect of data sharing also appears in the cache capacity as if there are fewer independent cores which store their working sets in the shared cache, because a shared line stored in the cache by one core belongs to the working sets of several threads that run on different cores. Thus, to incorporate the effect of data sharing in our model, we define an abstract CMP configuration that has $P'_2$ independent cores, where $P'_2 < P_2$, such that our CMP power law model holds:

$$M_2 = \left( \frac{P'_2}{P_1} \right) \cdot \left( \frac{C_2/P'_2}{S_1} \right)^{-\alpha} \cdot M_1$$

(2.13)

where $P'_2$ is an unknown value that must be estimated. Let $f_{sh}$ represent the average fraction of data cached on-chip that is shared by threads executing on different cores. To estimate $P'_2$, we analyze how $f_{sh}$ directly affects memory traffic requirement. Note that each core fetches its own private data (hence we have $(1 - f_{sh}) \cdot P_2$ fetchers), but only one core fetches the shared data (hence we have $f_{sh}$ fetchers). Thus, there is a total of $f_{sh} + (1 - f_{sh}) \cdot P_2$ cores that fetch data from off-chip memory. However, we also know that in the abstract CMP model, we have $P'_2$ independent cores that fetch data from off-chip memory. Since they must be equal, we can estimate $P'_2$ as:

$$P'_2 = f_{sh} + (1 - f_{sh}) \cdot P_2$$

(2.14)

Figure 2.13 shows the impact of data sharing with various fraction shared ($f_{sh}$) values on memory traffic requirements, normalized to our baseline CMP of 8 cores and 8-CEA caches. Each line corresponds to a future technology generation where we would like to continue proportional core scaling to 16, 32, 64, and 128 cores. An important observation from this figure is that in order to keep the memory traffic constant (at 100%) while still allowing proportional core scaling in each subsequent future technology generation, the fraction of shared data in an application must continually increase to 40%, 63%, 77%, and 86%, respectively. Even if the bandwidth envelope grows in each subsequent generation, the trend of a required increase in the fraction of data sharing in future generations still holds.

Interestingly, in practice the fraction of data sharing in an application tends to decline (rather than increase) as the application runs on more cores. To illustrate this case, we run PARSEC [11] benchmarks on a shared L2 cache multicore simulator, and measure the actual fraction of data sharing. Specifically, each time a cache line is evicted from the shared cache, we record whether the block is accessed by more than one core or not during the block’s lifetime. The results are shown in Figure 2.14, which displays the average fraction of cache lines that are shared by two or more cores. The figure shows that the fraction of shared data in the cache

---

1If private caches are assumed, a shared block occupies multiple cache lines as it is replicated at multiple private caches. Thus, the cache capacity per core is unchanged. In the equation, $\frac{C_2}{P'_2}$ should be replaced with $\frac{C_2}{P_2}$. 

28
tends to decrease with the number of cores, which is the opposite of what is needed to hold down the memory traffic requirement (Figure 2.13). A possible reason for this is that while the shared data set size remains somewhat constant, each new thread requires its own private working set [11]. Therefore, while an increase in data sharing can reduce the memory traffic requirement, it is likely that such an increase will not occur automatically and instead require a significant reworking of the algorithm or parallelization techniques.
2.4.4 Discussion and Recommendations

In this section, we compare the various memory traffic reduction techniques described in Sections 2.4.1, 2.4.2, and 2.4.3 in their effectiveness at enabling CMP scaling in several future generations. We start with a comparison of individual techniques.

Figure 2.15 plots the number of supportable on-chip cores given by our model for four future technology generations (with 2, 4, 8, and 16 times the amount of on-chip transistors available), assuming a baseline of 16 available CEAs (8 for cores, 8 for cache) and memory traffic that is kept constant across generations. The IDEAL case shows proportional core scaling, while the BASE case corresponds to the number of supportable cores if no memory traffic reduction techniques are used. Each remaining configuration corresponds to one of the traffic reduction techniques. Data sharing improvement techniques are excluded because they require program, rather than architecture, changes, and they have been covered in their respective section. The labels used on the x-axis are specified in Table 2.2. Each data point in the figure shows a main point representing a realistic assumption, with a candle bar range representing the spread between optimistic and pessimistic assumptions (Table 2.2). The case of a 3D-stacked cache only shows a single point because we only consider an additional layer of SRAM die.

The figure clearly shows the growing gap between ideal versus baseline core scaling. Comparing the effect of various memory traffic reduction techniques, in general we observe that the "indirect" techniques which increase the effective amount of data that can be cached on-chip (CC, DRAM, 3D, Fltr, and SmCo) are much less beneficial than "direct" techniques which affect the actual amount of data that must be brought on-chip to execute a workload (LC and Sect), which are in turn less beneficial than "dual" techniques (SmCl and CC/LC). The reason for this is evident from our model (Equation 2.5): the benefit of indirect techniques is damped by the \(-\alpha\) exponent, while the benefit of direct techniques directly reduces memory traffic requirements. One exception to this general trend is DRAM storage for on-chip caches, but only because of the huge 8\(\times\) density improvement over SRAM.
It is also interesting to note that the ranges of potential core scaling benefits offered by the direct techniques are quite large, depending on the effectiveness of the technique. While under realistic assumptions, no technique can outpace the ideal proportional scaling, under optimistic assumptions, small cache lines can outpace proportional core scaling for up to three future technology generations. The variability between the impact of optimistic versus realistic assumptions implies that the actual workload characteristics should be fully understood to accurately evaluate the impact of small cache lines.

Table 2.2: Summary of memory traffic reduction techniques.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Label</th>
<th>Realistic</th>
<th>Pessimistic</th>
<th>Optimistic</th>
<th>Effect</th>
<th>Range</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Compress</td>
<td>CC</td>
<td>2x compr.</td>
<td>1.25x compr.</td>
<td>3.5x compr.</td>
<td>Med</td>
<td>Low</td>
<td>Med</td>
</tr>
<tr>
<td>DRAM Cache</td>
<td>DRAM</td>
<td>8x density</td>
<td>4x density</td>
<td>16x density</td>
<td>High</td>
<td>Med</td>
<td>Low</td>
</tr>
<tr>
<td>3D-stacked Cache</td>
<td>3D</td>
<td>3D SRAM layer</td>
<td>40% unused data</td>
<td>10% unused data</td>
<td>80% unused data</td>
<td>Med</td>
<td>High</td>
</tr>
<tr>
<td>Unused Data Filter</td>
<td>Fltr</td>
<td>40% unused data</td>
<td>10% unused data</td>
<td>80% unused data</td>
<td>Med</td>
<td>Med</td>
<td>Med</td>
</tr>
<tr>
<td>Smaller Cores</td>
<td>SmCo</td>
<td>40% less area</td>
<td>9x less area</td>
<td>80x less area</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Link Compress</td>
<td>LC</td>
<td>2x compr.</td>
<td>1.25x compr.</td>
<td>3.5x compr.</td>
<td>High</td>
<td>Med</td>
<td>Low</td>
</tr>
<tr>
<td>Sectored Cores</td>
<td>Sect</td>
<td>40% unused data</td>
<td>10% unused data</td>
<td>80% unused data</td>
<td>Med</td>
<td>High</td>
<td>Med</td>
</tr>
<tr>
<td>Cache+Link Compr.</td>
<td>CC/LC</td>
<td>2x compr.</td>
<td>1.25x compr.</td>
<td>3.5x compr.</td>
<td>High</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Smaller Cache Lines</td>
<td>SmCl</td>
<td>40% unused data</td>
<td>10% unused data</td>
<td>80% unused data</td>
<td>High</td>
<td>Med</td>
<td>Med</td>
</tr>
</tbody>
</table>

Table 2.2 also summarizes qualitative attributes of each technique: the expected benefit to CMP core scaling (Effectiveness), the variability of its benefit to core scaling (Range), and our estimates of the cost and feasibility to implement it (Complexity).

As previously noted, using smaller cores as a memory traffic reduction technique has a low level of effectiveness in aiding CMP core scaling, because the die area freed up for additional cache space has a $-\alpha$ exponent damping effect on memory traffic reduction. However, since smaller cores are slower, the memory traffic will be stretched over a longer time period, allowing them to fit the bandwidth envelope more easily, but at a direct cost to performance. We classify cache compression, 3D-stacked cache, unused data filtering, and sectored caches as techniques with a medium level of effectiveness. While these techniques also increase the effective cache capacity, compared to using smaller cores, they have the potential to increase the amount of cache per core much more drastically (up to 5×). Therefore, they provide a moderate benefit to CMP core scaling. Sectored caches directly impact memory traffic requirements by reducing the amount of data that must be fetched on-chip. However, since the fraction of memory traffic that can be eliminated is modest (40% in our realistic assumptions), it only has a medium level of effectiveness. The most effective techniques for CMP core scaling are DRAM caches, link compression, smaller cache lines, and cache+link compression. Interestingly, although DRAM caches only affect memory traffic indirectly, their huge 8× density improvement over SRAM allows very effective core scaling.

Cache compression, 3D-stacked cache, and smaller cores have a narrow range of variabil-
ity, due to their small range of reasonable assumptions. This implies that the benefit of these techniques on core scaling is more predictable and does not vary much based on workload characteristics. DRAM caches, unused data filtering, and link compression have a medium variability. It is not surprising that more effective techniques have a larger variability. Finally, sectored caches, smaller lines, and cache+link compression have a large variability, and their potential benefits can vary drastically with different workload characteristics (e.g., the compressibility of data, and the degree of spatial locality). It is also interesting to note that the majority of variability for these schemes is in the positive direction, i.e., the number of supportable cores with realistic assumptions is much closer to the number of cores with pessimistic assumptions. The fundamental reason is that the impact of these techniques becomes magnified as they become more effective. For example, sectored caches can reduce the memory traffic by $5\times$ for a workload which does not use 80% of its fetched data, versus $10\times$ for a workload which does not use 90%.

DRAM caches, smaller cores, and link compression have relatively low complexity compared to the other techniques, and in some cases they could be implemented in current-generation systems. Because of the low complexity of DRAM caches, as well as their high effectiveness and low variability, this is an extremely promising technique to enable CMP core scaling in the face of bandwidth wall. Cache compression, unused data filtering, sectored caches, cache+link compression and smaller cache lines are all likely more complex to implement. Cache compression requires a cache organization that handles variable-sized lines, in addition to the compression engine. Unused data filtering, sectored caches, and smaller cache lines will likely require a mechanism to accurately predict which sections of a cache line will not be referenced. Link compression seems more promising than cache compression both because of its effectiveness and its lower complexity because the cache design does not change. Lastly, we rank 3D-caches as the most complex option because it relies on a complex new technology. 3D-stacked caches on their own have a limited effectiveness and the complexity hurdle, but they become a lot more effective when combined with other techniques.

Figure 2.16: Core-scaling with combinations of various techniques for four future technology generations.
Figure 2.16 shows similar results as Figure 2.15 for various combinations of memory traffic reduction techniques, with an increasing number of techniques as we go to the right of the x-axis. This figure shows several significant insights. The rightmost case, which combines all of the highly effective techniques we evaluate, shows a very promising result: even with realistic assumptions about the effectiveness of individual techniques, super-proportional scaling is possible for all four future technology generations (183 cores vs. 128 cores with proportional scaling, at the fourth future generation). It follows that the most effective individual techniques from Figure 2.15 also have the most benefit when combined with other techniques. Together, the combination of link compression and small cache lines alone can directly reduce memory traffic by 70%, while the combination of 3D-stacked DRAM cache, cache compression, and small cache lines, can increase the effective cache capacity by $53\times$, indirectly reducing memory traffic by 84%.

Finally, Figure 2.17 shows the effect of different $\alpha$ values on CMP core scaling. Recall that $\alpha$ is essentially a measure of an application’s sensitivity to changes in cache size. In this figure, we evaluate the minimum and maximum $\alpha$ values that we observed in Figure 2.1 on a select group of techniques starting with DRAM caches and working up through combinations of DRAM caches with several other techniques. An important observation from this figure is that $\alpha$ can have a large impact on core scaling in future generations. In the baseline case, a large $\alpha$ enables almost twice as many cores as a small $\alpha$. When traffic reduction techniques are applied, the gap in number of supportable cores between a large and small $\alpha$’s grows even more. A small $\alpha$ prevents the techniques from achieving a proportional scaling, while a large $\alpha$ allows a super-proportional scaling.

2.5 Related Work

This study is motivated in part by Emma and Puzak’s work presented in [86]. Our model is built upon the study by Hartstein et al. [33], that validates the long-observed $\sqrt{2}$ rule of how cache miss rate changes with the cache size. Our work extends this general model to a CMP architecture by modeling how die area allocation to cores and caches affect the off-chip memory traffic in current as well as future technology generations. In [26], the impending memory bandwidth limitations as CMP core scaling continues is noted, and the effect of the power law of cache miss rates on the relationship cache and bandwidth is discussed. Again, our work takes this idea further by developing a formal CMP memory traffic model, as well as studying the effectiveness of a variety of bandwidth conservation techniques at reducing memory traffic. Hill and Marty [5] develop a novel analytical model to describe the performance of various CMP designs. This model focuses on the performance of chips with cores of a varying number and capability on workloads with different amounts of parallelism, while our model focuses on how
memory bandwidth limitations impact CMP design.

Other prior studies have also identified the bandwidth wall as a performance bottleneck for CMP systems [39] and even for high-performance single processor systems [18]. However, these studies do not quantify its effects on performance and throughput for large-scale CMP systems as we do in this work. Additionally, some studies have looked at CMP design space search [39, 58] to determine the best chip configuration (e.g. number of cores, complexity of cores, cache organization, etc.) for a target workload mix. However, these studies focus on smaller-scale CMP systems, where the memory bandwidth bottleneck is not as critical.

A more closely related study to our work is Alameldeen’s PhD thesis [2], which uses an analytical model to study how to balance cores, caches, and communication to maximize IPC in a current generation CMP that employs cache and link compression. In contrast, our work focuses on modeling how memory traffic limits the the number of on-chip CMP cores across technology generations. We evaluate the impact of many bandwidth conservation techniques, including cache and link compression, and provide a comparison on their effectiveness.

2.6 Acknowledgments

This work was done in collaboration with Brian Rogers, Gordon Bell, Xiaowei Jiang, Yan Solihin and Ken Vu.
Chapter 3

Data Sharing in Multi-Threaded Applications and its Impact on Chip Design

This chapter is organized as follows: Section 3.1 presents a new methodology to measure the impact of data sharing and compares it to prior art. Section 3.2 uses this new methodology to measure the impact of data sharing in an area constrained CMP running contemporary multi-threaded workloads. Section 3.3 describes how we extend an analytical model for CMP performance to account for the impact of data sharing. Section 3.4 presents a case study that uses the analytical CMP performance model, and explores optimal CMP design in the presence of realistic data sharing in multi-threaded applications. Section 3.5 describes prior and related art.

3.1 Measuring the Impact of Data Sharing

3.1.1 Why is measuring the impact of sharing difficult?

In multi-threaded applications data sharing can help improve cache miss rates. For example, if one thread brings a cache block on-chip and one or more of the other threads are able to use the cache block, then, effectively, data sharing has avoided additional misses and has reduced the cache miss rate. Although it is easy to see this qualitative impact of data sharing, it is not easy to quantify it. There are three main reasons for this.

Sensitivity to cache geometry First, an application’s algorithmic sharing behavior may be quite different from the sharing behavior realizable on-chip. This is because on-chip caches can only store a temporally and spatially local subset of the application’s full memory footprint.
For example, application threads may, according to the algorithm, all work on the same data; however, unless they work on that data while it is resident in the on-chip cache, the full potential of algorithmic sharing is not realized. Therefore, the cache size, geometry and configuration (private per thread or shared across threads) can potentially impact the realizable data sharing potential.

Figure 3.1: Miss rate profiles for ferret(a), x264(b) and fluidanimate(c) - x-axis is L2 cache size in bytes, y-axis is the cache’s data miss rate

**Sensitivity to thread count** Second, an application’s sharing behavior, typically, changes with the number of threads employed by the algorithm. On the one hand, as the application employs a greater number of threads, the potential for realizable data sharing among those threads increases. For example, with 2 threads, a block can, at most, be shared by those 2 threads; however, with 4 threads the block may be shared by all 4. On the other hand, since the effective cache space available to each thread reduces as the application employs more threads, the realized sharing might fall. Continuing from the previous example, even if the 4 threads shared a block, the increased competition for cache space among the threads may not allow the block to stay long enough in the cache to be shared, thereby nullifying the potential of the increase in thread count.

A common misconception related to this point is that the miss rate profile (miss rates across a range of cache sizes) of an application does not change much between its single-threaded and various multi-threaded incarnations [82]. Figure 3.1 shows that this is not always a good assumption. The figure plots the miss rate profiles for 3 benchmarks in PARSEC, each running with 1, 4 or 16 threads. The plots show that the multithreaded miss rate profiles are different from the single threaded miss rate profiles. Further, there is no clear pattern. For example, Figure 3.1(a) shows that for the ferret benchmark the miss rates typically become worse (higher is worse) as the number of threads increase. For x264 (Figure 3.1(b)), however, miss rate worsens going from 1 to 4 threads, before improving for 16 threads. For fluidanimate (Figure 3.1(c)), miss rates generally improve with more threads. This shows that using the
single-threaded miss rate profile to project multi-threaded miss rate profiles is incorrect and may lead to inaccurate conclusions. It also highlights the difficulty in predicting which of the two opposing effects due to increasing thread counts dominates – shrinking cache space per thread versus the increasing data sharing opportunity.

**Mimicking the absence of sharing** Third, in order to attribute any improvement in the cache miss rate to data sharing, we must be able to contrast the miss rate with data sharing against what the miss rate would have been had there been no sharing.

### 3.1.2 How do current approaches measure this impact?

Techniques have been proposed in literature to quantify the impact of data sharing on cache miss rate. Most existing approaches to isolating the impact of data sharing simulate a multi-threaded incarnation of the application, and measure the baseline miss rate. In addition, they also identify the average number of sharers for a typical cache block. They then use this *average sharer count* to project what the cache miss rate would be if sharing were absent. The difference between the measured miss rate (with sharing allowed) and the projected miss rate (no sharing) estimates the impact of data sharing on miss rate reduction.

**Projection Technique 1** In this approach [2, 74, 82], sharing is assumed to be responsible for an increase in the effective cache size by a factor equal to the average sharer count, \( n \). The reasoning behind this is that every cache block that is shared by \( t \) threads saves \( t - 1 \) cache blocks worth of space in the cache, that would have been needed if the block were not shared. The difference in the effective cache size with and without sharing is then tied back to the effect on miss rates with and without sharing, by using the well known Power Law of cache misses [33].

**Projection Technique 2** This, more aggressive, approach is also foreseeable. In this approach, a sharer count of \( n \) is assumed to be responsible for a direct factor-\( n \) reduction in the miss rate; there is no Power Law based tempering of the impact of \( n \) on the miss rate in this approach. The reasoning behind this approach is that a block shared by \( t \) threads saves \( t - 1 \) misses, which would have been exposed if sharing were absent.

### 3.1.3 Why are current approaches insufficient?

Even though these projection techniques correctly use the base line miss rate of the appropriate multi-threaded version of the application, they still fail to correctly project the miss rate to the no-sharing case. In fact, these techniques almost always underestimate data sharing’s role in miss rate reduction. We quantify this effect in Section 3.1.6. Here we provide an insight about why these techniques may end up projecting pessimistically.

Projection Technique 1 is overly conservative in that it fails to take into account the direct
reduction in misses that sharing enables. Projection Technique 2 fixes this aspect of technique 1; however, it ignores two subtle details. Not only does a shared block affect the miss rate directly by avoiding a miss, it also saves cache space by not having to use up another line. That is, without data sharing, additional cache blocks must be pulled into the cache. This fundamentally alters the temporal and spatial locality of the data, and therefore the cache behavior. Further, sharing naturally enhances the block’s lifetime in the cache. Multiple interleaved accesses from different sharers keep a shared cache block warmer in the cache. This warming benefit cannot be captured simply by counting the number of sharers. We argue that the average number of sharers parameter is insufficient to accurately measure the impact of data sharing, and this is a fundamental limitation in the current approaches.

In order to make our argument clearer, we provide a simple, yet illustrative, example. Consider a cache capable of holding only one cache block, and shared by 2 threads, where each thread alternately requests access to a given cache block as follows - $0A,1A,0A,1A$ ($0$ and $1$ represent the requesting threads, and $A$ is the cache block address). In this case the miss rate is 25% (the first access misses and the other 3 hit), and the number of sharers is 2. If this is used to project the miss rate for the no-sharing case, using the aggressive Projection Technique 2 we get $2 \times 25\% = 50\%$. In reality, however, with no sharing, each thread would have to displace the other thread’s block in the cache since each thread would have to bring in a private copy of the cache block. The same access sequence would result in a 100% miss rate. This is a contrived example that, while exaggerating the issue, highlights the fundamental problem with prior techniques, as well as develops an intuition for why they always underestimate the impact of sharing. There is, therefore, a need to find a better way to measure the true impact of sharing.

3.1.4 Our approach

Instead of estimating the miss rate with no sharing using these projection techniques, we actually simulate the no-sharing case. We modify our cache model so that it can run in either the Sharing Mode or the No-Sharing Mode. Sharing Mode refers to the default mode of operation where threads can share data via the cache. No-Sharing Mode simulates a scenario where data sharing is artificially disallowed. This is accomplished by tagging each newly installed cache block with its requesting thread’s identifier and granting a cache hit to future accesses only when both the cache tag and the thread number match. This effectively forces threads to create private copies of shared cache blocks, which accomplishes simulating both the extra cache miss as well as the extra cache space usage expected in the absence of sharing.
3.1.5 Methodology

We use 12 of the 13 benchmarks from the PARSEC 2.1 multi-threaded benchmark suite (all except raytrace, which we had trouble compiling), and 8 benchmarks of the NAS parallel benchmark suite. We use the Virtutech Simics [61] full-system simulator with the Linux operating system as our simulation infrastructure. We run each benchmark in the single-threaded mode as well as in 2, 4, 8, 16 and 32 threaded mode and create checkpoints at the start of the main work loop (PARSEC benchmark source code clearly identifies this code region). In order to ensure that threads do not move between cores during simulation, we modified the benchmark source code to affinitize each thread to a specific core. We run the simulation for 10 billion instructions or till the end of the region of interest, whichever happens sooner. The number of cores simulated under Simics matches the target thread count for each run; we assume each core is single-threaded. We model an 32KB L1 instruction cache and 32KB L1 data cache per core, and a 64B block size; these remain unchanged through the rest of the studies in this dissertation.

For every benchmark, and every thread count, we vary the L2 cache behavior along three dimensions: cache size, cache configuration, and, sharing allowance. We vary the L2 cache size from 256KB through 32MB. We vary the configuration of the L2 cache to either be a single shared L2, shared by all simulated cores, or be a collection of per-core private L2 caches. In a private L2 configuration with \( k \) cores, each core has \( \frac{1}{k} \) the L2 space of the corresponding shared L2 configuration. The cache miss rate we talk about in this work always refers to the total on-chip L2 cache - regardless of the configuration being private or shared. That is, a miss implies the data was not found anywhere on chip; if the data were available on a private L2 slice different from the requesting core’s L2, we assume cache-to-cache forwarding will be able to provide that data, and it is treated as a cache hit. We implement the standard MESI protocol to ensure that the coherence state is correctly updated among all caches on chip. We model the L2 (of any size and configuration) to operate in one of two modes - Sharing or No-Sharing (Section 3.1.4).

In addition, we track the number of sharers per cache block (logged at block eviction time), and use the average sharer count to project the no-sharing miss rate in accordance with the two projection techniques described in Section 3.1.1. We found one implementation detail to be important to get right, particularly for shared cache configuration. We implement each private L2 cache to be 4-way set associative. With a shared cache, especially with large number of cores sharing it, or in the replication-rich, no-sharing mode, we must ensure that associativity limits do not artificially increase the miss rate. To ensure that, we grow the associativity of a shared cache in proportion with the number of sharers. While it is certainly not practical to implement a 128-way shared L2 cache for a 32-core system, it allows the analysis to focus
Figure 3.2: Impact of data sharing on miss rates across projection techniques, cache sizes and thread counts for *vips* with private caches (a), *vips* with shared cache (b), *canneal* with private caches (c), *canneal* with shared cache (d), *bodytrack* with private caches (e), and, *bodytrack* with shared cache (f).

on the qualitative differences between a private and shared cache configurations, unaffected by associativity-related sensitivity.

### 3.1.6 Results

Figure 3.2 shows the cache miss rates measured for data accesses (as opposed to instruction accesses\(^1\)) for three benchmarks across varying thread counts, cache sizes and cache configurations. Each stacked bar shows four quantities - the miss rate with data sharing allowed as normal, projected miss-rate in the no-sharing case using Projection Technique 1, projected miss rate with Projection Technique 2, and the simulated no-sharing miss rate.

Figure 3.2(a) shows the results for the *vips* benchmark under a shared L2 cache configuration, whereas Figure 3.2(b) shows the results for the *vips* benchmark under a private L2 cache configuration. The figure shows that regardless of the configuration, and thread count, a larger cache has a lower miss rate. The figure also shows that a private cache has a higher base miss rate (the miss rate with sharing allowed) than a shared cache configuration. We attribute it to the fact that in a shared cache there is only 1 copy of a shared block, whereas in

\(^1\)Instruction accesses are typically much smaller in number and we find that their inclusion in this plot does not change the results noticeably
a private cache a block may be replicated (multiple lines in the shared coherence state) which effectively reduces capacity and increases miss rates. Additionally, the shared cache has the advantage of a larger associativity. Note, however, that the intent of this work is not to compare the cache configurations, but rather, to understand data sharing in each cache configuration independently.

The main takeaway form this figure is that the overall impact on miss rates due to sharing is rather small, and mostly relegated to the cases with a small cache and a large number of threads. The two projection techniques do fall short in terms of projecting miss rate increases in the absence of sharing; however, it may not matter much in this case.

Figure 3.2(c) and (d) show the cache miss rate results for the *canneal* benchmark with shared and private cache configurations respectively. This figure shows that for the *canneal* benchmark, in both private and shared cache configurations, miss rates are significantly impacted by data sharing (increasing by up to 4 times in its absence). Further, it is quite clear from the plots that the existing projection techniques, while doing a pretty good job of covering the difference in miss rates, leave a significant fraction unaccounted for.

This is especially noticeable in the shared cache configuration, although even in the private cache configuration, the increase in miss rates is quite drastic in some cases. Existing projection techniques do very well in matching the simulated miss rate measurement in some cases; however, there remain some cases where they fall significantly short.

Figure 3.2 (e) and (f) shows the cache miss rate results for the *bodytrack* benchmark. We can see that the miss rates are drastically impacted by data sharing, especially for the smaller cache sizes (≤8MB).

Overall, Figure 2 shows that there exists noticeable variability in the impact of data sharing on miss rates across workloads and cache configurations. The figure also highlights the variability in how well existing approaches are able to measure this impact.

Figure 3.3 summarizes the findings from this study; it shows the relative increase in miss rates as projected by the existing techniques and as measured in simulation, averaged across the 12 PARSEC benchmarks we studied. The bars are normalized to the base case of the miss rate with sharing allowed as normal. The main takeaway from this figure, and this section, is that it is not possible to consistently project the impact of data sharing on the cache miss rate simply by using the average number of cache block sharers. Doing so tends to underestimate the positive role data sharing plays in reducing cache miss rates. It is important, therefore, to use simulation to understand the true impact of data sharing.
3.2 Impact of Sharing in a fixed-area CMP

In Section 3.1 we developed a methodology to measure data sharing’s impact on cache miss rates in an unconstrained design (cache size is independent of thread count). In this section we focus on a constrained design – a fixed-area CMP – where, as more of the chip area is devoted to computation (cores), less area remains for the caches. In the context of multi-threaded applications, as the number of thread contexts on chip increases linearly, the total on chip cache decreases linearly and the amount of effective cache space per thread decreases super-linearly. Therefore, in an area-constrained CMP, data sharing has the potential to play a more pronounced role in determining the optimal compute-to-cache resources. We will focus on the impact of data sharing on optimal chip design in Section 3.4. In this section we focus on understanding how the on-chip cache miss rate in an area-limited CMP is affected by the presence of data sharing in contemporary workloads.

3.2.1 Simulated Machine

We use the same simulator, benchmarks and baseline core methodology as described in Section 3.1.5. We assume that each core in the CMP is the same size as 1MB of L2 cache (we
elaborate on this assumption in Section 3.4.2). We constrain the CMP area to be equivalent to the 33 Cache Effective Areas (CEAs); a CEA is the area needed by 1MB of cache. The CMP can contain 1 core with 32MB of L2 cache, or, 32 cores and 1 MB of L2 cache, or, any intermediate combination. We simulate a private L2 configuration described in Section 3.1.5. We run each application with 1, 2, 4, 8, 16 and 32 threads, which corresponds to an equivalent number of cores. For each application and thread-count, we measure the L2 miss rates both with sharing allowed as normal, and with sharing artificially disallowed by keeping private copies of the block per sharer (as described in Section 3.1.4).

3.2.2 Results

Due to space limitations we highlight the findings of this study by showing the detailed results for 2 of the 20 applications, and then summarize the findings by showing the average behavior across all applications.

Figure 3.4(a) shows that as the number of threads (and cores) increases from 1 to 32, freqmine benefits more and more from data sharing (2nd y-axis). The absolute impact on miss rates is still relatively small - 13% decrease in miss rate due to sharing.

Figure 3.4(b) shows that as the number of threads in canneal increases, the impact of data sharing keeps increasing to about 3x with 16 threads, beyond which the impact of data sharing starts falling again. Considering what we know about canneal from the unconstrained experiments (Figure 3.2(c) and (d)) this decrease cannot be attributed to any sudden decrease in the available algorithmic sharing opportunity when going form 16 to 32 threads. Instead, the
fall in sharing impact must be attributed to the reduction in the cache space. This indicates that *freqmine* has tight sharing behavior (one that can manifest even with a small amount of cache); *canneal* has a sharing behavior that needs a minimum amount of cache to be effective.

Table 3.1: PARSEC and NAS benchmarks grouped by their data sharing trends

<table>
<thead>
<tr>
<th>Type</th>
<th>Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sharing Style 1 (Figure 3.5 (Group 1))</td>
<td>blackscholes, ferret, swaptions, x264, canneal, dedup, streamcluster,</td>
</tr>
<tr>
<td></td>
<td><em>bt, ep, ft, is, lu, mg, sp</em></td>
</tr>
<tr>
<td>Sharing Style 2 (Figure 3.5 (Group 2))</td>
<td>bodytrack, facesim, fluidanimate, freqmine, vips, cg</td>
</tr>
</tbody>
</table>

Figure 3.5: Data sharing impact for application groups 1 and 2: measured impact and a canonical trend line

Indeed, we see these two general trends across most of the applications we studied. We categorize the applications we studied into two groups based on the sharing trend they exhibit. Table 3.1 shows these groups. Figure 3.5 shows the data sharing impact trends as an average for each of these two application groups (darker line). We also show the canonical trend line; we find that a 2nd degree polynomial suffices to capture the behavior quite accurately for the purposes of this work. We use this canonical trend as an input to the analytical throughput model used in Section 3.4.
3.3 Analytical Model for Chip Performance

In this section we present the analytical model we use for projecting CMP throughput. This model projects the chip’s Instructions Per Cycle (IPC) metric, and is based on the approach developed by Wentzlaff et al. [94] and extended by Krishna et al. [48]. For a full derivation we refer the reader to these papers; however, for completeness, we provide a brief discussion of the model derivation approach, and the parameters it takes into account. Table 3.2 lists all the parameters used in the IPC model.

### Table 3.2: Parameters used in the IPC model

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$IPC_C$</td>
<td>Instructions per Cycle of the chip</td>
</tr>
<tr>
<td>$IPC_c$</td>
<td>Instructions per Cycle at a core</td>
</tr>
<tr>
<td>$CPI_C$</td>
<td>Cycles per Instruction of the chip</td>
</tr>
<tr>
<td>$CPI_c$</td>
<td>Cycles per Instruction at a core</td>
</tr>
<tr>
<td>$CPI_{pre-L2}$</td>
<td>CPI in the core and L1s</td>
</tr>
<tr>
<td>$CPI_{post-L2}$</td>
<td>CPI in the L2 (lookup time)</td>
</tr>
<tr>
<td>$n$</td>
<td>number of cores in the chip</td>
</tr>
<tr>
<td>$f_s$</td>
<td>Fraction of serial code in a parallel application</td>
</tr>
<tr>
<td>$api_{L2}$</td>
<td>L2 accesses per instruction</td>
</tr>
<tr>
<td>$mpi_{L2}$</td>
<td>L2 misses per instruction</td>
</tr>
<tr>
<td>$CEA$</td>
<td>Cache Equivalent Area (Area of 1MB of cache)</td>
</tr>
<tr>
<td>$A_C$</td>
<td>Total area of the chip for cores and cache, in CEAs</td>
</tr>
<tr>
<td>$A_c$</td>
<td>Area of each core in CEAs</td>
</tr>
<tr>
<td>$A_{L2}$</td>
<td>Area of each core’s private L2 slice in CEAs</td>
</tr>
<tr>
<td>$l_{L2}$</td>
<td>L2 hit latency</td>
</tr>
<tr>
<td>$m_{L2}$</td>
<td>L2 miss rate</td>
</tr>
<tr>
<td>$T_Q$</td>
<td>Memory Interconnect Queueing Delay</td>
</tr>
<tr>
<td>$T_M$</td>
<td>Memory Access Penalty</td>
</tr>
<tr>
<td>$m_{1MB_{L2}}$</td>
<td>L2 miss rate for a 1MB L2</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Sensitivity factor from the Power Law of Cache Misses</td>
</tr>
<tr>
<td>$E(n)$</td>
<td>Data Sharing Impact on miss rates</td>
</tr>
<tr>
<td>$\mu$</td>
<td>Mean per-core memory interconnect service rate (B/cyc)</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>Mean per-core memory request rate (B/cyc)</td>
</tr>
<tr>
<td>$\rho$</td>
<td>$\lambda/\mu$</td>
</tr>
<tr>
<td>$B$</td>
<td>Memory bandwidth in GB/s</td>
</tr>
<tr>
<td>$f$</td>
<td>Core frequency in GHz</td>
</tr>
<tr>
<td>$b_{L2}$</td>
<td>L2 cache block size in bytes</td>
</tr>
</tbody>
</table>

**IPC model** Equation 3.1 calculates the chip’s performance in terms of Instruction Per Cycle, $IPC_C$, in terms of each core’s Cycles Per Instruction metric, $CPI_c$.

$$IPC_C = n/CPI_c$$  \hspace{1cm} (3.1)
The implicit assumption that is reflected in this equation is that the chip is composed of a collection of homogeneous cores, and the workload being run on each core is also homogeneous. The latter assumption is reasonable for at least a subset of multithreaded workloads – those that employ data level parallelism (as opposed to pipelined parallelism).

It has been observed [5] that for multi-threaded workloads realistic parallel scaling is limited by the fraction of the serial, non-parallelizable, portion of the application’s instructions, represented by $f_s$. Equation 3.2 incorporates this effect.

$$IPC_C = \frac{n}{1 + f_s \cdot (n - 1)} \cdot \frac{1}{CPI_c}$$

**Components of CPI** $CPI_c$ can be calculated as a summation of the average number of cycles spent by an average instruction in three distinct phases - before reaching the L2, in the L2, and, beyond the L2. Equation 3.3 describes this.

$$CPI_c = CPI_{preL2} + CPI_{inL2} + CPI_{postL2}$$

$CPI_{preL2}$, the CPI of the application, assuming memory operations are always satisfied with a latency corresponding to the L1’s latency, can be obtained by simulation or measurements for the core designs and applications under consideration. The $CPI_{inL2}$ term corresponds to the cycles an average instruction spends in the L2. It depends on the average number of L2 accesses per instruction, represented by $api_{L2}$, and the latency of an L2 hit, represented by $l_{L2}$. It is given by Equation 3.4

$$CPI_{inL2} = api_{L2} \cdot l_{L2}$$

$api_{L2}$ can be obtained by simulation and depends on the L1 miss rate for an application; if the L1 satisfies most loads and stores, or, if there are very few loads and stores, then the number of L2 accesses per instruction will be low. The L2 hit latency, $l_{L2}$, depends on the size of the L2 and the transistor technology. In order to estimate the size of the L2 (in bytes) we first estimate the L2 area available to each core on the chip, $A_{L2}$, using Equation 3.5. $A_C$ represents the chip area available to cores and cache, and $A_c$ represents a single core’s area (all areas are measured in terms of Cache Equivalent Areas, CEAs). The equation subtracts the area for $n$ cores from the available chip area to get the cache area, and then divides the cache area evenly across the $n$ cores. Then we use the CACTI6.5 [34] tool to identify $l_{L2}$ for that L2 size and assume an appropriate technology node size (such as 22 nm).

$$A_{L2} = (A_C - A_c \cdot n)/n$$

$CPI_{postL2}$ depends on how often an average instruction misses the L2, and, on how long it takes
to return after a miss. *How often* the average instruction misses the L2 is given by Equation 3.6. This equation describes how the average number of L2 misses per instruction, \( mpi_{L2} \), is given by the L2 miss rate for a given application and a given L2 size, \( m_{L2} \), and the number of L2 accesses per instruction, \( api_{L2} \).

\[
mpi_{L2} = api_{L2} \cdot m_{L2} \tag{3.6}
\]

In order to calculate \( m_{L2} \), we use the well-known Power Law of cache misses [33] and \( E(n) \), the canonical sharing-impact curves from Section 3.1, as shown in Equation 3.7. We assume that \( \alpha \) and \( m_{1MBL2} \) are both obtainable from measurement or simulation.

\[
m_{L2} = m_{1MBL2} \cdot A_{L2}^{-\alpha} \cdot E(n) \tag{3.7}
\]

*How long* the average instruction takes beyond an L2 miss detection is given by the sum of the fixed latency component, \( T_M \), and the variable queuing delay component, \( T_Q \). We put these together into Equation 3.8.

\[
CPI_{postL2} = mpi_{L2} \cdot (T_M + T_Q) \tag{3.8}
\]

\( T_M \) is the memory access penalty beyond the L2, which, typically, a well understood quantity given the memory chip and memory controller in consideration. Coherence protocol overheads and average on-chip network delays (often, tens of processor clocks) may be added to the pure memory penalty (often, hundreds of processor clocks) to get \( T_M \). To calculate \( T_Q \), we assume a standard M/D/1 queuing model to represent the memory interconnect. We model the arrival of memory requests as a Poisson process. The mean arrival rate in bytes per cycle depends on the request rate (L2 misses per cycle) and the request size (\( b_{L2} \)). The per-core service rate, \( \mu \), and the per-core request rate, \( \lambda \), are given by Equation 3.9 and Equation 3.10.

\[
\mu = B/(f \cdot n) \tag{3.9}
\]

\[
\lambda = (mpi_{L2} \cdot b_{L2})/CPI_c \tag{3.10}
\]

For an M/D/1 queue, the queuing delay (\( T_Q \), is given by Equation 3.11.

\[
T_Q = \rho/(2 \cdot \mu \cdot (1 - \rho)) \tag{3.11}
\]

**Final CPI Model** Now we have all the components of Equation 3.3. Equation 3.12 puts

\[2\]Unlike off-chip interconnects, on-chip interconnects are not pin-limited, and are less likely to become the first bandwidth bottleneck. They do impact latency, and are accounted for in the \( T_M \) term.
it all together.

\[ CPI_c = CPI_{preL2} + (api_{L2} \cdot l_{L2}) + \]
\[ mpi_{L2} \cdot (T_M + \frac{mpi_{L2} \cdot f^2 \cdot n^2 \cdot b_{L2}}{2 \cdot B^2 \cdot CPI_c - 2 \cdot B \cdot f \cdot n \cdot mpi_{L2} \cdot b_{L2}}) \]  

(3.12)

Once all the parameters are set, we end up with a quadratic equation in \( CPI_c \). These parameters are the workload dependent parameters \( (\alpha, m_{1MBL2}, api_{L2}, E(n)) \), the microarchitecture and workload dependent parameter \( (CPI_{preL2}) \), the microarchitecture and technology process dependent parameter \( (l_{L2}) \) and design specific variables \( (CEA_C, n, CEA_c) \). We solve this quadratic equation to calculate \( CPI_c \) and incorporate it back into Equation 3.2 to compute the overall chip performance.

### 3.4 Chip Design: A Case Study

![Figure 3.6: Optimizing the core-to-cache ratio under No-Sharing (a), Sharing Style 1 (b), and Sharing Style 2 (c). Limited b/w = 30 GBps](image)

In this case study we use the analytical model developed in Section 3.3 to study how differently a future many-core chip must be designed when targeting multi-threaded workloads instead of multi-programmed workloads. In Section 3.4.1 we first define a hypothetical future many-core chip by assigning values to many of the parameters of the analytical model. We fix the parameters related to the process technology, area budget, workload characteristics and core microarchitecture using reasonable assumptions, thus allowing the model to explore a focused and relevant design space. In Section 3.4.2 we use this parameterized model to explore two important chip design questions. First we explore how the optimal on-chip core to cache area ratio changes between a workload with data sharing compared to a workload with no data sharing. Second we explore whether it is better to build a many-core chip with many small cores, or with fewer larger cores in the presence of data sharing.
3.4.1 Assumptions

We assume a 400\text{mm}^2 chip in the 22nm process technology. We assume that 70% of the chip area is available for placing cores and lower level caches (the remainder is used by the on-chip interconnect, memory controllers and other pervasive logic). Using CACTI6.5 [34] we calculate that 1MB of SRAM cache occupies about 1.2\text{mm}^2 (we assume 8-way set associativity). This gives us about 256 CEAs (CEA is equal to 1MB of cache).

For the baseline core microarchitecture, we use area estimates from three recent core designs - IBM’s latest embedded processor core, PPC476FP [40] (out-of-order, multi-issue superscalar, single-threaded core, with Floating Point support), ARM’s Cortex A9 core [7] (out-of-order, multi-issue superscalar, with Floating Point support) and MIPS 1004K core [66] (in-order, dual-core design with dual-threading per core, with Floating Point support). We find that at 1.2 to 1.5\text{mm}^2 (in 22nm technology) there are several baseline core designs which can be used as a building block for CMP. This baseline core choice allows us to approximate $CEA_c$, the number of CEAs needed per core, to 1. We will relax this core size restriction in Section 3.4.2.

For the workload characteristics we assume a memory intensive workload. We assume an $api_{L2}$, the probability of an average instruction accessing the L2, of 0.033. This corresponds to the $api_{L2}$ we measured for canneal, facesim, swaptions and x264 benchmarks. Most other workloads we studied have a much smaller rate to access to the L2, in particular NAS parallel benchmarks. However, we intentionally choose a memory intensive workload to illustrate the impact of sharing while the memory system is stressed. We assume a $m_{1MB}$, the miss-rate with a 1MB L2 cache, of 0.6. This again corresponds to a memory intensive workload, quite like streaming workloads in general purpose, single-threaded CPU performance benchmarks. We made the decision to go with such an example to more clearly illustrate the high-level insights. We assume the standard $\sqrt{2}$ miss-rate sensitivity [33] for the power law ($\alpha=0.5$). We set $CPI_{preL2}$ to 2 (this is varied in Section 3.4.2).

According to the projections from ITRS [43] chip pin-counts are growing at a rate of 5% per year. For a cost-effective bus signaling solution, the rate of growth of bus frequency is about 10%. This falls significantly short of the 59% per year growth in transistor density which is expected to continue for at least another decade. In order to understand the impact of data sharing on chip design in the presence of a potential bandwidth wall, we consider two bandwidth scenarios. We assume that either there is always sufficient bandwidth (infinite bandwidth), or, that the off-chip bandwidth is limited to 30 GBps (15 GBps in each direction). This limited bandwidth scenario mimics the usable bandwidth available with a 128-bit dual-channel DDR3-1066.

We assume that the processor frequency is 2GHz, the memory penalty (including coherence

---

3CACTI6.5 provides latency and area estimates only down to 32nm process technology; we measure area and latency for different cache sizes for 90nm, 65nm, 45nm and 32nm technologies and extrapolate these to 22nm.
protocol overheads) is 400 processor clocks, and the L2 cache line size is 64 bytes.

We assume that the multi-threaded code cannot be perfectly parallelized across threads, and that the serial section accounts for 1% of the code ($f_s = 0.01$).

We study the two data sharing trends identified in Section 3.2.2, and bookend their performance by the no-sharing case on one end and the ideal sharing case on the other. Ideal sharing refers to the hypothetical situation where only one thread needs to bring data on chip, and all the other threads are able to use this data and therefore suffer no misses. We use a conservative approach to extrapolating the sharing trends from our simulations with up to 32 cores (Figure 3.5) to the current case study with up to 256 cores. We do not increase the impact of sharing on miss rates beyond what we observed with our simulations. We saw that data sharing makes about 2x impact on miss rates with 16 or 32 threads. We assume that data sharing will still make only about a 2x impact with 128 or 256 threads. While we agree that there may be overly conservative and that more sharing opportunity may exist with more threads, there is no way for us to predict that potential. So in that sense, the results shown here may be treated to be a conservative estimate of how much data sharing can impact CMP designs of the future.

![Figure 3.7: Optimizing core size under No-Sharing (a), Sharing Style 1 (b) and Sharing Style 2 (c)](image)

### 3.4.2 Results and Analysis

#### Core to Cache Ratio Decision

Figure 3.6 shows how the chip performance (y-axis), in terms of IPC, varies as the number of baseline cores on the chip is increased (x-axis) from 1 to 255. The figure shows the following 3 sharing scenarios - No Sharing, Sharing Style 1 and Sharing Style 2 (sub-figures a,b and c).
For each sharing scenario we evaluate the performance trends under two memory bandwidth assumptions - infinite and limited (30 GBps).

With sufficient bandwidth available, the peak chip-IPC is obtained with 84 cores (33% of chip area) in the absence of sharing. With sharing, however, the peak chip-IPC is obtained with 112 or 126 cores (44% to 49% of chip area), depending on the sharing pattern. This shows that the optimal core-to-cache ratio can be boosted significantly by data sharing. Notice that not only does the on-chip cache become smaller with sharing, the number of cores increases too. This increased core-count helps improve the absolute chip IPC (by 10% to 58%) as can be seen from the figure.

When bandwidth is limited, however, there is no significant change in the optimal core-to-cache ratio due to sharing. Further, chip performance potential is gated by the bandwidth wall. This highlights the importance of considering the available bandwidth when modeling the potential of data sharing.

**Core Size Decision**

So far we have assumed a baseline core that is 1 CEA in size (Table 3.2) and has an infinite L1 CPI of 2. In this section we relax this constraint and consider 3 larger baseline core designs. We use Pollack's observation \[77\] to set the infinite L1 CPI's for these larger cores. Specifically, we consider cores which are 2, 4 and 8 CEAs in size and have infinite L1 CPIs of 1.41, 1 and 0.71 respectively (lower CPI is better). Figure 3.7 shows the chip performance (y-axis) as the CEAs devoted to cores is increased (x-axis). The figure shows 3 sharing scenarios - No Sharing, Sharing Style 1 and Sharing Style 2 (sub-figures a, b and c). For each sharing scenario we evaluate the performance trends under 8 conditions - 4 core sizes, each under 2 bandwidth assumptions.

Figure 3.7(a), the No Sharing case, shows that going to larger cores generally worsens the chip IPC under both limited and unlimited bandwidth. This result suggests that smaller cores are appropriate for multi-threaded applications which do not have data sharing, and perhaps also for multiprogrammed workloads.

Figure 3.7(b), the Sharing Style 1, shows that larger cores do degrade performance in the unlimited bandwidth scenario. However, when the offchip bandwidth is limited, larger cores may even outperform smaller cores (see 2CEA cores vs. 1CEA cores). While larger cores are expected to outperform smaller cores in terms of single-threaded performance, it is insightful that data sharing enables this advantage to be realized even for multi-threaded applications at the chip level. There are two main reasons for this behavior. First, larger cores can more efficiently utilize the available bandwidth, as is clear from the shrinking difference between the unlimited and limited bandwidth curves as core sizes increase; for the 8CEA core there is almost no difference. Second, larger cores lead to fewer cores on chip. This, in turn, reduces
the pressure on the on-chip caches, allowing smaller caches and a larger fraction of the chip to be dedicated to cores.

Figure 3.7(c), shows that for applications with Sharing Style 2, the difference in chip IPC is very small between the 1CEA, 2CEA and 4CEA designs, even in the presence of unlimited bandwidth. When bandwidth is limited, the optimal chip-IPC is nearly identical across several core sizes, again suggesting that larger cores should be considered with applications that share data. Comparing Figure 3.7(a) and (c), it is clear that while data sharing does not improve the peak performance for the chips based on 1CEA cores, it significantly improves the peak performance for chips based on the 2CEA, 4CEA and 8CEA cores.

These results confirm that larger cores should be considered when designing chips that target multi-threaded applications with even moderate data sharing. Further, these results show why larger cores are perhaps even more appropriate in limited bandwidth designs.

3.5 Related Work

**Analytical Model based studies** Alameldeen et al. [2] develop an analytical model to study the interplay between cores and caches in a CMP. While they do attempt to incorporate data sharing in their throughput model; they use a fixed sharing pattern of 1.3 sharers per cache block. The investigation of data sharing is not central to their work, and their examination of it is cursory. Rogers et al. [82] develop a simple analytical model to highlight the bandwidth wall and ways to overcome it. While they too consider data sharing, we find that their sharing model is simplistic. This is understandable given that data sharing was not central to their work. We address the weaknesses of this and other prior data sharing models in Section 3.1. Wentzlaff et al. [94] develop a detailed analytical model to estimate chip-level performance. They also explore a large design space of cache hierarchies for multicore chips. They do not consider data sharing since their work is focused on multiprogrammed homogeneous workloads in the cloud computing context. We use their work to develop our baseline throughput model, and extend it by incorporating the effect of data sharing into it. Loh et al. [59] present a high-level analytical model that provides insights into how the uncore resources (caches and interconnect network) become the bottleneck to the continued scaling of the number of cores in future many-cores. They tackle data sharing indirectly by studying a set of canonical, sub-linear, trends that capture the required growth in the last level cache size in order to support core scaling. Our work is different in that we incorporate real measurements of data sharing and its impact on sharing from contemporary multithreaded workloads, rather than performing a limit study. They find that after scaling to a certain level, core sizes will have to increase to keep the uncore costs manageable, which is similar in flavor to one of our findings. Oh et al. [74] explore the optimal cache hierarchy and configuration for CMPs using an analytical performance model.
They do incorporate data sharing at the L3 level into their analytical model; however, they say that the actual value for the parameter should be determined by system characterization. For their evaluations they assume, rather aggressively, that of there are $N$ cores on the chip, $N/2$ cores share an average L3 cache block.

Simulation or Hardware measurement based studies Zhang et al. [99] study cache level data sharing in PARSEC benchmarks on real hardware, by using real multi-core processors from Intel and AMD (with up to 8 threads, and fixed cache hierarchies). They find that program code transformations are needed before sharing can impact performance. Our work is different in that we study sharing behavior across a wider range of cache configurations and thread counts, and contrary to their findings we find that data sharing can impact CMP design even with existing code. Wu et al. [95] use a simulator to perform a resource unconstrained study with 1 to 256 cores in a tiled CMP configuration. They use 6 parallel benchmarks (mostly from the SPLASH2 suite) for the study which evaluates how on-chip network load and off-chip bandwidth requirements scale with core counts and cache size. Our work focuses on more contemporary, non-scientific, workloads in an area-constrained many-core chip. Hsu et al. [37] evaluate the optimal number of threads that should share a cache at each level of the cache hierarchy in a large CMP. The workloads they consider are multiprogrammed, and have no data sharing; however their work highlights the challenges in large CMP design. This work also motivates the need for practical alternatives to highly detailed simulations of hundreds of execution threads. Zhao et al. [100] develop a constraint-aware analysis methodology that uses chip area and bandwidth as the two constraints, and prunes the design space for future Large-scale CMPs (LCMPs). They too assume no data sharing between threads in their workloads. Huh et al. [39] evaluate a small in-order core and a 3x larger out-of-order core for use in a CMP and predict that lack of bandwidth scaling will promote the use of larger core in future CMPs. We come to a similar conclusion about core sizes, although in the context of data sharing, which they do not consider.

Hardavellas et al. [32] and Esmailzadeh et al. [27] make the case that core scaling cannot continue with the lack of power and bandwidth scaling in the future. Large areas of the chip will have to be left unpowered, which makes the case for heterogeneity, specialization and workload migration. While these works do not directly address data sharing and its impact on cache and bandwidth demand, they highlight the other constraints that must be considered in the design of future many-core chips.
Chapter 4

Architecture and Performance of the Hardware Accelerators in IBM’s PowerEN Processor

This chapter is organized as follows: Section 4.1 provides a high-level overview of the PowerEN chip. Section 4.2 highlights the unique architectural features of PowerEN while describing the data and control flow across the chip resources in a typical usage scenario. The next few sections describe in some depth the functionality and performance of the on-chip hardware accelerators. Sections 4.3, 4.4, 4.5, 4.6 and 4.7 describe the architecture and performance of the Host Ethernet Adapter, the Pattern-Matching accelerator, the Cryptography accelerator, the Compression and Decompression accelerator, and the XML accelerator respectively.

4.1 Overview of PowerEN

The PowerEN processor was designed to meet the demands of next-generation networks. It integrates 16 general purpose, multi-threaded processor cores with an array of accelerators. The combination of many, multi-threaded cores with multi-tenant accelerators exploits the natural parallelism found in network processing workloads where data packets very often can be processed independently of each other. This processing is orchestrated by several novel control mechanisms built into the PowerEN hardware that enable software to track and guide packets as they flow through chip. The challenge is to complete the processing steps (“use case”) for each packet while maintaining the overall bandwidth goals for the chip. As is typical with network processing, the goal for PowerEN was to sustain 50% of the peak line rate while also being able to absorb short periods of 100% line rate.

The PowerEN architecture is accelerator friendly. It provides architectural and microar-
rchitectural features that let software easily and efficiently use the hardware accelerators. In this section, we first describe the microarchitectural components of the PowerEN chip and a fully connected multi-chip PowerEN system. We then describe the control and data flows in a typical usage scenario. Finally, we highlight the specific architectural extensions in PowerEN that enable software to communicate with and use the accelerators efficiently. Where possible, we will provide insights into how some of the design and structure sizing decisions were made.

Figure 4.1: PowerEN Chip Layout

Figure 4.1 shows the PowerEN die. The PowerEN chip consists of a collection of 16 general purpose cores, two dual-channel memory controllers and a collection of accelerators (Host Ethernet Adapter, Multi-Pattern Matching, Compression/Decompression, Cryptographic Data Mover and XML Processing).

The A2 general purpose cores were designed for throughput and power efficiency over and above single-thread performance. Applications in the target space have large amounts of thread-
level parallelism due to the highly parallel nature of the input network stream. Network traffic is typically composed of a large number of flows that can be processed independently. As will be described, the Ethernet interface automatically separates such flows to efficiently expose this inherent parallelism.

Prior evaluations found that 16 highly-threaded in-order cores with integrated I/O and hardware accelerators provided a 50% power reduction at constant performance compared to a similar chip with 8 out-of-order cores focused on single-thread performance [38]. These investigations formed the starting point for PowerEN. Quantitative analysis of proprietary target workloads built on this foundation, guiding numerous sizing decisions, such as number of cores, size of L2 cache, DRAM bandwidth, etc. The strategy was always to meet or exceed application throughput requirements with minimal power expenditure.

The resulting A2 core is 64-bit PowerPC Book-E architecture compliant [78]. The core supports in-order execution and dual issue over four hardware threads in a fine-grained simultaneous multi-threaded (SMT) fashion. The pipeline is 12 stages long with 27-FO4 per stage (FO4 is a standard delay metric used in CMOS technologies). Each A2 has a unified branch/integer/load-store execution pipe and a floating point execution pipe.

With 16 cores, a single PowerEN chip supports 64 live hardware threads allowing PowerEN to sustain high instruction throughput with simple, power-efficient pipelines. However, the cache and memory subsystem must be quite sophisticated to support concurrent execution of so many threads in a high-bandwidth application. It starts with 16KB 4-way set-associative L1-I caches and 16KB 8-way set-associative L1-D caches. PowerEN uses 64B cache lines throughout. Miss-under-miss within and between threads is enabled using an 8-entry load miss queue and a 4-entry instruction miss queue. The L1 caches are relatively small because the 4-way SMT efficiently covers L2 hit latencies.

Address translation is performed by fully associative L1 instruction and data TLBs backed by a 512-entry 4-way set-associative unified TLB, with hardware page table walker. All TLBs support multiple page sizes (4KB, 64KB, 1MB, 16MB and 1GB).

Four cores share a unified 2MB 8-way set-associative L2 cache. The L2 data array is built from embedded DRAM, saving area and power. An L2 cache shared by four cores amortizes the storage of shared data, particularly instructions, and reduces the power of coherency snoops, while keeping L2 latency and the complexity of the core/L2 inter-connect low. Each L2 cache is inclusive of the eight L1 caches above it, and thereby acts as a snoop-filter for them. The four on-chip L2 caches are kept coherent using a sophisticated snoopy protocol.

In similar fashion, accelerator design was guided by quantitative analysis of target workloads, in contrast to formal analysis [Zidenberg et al. 2012]. Workload characteristics drove the selection of the specific accelerators, accelerator functionality and accelerated bandwidth. PowerEN includes accelerators that perform functions common across multiple target workloads,
show significant power advantages over execution on the A2 core, and typically provide much higher throughput than general-purpose cores are capable of.

Figure 4.2: PowerEN Chip Overview

Figure 4.3: A Fully Connected 4-Chip PowerEN System

All components, including the accelerators (and multiple PowerEN chips in a multi-chip system), are interconnected via a fabric called PBus that provides hardware cache coherence and data transfer support. PBus splits transactions into address and data components, supports multiple unit-to-unit links and implements a snooping protocol to increase the interconnect bandwidth. Each unit on the PBus has interfaces and control logic to communicate on a
command bus, a reflected command bus, a partial response bus, a combined response bus and read and write data buses. Figure 4.2 illustrates these components. Four PowerEN chips can be connected together into a Symmetric Multi Processor complex in order to handle higher traffic data rates. Each chip to chip link supports 20GBps data movement bandwidth. Figure 4.3 shows such a fully configured system.

PowerEN employs aggressive power reduction and power management techniques [38, 46] including static voltage and frequency scaling, multi-voltage design, multi-clocking, extensive clock-gating, multiple threshold voltage devices, dynamic thermal control, eDRAM caches and circuit design to enable low voltage operation. Each chip’s main $V_{dd}$ voltage required for operation at different target frequencies is determined at test and stored in on-chip non-volatile memory, retrieved at power-on and used to program off-chip voltage regulators. All circuits that operate under the $V_{dd}$ Adaptive Power Supply domain are designed to operate at 0.7V or lower. There are multiple voltage domains which lets each circuit be supplied only the minimum voltage to operate correctly. The clock distribution minimizes skew, and the implementation supports multiple clock domains. Clock gating is extensively employed to reduce dynamic power. Devices with different threshold voltages reduce leakage power; overall, across the chip, about 75% of the devices are Super-high $V_t$, about 20% are High $V_t$, and about 5% are nominal. Temperature monitors feed an on-chip power limiter which can insert wait cycles for new operations on the PBus, thus reducing switching activity and power. These power management techniques reduce power consumption by over 50% and maintain a power density of less than 0.25W/mm$^2$. These techniques allow PowerEN to fit into different power envelopes ranging from 25W to 75W. When all the cores and accelerators on the chip are powered on, the projected power breakdown at 2.0GHz is that the cores and caches consume 40%, the accelerators consume 11% and the I/O and the PBus logic consume 19% of the power. All numbers reported in this work assume nominal frequency/voltage settings.

Table 4.1 summarizes PowerEN components, and their capabilities.

### 4.2 Architecture of PowerEN

#### 4.2.1 Example Use Case

Figure 4.4 shows a typical PowerEN use scenario. A packet stream coming in over the 4 10Gbit/s Ethernet ports is received by the Host Ethernet Adapter (HEA) (Step 1 in the figure). The HEA strips the packet of the appropriate headers from the lower-layers of the network stack, identifies the flow this packet is a part of, updates state it maintains to remember that this particular packet was seen and identifies the software thread that must be notified (Step 2). If this is a new packet, not belonging to any existing flow, the HEA identifies a new software
Table 4.1: Overview of PowerEN Capabilities

<table>
<thead>
<tr>
<th>Unit</th>
<th>Capability</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2 Processor Core</td>
<td>4 threads, scalar FP, dual-issue, 16KB I- and D-caches</td>
<td>2.3GHz</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>2MB shared by 4 cores, 8-way set associative</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>4x channels, 64b+ECC each</td>
<td>Up to 1600MHz DDR-3, 64GB memory, 50GB/s bandwidth</td>
</tr>
<tr>
<td>Cryptography</td>
<td>Encryption, Decryption, Authentication, Asymmetric Math, Asynchronous Data Mover, Random Number Generator</td>
<td>up to 50Gbit/s (symmetric) and 10M ops/sec (asymmetric math)</td>
</tr>
<tr>
<td>Compression and Decompression</td>
<td>LZ77 and Huffman coding/decoding</td>
<td>9Gbit/s (average across Canterbury Corpus suite)</td>
</tr>
<tr>
<td>Host Ethernet Adapter</td>
<td>Packet pre-classification, automatic packet queue management</td>
<td>4x 10Gbit/s ports</td>
</tr>
<tr>
<td>Pattern Matching Engine</td>
<td>Up to 1024 pattern contexts, 1000s of regular expressions per context, 8 simultaneous scans</td>
<td>73Gbit/s peak, 15-40Gbit/s typical.</td>
</tr>
<tr>
<td>XML</td>
<td>XML Parsing, XPATH evaluation, Schema validation and more</td>
<td>&gt;10 Gbit/s (avg. across 14 workloads studied)</td>
</tr>
</tbody>
</table>

thread to handle this flow. The HEA places the packet data in a free memory buffer from a pool of memory buffers that the thread has access to. The PowerEN architecture supports a technique called cache injection, which comes in handy at this point. The HEA can directly install the packet’s data into an L2 cache (Step 3), even if none of the corresponding cache blocks were resident in any of the L2s. This avoids having to write the buffer in physical memory first before it can be used by the software thread or other hardware accelerators. We discovered that it was important to support several memory buffer pools with different buffer sizes (128B, 256B, 512B, 2KB). Without such support, the packets of all sizes would be aligned to some multiple of a single large buffer size and the cache injection mechanism would be forced to map placement to a fraction of the cache sets. Without multiple buffer sizes, this cache thrashing behavior caused a 21% drop in HEA performance.

To support cache injection, the L2 cache must be augmented to accept lines that they have not first requested. When an L2 cache becomes overwhelmed with cache inject requests, it will deny the request, resulting in the line being sent to another L2 cache or to memory. Cache injection was implemented without adding any new cache coherence state and with little change to the overall protocol. Lines are injected in the exclusive and modified state. Because PowerEN supports cache-coherent I/O with or without injection, the protocol already shoots down other stale copies of the line.

Once sufficient data has been collected, the HEA also “wakes up” the software thread.
responsible for handling this data further. The HEA need not wait even for a single packet to be fully received before starting to send work to the accelerators. The latency of each packet remains a key performance metric and a worst case latency is stipulated as part of the performance requirements for each algorithm. This “wakeup” mechanism highlights another novel feature of PowerEN’s architecture. A software thread, once it is done processing all the packets it is responsible for currently, can excuse itself from being scheduled on to a hardware thread context. The software thread does this by first writing an address into a special hardware register, and then executing a new instruction - “wake-up on loss of reservation”. This removes the thread from scheduling consideration, and places a watch on the address posted into the special hardware register. A store command on the interconnect targeting the same address “wakes up” the thread (brings it back into scheduling contention). The HEA is pre-programmed to know the wake-up address in use by each software thread that is responsible for handling HEA’s flows. The HEA simply generates a store to the appropriate address on the interconnect to inform the thread that there is work to be done (Step 4). The value stored by the same store command tells the software thread what exactly needs to be done with the buffered data.

At this point, user-level software routines can set up the necessary control structures in memory in order to send work to the appropriate accelerator. Let us assume that the data needs to be decrypted first. The software thread sets up the necessary control data structures needed to communicate with the Cryptographic Data Mover (CDM), and sends the request to the CDM (Step 5). The control data structures and the command to wake up an accelerator are standardized across all accelerators, and are part of the PowerEN Architectural specification. There are 3 main control structures that the software must set up - the Coprocessor Request Block (CRB), the Coprocessor Parameter Block (CPB) and the Coprocessor Status Block (CSB). Among other things, the CRB contains a pointer to where the input data is, how much input data must be processed, a pointer to where the output data must be placed, a pointer to where the CPB is and a pointer to where the CSB is. The CPB contains necessary parameters to control how the input data is processed by the accelerator. For example, for a
simple decryption algorithm, the decryption key would be part of the CPB. The CSB contains place holders for the completion status and error messages - this is the structure that the accelerator would use to communicate back to the control plane thread, informing it whether the task finished successfully or whether there were errors. Note that all pointers used in these control data structures are specified as virtual addresses belonging to the address space of the process within whose context the thread is running. Each accelerator has a Memory Management Unit (MMU) in order to translate the virtual addresses to physical addresses before accessing them. This is a key design decision in PowerEN, and provides several advantages. First, it allows user-level software to set up the control structures. Second, it allows the accelerators to deal with page boundary crossings locally, without having to bother the control plane thread for the physical address corresponding to the next virtual page.

After setting up the control structures, the user level software issues a single, newly added, instruction - icswx. This instruction’s payload specifies two things - the target accelerator and the pointer to the CRB. The PowerEN hardware securely attaches 2 pieces of additional information into placeholder fields in the CRB - the Process ID and the Logical Partition ID. This allows the accelerator to work in the same virtual address space as the thread issuing the icswx. The icswx is completed when the interconnect delivers the CRB’s contents to the target accelerator.

The CDM unit processes the CRB by dereferencing the embedded pointers and pulling in the input data and the CPB from the memory subsystem. The accelerator’s data accesses go through the hardware TLB first, which translates the addresses from virtual to physical. And then the requests participate in the interconnect’s cache coherence protocol, ensuring correct operation regardless of the physical location of requested cache lines. In this case, the input data, as well as the CPB and the CSB are likely to be in L2 cache. The L2s in PowerEN are capable of directly supplying data to the accelerator via cache intervention (Step 6). That is, the data in the L2 cache does not need to be first evicted to physical memory before the accelerators get the data. This keeps the data on chip and keeps the communication fast.

The CDM starts processing the data (Step 7). The CDM writes the output data, as it is generated, to the appropriate location pointed to by the CRB (Step 8). Again, the cache injection mechanism can be employed to send the output data directly into an L2 cache, thereby keeping it on chip. The CDM also writes to specific fields in the CSB to indicate completion and any relevant status information (Step 9).

Software observes completion either by polling on the status bit in the CSB, or by employing the “wake up on loss of reservation” mechanism described earlier. Once the work requested by the CRB is completed, the memory associated with the CRB, CPB and CSB is returned (preferably to a pool of such data structures, for easy reuse).

In typical scenarios, decrypted data may be further processed by additional accelerators.
(decompression, pattern matching, XML parsing, etc.) using a similar handshake. Once all on chip processing is complete, the software thread hands the pointer to the output data to the HEA, which then sends the packet out and returns any memory buffers that free up, preferably to a free pool for easy reuse.

### 4.2.2 Accelerator-Friendly Architecture

The above example illustrates many of PowerEN’s architectural enhancements, which simplify accelerator invocation, reduce startup overheads and minimize data movement. In this subsection we recap the key innovations in PowerEN.

The PowerEN ISA defines a new instruction, called *Initiate Coprocessor Store Word* or `icswx` (pronounced ik-switch). The `icswx` instruction copies a 64B aligned co-processor request block (CRB) to the target accelerator. The CRB contains all the parameters needed to get the accelerator going - such as, the target algorithm engine inside the accelerator, input and output data addresses and pointers to the parameter (CPB) and status (CSB) blocks.

Process and Virtual Partition identification information is securely added to the CRB prior to transmission, allowing the accelerator to run in the same virtual address space as the invoking thread. In addition, each accelerator has an associated MMU, including a TLB and hardware translation page table walker. The accelerator is thus able to perform its own address translation while concurrently processing requests from different processes on different virtualized operating systems, with minimal software involvement.

Each accelerator is directly attached to the coherent interconnect (called *PBus*) and participates in the hardware coherence protocol. Accelerators are, therefore, able to read or write all the data they need without special software support such as pinning virtual pages or DMAing data in to and out of the accelerator.

Each accelerator has a hardware queue to hold a handful of CRBs. Should this queue overflow, further CRBs will be rejected by the accelerator. The rejection is reported back to software through a condition code set by the `icswx` instruction, placed in the same condition registers set by normal compare instructions in the PowerPC architecture.

To minimize this occurrence, the PowerEN accelerators can extend the hardware queue into main memory. The *spill/fill feature* spills excess CRBs to a queue in memory. As CRBs are completed and space frees up in the hardware request queue, CRBs are automatically transferred to the hardware queue from the queue in memory. Other than initial configuration of the spill/fill queues, software is uninvolved in this process. While this can increase parallelism, it does increase memory and bus bandwidth, so some applications may not find it beneficial. The test results reported in the following sections do not use the spill/fill feature.

Since all accelerator activity involves data movement, managing data movement within the
system is a major focus of the PowerEN bus architecture. Besides traditional optimizations, such as prefetching and cache line locking, PowerEN provides a number of accelerator-friendly optimizations.

Data produced by an accelerator, and data ingressed onto the chip from the HEA, is not sent out to DRAM. Rather, accelerators can be configured to place generated data in a specific L2 cache, or spread it uniformly across all L2 caches. This dramatically reduces memory bandwidth requirements compared to traditional models. In a traditional system, every hop to and from every accelerator would require writing and reading the data to and from memory. In PowerEN, all data stays on the chip.

Furthermore, accelerators can update data in L2 caches in-place. This is useful, for instance, when accelerators report completion. Accelerators typically report completion by writing a status bit in the Coprocessor Status Block. The status bit is checked by software using normal loads. Rather than forcing the line into the DRAM or even into some other L2 cache, the status bit is simply updated in-place, in the L2 cache where it resides. Typically this is the L2 cache associated with the thread that submitted the request. And since typically the same thread also checks for completion, updating the CSB in-place ensures that the status ends up exactly where it is needed, rather than in some distant physical location.

To make polling on completion status even more efficient, PowerEN provides the wait-on-reservation instruction. Its operation was described in some detail in Section 4.2.1. Wait-on-reservation deschedules a thread until a store to a particular thread-specific address is detected by the interconnect. This mechanism enables an efficient alternative to polling by disallowing threads that are only waiting for an accelerator to finish up from using compute resources. This allows other threads doing active computation from utilizing the core’s compute resources. The wait-on-reservation instruction may also be used to implement inter-thread communication structures, such as locks.

4.2.3 Accelerator Software Architecture

Historically, software communicated with devices via a kernel-level device driver. The driver performed two basic functions - security checks and device abstraction. In PowerEN, security checks have moved into the hardware. Accelerators access memory using the same virtual addresses as the invoking process, and the same page-level protections apply. Device abstraction allows applications to be written to a higher-level standard API, while giving hardware vendors the freedom to innovate. In PowerEN, device abstraction occurs through the user-level libraries typically used to invoke the accelerator.

PowerEN’s Software Development Kit (SDK) provides standard libraries supporting encryption/decryption (OpenSSL, OpenVPN) and compression/decompression (zlib). Multi-pattern
search is a developing area, and standard APIs for this function are not mature. PowerEN provides a custom interface for this function. The interface to program the Host Ethernet Adapter is also based on a custom-developed user-level library code.

Finally, the libcopl library provides an efficient low-level interface to the accelerators. This allows application developers to port PowerEN function to their own internal APIs efficiently. A uniform architecture simplifies the software interfaces to heterogeneous accelerators, allowing the application layer to take full advantage of their capabilities.

4.3 Host Ethernet Adapter

The HEA unit provides network connectivity through four Ethernet ports, each configurable as either 1Gbit/s port or a 10Gbit/s port. The HEA provides a number of important packet acceleration functions, including:

- Programmable packet pre-classification
- Check-sum validation (IP header, TCP, UDP)
- Automatic packet scattering/gathering over 128 send or receive queues
- Hardware-assisted packet queue management

The HEA communicates with software through queue pairs, each of which contains a receive queue and a send queue. Each logical receive-queue can contain up to three internal queues to efficiently handle different packet sizes.

When the HEA receives a packet from the network, it selects a queue pair based on a hash of the 5-tuple.1 This spreads work across many threads, such that they can work largely independently; yet it still ensures that inter-packet dependencies are enforced. Within the selected logical receive-queue, the HEA places the packet on the correct internal receive-queue based on the packet size.

To transmit a packet, software places the packet on a send-queue, and notifies the HEA. For large packets, the packet buffer used for reception can also be used for transmission, accelerating applications such as security, which tend to retransmit packets with few modifications.

4.3.1 Performance

Figure 4.5 illustrates HEA throughput with minimum software overhead. An Ixia Ethernet test system [44] was used to drive or receive packets in to or out of PowerEN in order to measure peak packet throughputs. All packets are IPv4 packets (IPv6 is also supported), and minimal packet classification is done. For transmit tests throughput is measured using Ixia’s tracing capability to measure the elapsed time and total number of bytes received on Ixia. For receive

---

1 i.e. protocol, source address and port, destination address and port.
tests the HEA timestamp associated with each frame was used to compute the elapsed time and the number of bytes received.

![Graph showing HEA transmit and receive bandwidth](image)

Figure 4.5: HEA transmit and receive bandwidth

For small packet sizes, HEA sustains 12Gbit/s to 28Gbit/s in each direction. However, for packets of 320B and larger, HEA supports nearly 100% utilization on all four Ethernet ports simultaneously. Since typical Internet packet mixes have an average packet size of around 384B, the HEA will support the full port bandwidth for realistic traffic patterns.

### 4.4 Pattern Matching Engine

The pattern matching engine (PME) scans input data for large numbers of regular expressions. Typical applications group tens to thousands of expressions into pattern contexts, and may contain hundreds of such pattern contexts. Given an input byte stream, the PME reports all match locations for all patterns in a particular context. Theoretical peak throughput is 73.6 Gbit/s. Realistic workloads typically obtain 15 Gbit/s to 45 Gbit/s.

Supported regular expression constructs include:
- Bounded/unbounded repeat operators (\{\ldots\}, *, +)
- Wild-cards, character classes, case-insensitivity (\[\], .)
- Alternatives (\|)
- Start-anchored, end-anchored, unanchored patterns (\^, $)

The only hardware feature specific to ASCII encoding is case-insensitivity. The PME is therefore suitable for scanning any byte-oriented data stream.
4.4.1 Software Interface

Figure 4.6 shows the PME software stack. Pattern contexts are compiled into an internal form understood by the hardware.

Compiled contexts are uploaded to the PME via the PME *Upload Manager* (UM). Application software then submits searches. During the performance critical search function applications interact directly with the PME engine through icswx, embedded in the PME search library. Match reports and completion status are produced by the PME into normal user memory, and processed by the application using normal loads and stores.

Network traffic can be scanned one packet at a time, without first reassembling traffic streams. To accomplish this the PME saves scan state at the end of one data buffer, and then restores that state before the next packet is scanned. The results produced are identical to scanning both packets at once. State is stored in normal user memory, so the number of active network flows is bounded only by system memory size.

![Figure 4.6: PME Software Stack](image)

This capability creates dependencies between successive scans. These hazards are checked and enforced by the PME. Software can submit multiple dependent scans in parallel, and the PME will ensure they are performed in the correct order.

4.4.2 Accelerator Design

The PME follows a RISC design philosophy. Simple hardware performs a minimal number of critical functions as fast as possible. An intelligent compiler makes efficient use of hardware primitives.

Like many modern multi-pattern-search engines [21, 55, 76, 84, 88], the PME contains both
NFA and DFA features (Figure 4.7). The core engine, the \textit{B-FSM} \cite{93} is a DFA state walking engine. Four of these are combined together into a \textit{physical lane}. The compiler will optimally distribute patterns across the B-FSMs to prevent the state-explosion that DFA-based algorithms are prone to.

State explosion is further avoided via the \textit{local-result processor} (LRP). While the B-FSMs can identify many patterns directly, some regular expression constructs, such as the common .\* construct, cannot be encoded efficiently on the B-FSM. Patterns are broken into sub-expressions at such constructs, and the B-FSMs search for these sub-expressions. When a sub-expression is found, the B-FSM forwards an instruction to the LRP, which determines when all the sub-expressions have been identified in the correct order. The LRP then reports the pattern match.

When a context becomes too large to fit efficiently on four B-FSMs and/or one LRP, the compiler will break the context into multiple sub-contexts. Similar to the dispersal across the B-FSMs, the compiler intelligently distributes patterns to minimize pressure on the B-FSMs and LRPs.

This dramatically reduces the size of the resulting compiled context, but does come at a cost. To the hardware, these sub-contexts look like independent contexts that happen to scan the same data. The PME search library must submit a separate scan for each sub-context in the logical context. To improve the performance of large contexts, a single PME icwsx command can invoke two scans.

Four independent physical lanes are instantiated on the PME. Furthermore, each physical lane is time-multiplexed between two virtual lanes, each of which can run an independent search. Thus the PME can work on eight scans simultaneously from the same or different software threads or processes. Each of the eight virtual lanes can scan one byte every other cycle at 2.3GHz.
The PME contains 512KB of SRAM cache for compiled pattern rules, organized as 32KB of cache per B-FSM. A B-FSM can read the rules required to process a single character in a single cycle, as long as the rules are in its local cache. When a rule miss occurs, the rule is fetched from main memory. While the rule is being fetched, the scan is disengaged from the virtual lane, which may be assigned to a different scan, reducing the effective miss penalty.

If a context is frequently used, it may benefit from more bandwidth than that provided by a single physical lane. In this case, the upload manager can replicate the context across multiple physical lanes. This requires replicating the rules in the B-FSM caches, thereby reducing the effective cache capacity. The UM seeks to maximize throughput, balancing the cost of increased rule miss rate, vs. improved load balancing across the physical lanes. The search application is uninvolved in this optimization. Replication is done asynchronously and transparently by the UM, based on hardware-assisted profiling of context and rule usage.

4.4.3 Performance

Figures 4.8 and 4.9 illustrate the performance of the PME for a wide range of synthetic workloads. Multi-pattern search workloads containing large numbers of complex patterns are not publicly available at this time. By using synthetic workloads, we can vary key parameters, and make the set publicly available [41].

Performance is measured using multi-threaded user-level Linux applications that leverage the libcopl library. The tests stream scans from 32 threads, an excessive number designed to ensure the PME is saturated. We measure the steady-state bandwidth on the processor threads using the processor’s timebase register over about 20K scans. These measurements are verified using performance counters on the PME engine. The average of three measurements is reported.
The figures show the performance in Gbit/s on the y-axis and total number of patterns on the x-axis. The number of patterns are increased either by adding more patterns to a single context (Single Context), or by increasing the number of different fixed-sized contexts (Multiple Contexts). In the later case, each context contains 1000 patterns.

The simple string patterns (Figure 4.8) consist of patterns made up of a uniform distributions of upper and lower case letters and digits. Each pattern is 10 characters long. All are case sensitive. None are anchored.

![Figure 4.9: PME Performance: Complex patterns](image)

<table>
<thead>
<tr>
<th>Example Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6zfe4t0ikumP</td>
<td>Simple string pattern</td>
</tr>
<tr>
<td>cbPud(WT)?</td>
<td>Extended string pattern with optional element</td>
</tr>
<tr>
<td>^jVCE(Wj/sq{AT}uX{\n\r}*3FFh3GAOz</td>
<td>Two string elements with separator, front-anchored</td>
</tr>
<tr>
<td>E[a-z]K8qDFj.*ebgyHH.*91a3UbnpDhk4tg{A-Z}</td>
<td>Three string elements with separators</td>
</tr>
</tbody>
</table>

Table 4.2: Example patterns from the complex pattern sets

The complex pattern sets (Figure 4.9) consist of a mix of simple strings, extended string-like elements and separator elements. The extended string-like elements contain broad character classes ([a-z], [A-Z], [0-9]), alternatives ( (abc|xyz|123) ) and optional components ( abc(123)? ). They represent 35% of all string elements. In addition, 20% of patterns contain multiple such string-like elements separated by wildcards and repeat constructs ( , , , *, [““]*, [\“\n\r]*, [\“\n\r]* , ... ). Of the patterns, 50% are case insensitive, and 20% are front-anchored (^). Table 4.2 shows several example patterns from the complex pattern sets.

The input data is also an important factor affecting pattern matching performance. Most
of the input characters are chosen from a random uniform distribution over letters and digits. However, random data does not stress the B-FSMs realistically because they will not tend to traverse a wide variety of DFA states. By interspersing byte sequences that match the initial pieces of the patterns, the B-FSMs are driven to explore a much wider variety of states, stressing the rule cache hierarchy. The distance between such sequences is an exponential distribution with a mean of 100 characters. The length of matching sequences is exponential with a mean of 2. The target pattern for the pattern-based sequence is random uniform over all patterns in the pattern context. All scans are 1000B long.

Where multiple contexts are used, searches are distributed in a uniform random distribution over the contexts. This is a worst case situation. Applications with multiple contexts commonly use a few contexts frequently, and many rarely.

For a single context of simple patterns (Figure 4.8), performance is a constant 45 Gbit/s up to 600 patterns. In this range, there are no rule misses. Beyond this performance starts to fall due to increasing miss rate. At 1000 patterns (36 Gbit/s) a single logical lane is full. After this two logical lanes are used\(^2\), reducing the miss rate, but at the cost of more physical scans. The trend continues resulting in a gentle decline in performance as patterns are added, culminating with 3500 patterns on four logical lanes at 17 Gbit/s. At this point the context fills four sub-contexts, which is the compiler’s self-imposed limit on context size.

With multiple contexts, performance remains constant at about 33 Gbit/s out to 8000 patterns, even though the number of rules increases as contexts are added. This is due to physical lane replication described previously. With a single context, the UM replicates that context on all four physical lanes for maximum throughput. Essentially four contexts are being cached. With two contexts, replication is reduced to two copies, again yielding four cached contexts and leaving the effective cache pressure unchanged, even though the number of live rules have doubled. Four contexts are similar. Each physical lane can comfortably hold 2 large, heavily used contexts. Hence, 8 contexts result in similar performance.

The complex pattern sets show similar trends, but put more pressure on the LRP and B-FSM rule cache hierarchy (Figure 4.9). Approximately 500 patterns, rather than 1000, fit in a logical lane. Performance is flat (51 Gbit/s) up to 200 patterns. Every 500 patterns, another logical lane is added, trading off increasing miss rate for additional scans. Performance degrades reasonably to 16 Gbit/s with 2000 patterns on four logical lanes. Note that the 1000 pattern point is relatively low, due apparently to random variation in the synthetic pattern set.

The multi-context results are relatively flat up to 4000 patterns (4 contexts). Replication again plays a key role in this region. Each context with 1000 patterns is compiled into two sub-contexts. Similar to the multi-context simple string results, the upload manager puts four

\(^2\)The proto-type compiler logical lane selection heuristics are still being tuned. The number of logical lanes was selected using command-line switches.
copies of each sub-context on each of the four physical lanes, physically caching eight total sub-contexts. With two contexts, the upload manager switches to replicating each context twice. In fact, with two logical contexts, four total sub-contexts, the UM has some flexibility to decide which sub-contexts are paired on the physical lanes, leading to a lower miss rate and better performance (20 Gbit/s, up from 15 Gbit/s).

There is a dip again with three contexts, because six sub-contexts don’t pack cleanly on four physical lanes. Two contexts are unreplicated, and one is replicated twice. As with two contexts, performance increases again with four contexts. Each context is unreplicated, and the eight sub-contexts are packed on the four physical lanes.

Performance is therefore relatively flat, varying between 15 Gbit/s and 20 Gbit/s from one to four contexts. After this point, the miss rate increases, and performance slopes off as shown.

In summary, the PME is capable of scanning data at 15 to 50 Gbit/s on multiple contexts containing as many as 8000 patterns. A RISC design philosophy coupled with tight integration produced a flexible high-performance pattern matching accelerator capable of handling a rich regular expression language at wire-speed.

4.5 Cryptographic Data Mover Accelerator

Encryption, decryption, and authentication of streaming data at the edge of a datacenter can substantially free up general purpose compute resources inside the datacenter. The Cryptographic Data Mover (CDM) is capable of handling a wide range of cryptographic algorithms, often at wire speeds, including combinations of two operations (encryption and authentication) in a single command.

In addition to accelerating cryptography the CDM also supports block data moves (generic DMA) at over 60 Gbit/s, and random number generation at 12K 64-bit numbers per second.

4.5.1 Accelerator Design

The CDM is composed of eight channels. Each channel contains one or more algorithm engines, and a DMA engine to stream data in and out of the algorithm engines. Only one algorithm engine is active within a channel at a time. Each algorithm engine is capable of handling one or a small subset of the supported algorithms. Algorithm engines are replicated across the channels to meet throughput targets for each algorithm.

The CRB indicates which algorithm is required. For a given algorithm, there is a fixed subset of applicable channels – those containing the appropriate algorithm engine. The CDM will select the next available, applicable channel to handle the CRB request.

The channel works on the incoming data in a pipelined fashion. The DMA unit in the channel starts fetching the packet data. The algorithm engine will begin work as soon as
a sufficient minimum amount of data has arrived. The output data is DMAed back to the
destination address, as pointed to by the CRB, as it is generated.

4.5.2 Performance

![Figure 4.10: Best-case throughputs for individual, symmetric, crypto algorithms. x-axis is payload size in Bytes.](image1)

![Figure 4.11: Best-case throughputs for symmetric, crypto algorithms including combined encryption/authentication algorithms. x-axis is payload size in Bytes.](image2)

The performance tests are written as user-level code under Linux, leveraging the libcopl library provided with the PowerEN Software Development Kit. Each test submits requests of a particular size for one particular cryptographic algorithm. The requests are submitted from a programmable number of software threads, spaced uniformly across the 64 available hardware threads. We vary both the packet size and the number of software threads to investigate how
these parameters affect throughput. We only show the encryption rates; the rate of encryption and decryption are equal for these cryptographic algorithms.

The steady state icswx acceptance rate directly correlates with the steady state icswx completion rate, and this is used to measure throughput. We validate this measurement using both hardware performance counters on the CDM, and the processor timebase register.

We first enumerate the absolute throughput achieved for each algorithm tested. We then discuss the relative advantage of acceleration compared to running code on the general purpose cores.

**Accelerator Throughputs**

Figure 4.10 and Figure 4.11 show the absolute throughputs measured for the individual symmetric encryption and authentication (message digest or hashing) algorithms, including combined operations (last six workloads in Figure 4.11). The results do not show all possible sub-functions within each algorithm. For example, AES supports a handful of different flavors. We just consider one (ECB) with a 128-bit key size. This was a conscious decision to help focus on insights, while gaining a reasonably good feel for the performance ranges. Each stacked bar indicates the throughput that can be reached by increasing the number of threads participating in sending work to the accelerator. For each algorithm the figure also shows the throughput improvements as the size of the payload per CRB increases. As can be seen, often, only 1 to 4 threads (out of the 64 on a chip) are needed to keep the accelerator well fed with work - only AES and DES need 8 or 16 threads, and that too only for payloads which are 128B or smaller. As the payload size increases the number of threads needed decreases; fewer threads are able to keep the accelerator busy if each request demands more work. This allows the other threads to be used for other useful work.

The combined algorithms shown in Figure 4.11 combine encryption and authentication operations on input data into a single command. It is also possible to perform the same two operations individually using two separate CRBs for a given payload - we call this the 2-call approach (not shown in the figure), as opposed to the baseline 1-call approach (shown in the figure). Intuitively, one would expect the 2-call approach to be worse both in terms of latency and throughput. However, we found that there is a subtle performance advantage to this approach in some specific scenarios. The baseline 1-call combined operations must be directed to channels which support both the encryption and authentication algorithm requested. This can disqualify some of the channels which do not contain both the necessary engines. For example, TDES is supported by all 8 channels, while SHA is supported by 6. This restricts the 1-call TDES,HSHA* operations to only 6 of the 8 channels. A 2-call implementation enables engaging all 8 TDES channels, and, indirectly, frees up the 6 SHA channels by a small amount enabling them to perform more of the SHA authentication. This insight can be used by software
developers to extract more performance from the PowerEN’s cryptology hardware.

Figure 4.12: Best-case throughputs for asymmetric m* crypto algorithms. x-axis is operand size.

Figure 4.13: Best-case throughputs for asymmetric m2*, p* and p2* crypto algorithms. x-axis is operand size.

The CDM supports a wide range of asymmetric mathematical functions, used to accelerate asymmetric cryptographic algorithms including Rivest-Shamir-Aldeman (RSA) and Elliptic Curve Cryptography (ECC) [22, 23, 92]. The asymmetric operations are basically modular arithmetic operations - addition, subtraction, multiplication, inversion, reduction, exponentiation and exponentiation based on the Chinese Remainder Theorem (CRT) - done on operands belonging to different Galois Fields (GF). In Figure 4.12 operations labeled M* operate on operands in GF(2^m). In Figure 4.13 operations labeled M2* operate on ECC operands over GF(2^m); operations labeled P* operate on operands in GF(P), and operations labeled P2*
operate on ECC operands over GF($P$).

Unlike the symmetric algorithms, these operations are not done over variable-sized input data. Instead the operations are mathematical functions performed over input operands of well-defined sizes. Performance for these operations is therefore measured in operations per second, rather than in bits per second.

Figure 4.14: Performance advantage of acceleration

Figure 4.15: Performance advantage of acceleration when engaging 8 threads
Acceleration Advantage

We used an internal software tool called **sslbench** to measure performance with and without acceleration. Sslbench uses the OpenSSL library to submit work. This makes it easy to compare accelerator and software performance, by either linking against the PowerEN-specific version of OpenSSL (with accelerator support), or, the generic software-based version of OpenSSL. This allows us to measure the latency of encrypting a large, 32KB payload with and without acceleration for a number of algorithms.

Figure 4.14 and Figure 4.15 summarize our findings. Figure 4.14 shows the performance of the tests that offload work to the accelerator, as more and more software threads are employed. The results are normalized to the best performance we could get using the software-only approach; in most cases the software-only approach required employing all 64 threads available on chip for best performance. The main takeaway from this figure is that for most algorithms acceleration can improve performance by nearly an order of magnitude, while engaging many fewer A2 threads than the purely software option would require. Indirectly, this also indicates a potentially significant power advantage by freeing up the general purpose cores. That said, we found one algorithm, rc4 (and its variant, rc4_40) for which by using sufficiently large number of threads the software-only approach does better than acceleration.

A different way to compare the performance with and without acceleration is to employ a fixed number of software threads in both cases. Figure 4.15 shows such a comparison with 8 A2 threads engaged. In one case the 8 threads run the cryptographic algorithm on the A2 cores; in the other, the same 8 threads are used only to set up CRBs and offload work to the accelerator. The performance advantage of acceleration shown in this figure is likely conservative. Fewer than 8 threads may be able to fully engage the accelerators, where as software performance will degrade proportionally with fewer threads.

4.6 Compression Accelerator

Data compression and decompression are extremely important for highly streamlined datacenter applications, often utilized in both high traffic networking and disk management to yield higher efficiency by shrinking data footprints. The nature of these algorithms lend themselves to hardware acceleration, and several standalone compression/decompression hardware engines have emerged to offload these operations [1, 69].

The PowerEN compression/decompression accelerator supports up to 8 Gbit/s of simultaneous compression and decompression bandwidth, using the RFC 1951 [72] compliant DEFLATE/INFLATE algorithms, based on LZ77 and Huffman coding. As they share the same base algorithm functionality, the engine is also capable of handling zlib (RFC1950 [70]) and
gzip (RFC1952 [71]) stream operations as well.

The accelerator can work on multiple compress/decompress commands at once from the same or multiple threads from the same or multiple processes. Furthermore, a single data stream can be submitted in separate pieces. Compressed packet streams do not need to be reassembled prior to decompression, nor be completed prior to compression and transmission.

Performance being the key component of the architecture, memory sizings for the accelerator were very important. For decompression, both static and dynamic Huffman decoding are supported using a 32KB history table. For compression, the engine emphasizes speed, latency, and die area over maximum compression ratios – only static Huffman encoding is supported.

The 32KB history table was sized according to a number of factors, including throughput performance, power, and physical chip size. LZ77 frequently employs a sliding history window (table) of anywhere between 2KB and 32KB; a larger history table allows compression to build a much richer comparison when deflating data, and allows decompression a much longer window in which to look back when inflating data. The bigger the window, the better the potential performance. The engine also contains a 4KB history cache to cache recently used history during decompression. In the time during which missing history is fetched from memory the engine does not stop. Instead, a “hole” is created in the output stream, which is filled once the missing history returns from memory. The size of the history cache was chosen as a direct result of simulated benchmarking done on the Canterbury Corpus benchmark suite.

A combination of both functional and performance-based software models written in C++/CSIM were used to ascertain hit/miss rates and relative compression/decompression history utilization based on the accelerator engine architecture. Numerous sizes were run through the models for both the table and cache, and it was ascertained that these sizings not only fit well within the physical accelerator space, but most importantly hit the mark on performance in terms of minimizing stalls during decompression.

4.6.1 Performance

Performance is gauged using the Canterbury Corpus benchmark set [64]. The benchmark set contains 11 files chosen specifically to stress various aspects of compression and decompression, and to deliver representative results. The testcases are single threaded. All reported results are averaged over 20 runs. An enhanced version of the zlib library capable of utilizing the accelerator was used. All measurements of time rely on the processor’s timebase register. Table 4.3 lists each file’s size before and after compression with gzip -9, as well as the compression ratio. The compression ratio ranges from 2.43 (xargs.1) all the way up to 9.83 (ptt5), with a harmonic mean of 2.26\(^3\). Results are sorted by this compression ratio throughout this section.

\(^3\)The harmonic mean corresponds to an average input made up of equal parts of each component, and is used throughout this section.
Table 4.3: Workload file sizes and compression ratios

<table>
<thead>
<tr>
<th>File</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>xargs.1</td>
<td>4227</td>
<td>1736</td>
<td>2.43</td>
</tr>
<tr>
<td>plrabn12.txt</td>
<td>481861</td>
<td>194332</td>
<td>2.48</td>
</tr>
<tr>
<td>asyoulik.txt</td>
<td>125179</td>
<td>48778</td>
<td>2.57</td>
</tr>
<tr>
<td>alice29.txt</td>
<td>152089</td>
<td>54170</td>
<td>2.81</td>
</tr>
<tr>
<td>lcet10.txt</td>
<td>426754</td>
<td>144439</td>
<td>2.95</td>
</tr>
<tr>
<td>sum</td>
<td>38240</td>
<td>12838</td>
<td>2.98</td>
</tr>
<tr>
<td>grammar.lsp</td>
<td>3721</td>
<td>1222</td>
<td>3.05</td>
</tr>
<tr>
<td>cp.html</td>
<td>24603</td>
<td>7940</td>
<td>3.10</td>
</tr>
<tr>
<td>fields.c</td>
<td>11150</td>
<td>3115</td>
<td>3.58</td>
</tr>
<tr>
<td>kennedy.xls</td>
<td>1029744</td>
<td>207029</td>
<td>4.97</td>
</tr>
<tr>
<td>ptt5</td>
<td>513216</td>
<td>52221</td>
<td>9.83</td>
</tr>
<tr>
<td>HMean</td>
<td></td>
<td></td>
<td>3.17</td>
</tr>
</tbody>
</table>

**Throughput**

Figure 4.16 illustrates the raw bandwidth (measured in terms of the uncompressed data stream) of the accelerator for each file in the Canterbury Corpus benchmark set. The average reported compression bandwidth is 9.1 Gbit/s, and given the straightforward static Huffman coding, is fairly constant.

Decompression has an average bandwidth of 8.9 Gbit/s, but varies by almost a factor of two from 6.7 Gbit/s (asyoulik.txt) all the way up to 12.8 Gbit/s (ptt5). The general upward trend across the graph demonstrates that high compression ratios lead to better decompression rates, as fewer history buffer reads are required.

While a full end to end application that simultaneously engages multiple accelerator engines was beyond the scope of this work, a subset of such a test was attempted involving the HEA, CDM and the compression unit. This corresponded to a case where a packet stream received over the HEA is first encrypted and then compressed before being retransmitted out. The overall throughput was, as expected, limited by the lowest performing individual accelerator in that case, which happened to be the compression unit. This provides a glimpse at how PowerEN’s efficient data movement avoids creating additional performance bottlenecks when multiple accelerators are chained together in a pipelined fashion.

**Comparison to Software**

Accelerator and software performance can be directly compared using the zlib library on the PowerEN processor core. Figure 4.17 shows the difference in compression ratios between the software and hardware implementations. As expected, software with dynamic Huffman encoding and variable levels of compression, achieves higher compression ratios than the accelerator. On average, the accelerator produces compressed files 22% larger than zlib -0, and 42% larger
than zlib -9. However, the resulting increase in throughput is enormous.

Table 4.4 shows the speedup of the accelerator over zlib. Decompression averages 22 times faster. The ptt5 input, with its high compression ratio, does relatively well with software (only 14x slower). Other than this, there is little trend to the data.

Compression is even more impressive. Compared to zlib -0, the accelerator is more than 100 times faster. With level-9 compression, zlib is more variable in performance, resulting in a wide range of speedups from 169x (grammar.lsp) to 7842x (kennedy.xls), with a harmonic mean of 464x. The mean drops only slightly to 424x if the high outlier (kennedy.xls) is removed.

In summary, compared to a general purpose processor core, the accelerator provides a 20x performance improvement on decompression. On compression, the accelerator yields hundreds to thousands of times more bandwidth. Offloading compression/decompression leaves processor cores available for more suitable tasks, and enables wire-speed processing rates.
Table 4.4: Throughput Ratios HW vs. SW

<table>
<thead>
<tr>
<th>File</th>
<th>Decompression</th>
<th>Compression zlib -0</th>
<th>Compression zlib -9</th>
</tr>
</thead>
<tbody>
<tr>
<td>xargs1</td>
<td>30.9</td>
<td>116.3</td>
<td>345.6</td>
</tr>
<tr>
<td>plrabn12.txt</td>
<td>24.2</td>
<td>131.9</td>
<td>919.0</td>
</tr>
<tr>
<td>asyoulik.txt</td>
<td>23.6</td>
<td>106.9</td>
<td>560.4</td>
</tr>
<tr>
<td>alice29.txt</td>
<td>22.5</td>
<td>121.4</td>
<td>722.1</td>
</tr>
<tr>
<td>lce10.txt</td>
<td>22.1</td>
<td>129.9</td>
<td>650.3</td>
</tr>
<tr>
<td>sum</td>
<td>22.3</td>
<td>89.3</td>
<td>2416.0</td>
</tr>
<tr>
<td>grammar.lsp</td>
<td>30.7</td>
<td>115.1</td>
<td>169.4</td>
</tr>
<tr>
<td>cp.html</td>
<td>21.3</td>
<td>90.7</td>
<td>249.3</td>
</tr>
<tr>
<td>fields.c</td>
<td>21.7</td>
<td>99.0</td>
<td>244.5</td>
</tr>
<tr>
<td>kennedy.xls</td>
<td>19.8</td>
<td>137.7</td>
<td>7842.7</td>
</tr>
<tr>
<td>ptt5</td>
<td>14.2</td>
<td>105.4</td>
<td>2183.7</td>
</tr>
<tr>
<td>HMean</td>
<td><strong>22.1</strong></td>
<td><strong>110.9</strong></td>
<td><strong>464.3</strong></td>
</tr>
</tbody>
</table>

4.7 XML Accelerator

4.7.1 Architecture

The XML accelerator parses XML documents, and can, optionally, post-process the results in different ways. Post-processing can include schema validation, Xpath matching and filtering, and other streamable XML processing options. The accelerator receives documents to parse in chunks (of up to 64KB) and generates output incrementally, as a collection of 4KB buffers, as the document is parsed. There are two main output formats - TLA (Type-Length-Attribute) format, which can be incrementally consumed by the software application, and the Tree format, which is intended to be used by the software once the entire document is processed. The size of the output is highly variable depending on the output format requested, the post-processing being performed, and the characteristics of the document itself. A 64KB chunk of input can generate anywhere from 1 to 32 4KB output buffers.

The accelerator can parse multiple documents simultaneously - and both the input document chunks and output data buffers can be interleaved. The accelerators can handle a large number of simultaneously live sessions, each of which is associated with one document and is identified by a unique session ID. Software communicates with the hardware accelerator by setting up a Coprocessor Request Buffer (CRB) structure in memory and using the icswx command. The first CRB associated with the processing of a new document requests that the accelerator open a new session. From that point on, all input data transfers from the software to the accelerator and output data or responses from the accelerator to software are tagged with the unique session ID. The session ID can be reused once the processing of the document currently associated with it completes.

The accelerator also contains a post-processing engine (PPE) that may be optionally engaged to perform in-line post-processing on the parsed document chunks. The PPE can sup-
press or selectively include parts of the result document in the final output, switch output formats mid-stream, or insert tags that annotate specific items in the result. The PPE is a programmable microcontroller with its own internal Instruction Set Architecture (ISA) and can execute sophisticated post-processing operations on the parsed output. If the PPE is to be used for a document, the CRB that opens a new session must indicate this and also point to the pre-compiled PPE program in memory to be used in the post-processing step.

The accelerator can be used by up to four processes at a time, be they across different Virtual Machines or within a single Operating System. The accelerator can handle 4 separate virtual-to-physical address mappings.

Internally, the accelerator implements 4 accelerator engine pipelines that can parse, post-process and generate output buffers in parallel. There is some function that is shared by these pipelines - address translation, data ingress and egress, request queuing etc. Load balancing is achieved by statically and predictably associating any given session ID to only one of the parallel pipelines.

4.7.2 Performance

We measured performance of the XML accelerator using 14 realistic workloads. Each workload repeatedly processes a unique, real-life, XML document. The testcases that exercise the accelerators were written as multi-threaded (1 to 4 threads) C programs running under Linux. The library used to communicate with the XML accelerator was a kernel-level library, although a corresponding, lightweight, user-level library is currently under development. Time is measured using the processor timebase register. These results illustrate the range of performance numbers that may be expected, although the actual performance depends on the characteristics of the document itself.

Figure 4.18 shows the throughputs of processing the workloads studied. The x-axis shows the size of each XML file and the number of TLAs in it. The first 14 pairs of bars correspond to parsing the input XML file in the XML Unit’s Parse engine. The last 3 pairs of bars correspond to the case where the input XML file is sent through the Parse engine followed by the Packet Processing Engine. For most workloads, the XML accelerator is able to sustain a throughput of over 10 Gbit/s.

In addition to throughput, the latency of document processing is an important performance metric for the XML unit. Figure 4.19 shows the latencies of processing a document under each of the workloads studied. The processing latency is dependent on the size of the file, the output format (tree format takes longer) and the post-processing steps if any (last 3 workloads).
Figure 4.18: Throughputs for XML workloads when generating TLA and TREE output (size in bytes and number of TLAs in workload shown along x-axis)

Figure 4.19: Latency to process XML documents (size in bytes and number of TLAs in workload shown along x-axis)

### 4.8 Acknowledgments

This work was done in close collaboration with Timothy Heil, Nicholas Lindberg, Farnaz Toussi and Steven VanderWiel. The work benefited from the contributions of the whole PowerEN team at IBM; in particular, we thank Brian Bass, William McNeil, Heather Achilles, Colin Verilli, Ken Inoue, Kay Muller, Sairam Kamaraju, Shaul Yifrach, James Xenidis, Chris Arges, David Maze, Kelly Carpenter, Randall Kunkel, Thomas McSweeney, Shane Lardinois, Srinivasan Ramani, James Appleman, Stephen Eaton, James Tate, David Moscrip and many others for their technical advice, tools support and lab support. We received high level feedback and suggestions from Ahmad Samih and Yan Solihin. We thank them all.
Chapter 5

ReSHAPE - Resource Sharing and Heterogeneity-aware Analytical Performance Estimator

This chapter is organized as follows: Section 5.1 describes how ReSHAPE works. Section 5.2 presents the methodology and results related to validation of ReSHAPE. Section 5.3 provides several use cases to highlight how ReSHAPE can be used to tackle interesting design space exploration and optimization problems. Section 5.4 places this work in the context of prior and related art.

5.1 The ReSHAPE Model

The Resource Sharing and Heterogeneity-aware Analytical Performance Estimator, ReSHAPE, is a chip-level performance estimation tool. It is written in C++ as an iterative solver of an underlying analytical model. By not attempting to be a closed-form mathematical model, ReSHAPE avoids the need to oversimplify either the uniqueness of a workload’s characteristics, or the effects of interaction between applications when sharing microarchitectural resources on a chip. ReSHAPE’s goal is to be able to predict the performance (measured in Instructions Per Second) of a multi-core chip when running any statically scheduled multi-program workload mix. ReSHAPE is not intended to be a replacement for detailed cycle-accurate simulation. Rather, it is meant to be used during early design space exploration (DSE), where the explosion in the design space can only be tamed by using a very fast and reasonably accurate estimator. ReSHAPE does not explore the design space on its own. It is only a fast performance estimator that can be used to evaluate a particular point in the design space.

In this section we first describe the two key inputs needed to use ReSHAPE - the charac-
teristics of an application-core pair and the chip configuration. We then explain the three key aspects of ReSHAPE’s approach to modeling a multi-core chip - handling shared caches, handling shared memory bandwidth and engaging the feedback loop between the core and memory system performance.

Table 5.1: Workload Characteristics inputs to ReSHAPE

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Example Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>hrpI</td>
<td>Hit Rate Profile for the instruction stream (hit rate with each additional slice of cache)</td>
<td>0.000000, 0.166224, 0.621715 ... 1.000000</td>
</tr>
<tr>
<td>hrpD</td>
<td>Hit Rate Profile for the data stream (hit rate with each additional slice of cache)</td>
<td>0.000000, 0.000117, 0.000548 ... 1.000000</td>
</tr>
<tr>
<td>apiI</td>
<td>Instruction-side Cache Accesses Per Instruction (beyond the L1 caches, which are considered part of the core microarchitecture)</td>
<td>0.000188</td>
</tr>
<tr>
<td>apiD</td>
<td>Data-side Cache Accesses Per Instruction (beyond the L1 caches, which are considered part of the core microarchitecture)</td>
<td>0.053471</td>
</tr>
<tr>
<td>baseIPC</td>
<td>Base Instruction Per Cycle (assuming a 1GHz clock and perfect L1 caches)</td>
<td>0.622839</td>
</tr>
</tbody>
</table>

5.1.1 Inputs to ReSHAPE

One key set of inputs into ReSHAPE are metrics that characterize the behavior of each application of interest when it runs on each core microarchitecture of interest. Collecting these metrics requires detailed simulations, but must be done only once per every unique application-core pair. The metrics collected in this step are listed in Table 5.1.

The Hit Rate Profiles, hrpI and hrpD, capture the projected cache hit rates at different cache sizes for the Instruction and Data traffic streams for a given application-core pair. Though not strictly necessary, we collect separate profiles for the application’s instruction stream and data stream because these memory access streams tend to have very different behaviors. The Hit Rate Profiles are collected as cumulative Stack Distance Profiles [47], which allows us to use a single simulation to project the approximate hit rates across a large range of cache sizes (which otherwise would have required multiple simulations per cache size).

The Accesses Per Instruction metric, apiI and apiD, capture the cache access aggressiveness of the application, and are similarly collected for the instruction and data streams separately.

The base IPC, baseIPC, is the Instructions Per Cycle rate that the given microarchitecture can achieve for a given application assuming an infinitely fast and infinitely large cache at the second level. The first level cache is considered to be an inherent part of the microarchitecture, and is modeled as part of the detailed simulation. The impact of this first level of cache is

84
reflected in the base IPC, which also captures the essential control flow and data flow in the application and the microarchitecture’s inherent ability to work with that. For example, the microarchitecture’s superscalarity, its ability to expose Instruction Level Parallelism, its ability to predict control flow using the branch predictor etc. is all captured in the base IPC.

Table 5.2: Example configuration input to ReSHAPE

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Example Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>nCores</td>
<td>Number of Cores on chip</td>
<td>4</td>
</tr>
<tr>
<td>freq</td>
<td>Frequency in MHz of each core</td>
<td>2000, 2000, 667, 333</td>
</tr>
<tr>
<td>apps</td>
<td>Applications running in each core</td>
<td>xalan, omnetpp, mcf, hmmer</td>
</tr>
<tr>
<td>nStrmsPerCore</td>
<td>Traffic Streams Per Core (default 2 - instruction and data)</td>
<td>2</td>
</tr>
<tr>
<td>nCLvls</td>
<td>Number of cache levels (beyond the L1 which is assumed to be part of the core microarchitecture), for example L2 and L3</td>
<td>2</td>
</tr>
<tr>
<td>nCPerLvL</td>
<td>Number of caches at each level</td>
<td>4, 3</td>
</tr>
<tr>
<td>nStrmsPerC</td>
<td>Number of traffic streams at each cache</td>
<td>2, 2, 2, 4, 2, 2</td>
</tr>
<tr>
<td>strmsAtC</td>
<td>IDs of the traffic streams at each cache</td>
<td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15</td>
</tr>
<tr>
<td>cSize</td>
<td>Size of each cache in KB</td>
<td>128, 128, 512, 128, 1024, 0, 512</td>
</tr>
<tr>
<td>lnSz</td>
<td>LineSize in Bytes</td>
<td>64</td>
</tr>
<tr>
<td>cLat</td>
<td>Hit latency of each cache in ns</td>
<td>3, 3, 5, 3, 6, 0, 5</td>
</tr>
<tr>
<td>mLat</td>
<td>Memory Penalty in ns</td>
<td>100</td>
</tr>
<tr>
<td>availBW</td>
<td>Available Memory Bandwidth in MBps</td>
<td>10000</td>
</tr>
<tr>
<td>err</td>
<td>Maximum difference in the performance estimates between successive iterations before model stops</td>
<td>0.01</td>
</tr>
<tr>
<td>partStyle</td>
<td>Partition strategy for shared cache space - MinMiss, Greedy, Natural</td>
<td>minMiss</td>
</tr>
<tr>
<td>prf1StcSz</td>
<td>Size of each additional slice of cache in KB in the application characteristics data</td>
<td>8</td>
</tr>
</tbody>
</table>

In addition to the application-core pair characteristics, ReSHAPE requires an input file that describes the chip configuration - how many cores the chip integrates, what frequencies they operate at, what application is running on each core, what the cache hierarchy beyond the L1s is, and, how much off-chip memory bandwidth is available. Table 5.2 lists these configuration parameters expected by ReSHAPE, along with some example values. Figure 5.1 shows the layout corresponding to the example configuration in Table 5.2. Notice that the configuration allows an arbitrary mixture of shared and private caches. It also allows different levels of hierarchy for different cores - a 3-level hierarchy for cores 0, 1 and 3, and a 2-level hierarchy for core 2 in the example.

ReSHAPE considers each core to be a memory traffic generator, generating several streams of traffic. As discussed earlier, the default implementation of the ReSHAPE model assumes two streams of traffic per application - the instruction traffic stream and the data traffic stream.
However, this assumption can be modified as needed - to more or fewer streams per core, or, to an entirely different categorization of streams (for example, Reads versus Writes). We leave the evaluation of how such stream definitions impact accuracy for future research. Based on our default categorization of streams, there are a total of $2^N$ traffic streams in a chip with $N$ cores.

The cache hierarchy to be simulated is specified using parameters that define the number of cache levels (beyond the L1s), the number of caches at each level, the size and access latency of each cache, and the identities of the streams flowing into and out of each cache. Every level of caches has $2^N$ streams entering and leaving that level. The streams are numbered in ascending order starting at 0. In Figure 5.1 streams 0 through 7 (marked by numbered bubbles) enter the first level of caches and streams 8 through 15 leave the first level of caches to enter the second level of caches. Streams 16 through 23 leave the second level of caches and contend for the off-chip memory.

The configuration file also specifies the off-chip memory bandwidth and the off-chip memory latency. Some of the other parameters, such as, err, partStyle and prflSlcSz, will become
clearer in the next section.

5.1.2 How ReSHAPE works

We will use Figure 5.2 as a simple illustrative example to explain how ReSHAPE computes chip performance. For this example, we assume there are two cores, each running a different application. For simplicity we show a single traffic stream emanating per core. For each core and application pair, the baseIPC and api is available from the detailed characterization. The product of the baseIPC, api, and, the core frequency, estimate the average memory traffic (in bytes per second) generated by each core stream. The hrp is then used to estimate the reduction in traffic that can be achieved at each cache level. This is roughly illustrated in the figure by the progressive narrowing of the downward block arrows that represent each stream. Before applying the hrp, however, ReSHAPE partitions the space in shared caches across the streams sharing the cache (for example, the L3 in Figure 5.2 is partitioned between stream 2 and stream 3). Similarly, ReSHAPE partitions the shared off-chip memory bandwidth across the streams at that level (stream 4 and stream 5 in the figure). The next few sub-sections elaborate on this partitioning process, at the end of which the chip can be effectively broken into a collection of vertical silos of execution, each of which corresponds to one core. Thus, ReSHAPE breaks the chip into a a collection of cores with effectively private cache and memory hierarchies. The throughput of each vertical silo is computed in terms of Instructions Per Second (IPS). The chip’s throughput is the sum of the individual throughputs of each core.

Handling shared caches

The Hit Rate Profiles are used at each cache level to identify how well that cache level can filter the traffic stream coming into it. While this is easy if a cache is private to a given stream, given our classification of streams it is very likely that a given cache will be shared by at least two streams (data and instruction stream from a given application), and by more streams if the cache is shared across cores. ReSHAPE deals with caches shared by multiple streams by approximating how the cache space would be partitioned across those streams in steady state.

ReSHAPE currently supports three cache partitioning strategies - MinMiss, Greedy and Natural. The strategies differ in terms of how the Hit Rate Profiles (hrpI and hrpD) are used in determining the steady state partitions. During detailed simulation described in Section 5.1.1, we collect the hrps at the granularity of a slice, s (the prftSlcSz parameter in Table 5.2). Each hrp is therefore the collection of hit rates at cache sizes s, 2 \cdot s, 3 \cdot s, etc. until the cache size becomes large enough to cover all cache sizes of interest (we use a maximum cache size of 64MB in this work).

The MinMiss strategy identifies partitions that minimize the cache’s total miss traffic.
ReSHAPE evaluates all possible partition combinations for a given cache with the minimum reassignment granularity of $prflSlcSz$. The number of traffic streams sharing the cache, their individual Hit Rate Profiles, as well as their access aggressiveness (product of core frequency and $api$) impact the total misses per unit time leaving the cache. This leads to a total of $n+1C_{k-1}$ partition combinations that need to be evaluated - $n$ is the number of slices and $k$ is the number of streams sharing the cache. This computation is reasonably fast (in the milliseconds range on a single thread on a workstation class machine) if the number of sharing streams per cache is no more than 16 or so. As the sharer count increases this approach may become the performance bottleneck to running ReSHAPE.

The *Greedy* strategy is similar to *MinMiss* in spirit, however it is much faster, scales well with increasing sharers per cache, and generally quite accurate. Instead of finding the globally optimal partition that minimizes misses, it adopts the greedy approach of assigning one slice of the cache at a time to the application that can most benefit (in terms of reducing miss rate) from that slice of extra cache space. This ensures that the number of computations is bounded by $n \cdot k$, where $n$ is the number of slices and $k$ is the number of streams sharing the
cache. In theory, the disadvantage of using this strategy stems from its restricted visibility to an application’s full hrp. A stream that could see a huge benefit in hit rate after accepting no benefits from the next handful of slice allocations, could forever starve while the other streams keep getting awarded slices that provide them no more than marginal benefits. One way to reduce the probability of this pathological scenario is to pick a strategy that bridges MinMiss and Greedy approaches, and allocates a group of slices at a time. However, we leave that for future evaluations, especially since we found that, in practice, the Greedy strategy performs very similarly to the MinMiss strategy.

Both the MinMiss and Greedy strategies try to minimize misses. We also implemented a Natural strategy that attempts to approximate the partitions that the cache would naturally end up with when not attempting to minimize overall misses. This is a difficult problem and has been researched in greater depth than we can delve into here by Chandra et al. [24] and Kim et al. [47]. However, for completeness, we describe the intuition behind the Natural strategy as we implemented it. If an application’s cache partition shrinks, and, as a consequence, its miss rate increases, the lines installed by those misses work towards increasing the application’s cache partition (assuming no strict partitioning limits and replacement rules are enforced). This increase in effective partition size can, in turn, help reduce its miss rate. However, the increase in this application’s partition size reduces the partition size of the other applications sharing the cache. A simple invariant that describes this balance is $\frac{p_i}{p_j} = \frac{m_i}{m_j}$, where $i$ and $j$ represent two applications, $p$ represents the partition size for a given application and $m$ represents the misses per unit time for a given application. However, further study is required to fine-tune this invariant - for example, it is possible that the relationship between partition sizes and miss rates is non-linear. We leave that for future work.

We use the MinMiss strategy for all results reported in this work.

**Feeding back the impact of memory bandwidth**

Memory bandwidth is also a resource shared across the streams on a chip. Therefore it is also partitioned just like the caches. We adopt a simple but effective partitioning strategy - we partition the total bandwidth into parts in proportion to the per-stream traffic demand at that level. For example, if there are only 2 streams, A and B, and stream A requests twice as much bandwidth as stream B, the available bandwidth is first divided between the two streams in 2:1 ratio.

The core’s base IPS is given by Equation 5.1. Once the available memory bandwidth per core is determined the impact of the memory bandwidth must be tied back to the core’s effective IPS. This is achieved by using the analytical model in Equation 5.2 for each core.

$$baseIPS = baseIPC \cdot freq$$  \hspace{1cm} (5.1)
\[ IPS = \frac{1}{\text{baseIPS} + \text{api}_2 \cdot \text{lat}_2 + \text{api}_3 \cdot \text{lat}_3 + \cdots + \text{api}_M \cdot (\text{lat}_M + \text{lat}_Q)} \]  

(5.2)

where \( \text{api}_x \) represents the accesses per instruction that reach level \( x \) of the cache and memory hierarchy - a value that is computed by ReSHAPE’s knowledge of the application’s hrp as well as the partition size at each cache level. \( \text{lat}_x \) represents the access latency (in seconds) at the corresponding hierarchy level. \( \text{lat}_M \) represents the average off-chip memory access penalty (in seconds). \( \text{lat}_Q \) represents the queuing delay (in seconds) due to contention, before the memory interconnect may be accessed. For a given core, this \( \text{lat}_Q \) term is affected by the bandwidth partition available for the core and the rate of requests made by the core.

ReSHAPE computes \( \text{lat}_Q \) by modeling the memory interconnect as an M/D/1 queue, where the mean request rate is given by \( \text{api}_M \cdot \text{baseIPC} \cdot \text{freq} \) and the mean service time is given by the computed available bandwidth for that core (both rates measured in requests per second). Equation 5.2 is used at each core to compute their effective Instructions Per Second value.

ReSHAPE’s Iterative Solver

This newly computed IPS at each core can potentially change the traffic streams emanating from each core sufficiently that all the computations related to shared resources must be re-done. These new IPS values are fed back to the core streams as their new baseIPS values, which ReSHAPE uses to recompute the cache resource partitions, the resulting hit rates, the downstream traffic at each level, and memory queuing effects of the available bandwidth. This generates a new IPS value for each core. ReSHAPE iteratively continues this process until the solution stabilizes and the error is smaller than the configured threshold (\( \text{err} \) in Table 5.2). The chip-wide IPS may then be computed by simply adding up the individual IPSs, although, for certain fair metrics we find that tracking individual IPSs at each core is necessary.

5.2 Validation of the ReSHAPE Model

5.2.1 Methodology

We validate ReSHAPE’s accuracy by comparing its projections for core IPCs, cache miss rates, and chip throughput to the same metrics as projected by detailed simulation using the Simics [61] full-system simulator. The configurations we validate vary the core count (1, 2 and 4 core), cache sizes, cache configurations (shared and private caches), memory bandwidth, and, workload mixes.
Workload Mixes

For validation as well as all evaluations described in the next section, we use 27 of the 29 SPEC2006 applications (we could not run 2 of the 29 benchmarks under Simics). Each benchmark is run against the reference input set in the fast-forward mode (functional simulation mode) for 100 billion instructions, at which point a checkpoint is saved. For all timing simulations under Simics we simulate 1 billion instructions beyond this checkpoint. When running multi-application workload mixes, we ensure that the slowest of all applications is able to run 1 billion instructions (thus ensuring that all other applications run at least 1 billion instructions). For applications that run longer than 1 billion instructions, all statistics collection stops when 1 billion instructions have been completed.

For multi-core simulations, we needed to construction multi-application workload mixes. In order to get a wide range of workload behaviors, we first studied the cache locality behavior of each of these applications individually by simulating a single core system under Simics equipped with a Stack Distance Profiler module. We then categorized these benchmarks into 3 groups
based on their cache locality behavior - TIGHT, MEDIUM and LOOSE locality. An application with a TIGHT locality fits in a small cache, and one with a MEDIUM locality needs a larger cache to start seeing good hit rates, while those with LOOSE locality may not see good hit rates even with very large caches. We then constructed multi-application workload mixes from these 27 applications, while ensuring that the workload mixes covered a varied range of locality behaviors. Figure 5.3 shows the locality profiles of 9 benchmarks which figure prominently in the workload mixes we use. Benchmarks with LOOSE locality are shown in the top row, with MEDIUM locality benchmarks in the middle row and the TIGHT locality benchmarks in the bottom row.

Table 5.3 lists the workloads used in this work - for 1, 2, 4 and 9 core simulations. The 1, 2 and 4 application mixes are used for the validation studies described in this section, as well as ReSHAPE use cases described in Section 5.3. The 9 application mix is used for a study described in Section 5.3.1. All mixes are constructed with the view to cover a wide range of cache locality behavior, as indicated by the string of characters T, M and L in parentheses next to each mix in the table.

Configurations

In order to validate ReSHAPE we first defined several chip configurations. The full set of configurations used for validation are listed in Table 5.4. Each of the configurations is set up both under Simics and ReSHAPE, and for each configuration we run the appropriate set of workloads listed in Table 5.3. For each validation run we compare the projected per-core Instructions Per Cycle (IPC) and cache miss rates from ReSHAPE against the corresponding values reported from the detailed simulation. When simulating with a shared cache the accuracy of ReSHAPE depends on how well it estimates the ratios in which the cache would effectively get divided between the applications sharing it. Therefore, as part of validation of a shared cache configuration, we also compare the cache partition sizes projected by ReSHAPE to the average partition sizes seen during detailed simulation.

For validation studies in this section, all the cores are all assumed to be running at 1GHz. Cache access latencies are assumed to be correlated with the cache size as follows. A 64KB cache is assumed to require 2 cycles at 1GHz for access; each progressive doubling of the cache size is assumed to require 1 additional cycle for access - for example, 3 cycles for a 128KB cache, 4 cycles for a 256KB cache, and so on. The cores are assumed to have a 32KB L1I and 32KB L1D. The core modeled by Simics is a simple single-issue in-order processor with blocking cache accesses - that is, while a miss is outstanding the core can not continue to make progress on the application. While this is not representative of modern high performance, superscalar, out-of-order processors, we believe that it suffices to prove the validity of the approach ReSHAPE takes to estimate chip-level performance. ReSHAPE's novelty and value-add is not in how
Table 5.3: Workload mixes used in this work

<table>
<thead>
<tr>
<th>Cores</th>
<th>Workloads (T=Tight, M=Medium, L=Loose)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Core</td>
<td>astart, games, lbm, omnetpp, tonto, bwaves, gcc, leslie3d, perl, xalan, bzip, gems, lib, povray, zeusmp, cactus, gromacs, mcf, sjeng, calculix, h264, milc, soplex, deal2, hmmer, namd, sphinx</td>
</tr>
<tr>
<td></td>
<td>each app. skips 100B insts. and runs 1B</td>
</tr>
<tr>
<td>2 Core</td>
<td>m00: xalan, namd (MT)</td>
</tr>
<tr>
<td></td>
<td>m01: xalan, xalan (MM)</td>
</tr>
<tr>
<td></td>
<td>m02: omnetpp, lib (ML)</td>
</tr>
<tr>
<td></td>
<td>m03: povray, povray (TT)</td>
</tr>
<tr>
<td></td>
<td>m04: mcf, namd (LT)</td>
</tr>
<tr>
<td></td>
<td>m05: milc, milc (LL)</td>
</tr>
<tr>
<td></td>
<td>m06: omnetpp, tonto (MT)</td>
</tr>
<tr>
<td></td>
<td>m07: leslie3d, omnetpp (MM)</td>
</tr>
<tr>
<td></td>
<td>m08: xalan, mcf (ML)</td>
</tr>
<tr>
<td></td>
<td>m09: tonto, namd (TT)</td>
</tr>
<tr>
<td></td>
<td>m10: milc, tonto (LT)</td>
</tr>
<tr>
<td></td>
<td>m11: lib, mcf (LL)</td>
</tr>
<tr>
<td></td>
<td>each app skips 100B insts. and runs at least 1B</td>
</tr>
<tr>
<td>4 Core</td>
<td>m00: povray, povray, tonto, namd (TTTT)</td>
</tr>
<tr>
<td></td>
<td>m01: povray, tonto, tonto, xalan (TTTM)</td>
</tr>
<tr>
<td></td>
<td>m02: mcf, tonto, namd, namd (TTTL)</td>
</tr>
<tr>
<td></td>
<td>m03: omnetpp, xalan, leslie3d, povray (MMMT)</td>
</tr>
<tr>
<td></td>
<td>m04: omnetpp, leslie3d, leslie3d, xalan (MMMMM)</td>
</tr>
<tr>
<td></td>
<td>m05: omnetpp, leslie3d, xalan, lib (MMML)</td>
</tr>
<tr>
<td></td>
<td>m06: mcf, lib, milc, povray (LLLL)</td>
</tr>
<tr>
<td></td>
<td>m07: omnetpp, mcf, milc, lib (LLLL)</td>
</tr>
<tr>
<td></td>
<td>m08: mcf, lib, lib, milc (LLLLL)</td>
</tr>
<tr>
<td></td>
<td>m09: povray, namd, leslie3d, xalan (TTTMM)</td>
</tr>
<tr>
<td></td>
<td>m10: mcf, milc, tonto, namd (TTL)</td>
</tr>
<tr>
<td></td>
<td>m11: mcf, xalan, leslie3d, lib (MMLL)</td>
</tr>
<tr>
<td></td>
<td>each app skips 100B insts. and runs at least 1B</td>
</tr>
<tr>
<td>9 Core</td>
<td>m00: povray, tonto, namd, deal2,games, astart, leslie3d, xalan, omnetpp (TTTTTTTTMM)</td>
</tr>
<tr>
<td></td>
<td>m01: deal2, games, astart, perl, calculix, gromacs, lib, milc, mcf (TTTTTTTTLLL)</td>
</tr>
<tr>
<td></td>
<td>m02: perl, calculix, gromacs, leslie3d, xalan, omnetpp, hmmer, soplex, bzip (TTTTTTTTTTTMM)</td>
</tr>
<tr>
<td></td>
<td>m03: povray, tonto, namd, leslie3d, xalan, omnetpp, lib, milc, mcf (TTTTTTTTTTML)</td>
</tr>
<tr>
<td></td>
<td>m04: perl, calculix, gromacs, lib, milc, mcf, lbm, sphinx, gems (TTTTTTTTTTTLLLL)</td>
</tr>
<tr>
<td></td>
<td>m05: leslie3d, xalan, omnetpp, hmmer, soplex, bzip, lbm, sphinx, gems (MMMMMMMMML)</td>
</tr>
<tr>
<td></td>
<td>m06: hmmer, soplex, bzip, lib, milc, mcf, lbm, sphinx, gems (MMMMMLLLLLL)</td>
</tr>
<tr>
<td></td>
<td>not used for Simics-based validation, used for a 9-core chip design use case</td>
</tr>
</tbody>
</table>

the core is modeled, but rather in how individually-collected performance statistics of different application-core pairs are combined when they are running on the same chip sharing cache and bandwidth resources. This aspect of ReSHAPE remains true regardless of whether the cores models used to generate the workload-microarchitecture characterization data are in-order scalar or out-of-order superscalar. In a microarchitecture that exposes memory level parallelism the memory-accesses-per-instruction parameters (api\textsubscript{I} and api\textsubscript{D}) that feed ReSHAPE must be adjusted before being used in the feedback mechanism that adjusts each core’s individual base\textsubscript{IPS} at the end of each iteration through the model. The api parameters must be reduced
Table 5.4: Configurations used for validation of ReSHAPE

<table>
<thead>
<tr>
<th>Name</th>
<th>Configuration</th>
<th>Sims</th>
</tr>
</thead>
<tbody>
<tr>
<td>1c-256K-10G</td>
<td>1 core, 256KB L2, 10GBps mem. b/w</td>
<td>27</td>
</tr>
<tr>
<td>1c-1M-10G</td>
<td>1 core, 1MB L2, 10GBps mem. b/w</td>
<td>27</td>
</tr>
<tr>
<td>1c-1M-1G</td>
<td>1 core, 1MB L2, 1GBps mem. b/w</td>
<td>27</td>
</tr>
<tr>
<td>1c-1M-100M</td>
<td>1 core, 1MB L2, 100MBps mem. b/w</td>
<td>27</td>
</tr>
<tr>
<td>1c-1M-10M</td>
<td>1 core, 1MB L2, 10MBps mem. b/w</td>
<td>27</td>
</tr>
<tr>
<td>2c-256K-10G</td>
<td>2 core, 256KB shared L2, 10GBps mem. b/w</td>
<td>12</td>
</tr>
<tr>
<td>2c-1M-10G</td>
<td>2 core, 1MB shared L2, 10GBps mem. b/w</td>
<td>12</td>
</tr>
<tr>
<td>2c-1M-1G</td>
<td>2 core, 1MB shared L2, 1GBps mem. b/w</td>
<td>12</td>
</tr>
<tr>
<td>2c-1M-100M</td>
<td>2 core, 1MB shared L2, 100MBps mem. b/w</td>
<td>12</td>
</tr>
<tr>
<td>2c-1M-10M</td>
<td>2 core, 1MB shared L2, 10MBps mem. b/w</td>
<td>12</td>
</tr>
<tr>
<td>4c-512K-10G</td>
<td>4 core, 512KB shared L2, 10GBps mem. b/w</td>
<td>12</td>
</tr>
<tr>
<td>4c-2M-10G</td>
<td>4 core, 2MB shared L2, 10GBps mem. b/w</td>
<td>12</td>
</tr>
<tr>
<td>4c-2M-1G</td>
<td>4 core, 2MB shared L2, 1GBps mem. b/w</td>
<td>12</td>
</tr>
<tr>
<td>4c-2M-100M</td>
<td>4 core, 2MB shared L2, 100MBps mem. b/w</td>
<td>12</td>
</tr>
<tr>
<td>4c-2M-10M</td>
<td>4 core, 2MB shared L2, 10MBps mem. b/w</td>
<td>12</td>
</tr>
<tr>
<td>4c-512K-x4-10GBps</td>
<td>4 core, 4x512KB private L2s, 10GBps mem. b/w</td>
<td>12</td>
</tr>
<tr>
<td>4c-256K-1M-x2-10GBps</td>
<td>4 core, 256KB, 1MB, 256KB, 1MB prvt. L2s, 10GBps mem. b/w</td>
<td>12</td>
</tr>
<tr>
<td>4c-128K-2M-x2-10GBps</td>
<td>4 core, 128KB, 2MB, 128KB, 2MB prvt. L2s, 10GBps mem. b/w</td>
<td>12</td>
</tr>
</tbody>
</table>

Total validation simulations 291

by an application-dependent constant such that the \textit{apis} only correspond to the truly exposed fraction of the accesses to memory. The rest of ReSHAPE’s methodology continues to work unchanged. While we leave the formal validation of this specific aspect of ReSHAPE for future researchers, we believe that ReSHAPE’s approach to chip-wide performance projection will continue to hold independent of the microarchitectural differences.

5.2.2 Validation Results

1 Core

While ReSHAPE’s main contribution is in speeding up performance projections for multi-core configuration by several orders of magnitude compared to detailed simulation, we start by comparing the IPCs and cache miss rate’s projected by ReSHAPE for a single-core configuration. We assume a 256KB L2 cache and 10GBps memory bandwidth for this configuration. Figure 5.4 shows the IPCs for the 27 SPEC2006 applications projected by Simics and the corresponding projections by ReSHAPE. The plot on the right shows the error in ReSHAPE’s projection - a deviation from the diagonal indicates a discrepancy between ReSHAPE and Simics. The IPC projection is very accurate - the average error is 1.5% (standard deviation is 1.4%). Figure 5.5 shows the L2 cache miss rates. The average error is 7.6% (standard deviation is 12.4%); ReSHAPE is able to capture the trends reasonably accurately. We also validated ReSHAPE with a single-core configuration with a 1MB L2 cache and found the accuracy trends to be similar.
to the 256KB L2 case.

2 Cores with Shared L2 Cache

The first multi-core chip we configure for validation is a 2-core chip with a shared 1MB L2 cache and 10GBps of memory bandwidth. With a shared cache, ReSHAPE must estimate how the cache would get partitioned between the two applications, and then use that estimate to project the effective cache miss rates and IPCs for each application. Figure 5.6 shows the IPCs for each of the cores as projected by Simics (left-most plot) and as projected by ReSHAPE (middle plot). As can be seen, ReSHAPE is able to project individual IPCs (and therefore the chip wide throughput) quite accurately across the workload mixes. The average IPC error is 2.7% (standard deviation is 2.1%).

Figure 5.7 shows the L2 cache miss rates, and Figure 5.8 shows the cache partition projections. The miss rate error is 13.4% on average (standard deviation is 12.6%); however, most of
the large error contributions come from applications that have a very small absolute miss rate. Also, we observed that the influence of the operating system code increases when running application mixes, as opposed to running single applications. ReSHAPE relies on cache behavior parameters collected in the single core mode, even when projecting in the multi-core configurations. However, the cache miss rate behavior can change under Simics in the multi-core mode. This also leads to the observed error. That said, we believe that ReSHAPE is able to sufficiently capture the high level interactions between applications sharing chip resources. The error in cache partitions is 3.7% on average (standard deviation is 4.5%). While the partitions are fixed in ReSHAPE at the beginning of each iteration through the model, the partitions in Simics keep changing throughout the simulation. We estimate the average effective partition sizes in Simics by first extracting the fraction of the shared cache in use by each application at every millionth simulation cycle, and then averaging across those samples. We also validated ReSHAPE with a 2-core configuration with a 256KB L2 cache; the accuracy trends remain similar to those observed with the 1MB L2 cache.
4 Core with Shared L2 Cache

With a 4-core configuration, we wanted to see how ReSHAPE’s accuracy scaled. Figure 5.9 shows the IPC projection accuracy of ReSHAPE for a 4-core system with a 2MB L2 cache and 10GBps of memory bandwidth. The average IPC error is 2.5% (standard deviation is 1.8%), showing that ReSHAPE continues to be reasonably accurate even as the number of cores increases. Figure 5.10 shows the miss rates - the average error is 12.8%(standard deviation is 13.1%). As can be seen from the rightmost plot, the large errors in miss rate projections come from the cases where the absolute miss rates are very small. As seen from the left and middle plots, ReSHAPE is able to follow the application’s cache behavior trends quite accurately.

Figure 5.11 shows that ReSHAPE is able to identify how the cache would get partitioned reasonably well across the board. The average error in projecting partitions is 20.9%(standard deviation is 12.8%), but this relatively large error must be understood in the context of two influencing factors. First, a small absolute error in projecting the partition size counts as a big relative error if the absolute partition size is small. For example, if a partition that should be 5% of the cache is projected to be 6% of the cache, that is counted as a 20% error. Second, every error in one partition takes away from a different partition, leading to additional error. For example, if a partition is under-projected by 1% of the total cache size, that same 1% must be over-projecting some other partition or partitions. Therefore, an overall 20% relative error in projecting partitions is not as bad as it seems. We also validated ReSHAPE with a 4-core configuration with a 1MB L2 cache, and saw promising accuracy numbers.

4 Cores with Private L2 Caches

All the multi-core validation thus far has used shared L2 caches of different sizes. In this section, we will show results from validation of ReSHAPE with 4 cores and private L2 caches. We evaluated 3 different private L2 configurations as shown in Table 5.4. Figure 5.12 shows the
IPC projection accuracy for a configuration where the 4 cores have a 128KB, 2MB, 128KB and a 2MB L2 cache respectively. The IPC error is 3.1% on average (standard deviation is 1.6%). Figure 5.13 shows the miss rates for each of the 4 L2s, and, the average error in the miss rate is 7.5% (standard deviation is 7.1%). The accuracy is greater with private L2 caches compared to shared L2 caches because with private caches ReSHAPE does not need to project partition sizes across applications. However, even with a private L2 there is still some partitioning going on to project how the instruction and data streams emanating from a single application share the L2 cache.

**4 Cores with Bandwidth Constraint**

All the validation cases described thus far have not exercised ReSHAPE’s ability to adjust performance projections based on the bandwidth. This is because we always assumed 10GBps of memory bandwidth, which is far greater than what these application mixes demand. However, once the bandwidth to memory starts getting scarcer, the per core performance drops. Projecting this accurately is important to ensure that ReSHAPE is able to scale to large systems with...
many cores, where memory bandwidth may become a first order constraint to chip performance.

We validate this aspect of ReSHAPE by setting up a 4-core configuration with a 2MB shared L2 cache and 4 different memory bandwidth settings - 10GBps, 1GBps, 100MBps and 10MBps - progressively constraining this aspect of the memory subsystem. Under Simics, we implemented a memory bandwidth module that adjusts the memory request service latency based on the available bandwidth, and queues requests as needed if the interconnect is in use. Under ReSHAPE, we mimic this behavior as described in Section 5.1.2.

Figure 5.14 shows the scatter plot of IPC projections of ReSHAPE on the y-axis, and the IPC projections from Simics on the x-axis, across the shrinking memory bandwidth. The plots show two things. First, the restriction of bandwidth does have a significant effect on the performance of the cores. This can be seen from how the cluster of dots tends to move from the top-right of the plot when there is ample bandwidth, to the bottom-left of the plot when memory bandwidth is severely limited. Second, the dots continue to remain very close to the diagonal axis. This means that ReSHAPE is able to continue to project performance quite accurately over a wide range of memory bandwidths. The average error in IPC projections is
17.3% with a standard deviation of 5.4%. The error is large here primarily because for the heavily bandwidth restricted cases the absolute IPCs tend to be very small, and small absolute errors are magnified. We also evaluated the effect of varying bandwidth similarly in a 1-core and 2-core configuration, and find that the general trends observed here hold. The average error is 10.3% (standard deviation of 4.0%) across the range of bandwidths for the 1-core case, and, 16.7% (standard deviation of 6.0%) for the 2-core case.

Overall, these validation results are very promising. These indicate that by using an iterative analytical model like ReSHAPE it is possible to get enough accuracy in projecting chip level performance that high level design evaluations and trend explorations can be attempted with ReSHAPE. The biggest benefit that ReSHAPE has over detailed simulation is speed. For example, a typical 4 core simulation runs in under 1 second on ReSHAPE compared to several hours on a detailed simulator. The kind of accuracy we see with ReSHAPE given this 1000x speedup...
speedup is what motivates us to actually apply ReSHAPE to interesting design questions, which ReSHAPE is uniquely placed to answer.

5.3 ReSHAPE Usecases

In this section we apply ReSHAPE towards answering several interesting and relevant chip-level design and schedule optimization questions. While the results from these use cases are interesting in their own right, the primary goal of this section is to show how ReSHAPE can be applied to a wide range of relevant problems.

5.3.1 Usecase 1: Heterogeneity in Cores versus Heterogeneity in Caches

Heterogeneity across computational cores on a chip is becoming more common [30, 46]. This is certainly true in the presence of application specific or special purpose processors such as hardware accelerators, Modulator-Demodulators, Digital Signal Processors and Graphics Processors on chip. However, there has also been an increasing interest both in academia and industry towards heterogeneity across the general purpose processor cores in a multi-core chip. This raises several interesting design questions. Is core heterogeneity necessary - what if cores remain homogeneous, while the caches are sized heterogeneously across the cores? What if both cores and caches are heterogeneous - does that add to the performance potential? How important is good application scheduling in heterogeneous processors - that is, can we lose performance compared to an equivalent homogeneous chip? Further, does greater heterogeneity help - for example, going from 2 core types and 2 cache sizes on a chip to 3 core types and 3 cache sizes?

While similar questions have been studied by chip architects over the last decade, there continues to be rich set of optimization challenges that remain unexplored. We suspect that the main hurdle to such exploration has been that the existing approaches that rely on timing approximate simulators, are impractically slow to address the design space. As the number of cores increases the simulator runtimes increase linearly. Further, for a given core count, the number of heterogeneous chip configurations and the corresponding application schedules (which application is run on which core) increase combinatorially. And evaluating this design space across multiple workload mixes further exacerbates the problem.

ReSHAPE, on the other hand, offers the right level of configurability and speed to attempt tackling these questions. We start this use case by evaluating a 4-core chip with private L2 caches configured at 4 different levels of heterogeneity, as shown in Figure 5.15. The blocks marked with a “C” are cores and the blocks marked with a “$” are L2 caches, with the thick double ended arrow connecting the core to its respective cache. In the homogeneous configuration (Figure 5.15(a)), we assume that each core is identical, and has 1 MB of L2 cache,
which occupies the same area as the core. All the cores in this homogeneous configuration are assumed to be 4 units in size, where a unit is an arbitrary measure of area which will allow comparison of core sizes across configurations. All the cores in this configuration are assumed to be operating at 1GHz and have 32KB I and 32KB D caches inside the area associated with the cores. In the configuration with heterogeneous cores and homogeneous caches (5.15(b)), two of the cores are smaller than the baseline core (1 unit) and the other two cores are larger (7 units). In the configuration with heterogeneous caches and homogeneous cores (5.15(c)), the cores are back to 4 units in size, but the caches are now smaller (1 unit, or, 256KB) or larger (7 units, or, 1792KB). And, lastly, in the fully heterogeneous configuration (5.15(d)), the cores and caches are both resized compared to the baseline - while ensuring that all the 4 combinations of core and cache sizes are honored. That is, there is a large core with a small cache, a large core with a large cache, a small core with a small cache and a small core with a large cache. Note that in all cases, we assume that the fraction of the chip allocated to cores and to cache remains unchanged. This assumption can be overridden as necessitated by the design space being explored.

We approximate the performance of larger and smaller cores using Pollack’s rule [77], and adjust the baseIPC parameter for each core size. Pollack’s rule states that the normalized performance of a core in a given technology correlates to the square root of its area. In a real-life application of ReSHAPE, actual statistics from the different microarchitectural core
designs could be provided to ReSHAPE.

Figure 5.16 shows the results of this evaluation. We ran the 12 4-application workload mixes described in Table 5.3 across each of the 4 configurations from Figure 5.15. For each workload mix and chip configuration, there are 24 (4!) permutations of how the applications are scheduled across the 4 cores. The plot shows the workload mixes and configurations for each mix along the x-axis. For each point on the x-axis a range of Weighted Speedups [85] compared to the homogeneous configuration with the default schedule (in the order indicated in Table 5.3) is plotted along the vertical axis. The minimum, maximum and average weighted speedup across the 24 application schedules for each point on the x-axis are highlighted. The weighted speedup metric ensures that the speedup of each application is measured relative to its own performance at the baseline configuration; thereby, avoiding the case where the results are biased by applications that have high absolute IPCs.

There are several interesting takeaways from this plot. First, there can be a big swing in performance depending on how the applications are scheduled. This is especially true when cores are heterogeneous. For example, mix m03 sees a 30% swing in performance depending on which application is scheduled on which core. This shows that ReSHAPE can be used to help guide the selection of an optimal static application schedule when scheduling multiple applications on a heterogeneous chip. Second, there are cases where heterogeneity in cores hurts performance compared to the base homogeneous case (mix m00, m01, m08, m11). This is attributable to the fact that going from 4 medium sized cores to 2 small and 2 large cores can hurt performance if the loss in performance on the smaller cores cannot be compensated for by the gain in performance from the larger cores. Third, heterogeneity in caches alone may be a safer bet, than adding heterogeneity in cores in a iso-area chip design, since it rarely loses performance over the homogeneous case. However, heterogeneity in caches alone does...
not provide a significant performance improvement in most mixes; mix m03, m04 and m05 see approximately 5% speedup potential assuming optimal scheduling.

Next, we use ReSHAPE to address similar questions in a chip with 9 cores. This case highlights the design space explosion problem.

Just as we did with the 4-core design, we wish to evaluate 9-core chip designs under the following 4 configurations - homogeneous cores and caches, heterogeneity in caches alone, heterogeneity in cores alone, and heterogeneity in both caches and cores. In addition, we also wish to explore the impact of a new dimension - the granularity of heterogeneity. We wish to understand whether allowing more unique types of cores and more cache sizes can further improve performance.

Each core in the baseline homogeneous chip is assumed to have an area of 4 units and own 1MB of private L2 cache. In one scenario we assume a granularity of heterogeneity of 2 - that is, 2 core types (1 unit and 7 units) and 2 cache sizes (256KB and 1792KB) to pick from. Since 9 cannot be equally divided into two groups, we assume 5 small cores and 4 large cores when configuring the chip with heterogeneous cores. Similarly we assume 5 small caches and 4 large caches when configuring the chip with heterogeneous caches. In the case where both cores and

Figure 5.17: Heterogeneity in cores vs. cache in a 9 core chip
caches are heterogeneous we ensure that we mimic as close to a uniform mix of core and cache sizes as feasible.

In the next scenario, we assume a granularity of heterogeneity of 3 - that is, 3 core types (1 unit, 4 units and 7 units) and 3 cache sizes (256KB, 1MB and 1792KB) allowed. The 9 cores are composed of 3 small, 3 medium and 3 large cores when the cores are heterogeneous. Similarly, the 9 caches are composed to 3 small, 3 medium and 3 large caches when the caches are heterogeneous. And when both cores and caches are heterogeneous, we ensure that all the various combinations of core and cache sizes are represented on the chip, thus leading to every core-cache pair looking different from every other such pair.

A single workload mix with 9 applications can be statically scheduled on a 9-core chip in 362,880 (9!) ways. We evaluated the above described 9-core design configurations across the 7 9-application workload mixes described in Table 5.3. This leads to a total design space of 362,880 static schedules per workload mix, 7 workload mixes per configuration, 4 configurations per heterogeneity granularity and 2 heterogeneity granularities, which corresponds to over 20 million points. We used ReSHAPE to visit every point on this space. This process was completed overnight by spreading the ReSHAPE runs across 300 2.7GHz Intel Xeons.

Figure 5.17(a) shows the results for the chip designs with 2 core types and 2 cache sizes. Figure 5.17(b) shows the results for the chip designs with 3 core types and 3 cache sizes. Several interesting observations can be made from the plots in Figure 5.17. First, comparing these plots to the one in Figure 5.16 it is clear that the benefits from supporting heterogeneity increases with core count. While on average heterogeneity was a small win (heterogeneous-caches) or a loss (heterogeneous-cores or heterogeneous-both) in the 4-core chip, with 9-cores heterogeneity in any form can lead to a big gain in chip performance. Second, the importance of good scheduling remains. In fact, without a good scheduler in place, a random schedule will likely lose performance compared to the homogeneous chip, as indicated by the Mean dot along each vertical range in the plots in Figure 5.17. Third, in general, heterogeneity in cores offers more potential benefit (10% on average over the baseline) than heterogeneity in caches alone (5% on average over the baseline). Heterogeneity in both cores and caches further enhances this potential (13% on average over the baseline). Fourth, comparing plots (a) and (b), we see that going form 2-types to 3-types, and thereby increasing the amount of heterogeneity does not make any noticeable difference to performance.

Overall, this use case teaches us that questions such as whether to add heterogeneity, what form of heterogeneity to add, and what granularity of heterogeneity to add, are all questions that must be carefully considered early in a chip’s design cycle. ReSHAPE can serve as a useful tool when making these decisions.
5.3.2 Use case 2: Application scheduling in presence of DVFS

In the previous use case we saw how ReSHAPE can be used to guide the design of a chip as well as application scheduling. In this use case we show how ReSHAPE can be used post design to optimize configurations of an existing multi-core chip. In particular, we consider a 4-core chip which allows each core’s dynamic voltage and frequency scaling (DVFS) to be independently configured over a range, a technique which is being widely adopted as a first step towards general purpose heterogeneity in an otherwise homogeneous multi-core design [45]. We assume that the chip supports 3 hypothetical, but realistic, operating points for each core - 250 MHz at 0.5 Watts, 1 GHz at 2 Watts and 4 GHz at 16 Watts. We assume that the 4 cores share a 2MB L2 cache and that each core has a 32KB L1I and 32KB L1D cache. We ran the 12 4-application workload mixes described in Table 5.3 across each of the 81 DVFS configurations that are possible; each of the 4 core can be in one of the 3 DVFS states. We define the base case to be the setting where all cores are at the lowest frequency (and voltage). Then, for each workload mix, we use ReSHAPE to compute the Weighted Speedups for each of the 81 configurations compared to the base case. We also compute other metrics such as Performance per Watt and the Energy Delay product for these configurations.

Figure 5.18 shows the most optimal configuration for each of the 3 metrics, across all the workload mixes. Several interesting observations can be made from these results. First, the optimal DVFS settings are different depending on the workload mix. This makes a case for using a ReSHAPE like tool to make an educated decision about DVFS settings at run time in
hardware. This option requires more research. For example, it requires estimating the necessary
inputs to ReSHAPE at runtime, perhaps by using the appropriate on-chip performance counters.
Second, even when the optimization target is maximizing the Weighted Speedup, which is a
purely performance metric, the best configurations do not run all the cores at full throttle. In
fact, for several configurations, 3 of the 4 cores run at the lowest frequency setting. This is
primarily because if an application is memory bound, running it faster does not benefit it much,
while it’s more aggressive cache usage may end up hurting other applications which could have
made better use of the cache. Third, for the Performance per Watt metric, which effectively
tries to minimize the total energy required to complete a task, it is not surprising to see that
the lowest frequency is the DVFS setting of choice in most cases. However, even here, there
are cases where the next higher setting, 1GHz, may need to be employed. This is because,
there are applications where the higher frequency can accomplish a task in sufficiently smaller
total time, that it may be worthwhile to spend the extra power during that time and still come
out ahead in terms of minimizing total energy used. Fourth, the configuration that minimizes
the Energy-Delay product can be starkly different from the optimal setting for the other two
metrics. While for some workload mixes the optimal setting looks similar to the optimal settings
for the Weighted Speedup metric, for others the optimal setting looks similar to the one for the
Performance per Watt metric.

Overall, this use case makes a case for using ReSHAPE (or a similar tool) to aid configuring
the DVFS settings of a multi-core chip - a problem that is otherwise difficult to tackle on a real
system, but holds the promise of providing significant impact and product differentiation.

5.3.3 Use case 3: Workload Characterization

ReSHAPE can also be applied to studying workload characteristics - how sensitive is the work-
load to core frequency and cache size? Is is computation bound or memory bound? We use a
1-core configuration in ReSHAPE and study the 27 SPEC2006 workloads listed in Table 5.3
across a range of core frequencies and L2 cache sizes. We divide the benchmarks into three
groups based on our findings. Benchmarks that are sensitive to core frequency are shown in
Figure 5.19. These benchmarks do not seem to be influenced by the amount of cache - as
long as the core runs faster, their performance improves. Benchmarks which are sensitive to
both core frequency and cache size are shown in Figure 5.20. For these benchmarks, greater
frequency improves performance especially if more cache size is also provided. And benchmarks
that are memory bound are shown in 5.21. These applications do not significantly improve
their performance as the core frequency is increased, or even as the cache size is increased.
5.4 Related Work

We first place ReSHAPE in the context of work on analytical models for multi-core performance estimation. In addition, given that one of the main usecases for ReSHAPE is as a tool for first pass optimization of a heterogeneous multi-core chip design, we place ReSHAPE in the context of other research in that area.

Analytical Modeling of multi-core chips Analytical modeling has been a standard approach for tackling fast design space explorations of multi-core chips. However, much of the current research has focused on tackling design of homogeneous multi-core chips, or assumes private cache hierarchies or both. Rogers et al. [82] develop a simple analytical model that studies how different microarchitectural techniques can enable continued core scaling while keeping the off-chip memory traffic unchanged. They do not consider heterogeneity in caches or the non-linear feedback between increasing memory bandwidth demand and core performance. Wentzlaff et al. [94] address this latter problem by developing an analytical model that ties the bandwidth to the core performance, however, they assume a homogeneous multi-core chip. Krishna et al. [49] extend the above model to incorporate the impact of data sharing between threads of a multi-threaded application running on the multi-core chip; however, the cores...
Figure 5.20: Workload characterization: Applications sensitive to both core frequency and cache size

are still considered to be homogeneous and the caches are still considered to be private. Li et al. [57] develop an analytical model to study how performance scaling across threads in a parallel application is limited by power constraints on a homogeneous multi-core chip. Yavits et al. [98] develop an analytical model that identifies the ideal cache hierarchy (number of levels and private versus shared configurations) to use on a homogeneous multicore chip.

One key aspect of ReSHAPE compared to prior work, which allows it to tackle many flexible cache configurations, is that ReSHAPE attempts to predict how shared caches would effectively get partitioned between applications. ReSHAPE uses the hit rate profiles to do this. Recent work by Wu et al. [96] uses similar profiles (which they call reuse distance profiles) to predict locality degradation in on-chip caches as the number of thread contexts on a chip scales. They consider multi-threaded applications - that is, they project the cache miss rates as the application is spread across more and more threads. On the other hand, ReSHAPE tries to project the cache miss rates as different applications start sharing the same cache.

While ReSHAPE only models heterogeneity in general purpose cores, there is some related work that attempts to bring accelerators into the scope of analytical models. We list these works here in the hope that these can inform future extensions of ReSHAPE. Nilakantan et
Figure 5.21: Workload characterization: Memory bound applications

al. [73] identify metrics to guide accelerator selection during the design of a heterogeneous multicore that supports hardware accelerators. They find that the communication between general purpose cores and accelerators must be considered in addition to computational speedups acceleration offers. Zidenberg et al. [101] cast the question of optimal accelerator selection in an area and power constrained chip into a formal analytical model. They find that at the optimum design point, there is no marginal benefit to giving more area to any one accelerator over any of the others.

**Heterogeneous multi-core optimization** The design space of a heterogeneous multi-core can be huge, rendering the standard approaches of cycle accurate simulation of each design point impractical. One way to speed up this process is by reducing the number of points in the design space to be evaluated as described by Navada et al. [68]. An alternative approach is by speeding up the evaluation of each point in the design space; this is the category ReSHAPE falls into. Note that ReSHAPE can benefit from the first category of techniques.

Optimizing a heterogeneous multi-core design refers to optimizing two key aspects of such a chip - core selection and application scheduling.

The first of these - what mix of cores to support on a chip - must be addressed at design time. One of the first works in the area of heterogeneous chip design is by Kumar et al. [52], where
they use detailed simulation to evaluate the performance of each application-microarchitecture pairing they were considering. The chip performance is then computed as a sum of the performance of the individual cores. Both these aspects are similar to ReSHAPE. The main difference is that ReSHAPE can handle a more general cache hierarchy and takes the effects of memory bandwidth into account, where as this prior art assumes private L2 caches and private memory channels for each core.

Post design, during run time, questions related to application scheduling must be addressed. ReSHAPE, as described in this work, optimizes a static schedule. That is, once an application is run on a core, it assumes the same type of application keeps running on the core. However, this aspect of ReSHAPE can be improved by treating applications as a collection of phases and calibrating ReSHAPE with each phase’s behavior. Simpoint [83] is a tool that can be used to break an application into a collection of representative phases. In an ideal scenario, ReSHAPE could be part of system firmware, where, with access to appropriate hardware performance counters, it could influence dynamic schedule optimization. Kumar et al. [51] show that dynamically migrating an application across cores in a heterogeneous multi-core can save power, although they use oracular scheduling. Navada et al. [67] propose bottleneck analysis to steer an application across heterogeneous cores on a chip. While both these works focus on dynamically scheduling a single application, the optimization problem become even harder when we consider multi-programmed workloads. Kumar et al. [53] propose sampling different schedules of multi-program mixes by running different permutations of the schedule across the different cores in a heterogeneous multi-core periodically. They then pick the best performing schedule for the next phase of execution. Becchi et al. [10] collect IPC statistics across a collection of core types per 1 million instruction phases for each application of interest. They use this information to schedule applications dynamically, though they do not consider shared caches and memory bandwidth. These prior works make a case for enabling a ReSHAPE-like solution in firmware, which can then estimate the best schedule based on key statistics collected thus far in hardware, while also taking cache sharing and bandwidth limitations into account.
Chapter 6

Conclusion and Future Work

The design of future computing systems faces several challenges. Bandwidth to off-chip memory, a resource which has always been plentiful, is, nonetheless, a chip pin-limited resource. As the number of computational contexts on a single chip grows, bandwidth becomes a first order design constraint. Multithreaded applications are becoming more prevalent on multi-core processor chips, and the impact of instruction and data sharing between threads in these applications must be considered for optimal chip design. Hardware Accelerators, while promising, need to be understood in terms of their performance potential and ease of programmability. And, finally, supporting heterogeneity among general purpose cores on a single chip, while highly promising, makes chip design space exploration nearly intractable without a fast and accurate modeling framework. In this report we present our findings from our study of these problems.

First, we developed a simple but powerful analytical model that allows us to study the bandwidth wall problem for CMP systems. We have sought to answer two key questions on this topic: (1) to what extent does the bandwidth wall problem restrict future multicorescale, and (2) to what extent are various bandwidth conservation techniques able to mitigate this problem. We found that the bandwidth wall problem can severely limit core scaling. When starting with a balanced, 8 core CMP, in four technology generations the number of cores can scale to only 24, as opposed to 128 cores which is desired under proportional scaling, without increasing the memory traffic requirement. We find that the impacts of the various bandwidth conservation techniques we evaluate have wide ranging benefits to core scaling. For example, using DRAM caches can potentially enable 47 cores in four technology generations, while other techniques such as using smaller cores provide a limited benefit to core scaling. Also, we find that when combined together, these techniques have the potential to enable super-proportional core scaling (183 cores for the combination of all techniques we evaluate) for up to four technology generations.
Second, we studied how instruction and data sharing impacts cache miss rates of multi-threaded workloads, and we developed an analytical throughput model that incorporates the effect of such inter-thread sharing. We showed how such a model can help system architects explore the interplay of design parameters, such as - number and size of cores, workload characteristics, and, cache, memory bus and memory organizations. The model reveals interesting insights. For example, the model reveals that though data sharing can significantly boost throughput compared to an application whose threads do not share data, in a bandwidth-constrained scenario, the benefit from sharing is severely restricted. Further, the model also shows that in future systems, which have a constrained off-chip bandwidth, it may be better to use fewer larger cores to build a CMP, rather than many smaller cores, without reducing the chip throughput.

Third, in a bid to understand the design tradeoffs involved in employing hardware acceleration alongside general purpose computation on a single chip, we studied in detail the design of IBM’s PowerEN processor. We also measure and report quantitative performance data for each of the hardware accelerators in PowerEN, across a range of test cases. The accelerators typically provide 10Gbit/s to 40Gbit/s of accelerated function, often orders of magnitude faster than software. This leaves many software threads free to address computation that is not conducive to acceleration. We found that while hardware accelerators can provide a significant performance boost for key algorithmic kernels compared to a general purpose processor, their usability relies upon techniques to minimize communication and data movement overheads.

Finally, we tackle the problem of fast and accurate performance evaluation for multi-core chips while supporting core heterogeneity and configurable cache hierarchies. We developed the Resource Sharing and Heterogeneity aware Analytical Performance Estimation tool, ReSHAPE. We validated ReSHAPE’s accuracy against detailed multi-core simulations. We then applied ReSHAPE to interesting chip level design optimization questions. We found that, within the constraints of our assumptions, in a 4-core design, a homogeneous design often outperforms a heterogeneous design, regardless of whether the heterogeneity is in the cores, the caches or both. Upon scaling to a 9-core chip, the benefit from heterogeneity starts to become more obvious; however, it still requires a good application to core mapper to deliver any benefits. We find that heterogeneity is cores is more important than heterogeneity in cache sizes. We also find greater heterogeneity may provide diminishing returns - a 9-core chip that supports 3 core types and 3 L2 cache sizes provides no advantage over one supporting 2 core types and 2 L2 cache sizes.

Overall, in this work, we investigate some of the interesting opportunities and challenges that researchers face in this era of the multi-core chip.
REFERENCES


