Cache Resource Allocation in Large Scale Chip Multiprocessors

by

Lisa Rufeng Hsu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctorate of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2009

Doctoral Committee:

Associate Professor Steven K. Reinhardt, Co-Chair
Professor Trevor N. Mudge, Co-Chair
Associate Professor Scott Mahlke
Associate Professor Dennis M. Sylvester
Ravishankar Iyer, Intel

© Lisa Rufeng Hsu 2009
All Rights Reserved

To my parents, De Dzwo and Ellen Hsu

ACKNOWLEDGEMENTS

This dissertation marks the single most trying, thrilling, educational, and lengthy period of my life. Through the research and experiences that have led to finishing this work, I have grown, both intellectually and emotionally, into a person I am proud to be.

I certainly could not have gone through this process alone — I have had the help of many wonderful people to guide me and support me through this doctoral process. First, I must acknowledge my parents, whom I feel lucky to have every single day. The level of friendship and trust I feel with my mother and father have enabled me to be myself and pursue my goals without fear of judgement. In the summer of 2005, I had an internship at Intel Corporation in Hillsboro, Oregon. I planned to drive from Michigan to Oregon so that I could have a car during the summer, and I asked my parents if they would join me on the road trip. When I tell people that I came up with this idea on my own, they think that I am the craziest person in the world. But I think this anecdote best represents how much I love and am loved by my parents. I am truly lucky. And, by the way, we had a wonderful time together.

Next, I cannot express the gratitude I feel for having married the best husband I could ask for. My husband Tom is my biggest fan, always encouraging me to be the best I can be. He genuinely feels happy and proud when I accomplish something, and knowing I have him in my corner makes the world an easier place to live in. When I was working on deadlines, Tom would cook meals for me and bring them to my desk as I stared at my computer screen, encourage me to go outside for fresh air when I

had closeted myself indoors for days, and give me unsolicited backrubs when I was at my most stressed. I am more grateful to him than he knows, and I am so glad he is my partner in life.

Of course, I must also thank my little sister, Leslie, for her support these last few years. I am several years older than Leslie, and have always been accustomed to taking care of her. As she has grown older herself, and the age gap feels narrowed, she has come to take care of me on occasion. I often turn to my sister first when I need cheering, and she is always successful in making me feel better. Her infectious bubbliness when I am down remind me that the world is a beautiful place and bring me back to my usual happy self. I am proud to call her my sister and my friend, and I am grateful for her love and support.

I would be in remiss if I did not also acknowledge my primary advisor, Steve Reinhardt. I truly believe I could not have asked for a better advisor; in fact, there are days when I am uncertain that I could have navigated the waters of graduate school without Steve's support. Steve created an environment that allowed me to get comfortable, grow, and learn; and he sometimes had to play the role of psychologist as well as advisor. The numbers don't lie — computer science can be a tough field for women, but if more advisors were like Steve I believe the attrition rate would be much lower. Steve, I thank you a hundred times for your role in my completion of this dissertation.

I must also thank the other members of my committee: Trev, who gamely stepped in as co-chair when Steve left the University, and advised me beyond my expectations even though I was not working on any project of his, and can always be counted on for a laugh or a wry comment; Scott, who never seemed annoyed that I often visited with his grad students and their group turtle, Bowser; Dennis, who was willing to step in and be on my committee when another member of my committee left the University; and to Ravi, my technical mentor during that summer at Intel, whom

I have always felt great affection for, and I regret not keeping in better touch with since my internship.

During my days as a graduate student, I also had a great amount of peer support. Within my research group, I could not have asked for a greater mentor than Nate Binkert, who taught me so very much about computers. He literally taught me nearly all of the tools I needed to be an effective researcher. Without him, I would not be who I am today. Ali Saidi, with whom I share great camaraderie, was a pleasure to have as an officemate, and he made dreary days in the lab go by much more quickly. I should also mention Kevin Lim, Ron Dreslinski, Korey Sewell, and Gabe Black, with whom I enjoyed working in our little office in CSE on developing M5 and other projects.

I am also incredibly grateful to my friends Victoria Fossum and Christine Vehar Jutte, with whom I shared a bond of being female graduate students in engineering entering the same year. We supported each other throughout the years, encouraging, venting, and propping up when the others were down, and I truly believe our little trio of PhDs was a major reason for all of us making it to the end. I love them both dearly and am so glad we met.

Gratitude also goes to Mike Chu, who was not only a friend during our tenure in the CSE department at Michigan, but who also helped me when I needed it as I tried to juggle finishing this dissertation with a full time job in Bellevue, Washington, where he now lives as well.

Thanks must also be given to the Lucent Foundation for funding the first four academic years of my PhD; and the Intel Foundation, for funding the last two, along with providing Joel Emer as my mentor, who is an incredible architect and took the time to speak with me and invite me to a talk at Intel Hudson. I am grateful for the opportunities and advice he provided.

Finally, I must thank my friends. I cannot name them all, but special thanks must

go to Karin Kin and Catherine Holahan, who are two of my dearest friends, and who would love me even if I never finished this dissertation.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES x

LIST OF TABLES xii

CHAPTER

1 Introduction 1

1.1 Why CMPs? 2

1.2 Large Scale CMPs and the Cache Hierarchy 5

1.3 Resource Allocation for a Shared Cache 7

1.4 Thesis Statement 9

1.5 Contributions 10

1.6 Organization 11

2 Background 12

2.1 The Basics of Cache Operation 13

2.1.1 The Shift to Multicore 14

2.2 Shared Cache Performance Optimization 15

2.2.1 Partitioning for Optimality 15

2.2.2 Other Approaches to Resource Management 20
2.3 Quality of Service 24
2.4 Other Related Work 27
2.4.1 Operating systems involvement in cache management 27
2.4.2 Latency optimizations for distributed caches 27
2.5 Conclusion 28

3 Shared Cache Metrics 29
3.1 Methodology 32
3.1.1 Performance Target Selection 32
3.1.2 Optimal Cache Allocations 35
3.1.3 CMP Thread Model 40
3.2 Results 41
3.2.1 Optimal Partitions 41
3.2.2 Comparing the *"-isms"* 45
3.2.3 Baseline Weighting Choices 48
3.2.4 Policy Metrics 49
3.2.5 Policy Evaluation 51
3.3 Conclusions 55

4 Machine Learning Techniques for Discovering Salient Characteristics of Poor Performance 58
4.1 Methodological Approach 61
4.2 Ridge Regression 62
4.2.1 Background 62
4.2.2 Isolating Features of Poor Performance 67
4.2.3 Evaluation Environment 71
4.2.4 Results 74

4.2.5 Conclusions 77
4.3 Decision Tree Analysis 78
4.3.1 Background 78
4.3.2 Isolating Features of Poor Performance 81
4.3.3 Evaluation Environment 83
4.3.4 Results 84
4.4 Conclusions 90
5 Scalable Lightweight Adaptive Management 92
5.1 SLAM 95
5.1.1 Detecting Poor Utilization 96
5.1.2 Mitigating Poor Utilization with Throttling 99
5.1.3 Honing SLAM 99
5.1.4 Using SLAM for QoS 102
5.2 Methodology 103
5.2.1 Metrics 104
5.2.2 Benchmarks 104
5.2.3 Simulation Environment 107
5.3 Results 108
5.3.1 Detailed Studies 108
5.3.2 Comprehensive Studies 110
5.3.3 QoS 112
5.4 Conclusions 117
6 Conclusions and Future Work 119
BIBLIOGRAPHY 123

LIST OF FIGURES

Figure

1.1 LCMP cache sharing . . . 5
3.1 Miss Rate Curves . . . 38
3.2 Two-application Partitioning Example . . . 41
3.3 Four-application Partitioning Example . . . 42
3.4 Partition Variation . . . 43
3.5 Communist vs. Utilitarian Histograms . . . 45
3.6 Utopian Histograms . . . 48
3.7 Baseline Weighting Choices . . . 49
3.8 Policy Correlations . . . 51
3.9 LRU vs. Perfect Raw MPC Partitioning . . . 52
3.10 Policy Evalutaions . . . 54
4.1 Sample Workload Plot . . . 70
4.2 Stacked Histogram Results . . . 76
4.3 Decision Tree: YAGGA Feature Set . . . 84
4.4 Decision Tree: New Feature Set . . . 85
4.5 Reduced Feature Decision Tree . . . 87
5.1 SLAM Intuition . . . 97
5.2 Varying MPAC Threshold . . . 100
5.3 SLAM with Dueling . . . 101
5.4 MPKI Graph . . . 105
5.5 MPKI for SPEC2000 Benchmarks . . . 107
5.6 Detailed Results . . . 109

5.7 Geometric Mean Results 110
5.8 Varying Cache Size 111
5.9 Scalability Results 112
5.10 Strawman Proof of Concept 114
5.11 QoS Results 116

LIST OF TABLES

Table

2.1 Cache Partitioning Proposals 16

3.1 Performance Targets 33

3.2 16 workloads 36

4.1 Sample Supervised Learning Data 59

4.2 214 Member Feature Set 67

4.3 125 Member Feature Set 82

4.4 Learning Results 90

5.1 Workload Mixes 106

5.2 Service Level Thresholds 113

CHAPTER 1

Introduction

The processor industry has recently undergone a rapid and dramatic shift from uniprocessor systems to chip multiprocessor (CMP) systems, where multiple compute cores reside on a single silicon die. Intel's latest notebook/desktop processor, the Core i7, has four cores on a chip [27], while their latest Xeon server processors, the 7400 series, have up to six cores on a single die [28]. Similarly, AMD has gone the multicore trajectory with six cores announced for the Phenom on the desktop, and six cores currently in the latest Opteron server offering [2, 3]. Sun's UltraSPARC T2 processor, known as the Niagara 2, is aimed at the server market and has eight cores on a single chip [59]. This, when as late as 2005, the desktop domain was universally uniprocessor, with each silicon chip having only a single core on it. Not only has the industry rapidly moved to a multicore paradigm, but there is ample evidence that the number of cores on each chip will continue to increase, as major vendors continually announce plans for chips with more and more cores.

On these CMPs, there are a number of resources that, for flexibility and/or cost reasons, are not replicated for each core. These resources are thus shared between all cores on the platform. This situation naturally leads to potential resource contention issues, and the problem is exacerbated when the number of cores on a chip increases. One of the primary resources vulnerable to contention issues is the on-chip cache. In

a uniprocessor system, the stream of memory accesses going from the CPU to the cache comes from a single application at a time, while in a CMP system the accesses are finely multiplexed between the simultaneously running threads, which may not be cooperative. The fundamental issue is one of destructive interference, where oblivious and disparate threads may disrupt the cache usage of the other threads sharing the cache.

This dissertation specifically examines the effects of contention for the shared on-chip cache resource in large-scale chip multiprocessors. As CMPs continue to scale, the potential for problems in resource allocation only increases. Additionally, as multi-core offerings increasingly become multi-threaded, as Sun's Niagara and Intel's Nehalem processors are, the issue of cache contention becomes even more dramatic when considering hardware threads, and not just cores.

Alongside contention issues, it is unclear how best to measure performance for a cache shared by large numbers of threads. Given that the resource is shared, there is a tension between overall good and the welfare of individuals. In this dissertation I present an analysis of ways to measure performance in a CMP system, present an analytical process based on machine learning techniques to examine the nature of sharing problems, and introduce a highly scalable mechanism for managing a shared cache to mitigate highly destructive sharing situations. This mechanism is also easily extensible to providing differentiated quality of service in shared caches.

1.1 Why CMPs?

In the continual march towards higher performance computers, there have been two overarching methods for extracting improvement: higher clock speeds (*i.e.*, working faster) and greater parallelism (*i.e.*, doing more at once). Combining these two methods has yielded the spectacular rise in computing performance in the last sev-

eral decades. Historically, achieving greater parallelism has meant finding ways to extract what is called instruction level parallelism (ILP) from running applications, *i.e.*, finding instructions from the same instruction stream to execute simultaneously. Designers have squeezed out more and more ILP over the years using techniques like out-of-order execution, having large instruction windows, using branch predictors, and superscalar execution in order to find more instructions to have in flight at the same time.

However, this two-pronged approach has become problematic in recent years due to thermal issues. Each time a transistor switches, it uses energy, resulting in power usage as shown by Equation 1.1, where P is power, C is the capacitance discharged at every transistor, V is the supply voltage used to power the transistor, and f is the operating frequency:

$$P \propto CV^2 f \tag{1.1}$$

At the same time, maximum operating frequency is roughly linear in V:

$$f \propto V \tag{1.2}$$

Thus, increasing frequency of operation causes roughly *cubic* increases in power. We have reached the point where further increases in power are infeasible, and as Mudge tells us, reducing power consumption should be a “first class” design constraint in computer architecture [44].

At the same time, extracting ILP is rapidly coming to a point of diminishing returns, for several reasons. First, the low-hanging fruit is has been picked, and it is becoming more and more difficult to eke out smaller and smaller increases in ILP. Second, adding more ILP-improving widgets generally means much higher power consumption; meanwhile reducing power consumption is now an important design consideration, as just described. Moore’s Law is still providing designers with more

transistors at each generation, but these transistors can no longer be used primarily to create more ILP-extracting widgets. These additional transistors can, however, be used for extra processor cores on a die.

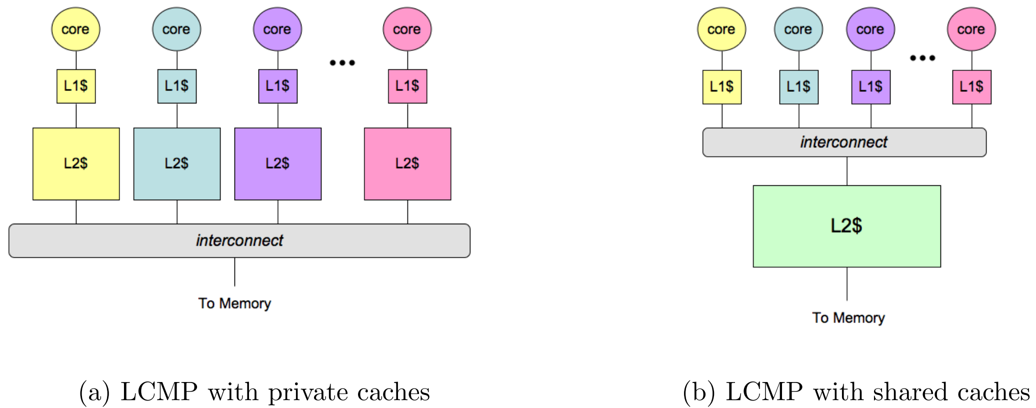
Putting multiple compute cores on a die has a number of effects. The first is that, in theory, a given task T could be completed in $1/Nth$ the time of a single-thread machine by merely splitting the task into N equal pieces and running it on N threads simultaneously. The second is that given this scenario, a chip with N simultaneous threads could reduce its operating frequency by $1/Nth$, and still finish T in the same amount of time as a single-thread machine. Recall the cubic relationship between frequency and power consumption, and suddenly multicore is a means to get off the trajectory of exponential relationships of power to performance. However, there is the practical consideration that it is much easier said than done to split a task into perfectly overlapping pieces. The amenability of a task or program to be easily subdivided into independent chunks is called Thread Level Parallelism (TLP).

The shift to CMPs has thus taken different trajectories. For the desktop/notebook domain, the “multi” in multicore is on a small scale. Intel’s Core line now has four relatively powerful cores on a chip, owing to the fact that desktop applications continue to have plentiful ILP but little TLP. In the server domain, however, TLP is abundant but there is much less ILP available [19]. In this market, it makes sense to have larger-scale CMPs, where many cores share a single die to maximize throughput rather than single-thread performance. Sun’s latest Niagara is a 64-way multi-threaded processor, while Intel’s plans for their Nehalem architecture to have up to eight cores with two threads each, or 16 simultaneous threads. I expect this number to increase in the coming decade, and the remainder of this dissertation focuses on large server-scale platforms.

1.2 Large Scale CMPs and the Cache Hierarchy

This dissertation focuses on the design and performance of the cache hierarchy for a large-scale CMP (LCMP). The most naive and straightforward design is to have all levels of the cache be private to their own cores; *i.e.*, a core cannot utilize cache space that does not belong to it. This scheme is demonstrated in Figure 1.1a in a CMP with a two-level hierarchy.

The downfall to this approach is the lack of flexibility. If core A's particular sub-hierarchy is not being fully utilized due to the nature of the application running on that core, while core B's sub-hierarchy is being taxed to the limit, there is no flexibility to allow for core B to use the unused space owned by core A. Even worse, in CMP designs where a core is shut down for power savings, the cache attached to that core is totally useless.



(a) LCMP with private caches (b) LCMP with shared caches

Figure 1.1: LCMP cache sharing - On the left is a naive cache hierarchy implementation for LCMPs. Each core has a private cache hierarchy of several levels all to itself. On the right is a more flexible implementation, where each core has potential access to the entire L2 cache.

Given the likelihood of this scenario, the natural next step is to consider shared caches. If all cores can access any available cache on the platform, flexibility is maximized and the likelihood of unused blocks is limited. However, this flexibility

also comes at a cost in terms of access times. If the L1 cache were shared across a platform, L1 access and hit times – vital to the performance of a platform – would be negatively affected. Thus, a typical platform will have private caches closer to the core, in order to preserve small sizes, close proximity, and fast access and hit times; but have shared caches closer to memory, where these latencies are not so tightly constrained. An example of this style of platform with a two-level cache hierarchy is shown in Figure 1.1b. This thesis focuses on this two-level topology.

The reasons for sharing a last-level cache (LLC) do not stop here. First, different software threads from server workloads often run the same binary but on different data, so that instruction addresses are shared between threads. Having every instruction address be replicated across all (private) last-level caches seems intuitively wasteful. Having a single copy of instruction data reside on the die to be shared amongst relevant software threads makes much more sense. Furthermore, many servers are operated as virtual machine hosts, and different clients who may be running the same guest operating system can also run the same operating system binary [61]. Both of these scenarios involve read-only instruction addresses, but studies show that there is also data sharing in a number of commercial server applications, where data is being passed between different threads for reading as well as writing [19, 36]. By sharing a cache, this shared data can be kept coherent without coherence traffic, which is yet another reason for sharing.

For these reasons, this shared last-level cache approach is used by Sun's Niagara 2, which has a shared L2 cache across all 64 threads on the platform; as well as all of Intel's multicore processors in a scheme known as Smart Cache, where the L3 is shared by all cores; and also AMD's multicore machines in a scheme known as AMD Balanced Smart Cache.

1.3 Resource Allocation for a Shared Cache

As with any shared resource, there comes a risk for contention issues and/or sub-optimal allocation. Caches as we know them today do not have explicit mechanisms to ensure fairness amongst sharers; indeed typical caches today do not even track which accesses are on behalf of which thread — thus all accesses from all threads are treated equally. The thread-blind nature of caches makes it difficult to perform any regulation of resource usage, and leaves the cache ripe for potential sharing problems.

Destructive interference occurs when threads spend their time evicting each others blocks and waiting while bringing their own blocks back into the cache rather than making forward progress. The lack of cooperation among threads can easily create situations such as these, particularly in the presence of streaming workloads, which stream their way through large numbers of accesses and fill the cache (thus evicting others' data) with blocks that will not be used again.

These scenarios are likely to be exacerbated and become more common as the number of threads on a platform increase, making protection mechanisms to circumvent such issues a necessary part of shared cache design. The essential problem is that of resource allocation—finding a way to intelligently allocate cache space amongst different threads in order to provide good performance.

The classic approach to solving resource allocation uses marginal cost analysis [21], which utilizes marginal gain calculations with incremental increases in resources to determine optimal allocations. There are several flavors of marginal cost analysis algorithms, depending on domain specifics such as whether the marginal gain function is convex or concave, whether the resource being allocated is continuous or integer, *etc*. However, all of these algorithms are compute intensive and assume a high level of abstraction when performing these calculations.

There have been a number of attempts [60, 57, 48], however, to translate these algorithms into cache management heuristics, where the general structure of the so-

lution follows a "assign, monitor/enforce, and adjust" format. Execution is divided into epochs, where re-allocations occur in between epochs, which cannot be too short in order to capture the full effect of the last re-allocation, but must be short enough to be nimble and capture changes in application behavior. Convergence to a solution would be tricky enough for four or eight threads, but convergence with hundreds of agents will be very difficult if not impossible under current design constraints. I will discuss these approaches in further detail in Chapter 2.

The distinct nature of the particulars of a cache resource allocation make the problem inappropriate for marginal cost analysis. There are few points to keep in mind for the shared cache resource allocation problem:

1. There is a distinct latency between the time cache space is given or taken away and the time to know whether that change has the desired effect. The implication is that changes in allocation cannot be made too rapidly because the effects of each change must be known before making the next change.

2. The nature of an application's memory access stream may change over time. The implication here is that not only would convergence to an optimal allocation take a long time due to the latency mentioned above, but the goal may itself be a moving target due to the transient nature of application behavior.

3. Hit times to a cache (*i.e.*, the amount of time it takes for a requested block that resides in the cache to be returned to the CPU), are a vitally important performance characteristic and cannot be sacrificed. In other words, the allocation decision mechanism must not affect hit times — any computation or calculation must not be on the critical path.

4. The cache is a highly optimized piece of hardware on a chip; ideally whatever mechanism is proposed would incur minimal changes to cache design.

5. In the domain of LCMPs, any mechanism would need to scale well. Many mechanisms that would have reasonable complexity in the small-scale can quickly become untenable in the large.

Based on this characterization of the problem, it is evident that marginal cost analysis is an overly complex approach to large-scale cache allocation problems.

I propose another approach that is “correlatively optimal”, which, rather than explicitly searching for optimal, exploits some characteristic that will trend towards a good solution in the general case. I take as my inspiration the simple Least Recently Used (LRU) replacement algorithm that dictates cache block allocation in caches today. Current caches to not try to explicitly perform optimal replacement in a cache; rather they use LRU to exploit the temporal locality properties of most programs, resulting in generally good performance. Similarly, cache prefetching algorithms do not try to explicitly perform perfect prefetching; instead they exploit some known tendencies of memory access patterns and prefetch cache blocks based on that principle. In both of these examples, the cost/benefit ratio falls strongly in favor of the simple solution that provides generally good (though not optimal) performance, rather than a complex solution that might provide minimal added performance benefit.

1.4 Thesis Statement

This thesis examines the interaction between cache space and system performance in the domain of large-scale chip multiprocessors in order to inform the design of a scalable, feasible shared cache management mechanism that provides good performance. This thesis demonstrates that selecting a specific definition of “good performance” for shared caches is inherently difficult, given that there are many competing definitions of “good” that can be quite incompatible with one another. As a result, a preferable approach is to search for correlatively optimal characteristics that would put a floor

on the performance of pathologically poor situations under any measure, instead of focusing on a single particular metric of performance.

This dissertation uses offline machine learning techniques to effectively extract salient characteristics in poor cache-sharing situations. These characteristics can be used to predict poor performance in previously unseen environs, *i.e.*, with application mixes not used in training the machine learning algorithm.

This approach to building a mechanism to improve cache sharing is validated by the success of SLAM (Scalable, Lightweight, Adaptive Management), a framework for shared caches which can detect and mitigate pathological cache sharing situations, designed with the information yielded from the machine learning techniques. SLAM demonstrates the validity of using correlatively optimal characteristics to shape cache allocation rather than a complex and explicit search for optimal; SLAM is scalable, feasible, has light-weight implementation requirements, and good performance results. In addition, SLAM is a natural candidate for providing differentiated quality of service (QoS) in caches, whereby certain threads are given preferential treatment because of higher priority.

1.5 Contributions

The contributions of this thesis include:

- an exploration of the objective function space for shared caches resulting in meaningful insights into the selection of cache design objectives;
- demonstrating the utility of existing machine learning techniques for effective offline analysis of the cache resource allocation problem;
- a comparison of two machine learning techniques, one effective, the other not, for this particular problem space;

- the presentation of a mechanism to provide scalable fairness in shared caches, suitable for implementation even in large-scale chip multiprocessors, costing only tens of bits per thread, and is actually feasible in present implementations;
- the presentation of two performance measurement techniques called MPAC (Misses Per Access Counter) and RIT (Relative Insertion Tracker) for tracking shared cache performance characteristics, which are the cornerstone of SLAM and could be useful in other system optimization domains;
- the demonstration of the SLAM framework as a natural provider of differentiated quality of service in the cache.

1.6 Organization

The remainder of this thesis is organized as follows. Chapter 2 discusses some background and related work for the cache resource allocation problem. Chapter 3 discusses my examination of the objective function and metric space for shared caches. Chapter 4 discusses the machine learning techniques used for the analysis in this dissertation. Chapter 5 discusses the major contribution of this thesis, the SLAM framework for managing shared caches. I conclude and discuss future work in Chapter 6.

CHAPTER 2

Background

A modern computer performs its tasks by following a program written prior to the execution of the task. To do this, a computer must access the stored program instructions, then execute the instruction in the CPU, which may involve accessing previously stored data. When finished, the CPU may be instructed to store results back to memory. Thus, there can be non-trivial amounts of data movement between the CPU and stored memory.

Unfortunately, from the earliest days of computing there has been a gap between processor instruction execution speeds and memory speeds. In other words, memory cannot keep up fast enough to feed the CPU with instructions nor the data required to execute the instructions.

To address this problem, designers began using caches (small, fast buffer memories between the CPU and memory) in order to try to hold portions of memory that will be used in the near future [52]. The IBM System/360 Model 85 is one of the first documented machines to utilize caching [40], and from there, caches have moved from being external to the silicon die to residing on-chip, increased in size, and changed from a mono-level system to two and three levels of cache on a chip. The cache has become an integral part of system design, and plays a large role in the overall performance of a computational system, particularly as the disparity between CPU

speeds and memory speeds has continued to increase.

2.1 The Basics of Cache Operation

Vital to the efficacy of these high-speed cache buffers is the selection of the subset of addresses to place in the buffers. At the crux of the question is not which addresses to bring into the cache, but *what to replace* when the cache is full. Research involving the paging of virtual memories in multiprogrammed environments had already led to the concept of a *working set* (*i.e*, a set of data of which quick access times would provide efficient performance of the corresponding process) [14]. Denning, who coined the term working set, posited that pages used in the recent past were a good predictor for pages to be used in the near future. Around the same time, Belady performed a study of replacement heuristics for virtual memory and found that employing a *least recently used* (LRU) algorithm, which evicted pages that had been used furthest in the past, generally provided performance closest to optimal [6]. These two ideas together led to the *locality principle*, which has become a cornerstone of resource allocation resolution in virtually all of computer science [15]. These principles apply just as well to cache memories as virtual memories.

For the purposes of cache management, the locality principle means that memory addresses from a given application tend to exhibit both temporal and spatial locality. Temporal locality implies that an address used now is likely to be used again in the near future. Spatial locality implies that an address used now is likely to be near or adjacent to an address to be used in the future. The intuition behind the success of the LRU algorithm is how it takes advantage of the temporal locality principle by keeping the most recently used address blocks close to memory.

Since fully associative memories tend to be both expensive and slow [52], caches are generally built as smaller sets of associative memories. Any address requested in

the cache is indexed into only a single associative set, and any address block to be replaced to make room for the incoming block is taken solely from within this set. The associativity of a cache indicates the number of blocks within a set, such that an n-way associative cache would have n blocks within each set, and the LRU block refers to least recently used block in that set of n blocks.

Addresses are used to index into a set based on a subset of the address bits; the remainder is stored in the cache along with the affiliated data in a *tag array* as an address tag. Address tags are used to distinguish between the multiple addresses that can index into a set [52].

Modern caches are generally all built as described, and this thesis evaluates all proposals under the lens of feasibility in this paradigm.

2.1.1 The Shift to Multicore

LRU is quite effective as a replacement policy for caches in uniprocessor systems because LRU acts as a predictor for which address blocks are to be used in the near future *within the stream of instructions from its affiliated process.* However, once a cache becomes shared among multiple simultaneous threads of execution, LRU may not be as effective because the stream of accesses to the cache is no longer from a single source. Rather, accesses can be finely interleaved among all simultaneous processes. As a result, researchers have moved their focus towards making caches thread-aware and differentiating between accesses from different threads in order to make space allocations decisions in the context of a shared cache [12, 18, 33, 35, 37, 48, 54, 57, 64, 65, 66]. Whereas the goal of the LRU replacement policy is to try to ensure that blocks to be used in the near future are kept close to the processor, the goal of any new mechanism has to balance the needs of multiple threads.

Likely the first piece of work on studying a cache shared between separate streams of accesses is by Stone, *et al.* While not involving a multicore platform, this study

developed a model for evaluating the optimal allocation of cache memory between distinct streams of accesses, noting that the overall miss rate of a cache shared by multiple streams is just the average of the miss rates of the contributing streams [55]. Thus, finding an optimal allocation among streams requires modeling a function of the form

$$F(CacheAllocation) = MissRate$$

for each stream, summing them, and solving for the minimum. Additionally, Stone, *et al.* performed empirical studies involving optimizing allocation for data streams versus instructions streams in a single application, and found that empirically, LRU did remarkably well relative to optimal—but they were unable to mathematically bound the delta between LRU and optimal in a rigorous manner.

2.2 Shared Cache Performance Optimization

With the move towards shared caches and the potential for inter-thread interference in cache usage, a number of proposals have emerged for shared cache performance optimization, many using the marginal gain analysis approach described in Chapter 1. The various proposals are described below.

2.2.1 Partitioning for Optimality

Cache partitioning is a prominent strategy for shared cache management, in which a scheme tries to dynamically determine a specific optimal cache partition for each hardware thread [57, 65, 18, 35, 48]. Each scheme monitors and enforces cache usage, and then adjust partitions based on recorded measurements on an epoch granularity. These schemes each have four attributes: 1) measurement techniques, whereby marginal utility is measured over an epoch, 2) partition assignment heuristics, whereby data from the measurement techniques are used to assign partition

Proposal	Measurement	Partitioning Heuristic	Partition Enforcement	Proposed Epoch Length
Dynamic Partitioning [57]	Per-way Hit Counters	Per-block Greedy	Modified LRU	5M cycles
Fast and Fair [65]	Per-Thread Shadow Tags	256KB Greedy	Modified LRU	100M cycles
Partitioning-aware LRU [17]	Per-Thread Victim Tags	Local Search	Modified LRU	2000 misses
Fair Caching [35]	Profile + Miss Rates	Local Search	Modified LRU	10K accesses
Utility Cache Partitioning [48]	Per-Thread UMON	Per-block Greedy	Modified LRU	5M cycles

Table 2.1: Cache Partitioning Proposals - This table describes the approaches for several optimal cache partitioning proposals with respect to four major attributes..

sizes to each hardware thread for the next epoch, 3) partition enforcement techniques, whereby the assignments are enforced during an epoch, and 4) epoch length. Table 2.1 describes the approaches for these schemes.

Dynamic Partitioning, proposed by Suh, *et al.* use per-thread, per-way hit counters to approximate the marginal gain of each way for each thread. Each time a hit occurs in a particular way for a particular thread, the affiliated counter is incremented. Thus each counter approximately represents the marginal gain in cache hits for its thread by having that way. Every 5 million cycles, it uses a greedy algorithm to partition the cache, block by block, to individual sharers based on the measured marginal gains. A modified LRU policy (described later in this section) enforces partitions [57].

Fast and Fair, proposed by Yeh and Sherwood, uses per-thread shadow tags, *i.e.*, a complete set of duplicated tag arrays, for each hardware thread in the system to track what *would* happen to the hit rates of each thread assuming sole ownership of the cache. Every 100 million cycles, the cache is partitioned in chunks of 256kB to individual sharers based on the shadow tag measurements. A modified LRU policy enforces partitions [65].

Partitioning-aware LRU, proposed by Dybdahl, *et al.* use per-thread victim tags to measure the marginal gain for each thread of having one more way. Each cache set has a single extra tag associated with it for storing the most recent eviction; hits to this victim tag indicate that one more way allocation would have resulted in more hits. There is also a counter per thread for the LRU way of the cache, which is also

incremented upon hits in that way. Every 2000 misses, if the thread with the most to gain from having an additional way as measured by the victim tag counter exceeds the thread with the most to lose from having its LRU way taken away, an additional block per set is allocated for the gaining thread for the next epoch. A modified LRU policy enforces partitions [17].

Fair Caching, proposed by Kim, *et al.* focus on maximizing fairness in a cache rather than minimizing miss rates; thus the measurement technique is quite different. Applications are profiled beforehand running alone on a shared platform to get baseline miss rates. At run time, shared miss rates are tracked and compared against these baseline rates. Every 10,000 accesses, blocks are taken from the thread performing the best relative to baseline rates and given to threads performing the worst. A modified LRU policy enforces partitions [35].

Utility Cache Partitioning, proposed by Qureshi and Patt, uses a monitoring mechanism called UMON (Utility Monitor) for each hardware thread. Essentially, a UMON is a sampled version of the shadow tags used by the Fast and Fair proposal; instead of having a full duplicate of tags for each thread, only a subset of sets are used for measurement in order to save tag storage space. The subset is presumed to be representative of the entire cache. Every 5 million cycles, a greedy algorithm allocates blocks to each thread, block by block, based on the measurements of the UMONs. A modified LRU policy enforces partitions [48].

An analysis of the overheads of each portion of each technique is described further below.

Measurement techniques: Dynamic Partitioning relies on per-way hit counters to measure the approximate marginal gain of a way for a thread. The technique is lightweight (one counter per way, per thread), but is prone to inaccuracies because of noise from inter-thread interference, which would increase as a platform scales to greater numbers of threads. Fast and Fair, on the other hand, is a storage heavyweight

and cites 32-way shadow tags per thread to measure the exact marginal gains of 256KB chunks of cache, which they estimate to be a 10% area overhead on a 4-thread platform. Partitioning-aware LRU needs two counters and one register per thread, plus a single shadow tag per set per thread, representing the most recent eviction of of each set, which is relatively lightweight. Fair Caching requires a static profile of a thread's entire execution, and miss rates are then measured online. While this is low storage overhead, in practice the profiling is unlikely. Finally, the UMONs used in Utility Cache Partitioning (UCP) are sampled shadow tags, *i.e.*, shadow tags for a subset of the sets in a cache. There is one UMON per thread in a system. Out of these proposals, only Partitioning-aware LRU, with a measurement storage overhead of $3N$ counters/registers and $TagWidth * NumSets * N$ bits of shadow tag storage; and UCP, with a measurement storage overhead of $Assoc * N$ counters and $Assoc * TagWidth * 32 * N$ bits for shadow tags, could conceivably be scaled into the hundreds of threads.

Partitioning heuristics: The partitioning heuristics used by these schemes can be divided into two camps — greedy algorithms and local search algorithms. The schemes with greedy algorithms attempt to achieve the optimal partition based on measurements from the previous epoch, while the local search algorithms only perturb allocations slightly, such that each epoch is only a step in the search for convergence to an optimal partition. The computation required for the greedy algorithms as number of threads increases simply does not scale; indeed most of the proposals were tested on two-thread systems. The exception is Fast and Fair, which was evaluated on a 4-thread system and mitigates potential scaling issues by allocating on a granularity of 256KB at a time rather than on a block granularity, but is still not scalable. The schemes employing local search require much simpler computation, but are slow to converge, even more so as platforms scale. Both Fair Caching and Partitioning-aware LRU make a single pass through their measurements from the previous epoch and

trade block allocation from the best and worst performers. Even on a platform with 64 threads, convergence could be painfully slow.

Partition enforcement: One thing all the schemes in Table 2.1 share is the use of a modified LRU replacement policy, whereby all blocks in a cache are tagged with a thread-ID bitfield, amounting to $log_2(N)$ bits for every block in the cache, where is N is the number of hardware threads. This is significant overhead as the value of N and cache sizes increase. The cache usage of every thread is tracked by incrementing its usage counter when a line is brought into the cache, and decrementing it when a line belonging to it is evicted. In order to maintain assigned allocations, upon replacement the LRU block *of the desired thread* is found and evicted. While replacement may not be on the critical path, this departure is not a trivial modification in modern caches, which by and large employ pseudo-LRU algorithms because true LRU is too high overhead for high-associativity caches. PLRU algorithms (described by Al-Zoubi, *et al.* [1]) take shortcuts to evict a block close to but not necessarily LRU, but the problem characteristics that are leveraged to achieve these shortcuts disappear when seeking the LRU block belonging to a certain thread, increasing the costliness of the replacement algorithm.

Worthy of note here is a proposal from Xie and Loh [64], which is essentially an alternative to partition enforcement and could be used in conjunction with any of the measurement techniques, partitioning heuristics, or epoch lengths described in Table 2.1. The authors present a novel way to pseudo-partition caches in order to avoid bit-tagging overheads from exact cache tracking. Promotion/Insertion Pseudo-Partitioning effects allocations close to prescribed partitions by manipulating the insertion and promotion policies of the cache, such that blocks can be placed in arbitrary locations in a set, and can be promoted arbitrary numbers of spots on an access, rather than always going to the MRU position. In practice, and particularly with respect to pseudo-LRU algorithms, this is very difficult to implement.

Epoch length: Epoch length admittedly should differ from between local search heuristics and greedy algorithms. In cases of local search, the algorithm takes small steps towards optimal every iteration; thus epoch length should be relatively short in the hopes that convergence could happen within a reasonable number of iterations. But since each iteration of the greedy algorithm is computationally intensive and hopes to find an optimal allocation each time, the epoch length should be longer in order to amortize computation cost, as well as measure enough information to make a good decision. However, an epoch that is too long risks missing phase changes in program behavior or context switches. In practice it is difficult to determine an epoch length that is appropriate for the plethora of workloads that could be potentially run on a CMP platform. Sherwood *et al.* [51] show that it is not possible to have a universal phase length that applies to workloads in a general sense and, even within an application, phase lengths can vary. Su *et al.* [56] find that, while reconfigurations of a shared cache based on optimal phase lengths determined offline can provide good performance, fixing an optimal phase length is not effective across workloads, nor is it easy to dynamically predict phase lengths that approach the offline optimal.

2.2.2 Other Approaches to Resource Management

In general, partitioning for optimality is unrealistic in the large scale for several reasons:

1. The convergence to an optimal solution in a system with many agents is likely to be overly complex and take multiple iterations (of epochs taking non-trivial amounts of time) to approach optimality.

2. At the same time, the nature of the domain being optimized is constantly shifting—in the large scale, phase changes and context switches are likely to occur before convergence.

3. Enforcement is also a non-trivial task, generally requiring the tracking of each thread's usage and the restriction of which thread's data can be replaced in the cache, which would represent a significant departure from the functionality of current caches.

4. Research has also shown that determining appropriate epoch lengths is not a simple task [51, 56], an area of research in and of itself.

Fundamentally, optimality is precise, and being precise when juggling the needs of N agents, where N is large, is bound to be complex and difficult. This is exacerbated when enforcing precision is not totally straightforward in practice, and the precise definition of optimality is not static. There are a number of other papers that approach shared cache resource allocation in a different way than the partitioning paradigm.

Chang and Sohi propose Cooperative Cache Partitioning (CCP) [12]. CCP consists of a hybrid approach, using Cooperative Caching(CC) [11] and Multiple Time-sharing Partitions (MTP). CC approaches the cache sharing problem from a different level of abstraction. It is fundamentally a LRU-based scheme and does not evaluate the relationship between cache space and performance, rather it is a mechanism for optimizing latencies within a distributed shared cache structure. MTP, however, is a partitioning approach that time-shares cache partitions rather than using a single partition allocation. While MTP is not exactly a proposal to partition for optimality, it maintains many of the drawbacks of the schemes described in the previous section. MTP essentially cycles threads through large, unfair allocations of cache so that over a larger time quantum, macro-level fairness is achieved. Thus MTP has two forms of epochs—the first is the macro-epoch, which identifies the time-share partitions and how long each should run, and the micro-epoch, which is the time of each time-share. MTP is an intriguing concept but can take very long periods of time not only to adapt to changes in program behavior due to the necessarily long nature of the macro-epoch,

but will be difficult to scale. Since all threads "take turns" at having generous cache allocations, the more hardware threads there are the longer each thread has to wait before getting its turn. The tension between having long enough micro-epochs and the resultant $O(N)$ increase in wait time before obtaining the generous cache allocations makes this an unlikely solution for large scale machines.

Srikantaiah, *et al.* present Adaptive Set Pinning [54], a mechanism to avoid misses that result from contention between hardware threads sharing a cache. Rather than focusing on space in general, they focus on addresses. The crux of their approach is to identify "hot blocks," blocks from a thread which are frequently accessed and the cause of the displacement of blocks belonging to other threads. To do this, they introduce *set pinning*, a way to pin a cache set to a particular hardware thread such that no references from other threads may displace anything from that set. Instead, references to pinned sets belonging to other threads are placed into a Processor Owned Private (POP) caches, which are basically small, associative caches operating under LRU replacement. In essence, these POP caches will capture the hot blocks of their associated threads, and since the sets of the main cache are all pinned to a certain thread, thrashing between threads is reduced to zero. The key to the performance of this scheme is judicious assignment of sets to threads, which Srikantaiah *et al.* do with saturating counters for each set roughly approximating hit/miss ratios of the pinned and unpinned threads. This mechanism does avoid the epoch-based approach of partitioning, while taking a fuzzier approach to partitioning by avoiding precise allocation prescriptions. However, this part-private, part-shared scheme necessitates a significant overhaul to current cache designs, while limiting cache usage flexibility. Not only do designers have to decide *a priori* the size of each private cache, but power-saving schemes that turn off cores when not in use would lose the ability to utilize the cache space in the POPs belonging to the powered down threads.

Jaleel *et al.* proposes a mechanism called Thread Aware Dynamic Insertion Policy (TADIP) for managing shared caches [33]. It is an extension of Qureshi and Patt's Dynamic Insertion Policy (DIP) [47]. Fundamentally, DIP is a simple mechanism for voting between the LRU replacement policy (which inserts new blocks into the Most Recently Used, or MRU, position), and a modified policy called BIP (Bimodal Insertion Policy) which inserts new blocks into the LRU position with high probability. The key observation in DIP is that a few sampled sets (empirically set at 32) dedicated to different policies can approximate the performance of that policy over the entire cache. Leveraging the knowledge that cache usage for a stream of accesses can be throttled by adopting LRU insertion, first used to deprioritize prefetching accesses [39], DIP chooses between normal operation and performing LRU insertion to improve cache performance in single-threaded platforms where the cache is not quite large enough to handle the footprint of the running application. TADIP is a natural extension of DIP for CMP platforms, where every hardware thread is assigned a pair of sampled sets (32 sets for each policy) to duel. TADIP samples the global miss rates of the cache between the two groups of pinned sets, which vary only the insertion policy for one thread. TADIP then selects the policy for that thread that yields a lower global miss rate, which that thread follows upon accesses to remaining sets. TADIP is the most scalable, feasible, and high-performing scheme for shared cache performance optimization in the literature.

Kron, Brooks, and Loh also extend Qureshi's DIP in a scheme they call Double-DIP [37]. The key observation for Double-DIP is that if a set is accessed by different threads at different rates, the more frequently accessing thread will very quickly push out the blocks of the lower frequency thread by virtue of constantly promoting its own blocks to the MRU position of the set. To mitigate this problem, a dueling mechanism is added to the original DIP to select between a standard MRU promotion policy, where a block moves to the MRU position every time it is accessed, or a less

aggressive promotion policy, where a is chosen to be promoted 1, 2, 4, or n steps towards MRU (where n is the associativity of the cache). This scheme avoids any requirement of knowing which accesses are from which threads, and uses only local information to lead to better performance; however it is unclear how this scheme would be implemented in an actual cache, which generally uses pseudo-LRU and is not literally a spectrum from LRU to MRU. Additionally, varied levels of promotion are not a trivial mechanism to implement in practice.

2.3 Quality of Service

Another aspect of shared caches is the idea of quality of service (QoS), most commonly used in the networking domain, where it is a mature concept. Cisco Systems defines network QoS in their *Internetworking Technology Handbook* [13] as:

> Quality of Service (QoS) refers to the capability of a network to provide better service to selected network traffic....The primary goal of QoS is provide priority....Also important is making sure that providing priority for one or more flows does not make other flows fail.

Essentially, networks are shared by numerous flows of network traffic, and QoS considerations guide the allocation of network resource among these flows. With the advent of virtualization in computing, where multiple distinct systems can each run multiple distinct programs on a single platform, there is an increased likelihood of non-cooperative processes running simultaneously on a CMP. In many commercial virtualization situations, some applications may be more important than others, and even in client environments, background processes may be less important than foreground processes. Thus, QoS is an emerging need in computing platforms as well.

As such, there have been a number of studies [31, 32, 23, 22] about the need for the ability to provide differentiated quality of service to different virtual machines

and present architectural frameworks to do so.

Iyer [31] describes the need for a priority-based resource allocation framework by illustrating the strong impacts threads could have on one another when sharing a platform.

Iyer, *et al.* [32] describe a comprehensive platform which enables a software layer to dictate several forms of QoS, as well as a taxonomy for describing them. *Resource Usage Metrics* (RUM) track the exact usages of each sharer on a resource, and employ the modified LRU replacement algorithm as listed in Table 2.1 and described in Section 2.2.1. A thread's RUM allowance could be dictated by the user, or be dictated by an algorithm trying to achieve QoS goals. Iyer's framework also describes *resource performance metrics* (RPM), which measure the performances of individual sharers and does local search optimization of the cache partition in order to satisfy the requirements set by the software layer. Finally, Iyer describes *overall performance metrics* (OPM), which allow for high priority threads to obtain more and more resources until the platform performance degrades to a certain level. While there is value in the taxonomy provided, the framework also has the same issues as optimal partitioning proposals involving epoch length, scalability of convergence, and enforcement/measurement overhead.

Rafique, *et al.* [50] propose an interface to allow for the setting of usage quotas for the cache via operating system and a hardware enforcement layer which is thread-aware. Essentially, it is a framework to enforce RUM with a thread-aware replacement policy, with a software interface to allow for software dictated allocations from arbitrary policies at the OS-level. Nesbit, *et al.* [45] have similar cache allocation enforcement within their framework of providing systemic QoS in a virtualized environment. Both rely on exact cache usage knowledge for enforcement.

Guo, *et al.* [23] proposes a detailed policy framework in which a CMP is able to assess its own ability for providing certain guarantees of performance for a given job,

and accept or reject the job based on this assessment. However, this work operates at a level different than this thesis. Even so, this work does presume cache allocation enforcement on the basis that each block is tagged with an owner bitfield.

In many ways, the QoS problem space as addressed by the above proposals faces the same obstacles as the shared cache optimization space; QoS just enforces allocations for a different end goal than for performance optimization. But underlying mechanisms for one can be used for another; likewise difficulties in implementation for one are shared by the other. Namely, the tracking of exact cache usage requires thread-IDs per block, and enforcement of exact allocations in the form of modified LRU are a burden in realistic hardware design.

As far as I know, all cache QoS frameworks in the literature require exact usage tracking of the cache in order to enforce exact allocations prescribed by some higher level policy, and is thus fundamentally limited by the need to attribute every cache block to an owner.

In this thesis, I make the case for not guaranteeing absolute levels of resources in for cache QoS because of the difficulties just listed. After all, even Cisco describes three different types of QoS levels: guaranteed service, differentiated service, and best effort service [13], where guaranteed service is an absolute reservation of resources, differentiated service provides preferential treatment to higher priority flows without absolute guarantees, and best effort makes no distinction between flows. Their *Internetworking Technology Handbook* also states, "Fundamentally, QoS enables you to provide better service to certain flows. This is done by either raising the priority of a flow or limiting the priority of another flow." The emphasis is not upon guaranteed allocation of resources. This thesis contends that the computing community should take a cue from the networking domain and focus on differentiated QoS rather than guaranteed QoS because not only is there a clear, established domain where customers accept such a model of service, but it is much simpler for platforms to be designed to

provide differentiation than exact resource guarantees.

2.4 Other Related Work

2.4.1 Operating systems involvement in cache management

Some studies move upward in the stack and utilize the operating system for managing the sharing of shared caches. Fedorova and Seltzer investigate using the OS scheduler to ensure that contentious processes are not run simultaneously [20], while Rafique, Lim, and Thottethodi present a mechanism for providing architectural support for OS-driven cache management to enforce partitions [50] (while not presenting a policy for generating partitions).

2.4.2 Latency optimizations for distributed caches

In distributed shared caches, the last level cache is distributed in that it is not a monolithic structure and is banked, but is shared in that any thread can have blocks in any bank. Accesses to a bank that is closer to a processor is lower-latency than accesses to a faraway bank, thus the aim for these studies is to make sure needed blocks are as close to their requesting processors as possible.

Beckmann, Marty, and Wood use dynamic monitoring techniques to assess the needs of running applications to decide whether to replicate data across banks [5]. The scheme, called Adaptive Selective Replication, determines the expected tradeoff between lower latencies and smaller effective cache size and makes its decision.

Other studies parameterize the level of private and shared use of each distributed bank [16, 66], adapting at run-time to maximize performance.

Cooperative Caching, presented by Chang and Sohi [11], uses private caching as its baseline but appends cooperative techniques to achieve the benefits of a shared cache. Cooperative techniques include the spilling of data from an overtaxed private

cache to an underutilized one, rather than evicting it from the chip entirely.

Qureshi takes this lateral spilling to another level with set dueling, the concept behind DIP, TADIP, and Double-DIP. In this scheme, a private cache is pegged either as a spiller (*i.e.*, belonging to a thread with a large cache footprint), or a receiver (*i.e.*, having capacity to spare for spiller cache victims), such that a cache cannot be both [49].

2.5 Conclusion

It is clear that while shared cache management is not a new area of study, there is opportunity for a scheme that scales, is feasible, provides good performance, and meets the emerging need for QoS in shared computing platforms. This thesis seeks to address this opportunity by first assessing the problem of metrics. In order to craft a good solution, a reasonable metric for measuring the quality of solution must be defined; in the area of shared caches there are a number of metrics to choose from.

The next chapter assesses this metric space in a rigorous manner, setting the stage for measurement for the remainder of this dissertation.

CHAPTER 3

Shared Cache Metrics

To begin studying the best approach to managing shared caches in a CMP context, the question is immediately raised as to what the *optimal* policy might be. Naturally, the answer to this question hinges on the objective for which we optimize. What is more important, maximal overall system performance, or fairness across threads? Given one of these goals, what specific metric do we seek to maximize or equalize? While the notion of performance in single-threaded uniprocessors is straightforward, multithreaded systems are amenable to multiple definitions, including instruction processing rates and miss rates, both raw (absolute) and weighted—and among weighted metrics, multiple weighting factors may be selected.

Thus, in a dissertation about shared cache management, I begin with a study about the impact of varying definitions of "optimal performance" on the partitioning of a shared cache among multiple threads. Specifically, I seek to answer the following questions:

1. To what extent does the definition of "optimal" impact the resulting partitions and performance?

2. In targeting one definition of "optimal", how well does the system fare with regard to other reasonable definitions? In particular, how do policies targeting

overall performance rate in terms of fairness, and how do policies targeting fairness rate in terms of overall performance?

3. If optimality is defined based on metrics that are not readily available online, are there online measurable metrics that correlate well that could be used to drive an online policy?

Borrowing loosely from economics terminology, I define four high-level policies for cache allocation. A "Communist" approach seeks to maximize fairness, ensuring that each thread bears an equal portion of the cache sharing penalty, or alternatively derives an equal benefit from the presence of the cache. The goal of a "Utilitarian" policy is to maximize the total benefit for the aggregate group – *e.g.*, by maximizing total throughput – without regard to individual thread performance. A "Utopian" policy attempts to balance throughput and fairness. Finally, a "Capitalist" cache policy is an unregulated free-for-all—the most common policy in use today.

Intuitively, either a Communist or Utilitarian policy seems preferable to a Capitalist policy, as these seek to maximize some desirable property, while the Capitalist policy makes no effort toward any form of "goodness". The Utopian policy seems even more preferable—as a balance of fairness and throughput it seems ideal. Studies have suggested the use of harmonic mean for effecting a balance between overall performance and fairness [41, 12], and this study evaluates harmonic mean as a way of implementing a Utopian policy.

In addition to the question of which policy to use, there is the issue of which metric to apply on these policies. Miss rate is a common metric for cache performance, but miss rate is not always proportional to perceived performance. Memory bandwidth is a scarce commodity in CMPs, so it may be appropriate for a Utilitarian policy to minimize total bandwidth. Throughput is often the bottom line, but raw IPC is not a very useful metric, since different threads have different levels of ILP (Instruction Level Parallelism, discussed in Chapter 1). Weighted IPCs [53] are attractive and

have been used in past studies, but raise the question of what weighting factor should be used.

Overall, I identify four distinct aspects of a cache partitioning policy: *performance targets*, *evaluation metrics*, *policies*, and *policy metrics*.

- A *performance target* is the end goal. A performance target may be for all threads to have the same miss rate, or that all threads should have the same IPC relative to some baseline configuration. The performance targets examined are: minimizing total memory bandwidth, maximizing weighted IPC with respect to various baselines, equalizing weighted IPC with respect to various baselines, and maximizing harmonic mean of weighted IPC with respect to various baselines.

- The *evaluation metric* is the metric used to express the performance target and to evaluate the extent to which the target is achieved. For example, if the target is to maximize total weighted IPC, then the evaluation metric is weighted IPC.

- A *policy* is the aspect of the cache implementation which makes allocation decisions. One well-known policy is LRU replacement, intended to achieve low miss rates in the general case. This study compares LRU, the general default policy, to policies that perform allocation based on metric values measured from application behavior. To keep the focus on fundamental behavior rather than specific implementations, these policies are modeling using static optimal partitioning based on offline analysis.

- A *policy metric* is a metric used to drive an allocation policy (the offline optimal policies in our case). Ideally, the policy metric and evaluation metric would be identical. However, the desired evaluation metric may not be measurable online, and thus may not be useful in driving policy decisions. Practical implementations may be forced to use policy metrics that are online observable metrics that merely correlate with the evaluation metric.

The primary contribution of this chapter is to provide a rigorous analysis of the objective function space in shared cache design along numerous axes. This study concludes that the optimal partition for a shared cache can vary greatly as different but reasonable definitions of optimality are applied. Additionally, none of these definitions of optimality encompass every positive attribute for shared cache performance; no metric is a functional superset of the rest. As a result, this investigation concludes that it is folly to pursue optimality along a particular metric in shared cache design. This conclusion further shows that marginal cost analysis in shared caches, which persistently pursue an optimal partition along some metric, is not the ideal approach for shared cache management. A better approach is to pursue a broadly applicable policy which provides good performance in the general case and prevents poor performance as measured under any reasonable metric. Experiments in this chapter show that LRU, the policy *de rigeur*, is insufficient for this purpose.

This study also makes the distinction between the pursuit of optimality along a metric, *i.e.*, using a metric as an objective function; and the use of a metric to measure performance. Results show that harmonic mean-based metrics do provide somewhat Utopian properties by not diverging very excessively from optimal fairness or throughput; in other words Utopian metrics are generally preferable to *measure* performance results, if not as an objective function.

3.1 Methodology

3.1.1 Performance Target Selection

A performance target is some function of the overall cache allocation which we seek to minimize or maximize by adjusting the allocation. The notation $(p_1, p_2, p_3, ...p_N)$ is used to denote the set of cache allocations for an N-thread workload, such that p_i is the cache allocation for thread i and $\sum_{i=1}^{N} p_i = C$, where C is the total cache space

	Target Name	Description
1	`MPC-None-Util`	Minimizing Overall Bandwidth
2	`IPC-128-Util`	Maximizing IPC weighted wrt 128KB performance
3	`IPC-256-Util`	Maximizing IPC weighted wrt 256KB performance
4	`IPC-512-Util`	Maximizing IPC weighted wrt 512KB performance
5	`IPC-1024-Util`	Maximizing IPC weighted wrt 1MB performance
6	`IPC-128-Comm`	Equalizing IPC weighted wrt 128KB performance
7	`IPC-256-Comm`	Equalizing IPC weighted wrt 256KB performance
8	`IPC-512-Comm`	Equalizing IPC weighted wrt 512KB performance
9	`IPC-1024-Comm`	Equalizing IPC weighted wrt 1MB performance
10	`IPC-128-Utop`	Maximizing Harmonic Mean of IPC weighted wrt 128KB performance
11	`IPC-256-Utop`	Maximizing Harmonic Mean of IPC weighted wrt 256KB performance
12	`IPC-512-Utop`	Maximizing Harmonic Mean of IPC weighted wrt 512KB performance
13	`IPC-1024-Utop`	Maximizing Harmonic Mean of IPC weighted wrt 1MB performance

Table 3.1: Performance Targets - Performance targets used in this study.

available.

As discussed previously, targets can be either Communist, Utilitarian, or Utopian in nature, emphasizing either fairness, overall performance, or a balance of the two. Each of these high-level targets can be applied to a variety of metrics.

A metric has two components: a “base” metric and an (optional) weighting factor. Included in this study are three base metrics:

- *Misses per access (MPA)*, *i.e.*, miss rate. This is a fundamental cache performance metric that is easy to measure online.
- *Misses per cycle (MPC)* measures bandwidth usage. We include this metric because off-chip bandwidth may be a precious resource in future servers.
- *Instructions per cycle (IPC)* is instruction rate. For a fixed program path length, *e.g.*, no spin-wait loops, IPC directly corresponds to throughput, and is thus a direct measure of observed performance.

A meaningful performance target should reflect some tangibly useful goal. Because IPC directly indicates performance, this study focuses primarily on IPC-based targets. Since MPC may prove useful as a target for severely bandwidth-constrained

systems, MPC-based targets are also included. MPA-based performance targets are not considered, since low miss rates do not directly translate into measurable improvement on the system level. Nevertheless, MPA is attractive because of its ease of online measurement; MPA will be revisited in Section 3.2.4 to consider whether it is useful as a policy metric, *i.e.*, a measurable proxy for other more significant metrics.

The other component, the weighting factor, was born out of SMT processor performance studies [53]. Any of the metrics defined above can be weighted, i.e.:

$$Metric_{weighted}(p_i) = \frac{Metric(p_i)}{Metric(baseline)} \tag{3.1}$$

The need for weighting came about because of the observation that simply maximizing raw aggregate IPC leads to policies that favor inherently high-IPC threads – threads which inherently have high ILP – at the expense of low-IPC threads. Using weighted IPCs, where the baseline is the IPC a thread achieves when it has the processor to itself, eliminates this bias, yielding a metric that does not reward starvation of inherently low-IPC threads. This same concept can be used when comparing policies for cache resource sharing.

For cache sharing studies, the choice of baseline for weighted metrics is less clear. In the context of a CMP system with 64 cores and 32MB of shared cache, for example, the performance of a single thread when it has the system to itself (*i.e.*, with all 32MB of cache) is largely irrelevant to its performance with the roughly 512KB of cache it would be expected to receive in shared operation. This baseline would also be costly to measure online, as it would require idling 63 cores for long enough for a single thread to warm up such a large cache, then repeating this task 64 times for each running thread. Instead, this study uses a set of baseline partition sizes that bracket the static uniform allocations of the studied systems – 128KB, 256KB, 512KB, and 1024KB – to see the impact of baseline selection on the resulting partitioning.

A useful performance target must combine a metric with a Utilitarian, Communist, or Utopian model to describe some meaningful optimization goal. A Utilitarian target minimizing raw MPC seeks to alleviate the demand for potentially limited off-chip bandwidth, and thus may be useful. However, there is little motivation for MPC targets based on weighted MPC or a Communist or Utopian model. As discussed above, raw IPCs also do not make meaningful performance targets due to their tendency to favor high-IPC threads at the expense of low-IPC threads, so this study considers only weighted IPC (WIPC) as performance targets. Table 3.1 lists all the performance targets evaluated in this study.

3.1.2 Optimal Cache Allocations

Having selected a set of performance targets, the question turns to determining an optimal allocation for a given target. Because this study examines the fundamental behavior of different performance targets, the optimal partition for a given target is determined using static offline analysis. I start by discussing partitioning using MPA-based targets, then use an analytic model to extend this methodology to the desired MPC- and IPC-based targets.

Past studies have observed that the overall cache miss rate (measured in terms of misses per access, or MPA) is a simple function of the sum of the miss rates of the contributing threads [55, 58]. That is, for N threads, where $MPA_i(p_i)$ is the the miss rate of of thread i with cache allocation p_i:

$$OverallMissRate = \frac{1}{N}\sum_{i=1}^{N} MPA_i(p_i) \tag{3.2}$$

Combining this observation with miss rate data for each application of interest on a range of cache sizes, a simple search algorithm can determine the statically optimal cache allocation $p_1, p_2, ...p_N$ for applications $1-N$ that minimizes the total misses

incurred.

A trace-based behavioral cache simulator called CASPER [30] is used to determine the miss rate functions of several benchmark applications for a spectrum of cache sizes. The CASPER simulator yields non-timing cache behavior information like miss rate, and is parameterizable according to size, associativity, and block size, among other things.

This study uses sixteen different benchmark traces. Five of the benchmarks are server workloads, which generally exhibit different behavior than user applications, while 11 are non-server. The intent behind the variety is to generate a varied cross-product of behavioral interactions with respect to cache sharing.

Workloads	Source
TPC-C™	Intel machine traces
SAP™	Intel machine traces
SPECjAppServer®	Intel machine traces
SPECweb®99	M5 full-system simulation
iSCSI	M5 full-system simulation
art	M5 EIO trace simulation
perlbmk	M5 EIO trace simulation
vpr	M5 EIO trace simulation
applu	M5 EIO trace simulation
apsi	M5 EIO trace simulation
bzip	M5 EIO trace simulation
twolf	M5 EIO trace simulation
equake	M5 EIO trace simulation
mcf	M5 EIO trace simulation
povray	M5 syscall-emulation simulation

Table 3.2: 16 workloads - The 16 workloads (five server, 11 non-server) used in this study.

All of the benchmarks as well as the source of their traces are listed in Table 3.2. Since CMPs are likely to be useful in the server market, common server workloads: TPC-C™, SPECweb®99, SpecjAppServer®, iSCSI, and SAP™are included in the benchmark mix. The TPC-C, SAP, and SPECjApp traces are from Intel and were generated from real machines, while the SPECweb and iSCSI traces were generated

with the M5 full-system simulator [7]. M5 is a modular, parameterizable simulator capable of performing full-system simulation for server workloads. For less system-intensive workloads, it can also run in syscall-emulation mode where system calls are emulated, not simulated. Additionally, users have a choice of ISAs and CPU models.

The non-server workloads are included to provide heterogeneity in application miss rates and behavior. These benchmarks are Simplescalar EIO traces of SPEC CPU2000 workloads [10], with the exception of povray, which was run in M5 syscall emulation mode due to the unavailability of an EIO trace. *Art* and *swim* are included as "pathological" workloads, in the sense that they are cache insensitive. On the other end of the spectrum are *perlbmk* and *vpr*, meant to represent threads that are not cache intensive.

Traces ranged from 40 million to 200 million instructions in length, with the shortest traces being the Intel machine traces, and the longest being from M5.

The sixteen benchmarks are measured with cache sizes varying in increments of 16KB from 16KB up to 1MB; results are shown in Figure 3.1. The top end size is limited to 1MB because experimental data showed that miss rate errors would have been unacceptably high with larger cache sizes. Error rates stem from the fact that in trace-based simulation, the first miss to a block may have been a hit had the simulation been running from the beginning, rather than from the middle of a trace [63]. So, all cold misses are actually of unknown status and comprise the error margin. Error margins can be reduced by running longer simulations, but the lengths of available traces dictated that the way to limit error in this study was via limiting simulated cache size.

One of the goals of benchmark selection is to encompass a variety of behaviors; this is especially important in multiprogrammed workloads in order to achieve a rich cross product of cache sharing behaviors. Based on the variety of miss rate curves and "knees" in the graph, the 16 benchmarks used do achieve this.

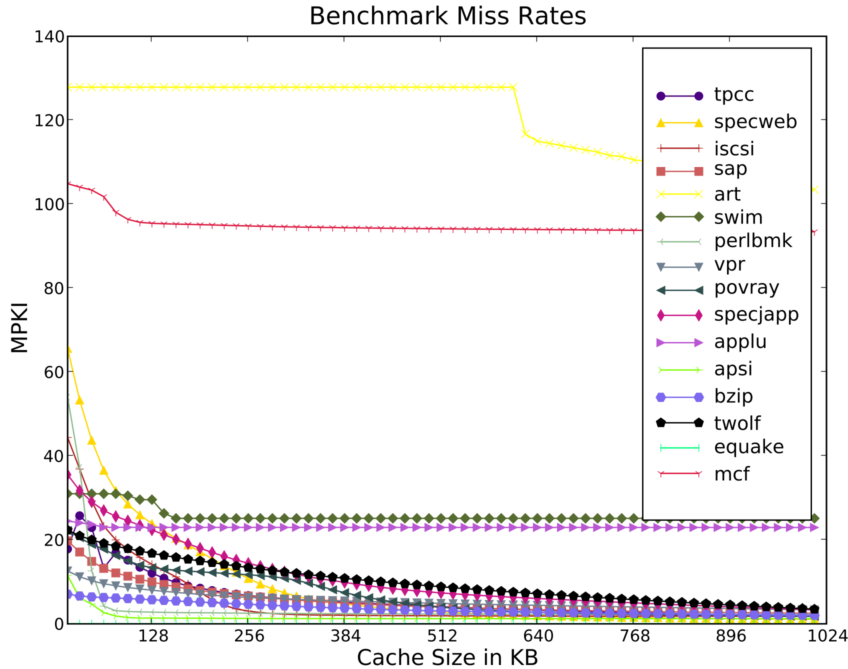


Figure 3.1: Miss Rate Curves - The sixteen benchmarks used in this study are plotted here. On the x-axis is cache size in KB, and on the y-axis is miss rate in terms of misses per thousand instructions (MPKI). For the X86 traces, instructions are in terms of micro-ops.

The simulated caches in this study have 64-byte blocks and as close to 32-way associativity as possible. Since some cache sizes were not perfect powers of 2 (*e.g.*, 48KB), 32-way associativity would be impossible. In cases like these, associativity moves to the nearest possible value to 32; in the case of 48KB, it is 24-way associativity. The use of high-associativity caches stems from the desire to emphasize the relationship between cache size and miss rate without undue interference from the relationship between limited associativity and miss rates.

In order to examine MPC and IPC metrics, this study uses an analytical model to translate the cache size-to-MPA mappings to obtain cache size-to-MPC and cache size-to-IPC mappings. Equation 3.3 shows the derivation of this mapping. MPI is misses per instruction, API is accesses per instruction, CPI_{base} is the base CPI of the

CPU, and CPI_{memory} is the CPI added due to miss penalties. The value p denotes cache partition size.

$$\begin{aligned} MPC(p) &= \frac{MPI(p)}{CPI(p)} \\ &= \frac{MPA(p) * API}{CPI_{base} + CPI_{memory}(p)} \\ &= \frac{MPA(p) * API}{CPI_{base} + MPA(p) * API * MissPenalty} \end{aligned} \tag{3.3}$$

Thus, Equation 3.3 determines MPC as a function of cache size with a given MPA curve. API is constant for each benchmark, and experimentally determined. Since the traces used are taken from both RISC and CISC sources, the API for the Intel traces are adjusted to reflect accesses per micro-op, so that numbers between the traces are comparable. This study assumes a simple 1-CPI machine, thus CPI_{base} is set to 1. $MissPenalty$ is set at 500 cycles, a large value to represent miss penalties to DRAM; in addition this study seeks to shed light on the qualitative behavior of cache sharing policies and by using a large cache miss penalty, relationships between cache size and IPC are more clearly highlighted.

Equation 3.4 shows the derivation of the mapping from MPA to CPI, using the same constants as previously. Note the use of the somewhat more awkward IPC form rather than CPI—this is because CPI is not additive across threads.

$$\begin{aligned} IPC(p) &= \frac{1}{CPI(p)} \\ &= \frac{1}{CPI_{base} + CPI_{memory}(p)} \\ &= \frac{1}{CPI_{base} + MPA(p) * API * MissPenalty} \end{aligned} \tag{3.4}$$

Converting these equations to weighted metrics is trivial; a thread's performance at a given cache size is divided by the thread's performance at the baseline size.

Given all of these size versus metric curves, determining optimal allocations is a

simple matter of evaluation. A brute force technique considers all possible partitions in granularities of 16KB. For Utilitarian targets, the partition that yields the best possible overall performance is selected. For Communist targets, the partition that yields the lowest standard deviation between all contributing threads is chosen. For Utopian targets, the partition yielding the maximum harmonic mean value across all contributing threads is selected. The equations below use weighted IPC with respect to 256KB as an example.

To find the optimal partition for `IPC-256KB-Utilitarian`, the equation below must be maximized:

$$WIPC_{total}(p_1, p_2, ...p_N) = \sum_{i=1}^{N} \frac{IPC_i(p_i)}{IPC_i(256\text{KB})} \tag{3.5}$$

For `IPC-256KB-Communist`, the following must be minimized over all threads i:

$$\sigma = \text{stddev}(\frac{IPC_i(p_i)}{IPC_i(256\text{KB})}) \tag{3.6}$$

3.1.3 CMP Thread Model

This study constructs workloads by choosing every possible two- and four-thread application mix from the 16 benchmarks described previously. Greater thread counts are modeled via logical replication, *e.g.*, a two-application mix is replicated four times to yield an eight-thread workload, or a four-application mix is replicated twice. This method enables the modeling of 2, 4, 8, 16, and 32 threads sharing 1MB of cache.

This technique is enabled by the static trace-based simulation methodology, in which a four-thread workload consisting of two threads from application A and two threads from application B sharing a C MB cache yields the same partition for each thread as a two-thread workload consisting of one A thread and one B thread sharing an $C/2$ MB cache. Thus the partition for a 32-thread workload on a 1MB cache can

be determined by evaluating the constituent four-thread workload on a 128KB cache. This technique shortens search time for optimal significantly since all possible ways to divide 128KB between four threads is a much more limited problem than evaluating all possible ways to divide 1MB by 32 threads.

3.2 Results

3.2.1 Optimal Partitions

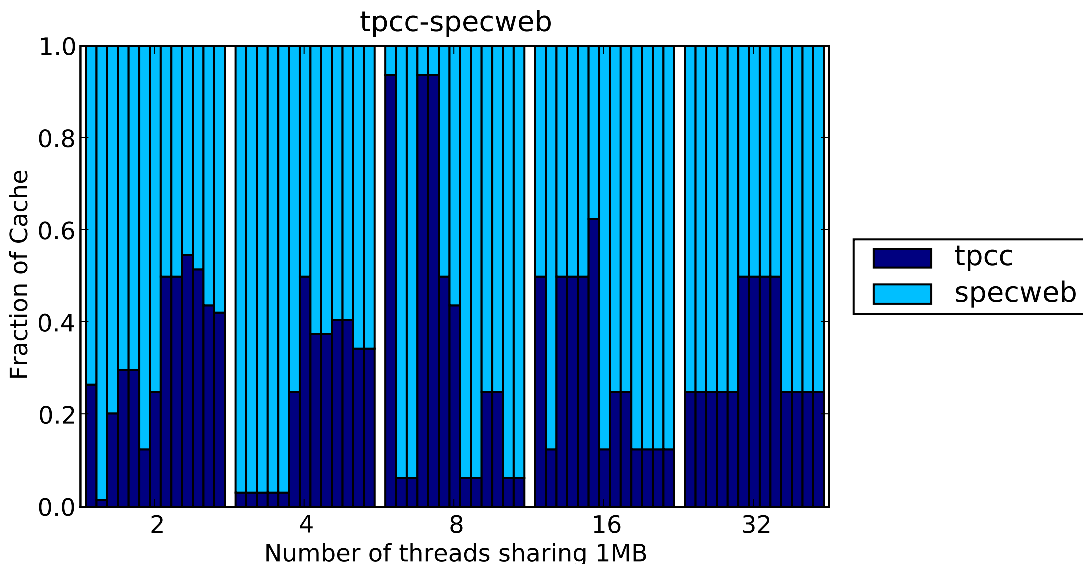


Figure 3.2: Two-application Partitioning Example - TPC-C and SPECweb sharing a CMP platform can yield widely varying ideal allocations, depending on the definition of ideal. Performance targets are unlabeled due to spacing reasons — however within one cluster of bars, the targets vary from left to right in the order listed in Table 3.1 from 1 through 13.

Figure 3.2 shows an example of how significantly partitioning can vary due to different definitions of optimality. This graph shows the various optimal cache allocations of the TPC-C/SPECweb application mix. The clusters along the horizontal axis represent different numbers of threads sharing a 1MB cache. Each member of a cluster is a different performance target. Due to spacing reasons, the targets are

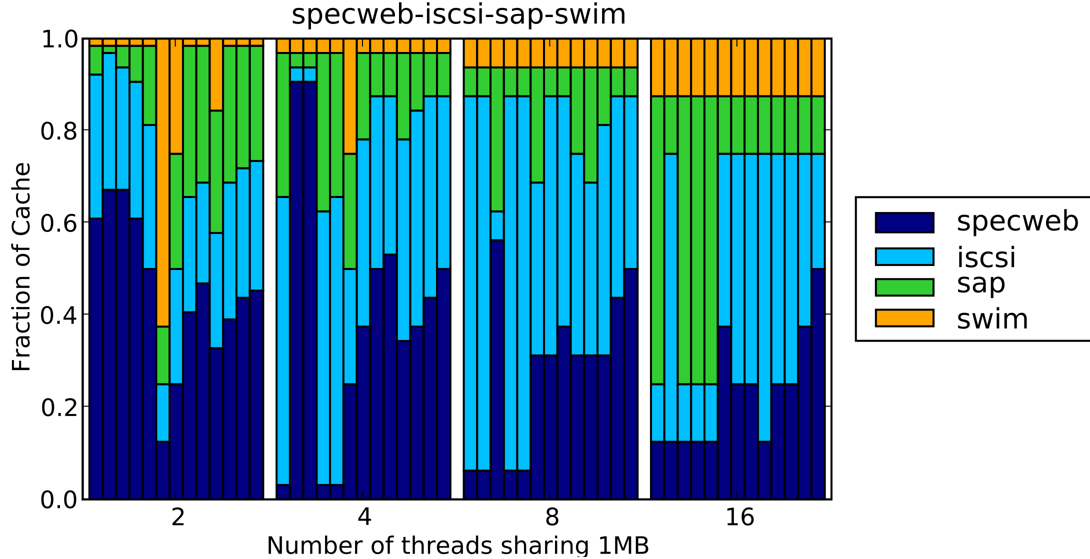


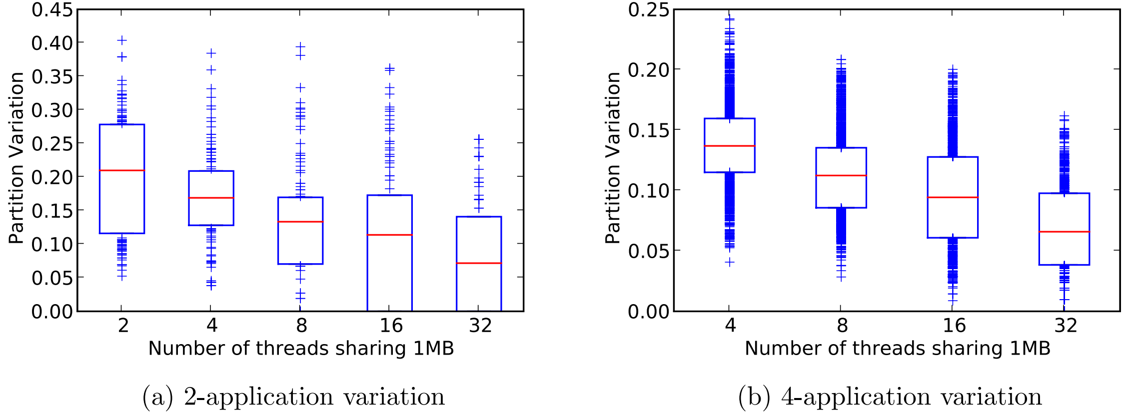
Figure 3.3: Four-application Partitioning Example - Another example of widely varying partitions, with four benchmarks sharing a CMP. Performance targets are unlabeled due to spacing reasons — however within one cluster of bars, the targets vary from left to right in the order listed in Table 3.1 from 1 through 13.

not labeled on the plot; however, each member from left to right corresponds to the targets 1 through 13 listed in Table 3.1. The vertical axis indicates the fraction of cache given to each benchmark. As is clear from the graph, the allocations can vary widely even from within a metric "family", like `IPC-*-Util` or `IPC-*-Comm`. This graph shows evidence that Communist, Utilitarian, and Utopian targets can differ greatly from each other, and that even choosing different weighting factors can yield significantly different results. The only generalization that can be made is that there are no generalizations evident from the graph.

Figure 3.3 shows the same results for a 4-application case, with benchmarks SPECweb, iSCSI, SAP, and *swim*. Optimal partition behavior varies widely and the impact of changing performance targets is difficult to predict. Trends caused by varying the baseline cache size are non-monotonic. For example, at 8 threads, going from `IPC-128-Util` to `IPC-256-Util` yields a larger partition for SPECweb, but its partition size shrinks again when moving from `IPC-256-Util` to `IPC-512-Util`. The

same phenomenon occurs for SAP, and the inverse occurs for iSCSI.

The details of the individual partitions are not important here, but Figures 3.2 and 3.3 do visually demonstrate the dramatic variety in optimal partitions there are for different valid and reasonable definitions of optimal.



(a) 2-application variation

(b) 4-application variation

Figure 3.4: Partition Variation - The variation in partition size between any given pair of performance targets. The values on the y-axis indicate the average relative change in cache allocation as the performance target changes. The 2-application plot shows the variation for all 120 $\binom{16}{2}$ application mixes; the 4-application plot shows the variation for all 1820 $\binom{16}{4}$ application mixes.

To quantify how much partitions can vary overall, this study uses a partition metric specifically devised for this study to measure partition variation. For any application mix, the metric determines the average difference in allocation for a single thread when going from any one performance target to another. Specifically, say the optimal partition for App_A-App_B with performance target i is such that App_A receives allocation p_{iA} and App_B receives allocation p_{iB}. Then the partition difference metric for a particular workload is defined as:

$$PartitionVariation = \frac{\sum_i \sum_{j>i} \sum_x |p_{ix} - p_{jx}|}{x_{max} * \binom{i_{max}}{2}} \tag{3.7}$$

The numerator represents the sum of all the allocation changes between all possible pairs of performance targets and all pairs of applications within the mix, while the

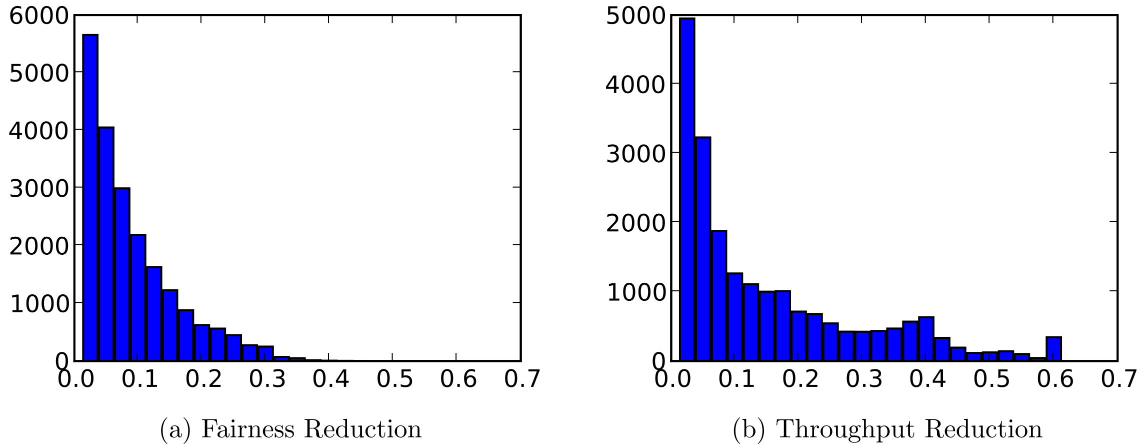
denominator represents the total number of possibilities, with the result yielding an average. Figure 3.4 shows the spectrum of partition variation for all the mixes used in this study. This box-and-whiskers plot shows the inner two quartiles inside the box, with the horizontal line representing the median. The whiskers extending outside the box show the remaining values of the outer two quartiles. The y-axis shows the partition variation values for each workload. When a plot is abutted on the x-axis, this indicates the entire bottom quartile (and likely some of the boxed middle two quartiles) have a value of zero.

Figure 3.4 shows the results for over all workloads in this study. Figure 3.4a shows results for when two applications out of the 16 used in this study are replicated to achieve the thread counts in the x-axis, while Figure 3.4b shows results for when four applications are replicated to achieve the thread counts in the x-axis. In the 2-application case, two threads sharing 1MB can yield partition variations up to 40%, with a median above 20%. The 16- and 32-thread cases have many zero-change instances resulting in lower medians because the 16KB partitioning granularity we use represents a significant fraction of the typical per-thread allocation in a 1MB cache. However, there are still cases where there is a significant partition variation between different performance targets.

In the 4-application graph, variation in the four thread case can go up to 25%, with a median at nearly 15%. Even in the 32-thread plot, optimal partitions between any two definitions of optimal can vary over 15%. In all cases, it is clear that varying the performance target selection among a set of reasonable choices can yield significant variations in the optimal cache partition, not just in the case studies shown previously in Figures 3.2 and 3.3, but in general.

These graphs demonstrate one aspect of the difficulty in selecting a single definition of optimal out of multiple reasonable definitions. It would be one thing if differing definitions of optimal led to more or less the same optimal partitions, in which case

selecting one definition or another may not be a decision with very big implications. However, selecting a particular definition can mean highly variable partitions, even when the definitions are highly related.



(a) Fairness Reduction

(b) Throughput Reduction

Figure 3.5: Communist vs. Utilitarian Histograms - Histograms indicating reductions in performance from optimizing throughput or fairness. On the left is a histogram of fairness measurements on caches optimized for throughput, and the delta with optimal fairness is shown. On the right is a histogram of throughput measurements on caches optimized for fairness, and the % difference from optimal throughput is shown along the x-axis.

3.2.2 Comparing the *"-isms"*

The last section demonstrated the high variability in optimal partitions for varying definitions of optimal. However, that could be unimportant if the different optimal partitions, despite their variability, led to relatively similar performance numbers. This section examines the differing levels of *performance* when optimizing for different metrics.

One of the key distinctions among previously proposed performance targets is whether they target overall performance (Utilitarianism) or fairness (Communism) or attempt to balance both performance and fairness (Utopianism). Both aggregate performance and fairness are important aspects of shared cache performance, and it

is difficult to say which is more important. Utopian ideals attempt to balance the two, but given the potential incompatibilities between the two "-isms", it is unclear how achievable that is.

This portion of the study evaluates the quantitative difference between these targets. Specifically, the compatibility of these targets are compared to determine how much of one metric is lost when optimizing for another.

Communism versus Utilitarianism

First, this study focuses on the compatibility between Communism and Utilitarianism, as these two approaches are most clearly at odds with one another. Figure 3.5 shows the results. The plots in this figure are both histograms of all WIPC workloads in the 4-application case, over all thread combinations. The raw MPC metric used previously is ignored in this set of graphs since it does not make sense under a Communist policy.

Figure 3.5a is a histogram of the delta in fairness performance between workloads optimized for WIPC fairness and workloads optimized for WIPC aggregate performance. Essentially, what Figure 3.5a demonstrates is the amount of potential fairness lost when optimizing for throughput.

Figure 3.5b shows the opposite scenario; a histogram of the delta in overall WIPC performance between workloads optimized for maximal WIPC and workloads optimized for WIPC fairness. Thus, this plot demonstrates the amount of potential aggregate WIPC lost when optimizing for fairness.

Both histograms have most of their samples in the bin closest to zero, meaning that for the most part, Communist and Utilitarian metrics are comparable; *i.e.*, optimizing for throughput tends to provide near-optimal fairness and vice versa. However, both distributions have a significant tail. In other words, Communist targets mostly yield partitions that have near-optimal overall performance, but there are a few cases

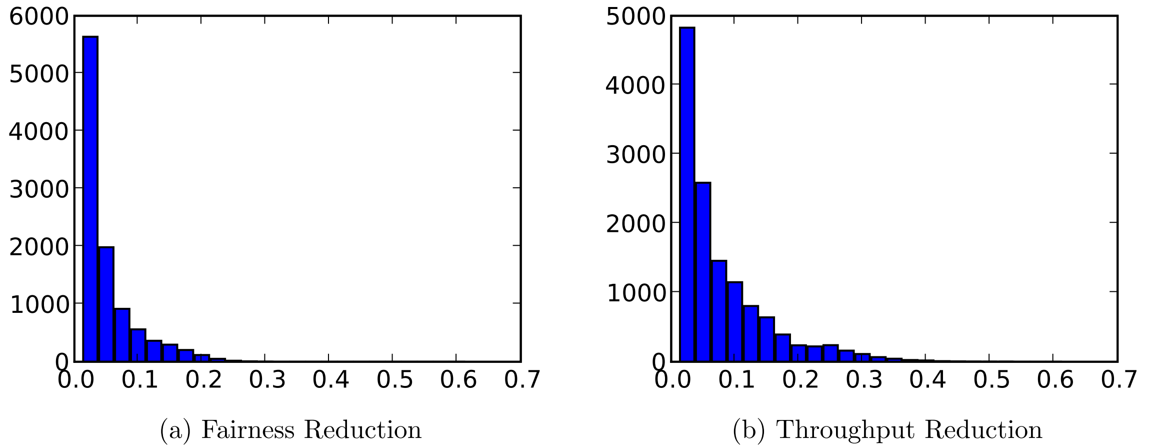
where the optimal Communist partition yields extremely poor overall performance. Likewise, Utilitarian targets mostly yield optimal partitions that are not extremely unfair, but there are a non-trivial number of cases which are.

These two graphs do not indicate which target is preferable, but they do indicate that whichever target is used, it is important to guard against these heavy tails such that the resultant partitions can be both fair and have good overall performance.

How Utopian is Utopia?

Since the previous section has shown Communism and Utilitarianism to be occasionally fundamentally incompatible, the question naturally turns to whether optimizing for a harmonic mean truly effects a Utopian balance between the two, the way minimizing standard deviations effects Communism and maximizing overall performance effects Utilitarianism.

Figure 3.6 shows the fairness reduction from optimizing for Utopianism rather than Communism on the left, and the throughput reduction from optimizing for Utilitarianism on the right. The appropriate comparisons are between Figures 3.6a and 3.5a, and between Figures 3.6b and 3.5b. What is clearly noticeable is that the heavy tails in Figure 3.5 are reduced somewhat in Figure 3.6. What this indicates is that optimizing for Utopianism leads to a lesser loss of fairness from optimal than optimizing for Utilitarianism. It also shows that optimizing for Utopianism leads to a lesser loss in overall performance from optimal than optimizing for Communism. Thus, using a harmonic mean as a means of effecting Utopianism, *i.e.*, a balance between fairness and throughput, is a reasonably good approach. However, it should also be noted that heavy tails still exist, such that a system optimized for Utopianism may still suffer from poor fairness and/or throughput. For this reason, while Utopian metrics may be better measures of shared cache performance because they are less likely to mask very poor performance in terms of fairness or throughput, they are still



(a) Fairness Reduction

(b) Throughput Reduction

Figure 3.6: Utopian Histograms - Histograms indicating reductions in performance from optimizing for Utopianism. Note the contrast between Figure 3.5a and the figure on the left. Optimizing for Utopianism is a lot less unfair than optimizing for Utilitarianism. At the same time, note the contrast between Figure 3.5b and the figure on the right. Optimizing for Utopianism maintains throughput much better than optimizing for Communism. In conclusion, the Utopianism seems to strike a balance between the Communism and Utilitarianism.

incompatible enough with other valid and reasonable metrics to discourage designing a shared cache entirely around them.

3.2.3 Baseline Weighting Choices

As previously discussed, one of the subtleties to choosing a weighted performance target is the selection of the baseline weighting factor. Prior work has taken the baseline as "what the performance would be if a thread had the whole system to itself". While this approach may be reasonable for small-scale systems, it may be both impractical and irrelevant on an LCMP system with tens or hundreds of threads and tens of megabytes of cache.

Figure 3.7 examines the impact of the choice of baseline by showing the correlation coefficients between weighted metrics with various baselines. A perfect correlation of 1.0 indicates that there is no difference between choosing one baseline or another. Lower correlation values indicate larger differences between the results of different

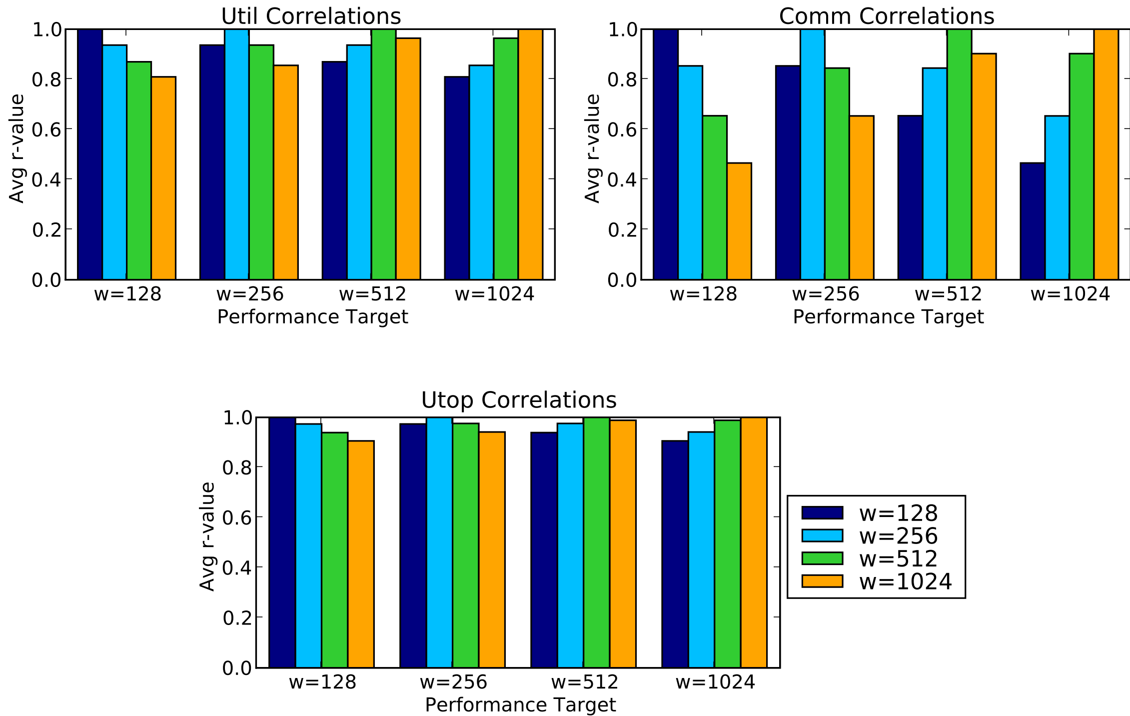


Figure 3.7: Baseline Weighting Choices - This graph demonstrates the dramatic differences there can be in choosing different baselines for weighted metrics for Communist metrics. The differences are more muted for Utilitarian and Utopian metrics. Each cluster has a bar with value 1 because every metric correlates perfectly to itself.

baselines. As can be seen from the graph, Utilitarian weighted IPC metrics are all reasonably close, with correlations staying above 0.8. Utopian WIPC metrics are even closer, with correlations staying at or above 0.9. However, with Communist weighted IPCs, the selection of weighting factor has a great impact on resulting performance. Given this sensitivity to the weighting factor, it seems difficult to define "fairness" in a robust and precise fashion. On the other hand, weighting factor is virtually irrelevant in Utopian targets, implying that these metrics are relatively baseline agnostic, at least in the "reasonable" spectrum tested. Utopian metrics thus seem the most robust.

3.2.4 Policy Metrics

As mentioned in the introduction, an evaluation metric may not be measurable on-line in real time. A weighted metric is difficult to measure online because knowing

how some thread would fare if it had some greater portion of the cache is impossible to know without one of several unappealing options. One of them, offline profiling, can be inaccurate when inputs vary, and typically does not capture time-varying program behavior. The other, online sampling, may be impractical if it requires running each of many threads alone sequentially in order to achieve unperturbed samples. Thus, it is useful to find *policy metrics*, online measurable metrics that serve as proxies for evaluation metrics and can be used to drive policy decisions that yield good performance with respect to the performance target. This section examines the correlations between metrics that are reasonably measurable online and all studied performance targets. Unweighted metrics are assumed to be measurable, as they can typically be captured online using simple counters without requiring special sampling phases.

The results are shown in Figure 3.8. The graph shows that miss rate (MPA) is actually correlates quite poorly with all Communist and Utilitarian performance metrics, but reasonably well for Utopian metrics. MPC and IPC, however, are relatively good indicators for Utilitarian targets, tepid predictors for Utopian targets, and poor for Communist targets. This figure indicates that none of the policy metrics considered is sufficient to drive an on-line policy for Communist weighted IPC targets. Thus, having a runtime shared cache policy which seeks to optimize fairness is not only unappealing because of the potential major loss in overall performance described in Section 3.2.2, but because it seems unlikely that fairness can even be feasibly and reasonably measured in such a way to provide adequate feedback on allocation decisions.

Thus, it may make more sense to use Utilitarian targets, while simply trying to avoid the unfair outliers seen in the previous section. Even better would be using Utopian targets, which largely capture both fairness and throughput, and for which MPA is a reasonably good policy metric.

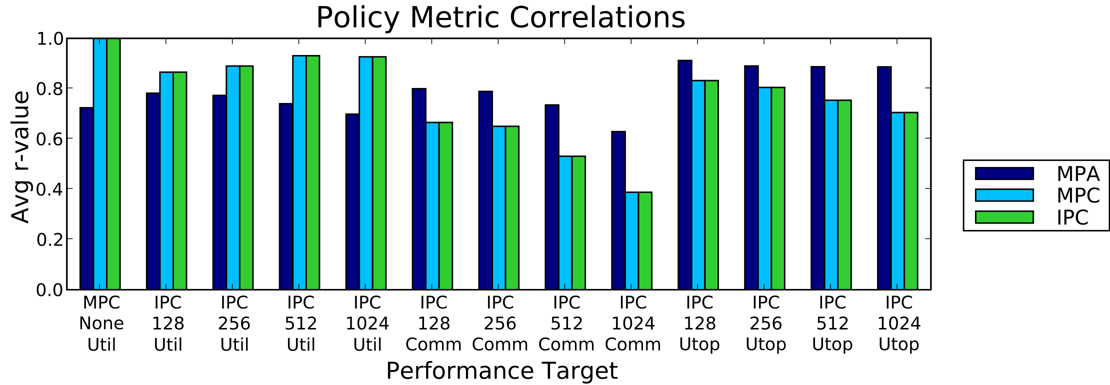


Figure 3.8: Policy Correlations - Correlations between possible policy metrics and performance targets. The y-axis shows the absolute value of r-values.

One interesting side result is that MPC (bandwidth) and IPC always correlate perfectly. It turns out that this is an artifact of the performance model described in Section 3.1.2. In this model, raw MPC and raw IPC are linearly related, such that minimizing raw MPC yields the same partitions as maximizing raw IPC. Due to this mathematical connection, this relationship will likely continue to hold, though less precisely, in real-world executions.

3.2.5 Policy Evaluation

Recall that one of the four aspects to cache partitioning policies is the policy itself—the aspect of the cache implementation which makes allocation decisions. LRU is the most common default policy today and is meant to effect high-IPC performance from the cache. In this portion of this study, LRU is compared against policies that are intended to effect high performance with respect to the various performance targets studied in this chapter. In order to avoid aliasing implementation issues with policy goals, LRU is compared against the optimal partitions for each of the performance targets. This way, the upper bound of performance differential between the default LRU policy the various performance targets can be exposed.

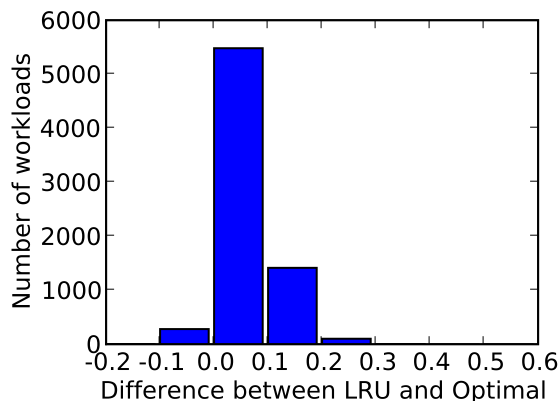


Figure 3.9: LRU vs. Perfect Raw MPC Partitioning - This plot shows that LRU achieves nearly perfect raw MPC performance. The value on the x-axis indicates the percentage difference between LRU and optimal `MPC-None-Util` partitioning. Recall that in this study, IPC and MPC are linearly related because of the nature of the analytical model used to generate these values, and so LRU, long used to effect high-IPC performance from the cache, does exactly that here.

First, Figure 3.9 shows that LRU really does achieve quite good raw MPC performance. In the figure, the x-axis designates the performance differential (in percent) between LRU and an optimal partition for `MPC-None-Util`, where a positive value indicates that LRU underperformed the optimal partition, and a negative value indicates that LRU achieved better raw MPC performance than the optimal partition. This is due to the dynamic nature of LRU, as opposed to the static nature of the partitions. Since there is virtually no tail on this plot, this indicates near ideal performance on the part of LRU with respect to raw MPC. While there are still some workloads that underperform optimal by up to 30%, they are a minimal portion of the distribution.

Recall also from the previous section that due to the nature of the analytical models used here, MPC and IPC are linearly related. So this plot also demonstrates the facility of LRU at providing high-IPC performance. So, LRU, long called upon to provide good raw throughput, is shown to truly do so here in most cases. However, high raw throughput can mean poor individual performances, as demonstrated by the

following graphs, which compare LRU against the more thread-aware performance targets studied in this chapter.

Figure 3.10 shows the histograms comparing LRU to the remaining 12 policy targets studied in this chapter. The x-axes are all in terms of percent relative to optimal, with the exception of the Communist targets. Since these performance numbers are in terms of standard deviations, a percentage different in standard deviation makes little intuitive sense. Instead, comparisons between LRU and optimal are done in terms of absolute differences in σ. Across the spectrum of histograms in Figure 3.10, it is clear that certainly, with respect to Utilitarian and Utopian targets, LRU vastly underperforms an optimally partitioned cache. Non-trivial numbers of workloads underperform optimal partitions for their affiliated performance targets by 30, 40, 50, and even 60%.

With respect to the Communist policies, the plots appear less dramatic but still indicate non-trivial departures from optimal fairness performance, no less because of the difference in measurement and scaling just described. These x-axis values indicate the raw difference in standard deviation between the contributing weighted IPC values. In other words, a value of say, 0.2 indicates that the average deviation from the mean of all values has grown by 0.2 in the LRU case from optimal, indicating a significant increase in the spread of weighted IPC values and thus a significant decrease in fairness.

The primary result from this collection of graphs is that LRU, while quite good at providing high performance with respect to raw throughput, leaves quite a bit to be desired when it comes to other performance targets that focus on shared cache performance. At the same time, while LRU does not provide performance in the general case for these other performance targets, it does do reasonably well in a significant number of cases. This results points to the idea that perhaps a better approach to shared cache management would be to use LRU as a default, and reacting

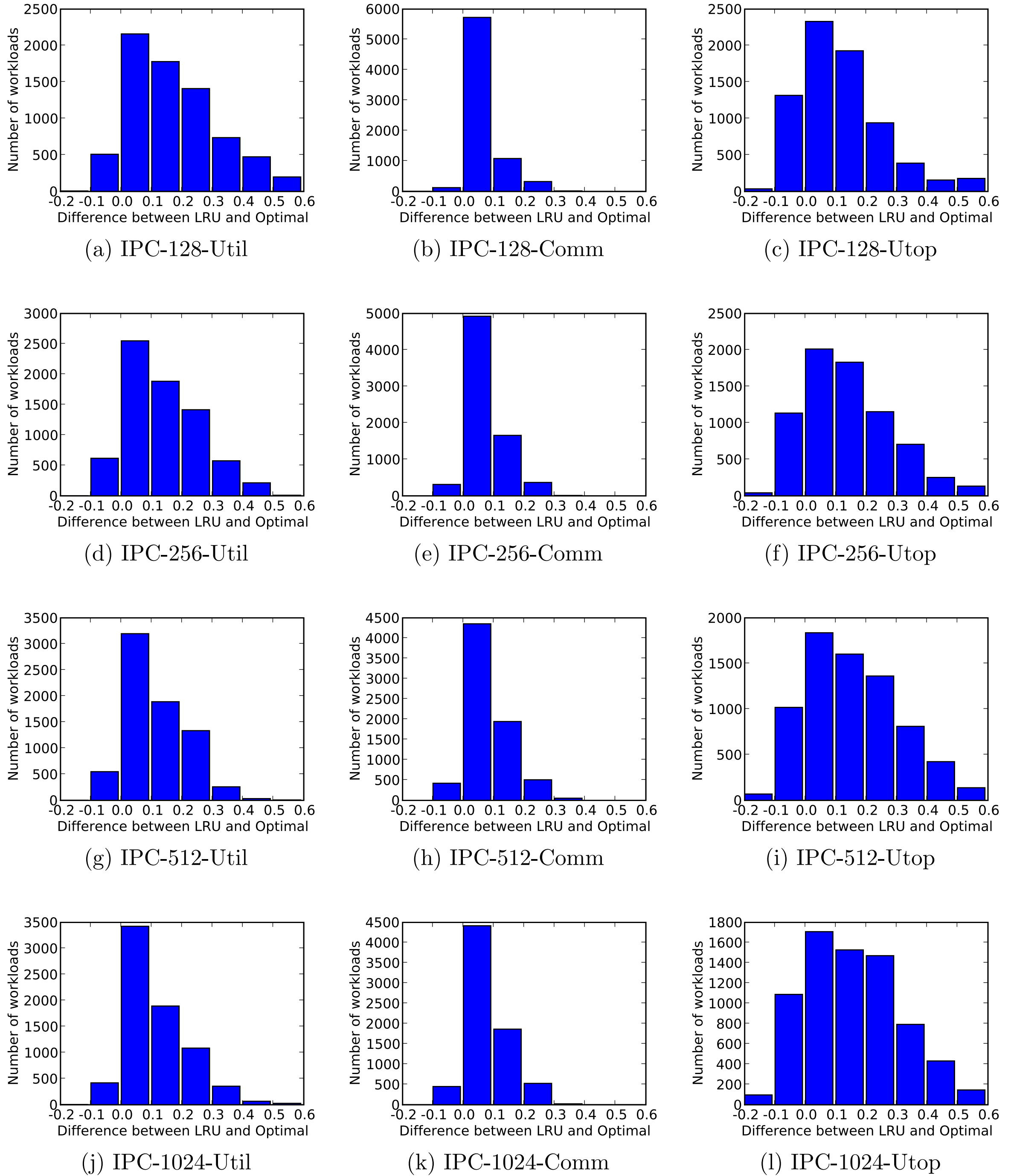


Figure 3.10: Policy Evalutaions - The histograms in this figure compare the performance of the LRU policy against optimal partitions for the affiliated performance targets indicated by the labels. X-axis values indicate the amount LRU underperformed the optimal partitions. In cases where x-axis values are negative, this means the dynamic nature of LRU outperformed the statically optimal partition.

when a poor sharing situation is detected, rather than actively pursuing an optimal partition for a particular performance target.

3.3 Conclusions

Controlling the allocation of shared cache resources among threads is a necessary task in future CMP systems. However, the selection of the objective function for guiding this allocation is a subtle issue, and different seemingly reasonable objectives can lead to significantly different results.

This study makes the distinction between *performance targets*, the goals of a cache; *evaluation metrics*, the metrics to measure the achievement of that goal; *policies*, the decision rules meant to achieve the performance target; and *policy metrics*, the actual metrics used to direct policy decisions. Many researchers use policy metrics and evaluation metrics interchangeably in the literature. However, our results indicate that a common policy metric, miss rates, is a poor proxy for both fairness and throughput, despite being commonly used as such in the literature. Additionally, selecting different performance targets can cause optimal cache allocations to vary greatly, which implies that the selection of a performance target cannot be done in an arbitrary manner.

In addition, this study compares the effect of applying Communist or Utilitarian goals in cache partitioning, *i.e.*, optimizing for fairness across threads or for optimal aggregate behavior. For most workloads, there is little conflict between these targets, in that Communist goals typically lead to near-global optima, and Utilitarian goals typically provide good fairness. However, in either case, there are outlying workloads for which a Communist goal severely degrades global performance or a Utilitarian goal severely impacts fairness. Thus choosing one or the other of these goals as primary may be less useful than guaranteeing that neither fairness nor global performance is

unduly sacrificed in pathological conditions.

Utopian policies, however, where harmonic means are used to balance fairness and throughput in a single measure, do reduce the risk of having a high performance measure but actually fare poorly in either fairness or throughput, and thus are a reasonable way to measure shared cache performance.

This study also shows that weighted metrics, a standard technique in the area of SMT research intended to alleviate unfairness in resource allocation, have unexpected issues as well. CPU-oriented SMT research uses standalone performance – *i.e.*, a single thread having all resources to itself – as a baseline. However, using standalone performance as a baseline for a multi-megabyte shared cache in a large-scale CMP is both less practical and less meaningful. Unfortunately, choosing different arbitrary baselines yields significant variations in performance for Communist targets, and it is unclear why there might be any reason to pick one over another. Utopian targets, however, can be robustly defined without regard to weighting factor in a reasonable range of values.

Additionally, this study investigates the ability of online measurable policy metrics to act as proxies for more desirable but less practical evaluation metrics. For the benchmarks selected, miss rate is a poor policy metric for both throughput and fairness, even though it is commonly used as such. However, miss rate is a good policy metric for Utopian policies, while raw IPC and MPC are relatively good policy metrics for weighted IPC evaluation metrics in the Utilitarian model. This study did not find a good policy metric for Communist performance targets.

Thus, for a variety of reasons, Utopian policies emerge as a pragmatic metric for measuring the performance of shared caches. These reasons include: the existence of a reasonable policy metric, in that miss rates seem to correlate well with Utopian policies; the policies seem to be generally baseline weight agnostic; and most importantly, they do seem to capture fairness and throughput in a single value. However,

given that there are still heavy tails in the performance histograms, showing that it is possible to have a high Utopian value but have poor fairness or poor throughput, it seems there is no one metric that truly encompasses the idea of optimal. This shows that selecting any one performance target as the goal of cache design is a mistake.

Instead, this study demonstrates that it may be more important to identify clearly poor cache sharing situations and alleviate them, rather than explicitly chasing a particular definition of optimal which may mask poor performance in some important way. This approach is further bolstered by the result that LRU, the *de facto* cache policy in use today, does well at providing Utopian performance in a significant number of cases; it would seem that taking advantage of this fact is wise. Thus, the primary conclusion of this chapter is that using LRU as a starting point, and only reacting to poor cache sharing situations, rather than proactively pursuing some definition of optimal, is the preferred approach. The key question then becomes: how can poor cache sharing be detected at runtime? The remainder of this dissertation takes this approach and seeks to answer this question.

CHAPTER 4

Machine Learning Techniques for Discovering Salient Characteristics of Poor Performance

Given the results of Chapter 3, this study seeks to identify some set of characteristics that correlate with poor performance (*i.e.*, detecting the heavy tails in the results histograms). If such characteristics can be found, the situation can be mitigated at runtime, thus leading to a reactive approach to shared cache management instead of a proactive one. Not only does this eliminate the complexity of explicitly seeking an optimal shared cache partition, but acknowledges the finding that there is no one metric that can definitively describe optimality anyway.

The studies in this chapter use supervised machine learning techniques in an attempt to determine what aspects of a multiprogrammed workload lead to significant losses in fairness or aggregate performance. The end goal is to use these discovered characteristics to build a runtime detection mechanism for identifying poor shared cache performance, and also to inform the crafting of a feasible and scalable solution for mitigating poor performance.

Supervised machine learning is the act of building a predictor that takes inputs (*features*) and uses them to predict specific *outputs* [24]. In this case, the goal is to use runtime application characteristics as features, and a classification of "acceptable"

Weight	Age	BP	Family History	Eye Color	Siblings	Heart Disease?
150	35	High	Yes	Blue	3	Yes
118	24	Low	No	Brown	2	No
250	55	High	No	Green	4	Yes
220	47	Low	No	Brown	1	No

Table 4.1: Sample Supervised Learning Data - Sample matrix for a supervised learning problem. Data points (patients) are mapped to characteristics (features) and outputs (heart disease).

or "poor" as outputs. In contrast, unsupervised machine learning does not seek to measure or predict outcomes, but rather seeks to describe how data are organized or clustered. For the remainder of this thesis, all references to machine learning (ML) will imply supervised learning.

A simple example of supervised learning is building a predictor for heart disease. If one is to be judicious about choosing features to build the predictor, features like `weight`, `age`, `blood pressure`, `exercise`, and `family history` would be taken into account. Less advisable features would include `hair color`, `eye color`, or `number of siblings`. To build a predictor, a matrix of data points and features must be created, along with known outcomes. See Table 4.1 for a sample table.

In this simple example, the given input matrix could be fed into any number of ML model-building techniques, and yield a model for predicting the presence of heart disease in an arbitrary patient, given the appropriate data inputs. However, anyone could guess that `number of siblings` and `eye color` would be essentially useless features in generating a predictor for heart disease. Some ML techniques are useful not just for model creation, but also for *subset selection* [25], which is the process of pruning the feature set down to useful features. This concept will be discussed further later in the chapter.

In this dissertation, machine learning is used as an analysis tool. The usage model here would be to collect large amounts of known input and output data, use ML to extract salient inputs for a given output through subset selection, and use this

knowledge to better understand the problem and inform the crafting of a solution. ML could also be used in an online and dynamic fashion by deploying a learner at runtime to discover techniques for given executions, such that continual feedback and adjustment to the learner are possible. However, this dissertation avoids that approach for the following reasons:

- Characteristics leading to poor performance are potentially generalized. In other words, being able to build a dynamic predictor in the hardware that can learn on the fly seems like overkill if the characteristics to poor performance are generally the same across executions of different workloads.

- The complexity of building a runtime dynamic learning mechanism is less appealing than finding simple factors leading to poor performance and building a static detector. While this dissertation does not evaluate the complexity of a dynamic learning mechanism, the result given by ML as an analysis tool does lead to an extremely simple static mechanism.

- Feedback in a runtime system is non-trivial matter; the system must know when something is performing poorly in order to train itself to detect a poor performer, which is essentially a chicken and egg situation.

The studies in this chapter primarily aim to use machine learning as an analysis tool for sifting through large quantities of data in order to discover characteristics correlating to poor performance. The end goal is to be able to use these discovered characteristics to not only build an on-line detection mechanism for poor performance, but also give hints as to how to alleviate the problem.

Since machine learning is used as a tool in these studies, this chapter also ends up providing some insights into using ML in architectural research. Machine learning can certainly be used effectively as a research tool when used properly, but cannot be counted on without some hands-on human guidance and intuition. Care must be

taken in selecting the ML technique used; additionally the set of features fed into the ML process must be selected judiciously.

The basic methodological approach to this chapter is described in Section 4.1. The two ML techniques used in this chapter, ridge regression and decision tree analysis, as well as the results of their application, are discussed in Sections 4.2 and 4.3. Conclusions about the use of machine learning in this study are described in Section 4.4.

4.1 Methodological Approach

The basic methodology of this chapter begins with generating large quantities of data using the CASPER cache simulator [30]. As discussed in Chapter 3, CASPER is a trace-based behavioral cache simulator that is highly parameterizable.

An important aspect of data generation is the selection of benchmarks. In order for the generated predictor to be generalized, the benchmarks used to generate the data must be varied enough to be representative. The benchmarks used here are exactly the same as the ones used in Chapter 3, and are listed in Table 3.2.

For this study, four of the sixteen applications are run simultaneously on CASPER, sharing a single 1MB last-level cache using an LRU replacement policy. The performance of these simulations are compared against the performance of an optimally partitioned cache (discussed in Chapter 3).

As discussed in Chapter 3, Utopianism may not make sense as a performance target, *i.e.*, as an overt goal to persistently achieve optimal Utopianism, but as a performance metric, it can be a helpful tool. The goal of this chapter is to determine a set of characteristics that indicate that a cache is exhibiting poor Utopian qualities.

Fair Speedup is a metric proposed by Chang and Sohi which is Utopian in nature [12]. It is defined as the harmonic mean of speedups over a baseline of private,

equal-share caches, *i.e.*,

$$FS(scheme) = \#app / \sum_{i=1}^{\#app} \frac{IPC_i(equal)}{IPC_i(scheme)} \tag{4.1}$$

With respect to the metrics studied in Chapter 3, this amounts to `IPC-Private-Utopian`, such that the weighting factor is always $1/N$th of the total cache space available. This is an intuitively pleasing metric because qualitatively, Fair Speedup (FS) measures aggregate performance relative to an equal share platform but weights high-value outliers less to measure fairness across sharers.

For the studies in this chapter, the "output" of the datasets used for model generation is the relative FS performance of LRU compared to the a statically partitioned cache optimized for FS. An LRU cache is considered to exhibit poor cache sharing performance when it performs at less than 80% of optimal. Thus, the datasets in these studies consist of a particular workload of four benchmarks and their feature data as the input, with relative FS performance as output. The goal is to find a robust model for being able to predict the latter given the former.

4.2 Ridge Regression

4.2.1 Background

Ridge regression is a subset of the more general linear regression analysis technique. Linear regression analysis has recently become popularized in the architecture domain as a processor design tool [29, 38, 34].

A linear regression model is a mathematical representation of the relationship between an array of input parameters and the observed output response, usually of the form:

$$y = \beta_0 + \sum_{i=1}^{p} \beta_i x_i + \epsilon \tag{4.2}$$

Note that the linear form of the model does not indicate that this method is only capable of expressing linear relationships between inputs and outputs. Input features can be non-linear transformations of variables, *i.e.*, $log(x)$, thus enabling the technique to capture non-linear relationships as well.

Linear regression analysis can be used for both investigating and modeling relationships between variables. Given a dataset of features to mapped to known outputs (*i.e.* feature matrix $\boldsymbol{X}$ and output vector $\boldsymbol{y}$ in Equation 4.2), simply solving for the $\hat{\boldsymbol{\beta}}$ that yield the smallest error yields a mathematical model between $\boldsymbol{X}$ and $\boldsymbol{y}$. Note that the hat above $\boldsymbol{\beta}$ indicates that the values are the solved-for values, not the actual values. Error between the solved and actual values is measured in the form of the residual sum of squares (RSS), shown in Equation 4.3 for N data points of p features each. Solving for the $\hat{\boldsymbol{\beta}}$ that provides the smallest RSS is called *least squares estimation* [62].

$$RSS = \sum_{i=1}^{N} (y_i - \hat{\beta}_0 + \sum_{j=1}^{p} x_{ij} \hat{\beta}_j)^2 \tag{4.3}$$

The ridge regression is essentially a least squares estimation problem with one additional constraint:

$$\sum_{j=1}^{p} \hat{\beta}_j^2 \leq s \tag{4.4}$$

Constraining the possible values of the $\hat{\beta}$s in this way serves to limit the variance of the models generated between two different datasets with the same relationship. For example, imagine a a single large dataset that represents a particular relationship between $\boldsymbol{X}$ and $\boldsymbol{y}$. Intuitively, one would imagine that dividing the dataset in half,

and building two models out of the separated datasets would yield the exact same model. However, this is not necessarily the case, particularly when the features representing the columns of $\boldsymbol{X}$ are not orthogonal. Through constraining the $\hat{\beta}$ values, ridge regression offers some stability to the achieved solution [25].

Another reason for using the ridge regression is the simple closed form with which one can solve for $\hat{\boldsymbol{\beta}}$. Solving for the ridge $\hat{\boldsymbol{\beta}}$ is just a simple linear algebra equation shown below, where $\mathbf{I}$ is the identity matrix (all zeros with a stream of ones through the diagonal) [25]:

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^{\boldsymbol{T}}\mathbf{X} + \boldsymbol{\lambda}\mathbf{I})^{-\mathbf{1}}\mathbf{X}^{\boldsymbol{T}}\mathbf{y} \tag{4.5}$$

This simple transform yields the vector $\hat{\boldsymbol{\beta}}$ providing the smallest model error for a given λ. Note that there is a one-to-one ratio between λ and the s constraint shown in Equation 4.4. The intuitive meaning of λ is that as $\lambda \rightarrow 0$, the ridge model approaches the simple least squares solution, as in Equation 4.3. As $\lambda \rightarrow \infty$, the βs are all pushed to 0 and the model will yield a simple β_0 intercept—regardless of input, the output predicted will be β_0. In short, the greater the λ, the more the ridge β values will be pushed toward 0 (and each other) to prevent wild swings in the variance of $\hat{\boldsymbol{\beta}}$ values from dataset to dataset [25].

Naturally, it is important to select an appropriate λ value. The idea of K-fold cross validation is to find a λ for the model that will have the least mean squared error when applying the model to new data. This can be done by taking a single set of input/output observations and dividing it into K separate subsets of equal size. Common choices of K include $K = 5$ and $K = 10$ [26].

Now, there at K training sets T, each called a *fold*. For each $k \in [1, \cdots, K]$, the ridge regression should be solved on all training subsets *except* the kth fold T_k. The resulting $\hat{\boldsymbol{\beta}}$ weights yield a model $f_k(\mathbf{x})$. The error measure when applying this model to the excluded fold T_k is described in Equation 4.6.

$$CV_k^\lambda = \frac{1}{|T_k|} \sum_{(\mathbf{X},\mathbf{y}) \in T_k} \mathbf{y} - f_k(\mathbf{x}) \tag{4.6}$$

The average model error over the entire cross validation of the λ model is then:

$$CV^\lambda = \frac{1}{K} \sum_{k=1}^{K} CV_k^\lambda \tag{4.7}$$

Over a range of λ, which in practice are generally small values ranging logarithmically from 0.0 to 1.0, the value which yields the lowest average error is selected for the final model.

There is one more relevant issue to this portion of the methodology, involving the standardization of inputs. Ridge regressions perform better when the inputs and outputs are scaled, so that magnitudes of values in feature $\mathbf{x_i}$ are directly comparable to the magnitudes of values in feature $\mathbf{x_j}$ [25]. More concretely, if the input features have different units, then the obtained $\hat{\boldsymbol{\beta}}$ vector could be skewed, hiding the truly salient input features. For example, imagine a relationship like the following:

$$y = 128 + 1024x_1 + x_2$$

At first, it might seem clear that the x_1 variable is the most dominant term of the relationship because the β associated with it (*i.e.*, 1024) is so large. However, if x_1 is an input measured in kilobytes, x_2 is an input measured in bytes, and y is in terms of kilobytes, then it becomes clear that a change of 1KB for either x_1 or x_2 will yield the exact same change in y. This not only has implications on the quality of the model generated, but when ridge regression is used for subset selection, the means of selecting features with high predictive value are based on the magnitudes of the $\hat{\boldsymbol{\beta}}$ weights.

In order to avoid this phenomenon, both the input matrix as well as the known output vector need to become dimensionless. In other words, each vector (whether it be a column of the input matrix of the output vector) should have a mean of 0 and unit variance. This is often called obtaining the *z-score*.

Equation 4.8 shows how to obtain a standardized version of $\mathbf{X}$, $\mathbf{Z}$. In the equation, i indicates the ith row and j indicates the column, $\bar{x}_j$ indicates the mean of the jth column of $\mathbf{X}$, and s_{x_j} indicates the standard deviation of the jth column of $\mathbf{X}$:

$$z_{ij} = \frac{x_{ij} - \bar{x}_j}{s_{x_j}} \; i \in [1, \cdots, N], j \in [1, \cdots, p] \tag{4.8}$$

The same thing is done for the output vector $\mathbf{y}$ to obtain the standardized vector $\mathbf{y}'$:

$$y'_i = \frac{y_i - \bar{y}}{s_y} \; i \in [1, \cdots, N] \tag{4.9}$$

Using these dimensionless matrices $\mathbf{Z}$ and $\mathbf{y}'$ ensure a dimensionless solution, which indicates high-weight features with more clarity. The following equations, where β' is the standardized β, demonstrates how to return standardized β' values back to their original units [4]:

$$\hat{\beta}_j = (\frac{s_y}{s_{x_j}})\beta'_j \tag{4.10}$$

$$\hat{\beta}_0 = \bar{y} - \sum_{i \in features} \beta'_i \bar{x}_i \tag{4.11}$$

Category	Feature Template	Features
1	N/A	$w = weight$ $1/w$ $log(w)$ e^{-w} w^2
2	N/A	$s = sharedsize$ $1/s$ $log(s)$ e^{-s} s^2
3	N/A	$\sigma(occs)$ $max(occs)$ $min(occs)$ $max(occs) - min(occs)$
4	transform(Metric(CacheSize))	transforms = $\{\sigma, \mu\}$ $\times$ Metrics = $\{IPC, WIPC, MPI, MPA\}$ $\times$ CacheSize = $\{16KB, w, s, p, lru\}$
5	transform(Metric(CacheSize1) - Metric(CacheSize2))	transforms = $\{\sigma, \mu, max, min\}$ $\times$ Metrics = $\{IPC, WIPC, MPI, MPA\}$ $\times$ (CacheSize1, CacheSize2) = $\binom{CacheSize}{2}$

Table 4.2: 214 Member Feature Set - Representation of 214 features used in this study for model generation.

4.2.2 Isolating Features of Poor Performance

This study uses the ridge regression technique to isolate runtime attributes of poor performance in a shared cache. As was alluded to at the beginning of this chapter, the choice of features to use in building a model can be extremely important. Leaving out a relevant feature can lead to a faulty model. Adding superfluous or useless features is a lesser evil—if the model generation technique is capable of pruning useless features.

This section discusses the selection of features used in this study, as well as the procedure used to generate composite features (like $A \wedge B$) from singleton features (A, B).

Singleton Features

In the hopes of having ML be the sole influence in selecting the most salient input features, this study uses many, many features in the initial dataset. This avoids having too much human intuition directing the potential solution. These features fall into several subclasses, described below and listed in Table 4.2:

1. Five features that are based on baseline weight (as described in Chapter 3) are used to find potential relationships between poor performance and weighting. They are: $weight$, $1/w$, $log(w)$, e^{-w}, and w^2.

2. Five features based on overall shared cache size are used to find potential relationships between poor performance and total shared cache size. They are: $shared$, $1/s$, $log(s)$, e^{-s}, and s^2.

3. Four features relating to cache occupancy are used to find potential relationships between the distribution of cache occupancy in an LRU cache among benchmarks and poor performance. Occupancy is measured as the average occupancy over the course of the entire simulation. The first feature is the standard deviation of the measured occupancies of all benchmarks (denoted hereafter as $\sigma(occs)$), the second is the maximum of all occupancies ($max(occs)$), the third is the minimum ($min(occs)$), and the last is the delta between the maximum and minimum occupancies ($max(occs) - min(occs)$). Mean is not used here because ostensibly the mean of all the occupancies would be C/N, where C is total cache size and N is number of hardware threads.

4. Forty features are related to various performance metrics at particular cache sizes. These are best explained by example. Figure 4.1 is a plot of a sample workload: TPC-C, *perlbmk*, *povray*, and *twolf*. The x-axis is cache size, and the y-axis is the resulting IPC response, produced by the analytical transformations

on miss rate described in Chapter 3. The machine learning features in this category are transforms of the y-values for all benchmarks at a particular x-value. The x-values considered are: 16KB, *weight*, the cache size of a private, equal-share cache (p), s, and the measured occupancy in an LRU simulation (lru). Thus, the feature $\mu(IPC(s))$ is the mean of all the IPC values of the workload benchmarks at cache size s. These transforms (mean and standard deviation) are performed on IPC, weighted IPC (WIPC), misses per access (MPA), and misses per instruction (MPI). The cross-product of five cache sizes, four metrics, and two transforms leads to 40 distinct features.

5. Finally, 160 features are related to performance at two different cache sizes. Essentially, in an effort to avoid having to dynamically determine performance online the way marginal gain techniques do, the intent behind these features is to search for deltas in performance between two particular x-values on a plot like Figure 4.1. Thus, an example of a feature from this category is: $\sigma(IPC(p) - IPC(s))$, *i.e.*, the standard deviation of every benchmark's delta in IPC performance between cache size p and cache size s. Thus, for every feature, there are two x-values chosen out of five, resulting in ten possibilities. Each of these ten are used on four metrics, and each of these are transformed to mean, standard deviation, max, and min, yielding 160 features.

Composite Feature Generation Using YAGGA

It is easy to imagine that the relationship between workload behavioral features and poor performance could take the form of some sequence of pairwise relationships such as $A * B$ or A/B. Clearly, generating composite features of arbitrary length using 214 features and various pairwise operations quickly explodes into an intractable problem. To avoid this issue, this study uses YAGGA, a generating genetic algorithm implemented in the RapidMiner data-mining tool suite. YAGGA is essentially a

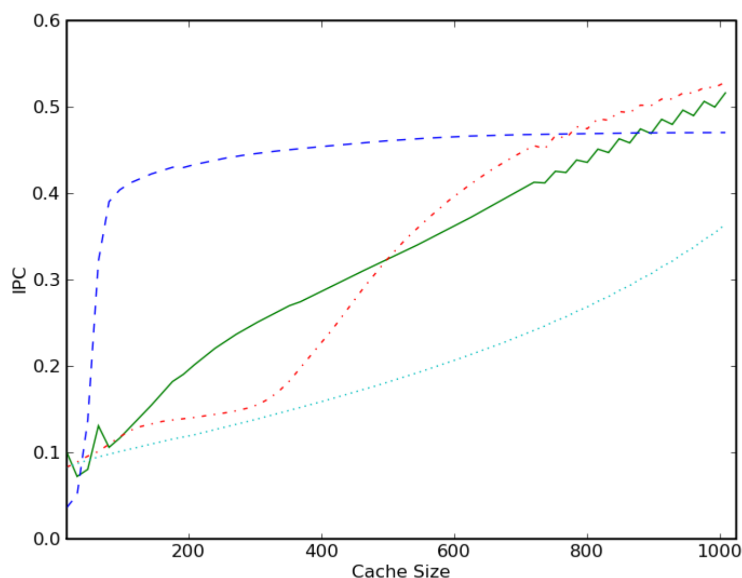


Figure 4.1: Sample Workload Plot - Sample plot of the workload consisting of benchmarks TPC-C, perlbmk, povray, and twolf. On the x-axis is cache size (in KB), and on the x-axis is IPC, as produced from CASPER miss rates and the analytical model described in Chapter 3.

search heuristic [42], and stems from a general class of genetic algorithms, which are computational procedures modeled after biological evolution and intended to yield the "fittest" features [43].

Genetic algorithms are aimed towards subset selection problems, by which the most relevant and valuable features in the context of a prediction model are selected from the feature set. Genetic algorithms do this in an evolutionary manner, building on ideas of inheritance, mutation, and survival of the fittest. Probabilistically generated subsets of the full feature set are used to build predictive models, and the best (fittest) feature subsets are kept with high probability for breeding the next generation of feature subsets. Breeding involves splicing two feature subsets together to form a new subset. There can also be mutations between generations to slightly perturb the solution space and avoid being stuck in local minima. Breeding new generations of feature subsets continues until M generations without improvement in the

generated model's predictive powers, or until a set number of generations is reached. The intuition is that by breeding "fitter" subsets, eventually the best feature subset for creating a useful model will be found, thus achieving the goal of subset selection.

While genetic algorithms are used for subset selection, they still do not address the problem of generating potential composite features. This is addressed with generating genetic algorithms, which modify genetic algorithms to also generate new composite features as part of the breeding process. YAGGA is a specific implementation of such an algorithm.

For both genetic and generating genetic algorithms, feature subsets are judged on their fitness and viability according to user-defined definitions of fitness.

4.2.3 Evaluation Environment

The 16 benchmarks used in this study can generate 1,820 four-thread workloads, each of which are run sharing a 1MB cache. These 1,820 workloads are broken up into a training set, consisting of 445 workloads, and a test set, consisting of the remaining 1,375 workloads.

YAGGA is trained on the 445-workload training set to both generate composite features from the 214 singleton features listed in Table 4.2, as well as select an optimal subset of the total feature set, consisting of both singleton and composite features. Fitness is judged by taking the selected subset and building a ridge regression model to model the relationship between the selected features and the relative performance between a statically partitioned cache for optimal Fair Speedup performance and an LRU cache. That model is tested for error, which is used at the fitness measure of that subset.

In this study, error is not so straightforward as raw error values—in this particular usage it is more important to predict correctly when there is a large delta between LRU and optimal performance. In other words, accurately knowing when LRU is

vastly underperforming optimal is more important than accurately knowing when LRU is only slightly underperforming against optimal. Thus, I use the measure shown in Equation 4.12 as the error calculation. By weighting the error value against the magnitude of the actual performance delta, mispredictions when performance deltas are large are punished more strongly than mispredictions when performance deltas are small.

$$error = abs(predictedValue - actualValue) * abs(actualValue) \tag{4.12}$$

The subset of features selected by YAGGA tends to be larger than what is needed by this study. This is partially on purpose; the seed parameter for deciding whether a feature is part of the selected subset or not is a user-defined probability. Having a probability that is too low reduces the "genetic variety" of each subset and makes for a breeding process that is not effective at feature generation. This study uses a value of $p = 0.25$, which means that the average feature subset examined from the initial 214 singleton features is around 54 features long.

In order to further reduce the feature subset into just the few most relevant, the final, fittest feature subset selected by YAGGA is then further fed into a simple ridge solver. This ridge solver provides the $\hat{\beta}$ values for each of the features selected by YAGGA. The ten features with the greatest magnitudes of $\hat{\beta}$, implying high relevance, are then selected for further evaluation.

These final ten are then selected, in order of the magnitude of their $\hat{\beta}$s, to create models using the top one feature, the top two features, and so on, all the way to using all ten. These simple ridge solvers are programmed in MATLAB®, which is simpler to program, enabling more flexible definitions of error. Recall that the final goal of this examination is to find salient runtime characteristics which indicate poor shared

cache performance; the implication is that if this poor performance can be predicted, then something can be done about it dynamically.

Fundamentally, this prediction of poor performance is a binary value. YAGGA is used to effectively generate and prune the feature space for predicting large magnitudes of performance deltas between an LRU and an optimally partitioned cache; now MATLAB is used to quantize these numeric predictions into binary "poor" and "acceptable" values. "Poor performance" is defined as having a Fair Speedup (FS) measure more than 20% worse than the FS performance of a statically and optimally partitioned cache of the same size.

With this sort of quantization, simple prediction error would merely encapsulate misprediction rates, *i.e.*, the percentage of times a workload was correctly labeled. However, recall that in this usage model it is more valuable to make correct predictions when there are large performance differences between LRU and optimal. In light of this distinction, the custom error measure shown in Equation 4.13 is used to determine the quality of a ridge model, assuming that a feature matrix $\mathbf{X}$ is multiplied against the model $\hat{\boldsymbol{\beta}}$ to yield a prediction vector $\hat{\mathbf{y}}$, with actual values represented by $\mathbf{y}$. Having an error measure as shown below creates a strong emphasis on getting predictions correct for large magnitudes of performance difference.

$$
\begin{aligned}
thresh &= 20\% \\
weightSum &= \sum_{i \in \hat{\mathbf{y}}} \begin{cases} 0 & \text{if } correctPrediction \\ (y - thresh)^2 & \text{if } mispredicted \end{cases} \\
N &= \sum_{i \in \hat{\mathbf{y}}} \begin{cases} 0 & \text{if } correctPrediction \\ 1 & \text{if } mispredicted \end{cases} \\
weight &= \sqrt{\frac{weightSum}{N}} \\
weightedError &= 0.75 * weight + 0.25 * mispredictRate
\end{aligned}
\tag{4.13}
$$

In the end, the best models with respect to the above weighted error are produced for one, two, three, and up to ten features. Each model is then tested against the 1,375 workloads not used in the training set to generate a final prediction error. From these, I make qualitative judgements on the marginal gain of reduction in prediction error versus increase in model complexity to determine a final model.

4.2.4 Results

From 214 singleton features, YAGGA selected 74 features, 42 of which are generated composite features. When these 74 features are fed into the simple ridge solver, the top ten features are:

1. $\sigma(occs)$
2. $\sigma(MPI(w))$
3. $\mu(IPC(p))$
4. $\sigma(MPI(p))$
5. $\mu(IPC(16KB) - IPC(lru))$
6. $\sigma(IPC(16KB) - IPC(lru))$
7. $\sigma(MPI(p)) * \mu(MPI(16KB) - MPI(lru))$
8. $max(WIPC(w) - WIPC(lru)) * \mu(MPA(w) - MPA(lru))$
9. $\mu(MPA(16KB) - MPA(w)) * \mu(IPC(w) - IPC(s))$
10. $\sigma(occs) * \sigma(MPI(16KB))$

Ideally, a model would include only a few features—obviously a 10-feature model using all the features just listed would be extremely complex. The only reason to

consider it would be if a model with 3 or so features does not provide reasonable predictive capability.

In the end, the model using the top three features, *i.e.*, the three features with the greatest $\hat{\beta}'$ values, yields the best "bang for buck" in attempting to optimize both complexity and accuracy. Excluding mispredictions within 5% of the 20% threshold (to emphasize accuracy when LRU and optimal performance differ widely), the model with the top two features yields a misprediction rate of 4.1%, while the top three yields a model with a misprediction rate of 2.76%, and the top four features yields a model with a misprediction rate of 2.83%. This highlights one of the unwieldy aspects of the ridge regression—increasing the number of features used in a model does not monotonically increase the accuracy of a prediction. This can occur because certain aspects of the model need to be present in groups; when using the top five features, the misprediction rate goes down to 2.03%, indicating the the fourth feature really needs the fifth feature to be included in the model to be useful. Clearly, the three-feature model optimizes simplicity and predictive capability.

The final features chosen from the list above are: 3, 6, and 8, leading to the model shown in Equation 4.14. A graphical representation of the results are shown in Figure 4.2.

$$
\begin{aligned}
y = \quad & 0.22673 \\
& -0.88603 * \mu(IPC(p)) \\
& +0.44529 * \sigma(IPC(16KB) - IPC(lru)) \\
& -2.64678 * \max(WIPC(w) - WIPC(lru)) * \mu(MPA(w) - MPA(lru))
\end{aligned}
\tag{4.14}
$$

Red bars indicate the model makes a correct prediction that LRU is within 20% of optimal FS performance. Blue bars indicate the model makes a correct prediction that LRU is more than 20% worse than optimal. Black bars indicate a false positive;

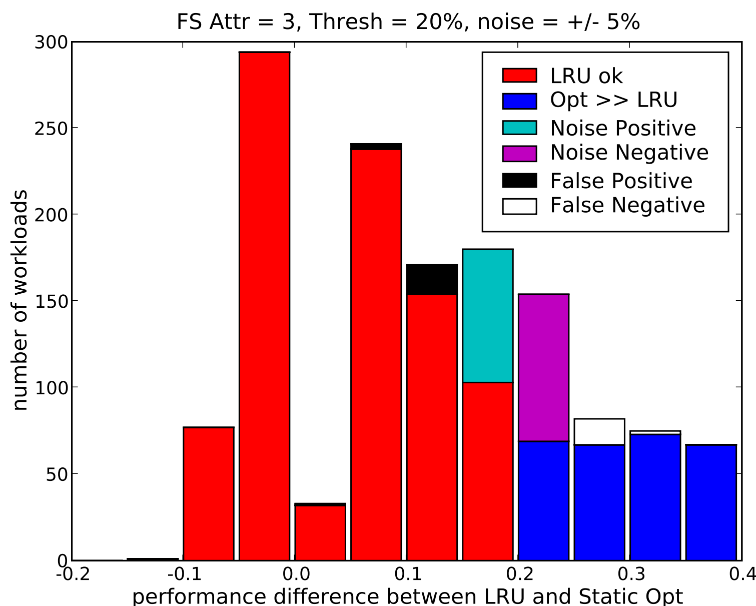


Figure 4.2: Stacked Histogram Results - A stacked histogram representing the results of the YAGGA/Ridge generated prediction model. The overall misprediction rate (the white, black, purple, and green bars bars, together) is 15.5%, with a $weightedError = 0.096$, and $weight = .080$, indicating a mean squared differential from the 20% threshold of 8%, for all mispredictions. When excluding the mispredictions that are within the 5% noise threshold, the misprediction rate goes down to 2.76%.

i.e., a workload was incorrectly predicted problematic, while white bars indicate a false negative, *i.e.*, a workload was incorrectly predicted to be sufficient under LRU. The light blue and purple bars indicate that, while predictions prove incorrect, they are within 5% of the threshold.

The results prove to be reasonably good—a total misprediction rate of 15.5%. The mean squared distance from the 20% threshold when mispredicted is 8%, meaning that mispredictions when far the the 20% threshold are rare. When ignoring the mispredictions within the 5% noise limit, the misprediction rate is only 2.76%. While the results prove to be reasonably good, particularly since the number of outright mispredictions at the extreme ends of the histogram in Figure 4.2 are very few, there is little intuition with this model. The ability to translate the model in Equation 4.14

to a runtime detector in a shared cache for detecting poor behavior is tenuous at best, and the utility of the model in guiding the mechanism for mitigating poor performance is virtually nonexistent. Thus, this study, while successful in building an accurate analytical model, does not provide the hoped for insight into building a simple run time predictor for poor performance.

4.2.5 Conclusions

This study makes a number of methodological decisions that, in hindsight, turn out to be sub-optimal. From this experience I draw the following conclusions:

- When using machine learning to extract salient features through feature selection for a certain outcome, it is important to realize that the power of ML to zone in on a uniquely optimal model is only as good as the differences between the features in the feature set fed to it. For example, given a feature set consisting of MPA(16KB) and MPI(16KB), the ML mechanism could be hard pressed to strongly be in favor of either feature because they are so highly related. ML can be counted on, however, to realize that MPA(16KB) is likely much more useful than the age of the owner of the machine a workload is running on for predicting poor cache performance. Thus, it is overly optimistic to count on ML for teasing out minimally optimal models in an extremely massive search space.

- In a related note, using such highly correlated features in conjunction with a generating genetic algorithm (GGA) for composite feature generation proves to be somewhat unstable. The idea behind a GGA is to expand the feature space prudently in a probabilistic way, and prune the feature space prudently in a probabilistic way. With such a huge feature space and such correlated features, it is not always clear how to be prudent. Something else is needed for composite

feature generation.

- Generating complicated singleton features such as $\sigma(MPI(s) - MPI(lru))$ is impractical, and the assumption that the very best and most salient features will be extracted and provide insights to solutions turns out to be optimistic. Not only does the model generated in this study not provide clear insights into how to alleviate poor shared cache performance, it also does not provide clear insights into how to build realizable detector in hardware. The intention behind feeding a wide, detailed, and varied set of features in to ML was to find a highly accurate model first, and then dial back the precision in order to build a realizable model. However, with this feature set and with this final model, this proves to be an impractical approach. At best, this provides a baseline comparison for more practical predictors.

4.3 Decision Tree Analysis

4.3.1 Background

This section relies on a different approach than ridge regression and YAGGA, and uses decision tree analysis to find and extract salient runtime features for poor cache sharing performance. Decision tree learning leverages the concepts in information theory to associate features to outcomes in a simple manner.

In a decision tree, each node represents a feature, while each branch represents a value, or a range of values, for that feature. The leaf nodes of the tree provide a classification into a bin for the outcome in question, which must be discrete. For the purposes of this thesis, the leaf nodes would classify a workload datapoint into "poor" or "acceptable" categories, where "poor" and "acceptable" are defined in the same way as in the YAGGA/ridge study.

The process of decision tree model creation utilizes the concepts within information theory and the idea of entropy [43], where entropy is the level of variability in the values of a random variable. Mathematically, for a dataset where y can have values of $\{y_1, y_2, y_3, ...y_m\}$ and $p(y_i)$ indicates the probability that $y = y_i$, entropy is defined as:

$$Entropy(\mathbf{y}) = -\sum_{i=0}^{m} p(y_i) log_2(p(y_i)) \tag{4.15}$$

As an example, a dataset with output vector $\mathbf{y}$ that has 50% of datapoints with $y = True$ and 50% with $y = False$ has maximal uncertainty—if the value of y is unknown, there is no educated guess that is better than another. This distribution has maximal entropy. However, if the distribution is such that there are 85% of datapoints with $y = True$ and 15% of datapoints with $y = False$, then even if the value of y for a new datapoint is unknown, betting that $y = True$ is a good guess.

The goal of a decision tree is to have as few nodes as possible en route to leaf nodes which classify datapoints into values of y with good certainty, *i.e.*, minimal entropy. The most basic algorithm recursively divides a dataset to maximize information gain, *i.e.*, choosing the best feature that will minimize entropy for the next node. Information gain $Gain(\mathbf{y}, F)$ is the expected reduction in entropy from sorting on feature F, and is mathematically defined as:

$$Gain(\mathbf{y}, F) = Entropy(\mathbf{y}) - \sum_{f \in F} \frac{|\mathbf{y}_f|}{|\mathbf{y}|} Entropy(\mathbf{y_f}) \tag{4.16}$$

One issue with recursing based on information gain is that attributes with many different values can be selected first, without providing much actual predictive gain. For example, an airline attempting to build a model predicting behavior of their frequent flyers could potentially build a model with frequent flyer number as the root node attribute. This would minimize entropy, but yield little actual value because of

the one-to-one relationship of frequent flyer number to customer. Building a decision tree based on information gain in this case would yield a model which begins with something like, “If passenger’s frequent flyer number is 1234567, then...” which clearly is not helpful to the airline.

The way around this is to use gain ratio as the decision-making parameter for forming a node, rather than information gain. The intuition behind gain ratio is to specifically counter the useless entropy reduction of a many-valued feature F by considering the entropy of $\mathbf{F}$ itself. Gain ratio is the ratio between information gain from sorting on F and the entropy of the vector $\mathbf{F}$ itself; selecting a node based on this attribute reduces the chances of the airline scenario just described. Mathematically, gain ratio is:

$$\begin{aligned} SplitInformation(\mathbf{y}, F) &= Entropy(\mathbf{F}) \\ GainRatio(\mathbf{y}, F) &= \frac{Gain(\mathbf{y}, F)}{SplitInformation(\mathbf{y}, F)} \end{aligned} \tag{4.17}$$

Decision trees can also easily be used for features with continuous numerical values rather than just discrete values—the values merely need to be converted to discrete form, *e.g.*, a feature like `Weight` from Table 4.1 could be converted into a true/false feature like `Weight > 135`, or even a multi-valued feature with possible values `Weight < 120, 120 <= Weight < 135, 135 <= Weight < 150`.

A decision tree (DT) model alleviates the feature generation difficulty faced by the YAGGA technique because composite features do not need to be generated explicitly—instead, the construction of the tree creates composite features. All paths from root to leaf represent a composite feature where the features of the path nodes are connected by an AND relationship. With the branches representing values, a tree could easily be constructed to mean $(A \wedge B)||(\neg B)$. While the ability to generate more complex pairwise relationships such as $\frac{A}{B}$ or $\frac{1}{A}$ (when A is numeric) is lost when moving from YAGGA to DTs, the experience with YAGGA indicates that simplicity

is a boon. Additionally, given that part of the problem with YAGGA is the explosion of problem space, eliminating unlikely pairwise relationships between singleton features is hardly a loss.

4.3.2 Isolating Features of Poor Performance

Based on the lessons from the YAGGA/ridge approach, this study reduces the feature set used by a great deal. Additionally, other modifications are made to the feature set as a result of experiences in the YAGGA/ridge study. First, the features based on w and s never showed up in any top models, so those are eliminated here. Second, in the YAGGA/Ridge study, features were constructed based on the interrelationships between all benchmarks running together, at a given cache point, *e.g.*, $\mu(IPC(w))$. While this approach does emphasize the dependence of a workload's behavior on the relationships between threads, this approach masks the effect of a single benchmark's behavior on shared cache performance. In particular, during the iterative process of building the YAGGA/ridge methodology, I found myself struggling with interpreting features that rely on standard deviations, for which I could not associate the high values or the low values with any benchmark. Even worse was when models were generated with features relying on the σ of one metric, in addition to the σ of another metric. In these cases I would wonder whether the high values of one were associated with the high values of another.

Thus, the features in this study are constructed based on two principles, detailed below:

- Primarily there is a focus on simplicity; rather than having features of the form $\sigma(MPA(w) - MPA(s))$, which are not only complicated to grasp immediately, but certainly difficult to measure effectively in a running system, this study focuses largely on simpler features.

Category	Feature Template	Features
1	$Metric1(Place(Metric2))$	$Metric1 = \{MPA, MPI, MPC, IPC, Occ\}$ $\times$ $Metric2 = \{MPA, MPI, MPC, IPC, Occ\}$ $\times$ $Place = \{Biggest, 2nd, 3rd, Smallest\}$
2	$transform(Metric)$	$transform = \{max, min, \sigma, \mu, max - min\}$ $\times$ $Metric = \{MPA, MPI, MPC, IPC, Occ\}$

Table 4.3: 125 Member Feature Set - Table describing the 125 features for the decision tree approach. Occ refers to the percentage of occupancies for each benchmark, not the absolute space usage.

- To avoid masking the behaviors of individual applications without losing a sense of interplay between them, this study ties individual performance to their relative ordering in the workload; *i.e.*, there is a set of features as follows: $Largest(MPA)$, $2nd(MPA)$, $3rd(MPA)$, $Smallest(MPA)$. Additionally, there are feature sets of the form $Occ(Biggest(MPA))$, $Occ(2nd(MPA))$, $Occ(3rd(MPA))$, $Occ(Smallest(MPA))$ to indicate the associated occupancies of each workload, in order. These sorts of features allowed for a clearer view of the relations between benchmarks in the workloads.

In the end, there are two classes of features for this study. The first is of the form: $Metric1(Place(Metric2))$, such as $Occ(Largest(MPA))$ as just described. $Metric1$ and $Metric2$ can be $\{MPA, MPI, MPC, IPC, Occ\}$, while $Place$ ranges from "Largest" to "Smallest", yielding 100 distinct features. The other class of features is more relational, as in the YAGGA/ridge study, and are of the form $Transform(Metric)$, *e.g.*, $max(MPA)$, totaling 125 features. A table of all features used in this study is shown in Table 4.3.

4.3.3 Evaluation Environment

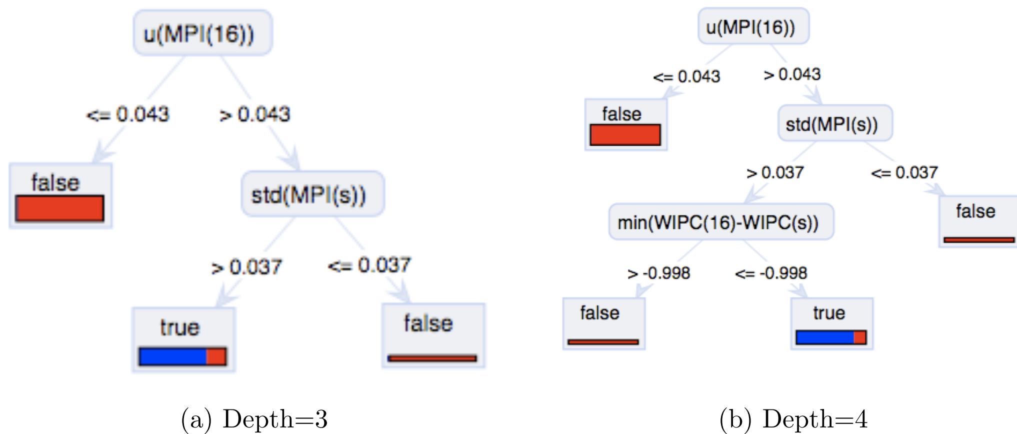
This investigation uses the decision tree implementation available in the RapidMiner tool suite [42]. There are two primary user-specified parameters in this tool. The first is the criterion for deciding which feature to use for the next node, *i.e.*, information gain or gain ratio, as described in the previous section. The other parameter is the depth of the tree to be used; in order to limit the complexity of the models produced, this study uses three to four levels.

The data points used in this study are the same as the YAGGA/ridge study, with 445 four-application workloads as the training set, used to create the model, and 1375 four-application workloads as the test set, used to test the model. All prediction results are with respect to performance on the test set.

The primary goal of this study is to learn something valuable for the construction of a useful mechanism for detecting and alleviating poor shared cache performance. The YAGGA/ridge study demonstrates the unwieldiness of using YAGGA for feature generation; it also shows the potential folly of using overly complex features as part of the dataset.

As a result, to separate the effects of ML technique from the selection of features used, this study begins first with using the exact same feature set as in the previous YAGGA/ridge study but with the decision tree model-building technique. Subsequent examinations involve the decision tree technique with the feature set just described in the previous section.

Note that in this study, because decision trees must have discrete classification at the leaves, there is no custom error measure as in the previous YAGGA/ridge study. Those custom error measures are enabled by quantitative error values; here, error is judged purely in misprediction rates.



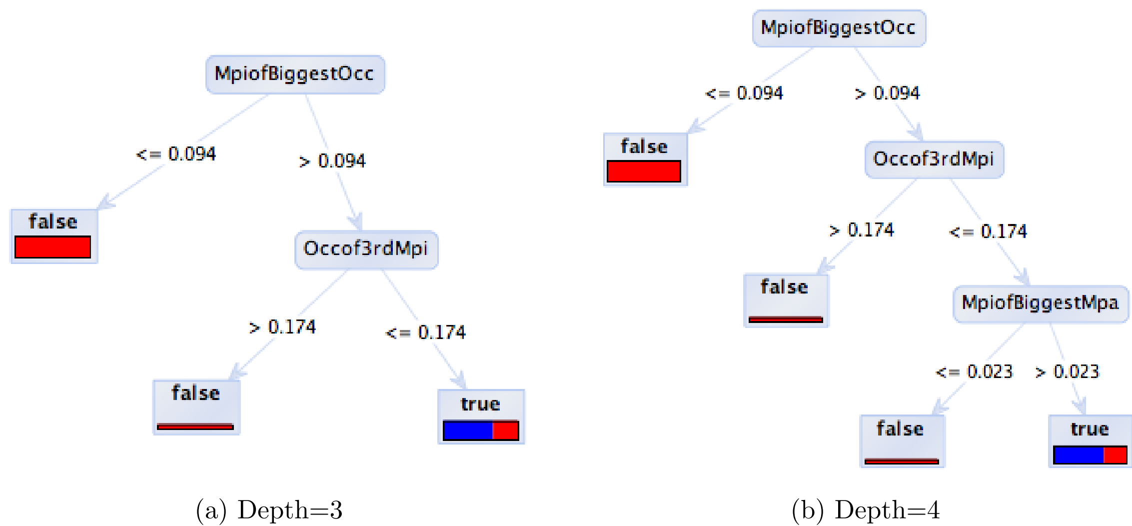
(a) Depth=3 (b) Depth=4

Figure 4.3: Decision Tree: YAGGA Feature Set - Decision Tree produced when using gain ratio as the node decision factor and the feature set used in the YAGGA/ridge study. A three-level model results in a 12.5% misprediction rate, while the four-level model results in a 10.7% misprediction rate. Compare this to the 15.5% misprediction rate achieved with the YAGGA/ridge methodology.

4.3.4 Results

The tree produced when using the feature set from the YAGGA/ridge study with three levels of tree and using gain ratio for making node decisions is shown in Figure 4.3. When using a tree depth of three, the model mispredicts 12.5% of the 1375 workload test set, and when using a tree depth of four, the misprediction rate drops to 10.7%. Not only does this model prove to be more accurate than the YAGGA/ridge model, but it is simpler to understand as well. This result demonstrates the better suitability of decision tree analysis for this investigation. In addition, fewer steps are required to produce this model (as indicated by the methodological descriptions), and the compute time is reduced by several orders of magnitude. From start to finish, a YAGGA/ridge model can take several hours to compute, while the decision tree models shown take several seconds.

In the depth four tree, the third order predictor for poor performance is $min(WIPC(16KB) - WIPC(s))$, which is a harder feature to grasp and certainly difficult to measure in real time on a system. Now that decision tree learning is



(a) Depth=3 (b) Depth=4

Figure 4.4: Decision Tree: New Feature Set - When using the full 125 features, a three-level tree results in a 16.4% misprediction rate, while a four-level tree results in a misprediction rate of 14.98%.

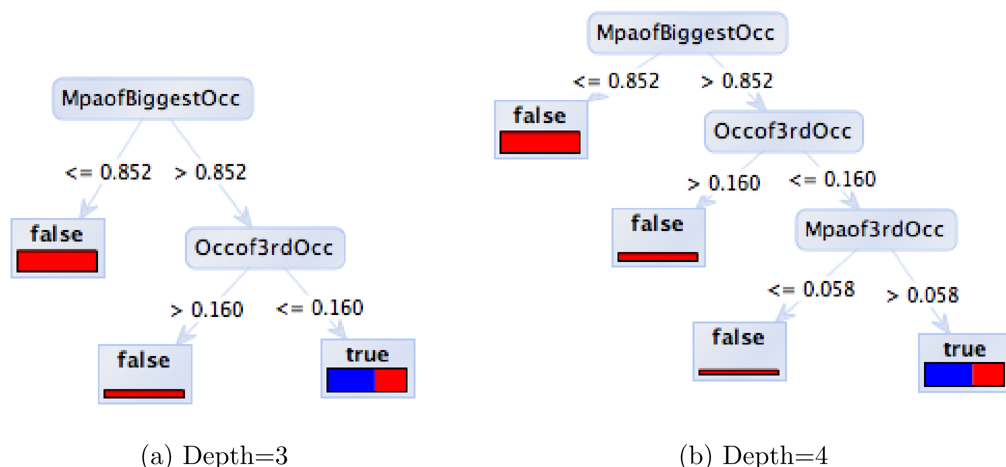
confirmed to be less complicated as well as more accurate than YAGGA/ridge, this study moves towards finding an even simpler model using more intuitive features.

Using the feature set described in Section 4.3.2, the decision tree technique is applied; again using three to four levels of tree depth and gain ratio as the node criterion. The results of using all 125 features are shown in Figure 4.4. What is clear from this is that the single most important factor in determining whether a shared cache will exhibit poor performance is if the MPI of the application occupying the largest portion of cache is above 0.094 (since this feature is the root node of both trees). Note that the DT algorithm discretized the MPI feature for optimal gain ratio, and the result is this 0.094 value shown in the figure. Put more abstractly, if an application is occupying a large amount of cache space but has an MPI higher than what should be expected, considering the cache space occupied, then the workload is likely to show poor cache performance. This is a helpful feature because not only is the meaning clear, but it points the way towards a possible way to mitigate the performance problem—reduce the cache occupancy of the biggest cache occupier.

The secondary factors are further down in the tree, and their meaning is less glaringly clear. The second node, which translates literally to "If the application with the 3rd largest MPI value, *i.e.*, the application with the 2nd best MPI (since lower MPI is better), has an occupancy less than 17.4%, then this workload is likely to have a performance problem." In the context of trying to build a realizable detector in hardware, this requires an additional level of translation to be meaningful. However, the crux of this DT model is not totally clear. This secondary factor could mean that the 2nd best performer is only nominally so, because the greedy top performer is pushing down the performance of all other threads. It could also mean that in caches where the 2nd best performer is capable of doing well despite a small cache allocation, there is a performance problem. The tree model is incapable of making this distinction without human intervention to pinpoint the true meaning with better precision.

Intuition would clearly imply that the true meaning of the secondary branch factor in Figure 4.4 is the former; that the 2nd best performer is actually a low performer because the best performer is greedy and has taken up much of the cache. Fortunately, since decision trees can be built in a matter of seconds, the feature set can be manipulated in order to create models with more clarity.

Through an iterative process, this study pares down the feature set to a limited subset in order to maximize the clarity of meaning of the resultant decision tree, as well as to maximize the buildability of an on-line predictor. References to MPI, MPC, and IPC are eliminated such that all features are related to either occupancy or MPA. The reasoning behind this stems from the idea of measurability in a real system. MPI and IPC rely on information about instructions, which is inherently tied to the execution core of a system, not the memory subsystem. Thus measuring this information cannot be done locally at the cache without some design changes extending beyond the cache. Since this study aims to find a simple way to detect



(a) Depth=3

(b) Depth=4

Figure 4.5: Reduced Feature Decision Tree - When using the reduced set of 25 features, a three-level tree results in a 20.2% misprediction rate, while a four-level tree results in a misprediction rate of 18.7%.

poor cache performance at runtime, removing these features from contention will show whether this can be done using only features that are easily measurable dynamically.

Removing MPI and IPC leaves occupancy, MPA, and MPC. Occupancy is clearly an important factor that is distinct from MPA and MPC, while MPA and MPC are clearly somewhat correlated. Design-wise, in a runtime prediction system the fewer things that need to be tracked the better, and for a small loss in predictive precision results from removing either MPA or MPC, the gain in hardware simplicity is likely to be worth it. MPA is kept in favor of MPC because of the locality of measuring both misses and accesses at the cache. Thus, out of the 125 features originally used, 25 remain. Results for this feature set are shown in Figure 4.5. These are the final results of this study, which qualitatively optimize tree simplicity, intuition provided, and potential detector buildability.

The clear trend in moving from a large, complex feature set to a smaller feature set consisting of simpler, more measurable features is a distinct reduction in prediction accuracy. However, the primary goal of this investigation is not to produce the best predictive model, but to produce the *best predictive model that leads to a simple,*

buildable mechanism for detecting poor performance in a shared cache. As expected, there is a tradeoff between accuracy and simplicity. However, even at the level of simplicity in the reduced feature set model at three levels of depth, there is a clear message and a reasonable amount of knowledge to build upon. This reduced feature set produces a three-level tree with a 20.2% misprediction rate. Put another way, nearly 80% of poor performers are discovered with this model.

Translating the DT model shown in Figure 4.5 indicates that if the miss rate of the largest cache occupier is greater than 85.2%, and the occupancy of the 3rd largest cache occupier is less than 16%, then the workload is very likely to have a performance problem. Thinking more deeply about this, a miss rate of the largest cache occupier says nothing about whether the largest cache occupier is actually large, *i.e.*, an application could have a miss rate of 92% and occupy 28% of the cache (the rest of the applications having 24% each), or an application could have a miss rate of 92% and occupy 80% of the cache, leaving the remaining 20% to be split among the remaining three applications. Increasing the accuracy of prediction requires the second node, which says the 3rd largest occupier is 16% or less. Thus, the smallest occupier is 16% or less as well, meaning the two largest occupiers of the cache take up at least 68% of the cache.

In more abstract terms, pegging workloads as poor cache performers when an application in that workload occupies a large amount of cache while still maintaining a high cache miss rate will net most of the actual poor performers. For this level of simplicity this is extremely useful information. In hindsight, it is also somewhat obvious. However, nothing in the literature takes advantage of this connection. The next chapter of this dissertation attempts to build a mechanism based on this result.

To further increase predictive accuracy, the four-depth tree has one more decision node to consider, which adds an additional stipulation for classifying a workload as a poor performer. This stipulation states that only when the MPA of the 3rd occupier is

greater than 5.8% is the workload problematic. What this really shows that if the 3rd occupier, which has been squeezed into 16% or less of the cache, has a very low miss rate, then there actually is not a performance problem. In other words, it is unlikely the 3rd occupier is being negatively affected by having a small cache allocation if its miss rate registers below 5.8%, so letting the large occupiers have their large allocation is not a problem. This minor addition improves prediction accuracy by 1.5%, to 18.7%, by refining the decision process for determining whether a workload has poor performance. The additional 1.5% increase in prediction accuracy represents about 20 workloads of the 1,375 workload test set.

With this result, this study achieves its aim of extracting a few simple and salient characteristics for detecting poor shared cache performance during runtime. While the prediction rate is not especially high, the utility of the model meets the intent of this study, and the knowledge that a few simple characteristics cannot fully predict poor performance is not unexpected.

At this point, it is worth breaking down the error rates of these models into more detail. The results for the reduced feature set decision trees are shown in Table 4.4. The vertical columns “Poor” and “Acceptable” represent the actual nature of the workloads tested, while the horizontal columns indicate the predictions generated by the tree. Class Precision and Recall are concepts in information science. *Precision* refers to the purity of predictions; in this case, if all “Poor” predictions are actually poor, then the tree has good precision. *Recall*, on the other hand, refers to ability of the predictor to detect every case correctly; in this study, good recall would entail tagging most poor workloads as “Poor”.

What is clear from these results is that the primary source of misprediction is in falsely predicting a workload has poor performance when in fact it does not. In general, the vast majority of workloads which truly are poor performers are correctly cast as such (*i.e.*, good recall). This is also valuable information to consider when

	Poor	Acceptable	Class Precision
Pred. Poor	366	266	57.92%
Pred. Acceptable	12	731	98.38%
Class Recall	96.83%	73.32%	

(a) Depth=3

	Poor	Acceptable	Class Precision
Pred. Poor	363	243	59.90%
Pred. Acceptable	15	754	98.05%
Class Recall	96.03%	75.63%	

(b) Depth=4

Table 4.4: Learning Results - Table showing the results of decision tree analysis on the 1375 workload test set using the reduced feature set models from Figure 4.5.

crafting a mechanism for detecting poor performance. This result indicates that what a mechanism built based on these models must guard against is false positives; at the same time, this mechanism is unlikely to miss many opportunities, as it will likely detect most problematic scenarios.

4.4 Conclusions

This study achieves the stated goal of using a machine learning technique to discover simple, salient characteristics for poor shared cache performance. Results show that the first-order predictor for poor performance is when the largest cache occupier in a workload also has a high cache miss rate. A second-order predictor adds that the "largest" must indeed be quite disproportionately large. Finally, a third-order predictor states that even if the two conditions above are present, there is only a performance problem if the smallest cache occupiers suffer from non-trivial miss rates.

These simple predictors can easily be used to craft a dynamic mechanism for both predicting and alleviated poor shared cache performance in a real-time system, which is done in the next chapter of this dissertation.

This chapter of the dissertation also provides useful conclusions about the nature of machine learning as a research tool. The results dispel the notion that machine learning can be relied on to automatically extract useful conclusions without intelligent human intervention. In the first place, it is extremely important to use the

appropriate ML mechanism. Ridge regression makes it difficult to generate composite features, and this study finds that rather than using a complicated iterative technique like YAGGA to probabilistically generate composite features, a much simpler technique like decision tree learning can be used to more directly find useful composite features, and with better accuracy. The experience from this chapter shows that the tuning required to make a complex mechanism like YAGGA/ridge useful makes the technique unattractive. Researchers are likely better served using simple ML techniques, particularly when seeking generalizations rather than precise models.

The other major conclusion of this chapter is the paramount role features play in building useful models. Machine learning is not an oracle; in order for it to give a useful answer it must be fed useful features from which to build an answer. At the same time, giving an ML block every potential feature imaginable may not result in the most useful model—unless the ML technique is sensitive enough to parse through noisy data to find an optimal model. For best results, the search space needs to be discrete enough that the ML can more easily separate good results from bad ones.

The next chapter attempts to build on the results from the decision tree models to create a feasible run-time detector for poor shared cache performance. Given that the crux of poor performance is having a large cache occupier which also has a poor miss rate, it seems logical that reducing the cache occupancy of this large occupier would be a viable way to improve shared cache performance. This hypothesis is tested in the next chapter by building a mechanism in an architectural simulator to investigate.

CHAPTER 5

Scalable Lightweight Adaptive Management

The work from the previous chapter indicates that a mechanism for detecting poor shared cache performance should rely on detecting when a cache sharer occupies a disproportionately large amount of space while maintaining a high cache miss rate. I now turn to the crafting of such a mechanism, propose a scalable and feasible solution based on the findings of the previous chapter, and provide an evaluation of this mechanism's ability to improve shared cache performance.

To motivate this approach, recall from Chapter 2 that most proposed cache management schemes *proactively* search for a partitioning of cache capacity that is optimal under some metric [18, 35, 48, 57, 65]. These proposals iteratively assign allocations to threads for the duration of an epoch, monitor the resulting performance, then adjust the allocations accordingly for the next epoch. This approach has several shortcomings. For large-scale platforms (potentially up to hundreds of threads by 2015 [9]), exploration of the space of possible allocations will be very slow, and may never approach a optimal solution within a relevant timeframe as the optimal solution shifts due to program phase changes and context switches. Additionally, determining the appropriate epoch length is difficult [51, 56]. Finally, identifying an appropriate and feasible on-line metric to optimize requires making seemingly arbitrary choices among a variety of reasonable throughput and performance metrics that can lead to

different allocations, as demonstrated in Chapter 3. Only several more recent approaches attempt to improve shared cache performance without explicitly seeking an optimal partition [33, 54].

In this study, I propose a new *reactive* approach to shared cache management. Instead of searching for an optimal partition, this mechanism seeks to detect poor performance and mitigate it when it happens, doing nothing otherwise. The detection mechanism design is guided by data from the machine learning study from Chapter 4 which indicates that the most important factors contributing to poor performance is when a sharer of the cache has a miss rate unbefitting its cache occupancy. Because common LRU and pseudo-LRU replacement policies perform well for shared caches in many situations, as shown in Chapter 3, there is often no need for further optimization. These results from the previous two chapters lend credence to the hypothesis that a reactive approach from an LRU baseline is feasible.

The experiments in this chapter show that

1. identifying and throttling individual threads that cause poor performance is sufficient to improve overall throughput, rather than attempting to find a global optimum operating point;

2. these problematic threads can be identified using very simple, scalable extensions of performance counters provided by existing platforms; and

3. this thread-local approach lends itself to providing differential quality of service merely by adjusting the criteria by which a thread is considered problematic.

Specifically, problematic threads are identified as those that use the cache inefficiently, bringing in (and displacing) a disproportionate share of data while still suffering a high miss rate. The genesis of these criteria from their ML roots are discussed further in Section 5.1. These problematic threads can be detected using two simple components—the Misses Per Access Counter (MPAC) and the Relative

Insertion Tracker (RIT)—which provide weighted historical averages of each thread's local miss rate and its cache insertion (miss) rate relative to other threads. These mechanisms require a total of three counters and one register per thread (less than 50 bits of state), plus one global counter for the platform, and are driven off the same events that are typically already collected to drive performance counters on current platforms. As a result, these components do not require any changes to the core of a conventional cache design and scale easily with increasing thread counts.

These lightweight performance monitors are combined with a thresholding mechanism and a simple throttling control to create a framework called Scalable, Lightweight, Adaptive Management (SLAM) for shared caches. A thread whose MPAC and RIT values both cross their respective thresholds, indicating a high miss rate despite a disproportionate rate of cache insertions, is prevented from displacing blocks belonging to more efficient threads by modifying the replacement priority of its newly allocated blocks [39, 47].

SLAM also lends itself to providing differential quality of service by using different MPAC and RIT thresholds to determine whether a thread should be throttled. Assigning larger thresholds to a thread indicates that the thread has higher priority, and thus can be allowed to use the cache less efficiently than its peers before being penalized. Conversely, lower thresholds cause SLAM to hold threads to a higher efficiency standard, penalizing them at efficiency levels that would otherwise be acceptable.

This chapter makes the following contributions:

- it presents a novel conceptual approach to shared cache management, based on reactive throttling of inefficient threads rather than proactive optimization of cache policies, which is more scalable as thread counts increase.

- it shows that inefficient threads can be detected adequately using simple, scalable, low-overhead mechanisms—MPAC and RIT—that do not require modifications to a conventional cache structure.

- it combines these simple monitoring mechanisms with a throttling mechanism based on LRU priority insertion to create SLAM.
- it demonstrates via simulation that SLAM provides performance optimization competitive with the best published approach across hundreds of workload mixes and several cache sizes.
- it shows that, unlike prior performance optimization approaches, SLAM also enables thread prioritization via varied per-thread thresholds.

5.1 SLAM

Recall from Chapter 4 that a workload has a high likelihood of poor shared cache performance if a thread occupies a large portion of the cache while maintaining a high miss rate. In essence, this is really making a statement about the efficiency of cache usage—intuitively a cache that occupies a large cache allocation should also have a low miss rate. Prediction accuracy can be improved by adding the stipulation that a low-occupancy thread have a non-trivial miss rate. In crafting the SLAM framework, the primary focus is proposing a mechanism that is not just effective, but scalable and feasible. With this in mind, the additional stipulation is not considered for the first incarnation of SLAM.

Also, recall from Section 2.2.1 that tracking the occupancy of multiple threads of a cache is a non-trivial endeavor, particular as thread counts scale. This makes measuring occupancy for this detection mechanism a difficult endeavor. However, it seems likely that the crux of the result from Chapter 4 is not about occupancy, but rather about cache efficiency. I make the observation that tying cache *insertions* to performance is a very similar measure of efficiency—if a thread is bringing in many blocks to the cache, then it should also have a low miss rate. Since insertions are much easier and much lower overhead to track, SLAM is built around insertion rates

rather than occupancy.

Thus, the SLAM framework seeks to detect when a hardware thread's utilization of the cache falls below some threshold of efficiency. If it does, then its usage of the cache should be reduced in order to allow the cache to be used by more efficient threads. The idea can be stated in simple terms: if a thread is responsible for bringing some proportion of blocks into the cache, it should have a reasonable miss rate to show for it. We describe the concept here; specific definitions of "reasonable" are addressed in Section 5.1.3.

Every workload execution can be characterized by a graph such as the one shown in Figure 5.1, where the relative insertion ratios of the various hardware threads in the execution are plotted on the x-axis, against the resultant miss rate on the y-axis. The figure shows results from a workload used in our experiments (Mix 8 in Table 5.1 sharing 8MB of cache). Viscerally, it seems clear that the right-most point, with a 48% miss rate and responsible for nearly 57% of the insertions to the cache, is not using the cache efficiently and should be restricted from displacing potentially useful blocks from other threads. Given a certain standard of efficiency, we can identify inefficient threads in this manner. While more complicated schemes are certainly possible, *e.g*, curve-based thresholds, we use a quadrant-based technique for implementation simplicity and leave exploration of more sophisticated mechanisms for future work.

5.1.1 Detecting Poor Utilization

The key to detecting whether threads are in the poor utilization zone at runtime is tracking the x- and y-axis values from Figure 5.1 effectively. Once a thread's behavior has been placed somewhere in the plot, it is easy to deem it inefficient based on threshold checking. We describe how we track these values in the following subsections, and discuss how we determined appropriate threshold values in Section 5.1.3.

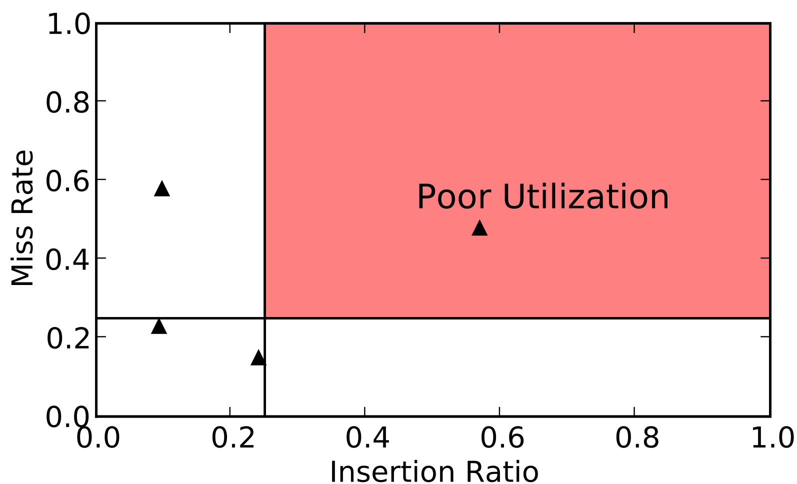


Figure 5.1: SLAM intuition illustration using real data from Mix 8 in Table 5.1 sharing 8MB of cache. Each point represents an application in the workload and its associated runtime properties. Clearly, the far-right point (*lbm*) is not using the cache efficiently.

MPAC - Misses Per Access Counter

An exact measurement of misses per access (MPA) would require a division, which is clearly to costly too implement in a real system. Instead, we present *MPAC* (Misses Per Access Counter), a novel technique which approximates the misses per access value and requires only two counters per thread. One counter is incremented when a miss is incurred by its thread, and another is incremented on every access from that thread. When the access counter saturates and rolls over, the miss counter is right-shifted a single bit, thus halving the value. This study uses a counter ten bits wide, which is experimentally shown to be sufficient for high performance. Essentially, this is a representation of the weighted history of miss rate. The miss counter will never exceed 11 bits[1], so the hardware overhead is merely two counters of 10- and 11-bits, plus the minimal logic to right shift upon rollover. We use this implementation for all of our experiments.

[1] This counter is upper-bounded by the geometric series $1024 \times \sum_{i=0}^{n} (\frac{1}{2})^i \rightarrow 2048$.

Because MPAC is a weighted history of misses rate, we can easily translate MPAC values to miss rate percentages. For example, an MPAC value of 2,047 represents a weighted history of 100% miss rate, while an MPAC value of 1,024 translates to a weighted history of 50% miss rate.

RIT - Relative Insertion Tracker

The typical measure of utility in a cache relates miss rate to cache occupancy. However, an exact measurement of cache occupancy would involve a counter per thread that increments whenever a line is brought into the cache on behalf of that thread, and decrements every time a line from that thread is evicted. This necessitates that every cache line be tagged with a thread ID to know which counter to decrement upon the eviction of a line.

To avoid this overhead, we present *Relative Insertion Tracker* (RIT) which tracks only the relative *insertions* into the cache, rather than occupancy. As a result, RIT obviates the need to pay attention to evictions. At the same time, RIT maintains the information levels needed to make an informed judgment regarding utility. A thread that inserts many lines ought to be using them, *i.e.* not have a high miss rate.

To implement this solution, we merely need one counter and one register per thread, plus an additional global counter per shared cache. Every time an insertion is made to the cache, the global insertion counter and appropriate thread insertion counter are incremented. When the global counter saturates and rolls over, per-thread insertion counter values for all threads are read into their per-thread registers. Then, the per-thread insertion counters are right-shifted one bit to provide a weighted history of relative insertions across all threads. When a threshold check is performed, it is done on the register, not the counter, to avoid sawtoothing from the right-shifts. For symmetry, we use the same counter width as MPAC for the global counter (10 bits), and each thread's RIT is subject to the same geometric series rule as MPAC,

with a maximum value of 2,047.

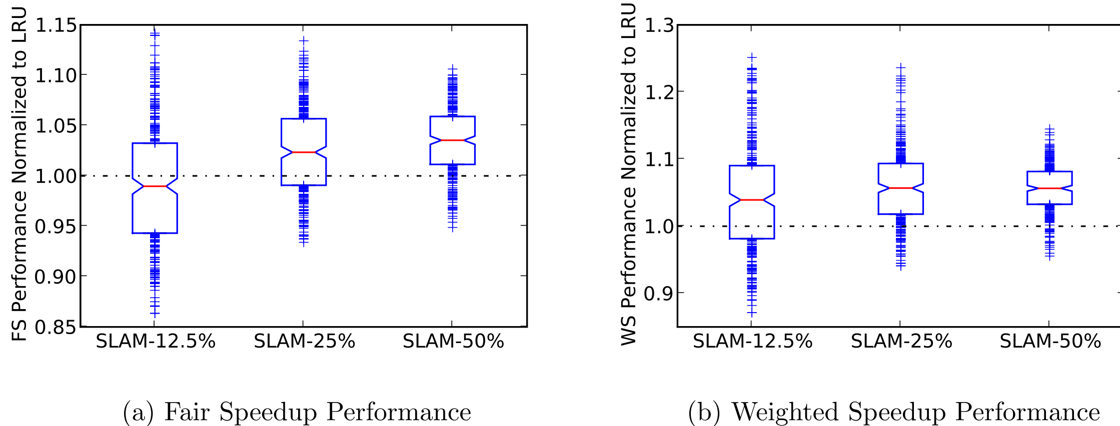
5.1.2 Mitigating Poor Utilization with Throttling

Once a thread is deemed to be using the cache inefficiently, a modified insertion policy is used to rectify the problem. Previous work indicates that accesses can be deprioritized by modifying the insertion policy to insert incoming lines into the LRU position of a cache set [39, 47]. SLAM uses this LRU insertion technique as the throttling mechanism to dial back the cache usage of inefficient threads. Specifically, SLAM uses the BIP policy [47], which inserts blocks into the LRU position with high probability. By doing this, the lifespan of that line is reduced, along with the possibility of displacing a useful line belonging to a more efficient thread.

The decision to throttle is made on a per-access basis. When a miss occurs, while the miss is sent to memory, the MPAC and RIT counters affiliated with the thread making the miss are checked against their threshold values. If the thread is to be throttled, then LRU insertion is achieved merely by not updating the state of the cache set, *i.e.*, not promoting the block to the MRU (most recently used) position.

5.1.3 Honing SLAM

With the theoretical framework of SLAM complete, some experimentation is required pin down specific design details. The exact threshold values generated from the ML decision trees of the previous chapter are not used. Recall that those models were generated based on data from trace-driven simulations; thus while the models are useful for discerning first order effects, it is unlikely that the exact miss rate threshold generated by the model is useful in a more realistic situation. Thus, all threshold parameters are experimentally evaluated here. First, several samples of MPAC thresholds are tested, using all combinations of four workloads chosen from a pool of 11 SPEC CPU2000 workloads (no repeats) yielding 330 combinations.



(a) Fair Speedup Performance

(b) Weighted Speedup Performance

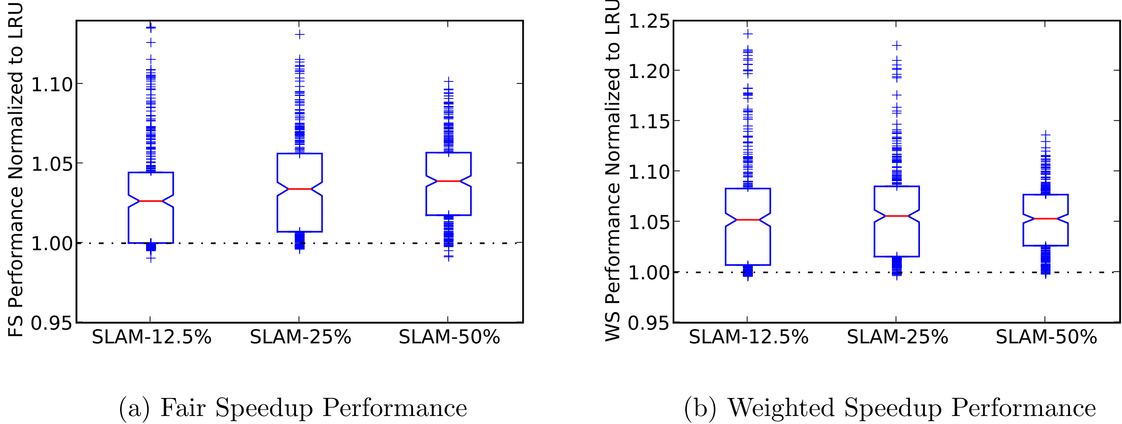
Figure 5.2: Varying MPAC Threshold - Varying the miss rate threshold in SLAM with four applications sharing 1MB of cache. Notice the clear tradeoff between increasing the sensitivity of SLAM with increasing the number of false positives.

The results shown in Figure 5.2 are for both Fair Speedup and Weighted Speedup, two common multiprocessor cache sharing metrics described further in Section 5.2.1. The n% in SLAM-n% indicates the threshold value for deciding an MPAC value is too high. All 330 combinations are plotted in a single column. The top quartile of values is shown by the markers above the box, the bottom quartile is shown by the markers below the box, and the mean of all values is indicated by the red horizontal line notched in the center box, which encapsulates the values of the middle two quartiles. Because of the number of workloads examined, we find this an efficient way to show a large volume of results without reducing values down to merely means or averages.

As expected based on the findings from Chapter 4, SLAM is subject to false positives, as evidenced by the workloads that fall below the LRU baseline of 1.0. To combat false positives, SLAM utilizes the dueling technique [47], which pits two policies against one another to determine the better policy during runtime in a low overhead manner. The results of dueling SLAM against a standard LRU policy are shown in Figure 5.3.

In Figure 5.3, SLAM-12.5% throttles when the MPAC has measured a weighted history of miss rates exceeding 12.5%. There is a distinct trade-off between max-

imizing the detection of poor utilization and the number of false positives. While SLAM-12.5% catches many more poor performers than SLAM-50%, it also catches many that should have been left alone, causing the LRU policy to kick in from dueling and losing opportunities to improve performance. Meanwhile, SLAM-50% is more cautious and has a higher average performance, but misses some opportunities at the top end. A happy medium is SLAM-25%, which provides an average FS improvement of 3.4% over LRU, and a maximum of 13.1%; and an average WS improvement of 5.7% over LRU and a maximum of 22.5%. We use SLAM-25% with dueling for the remainder of the paper unless otherwise specified.



(a) Fair Speedup Performance

(b) Weighted Speedup Performance

Figure 5.3: SLAM with Dueling - Fair Speedup and Weighted Speedup are common CMP shared cache metrics, described further in Section 5.2. For both metrics, 25% appears to maximize the top end of improvement (up to for 13.1% and 22.5% for FS and WS, respectively) without harming average performance (3.1% and 5.7%) over LRU.

The RIT threshold was initially set to be $1/N$ of the cache for N hardware thread for all experiments. This has an intuitive satisfaction, in that a thread would only be throttled if it overstepped its fair share of the cache. However, results show that setting a threshold of $1/2N$ allows for greater performance due to greater flexibility. There is very little room for reshuffling resources with a threshold of $1/N$, which limited the amount of performance gain to be had. Instead, when using a RIT

threshold of $1/2N$, performance gains are improved due to the increased latitude for resource management given to the system. For the remainder of these experiments, the RIT threshold is set at $1/2N$, *e.g.*, 12.5% of insertions to the cache on a four-thread system, unless otherwise specified.

5.1.4 Using SLAM for QoS

As discussed in Chapter 2, quality of service (QoS) has a clear definition in networking, but it is much less clear in the context of shared computing platforms. Nesbit, *et al.* [45] focus on the performance isolation aspect of QoS and defines it as performance on allocated resources at least as good as if those resources comprised a private machine. Iyer, *et al.* [32] propose three potential definitions, described in Section 2.3, as RUM, RPM, and OPM. Even in a single paper, there is not a consistent definition for what QoS means.

What is consistent is that proposals for QoS in the cache generally presume fine-grained control of cache allocation. However, lessons learned from shared cache performance optimization problem space indicate that this is unwieldy, especially since recent proposals for shared cache management avoid this necessity. Additionally, seeking a specific partition to best satisfy some stated goal can be difficult to converge, particularly as platforms scale.

This thesis seeks to push QoS research along the same road as performance optimization, such that prescriptive cache allocations are not the focal point of providing QoS. After all, even Cisco's *Internetworking Technology Handbook* describes QoS primarily as a means of providing prioritization, rather than primarily as a means to guarantee resources [13]. Given the difficulties in tracking and enforcing exact cache resource usage, on top of the existence proof of using differentiation rather than resource guarantees as a cornerstone of networking QoS, providing differentiated QoS seems like the more appropriate form of QoS for cache.

This dissertation presents SLAM, based on the simple yet information-rich MPAC and RIT techniques, as a provider of differentiated cache QoS in terms of a resource *efficiency* metric–decisions are not made based on cache usage, as in RUM, or on performance, as in RPM or OPM. By basing the decision on efficiency, and in concert with the observation that efficiency can be measured in a lightweight manner, SLAM can avoid the algorithmic and measurement pitfalls demonstrated from prior work in shared cache management while still maintaining meaningful guarantees.

By exposing the thresholds for determining poor utility to software, every thread can have individual standards according to priority. Essentially, rather than having fixed lines in the plot in Figure 5.1, there can be a pair of lines (one horizontal, one vertical) for every running thread according to their priority level, thus increasing or decreasing the leniency to which cache efficiency standards are applied. Thus, we can translate the priority differentiation provided by SLAM in a tangible and meaningful way; if a thread is very important, a service provider could guarantee that an application will get to use as much of the cache as it can take in competition with other currently running threads, as long as it falls within some standard of efficiency, *e.g.*, a 75% miss rate and 57% of insertions.

This concrete meaning could be valuable in the context of marketing in commercial virtualization, or even for system administrators. In the results section, we show that SLAM can provide QoS without seeking or enforcing fine-grained cache partitions.

5.2 Methodology

This section discusses the metrics and benchmarks used in this chapter, as well as the simulation environment used to test SLAM.

5.2.1 Metrics

Metric selection can be one of the more important aspects of system design because it has such a heavy influence on the design decisions. Chapter 3 discusses the differences among a number of cache-sharing metrics. Since there are various aspects of performance that can be considered important, there is no single metric that is all-encompassing. Least useful in a multicore context is raw IPC, which masks information about individual performances; thus it is not used in these studies.

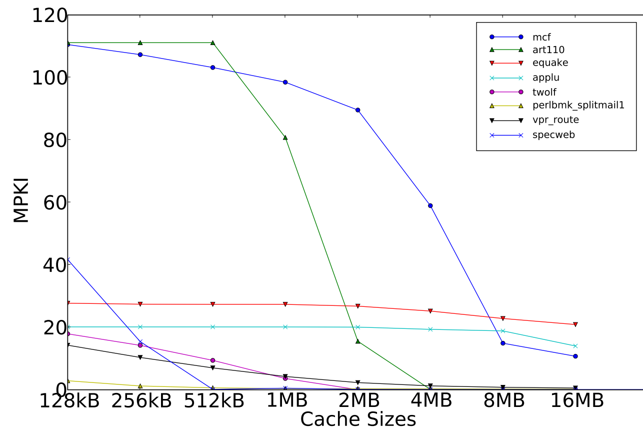
In keeping with earlier results in this thesis, one of the metrics used to judge SLAM is Fair Speedup. Fair Speedup (FS) was proposed by Chang and Sohi [12] as a metric that effectively captures fairness as well as throughput in a single measurement. The equation is given in Equation 4.1, but it is essentially the harmonic mean of weighted speedups over an equal-share private cache baseline. FS is an intuitively pleasing Utopian metric and measures performance relative to a private caching scheme.

At the same time, most of the literature on shared caches includes a form of Utilitarian metric. In order to effectively compare against other schemes, Weighted Speedup (WS) [53], is also used in these studies. WS typically uses a thread running alone on a platform as its baseline, but this makes little sense for large-scale platforms. Instead, this study uses a private equal-share cache as the baseline, essentially `IPC-Private-Util`.

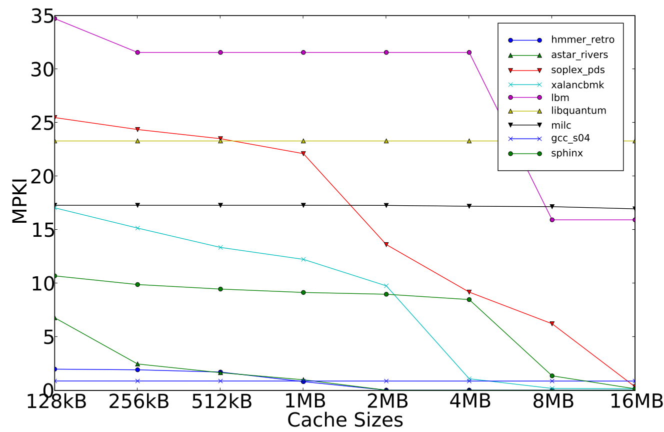
Note that, because FS represses the contribution of high-fliers, the range of FS performance improvements is generally smaller than that of WS, which includes large individual improvements over equal share "at value."

5.2.2 Benchmarks

This study begins with a detailed analysis on a small set of interesting workloads that include mixes of SPEC®CPU2000, SPEC®CPU2006, and SPECweb®99 benchmarks. The misses per 1000 instructions (MPKI) profiles of these benchmarks are



(a) SPEC CPU2000



(b) SPEC CPU2006 + SPEC Web

Figure 5.4: MPKI Graph - Misses per thousand instructions (MPKI) for hybrid workloads.

shown in Figure 5.4. To keep these detailed studies tractable, the following set of experiments focus on 16 different combinations of four workloads. The combinations used are shown in Table 5.1. The numerals at the end of application names indicate whether they come from CPU2000 or CPU2006, and preceding letters, if any, indicate the reference input set used if disambiguation is necessary. For example, *hmmer.r.06* indicates the hmmer benchmark from CPU2006 using the *retro* reference input set.

In addition to these detailed studies, this study also includes a set of experiments

Mix	App1	App2	App3	App4
Mix0	art110.00	applu.00	hmmer.r.06	xalancbmk.06
Mix1	mcf.00	art110.00	xalancbmk.06	specweb
Mix2	perlbmk.s.00	twolf.00	libquantum.06	specweb
Mix3	mcf.00	art110.00	soplex.p.06	specweb
Mix4	mcf.00	soplex.p.06	xalancbmk.06	specweb
Mix5	applu.00	twolf.00	hmmer.r.06	specweb
Mix6	art110.00	equake.00	soplex.p.06	specweb
Mix7	mcf.00	hmmer.r.06	soplex.p.06	xalancbmk.06
Mix8	mcf.00	hmmer.r.06	astar.r.06	xalancbmk.06
Mix9	applu.00	equake.00	xalancbmk.06	soplex.p.06
Mix10	twolf.00	equake.00	astar.r.06	specweb
Mix11	perlbmk.s.00	vpr.r.00	xalancbmk.06	astar.r.06
Mix12	equake.00	twolf.00	astar.r.06	lbm.06
Mix13	art110.00	xalancbmk.06	lbm.06	specweb
Mix14	xalancbmk.06	mcf.06	milc.06	gcc.s04.06
Mix15	sphinx.06	mcf.06	hmmer.r.06	soplex.r.06

Table 5.1: Workload Mixes - Workload Mixes for Detailed Studies

using a comprehensive set of workloads, comprising all combinations of N applications from a set of 11, ten being from SPEC CPU2000, and one from SPECweb. The miss profiles for these benchmarks are shown in Figure 5.5. For the baseline of four threads, this means 330 workload combinations. Since this thesis aims for scalability, there is also scalability study for which up to 16-application combinations are run simultaneously. Because more simultaneous applications make simulation time infeasibly long, a random number generator is used select a tractable subset from all combinations of 16 workloads from a set of 19 SPEC CPU2000 applications plus SPECweb, resulting in 28 16-application workloads. In these scalability studies, doubling the number of threads doubled the size of the cache, such that C/N is held constant.

For the comprehensive studies, computational limitations preclude testing exhaustive combinations of all SPEC CPU2006 benchmarks. Given that CPU2006 workloads tend to have larger cache and memory footprints, I had fewer machines capable of running such workloads, and running a comprehensive of tests using CPU2006 applications would have taken more computational time than desired.

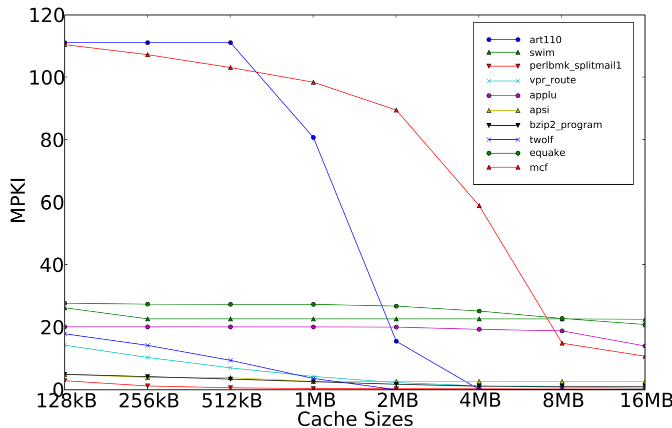


Figure 5.5: MPKI for comprehensive workloads. Note that due to the extremely high miss rates of *mcf* and *art110*, the variation of the remaining workloads appears muted from the scale of the graph.

5.2.3 Simulation Environment

This study uses the M5 Simulator [7] under the Alpha ISA to perform simulations. As opposed to the trace-based simulation platform used in earlier portions of this thesis, this environment is more realistic, incorporating timing in the memory system. The following experiments use an in-order 1 CPI, single-threaded CPU with a timing memory system with two levels of cache. Each core has a private IL1 and DL1 cache, each 16kB in size and 4-way set associative, with a 2-cycle access latency. The L2 cache is shared among all cores and is 16-way set associative with a 20-cycle access latency. Block size for all caches is 64 bytes. The memory latency simulated is 350 cycles. Writebacks do not update LRU state.

Simulations were run until all benchmarks in a workload reached one billion instructions, with the exception of *mcf* and *art110*, which were run to 500 million instructions in order to conserve computation time. Statistics for each benchmark were updated until it reached its instruction limit, but the benchmark continued to run in order to contend for the cache.

All CPU2000 benchmarks were run from their respective SimPoints [46], while all

CPU2006 workloads were run after fast-forwarding one billion instructions due to the lack of published Alpha SimPoints for SPEC CPU2006.

5.3 Results

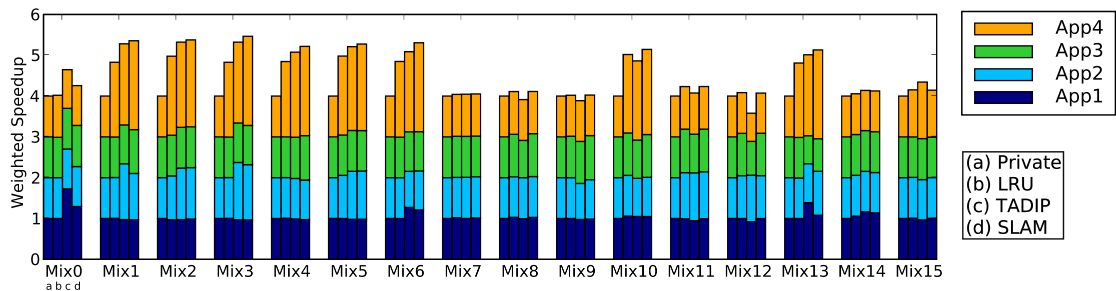
This section details the results along three major axes of study: detailed studies, for which the workloads in Table 5.1 are studied in depth; comprehensive studies, for with all combinations of SPEC CPU2000 are used; and QoS studies, which examine the suitability of SLAM for providing differentiated QoS.

5.3.1 Detailed Studies

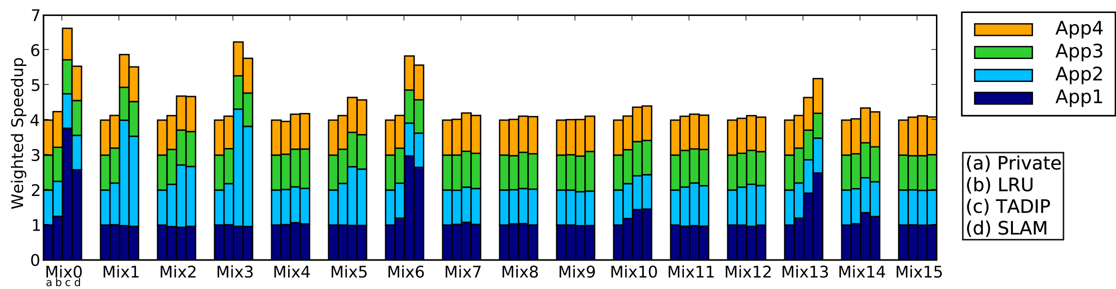
Figure 5.6 shows the 16 detailed study workloads described in Table 5.1 on the x-axis plotted against their relative performance against a private cache configuration on the y-axis. Thus, the sum of the bars is the total WS of its affiliated policy, while the harmonic mean of all the bar components is the FS. Consequently the left-most bars of each cluster, representing the private configuration, sum up to 4.0. This experiment compares a private cache configuration against LRU, TADIP, and SLAM.

What the graph shows is a small but clear advantage to using SLAM when the resources are most limited, at the 1MB cache size (Figure 5.6a). At other cache sizes, performance more or less matches that of TADIP, demonstrating that SLAM is robust and competitive at a variety of cache sizes.

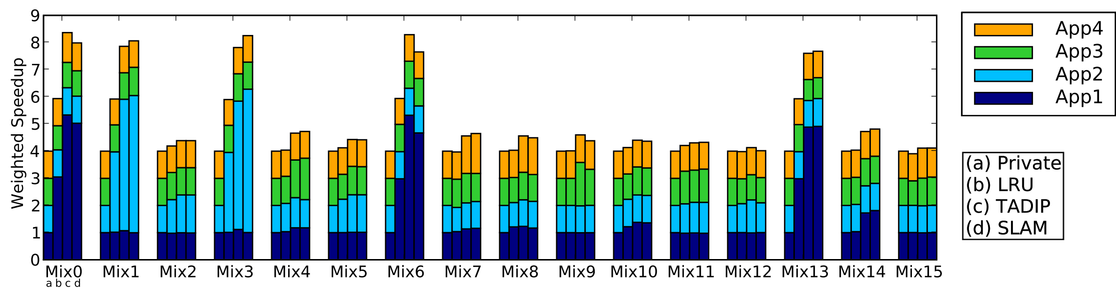
An interesting thing to note is that for the same workloads across different cache size, the gains to be had do not come from the same benchmarks. For example, looking at Mix2, under 1MB the SLAM is able to give more space to App4, yielding an improvement in performance. However, for 2 and 4MB, given the different region of operation within the application miss rate profiles, SLAM is able to appropriately give more cache space to App2. Finally, under 8MB, SLAM provides useful cache



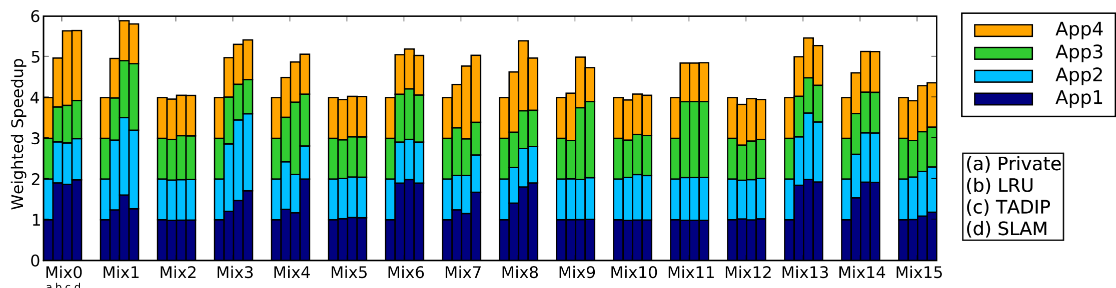
(a) Sharing 1MB



(b) Sharing 2MB



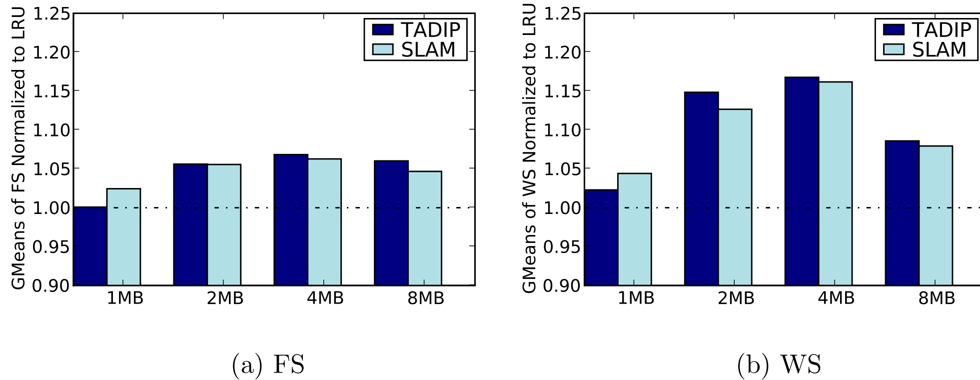
(c) Sharing 4MB



(d) Sharing 8MB

Figure 5.6: Detailed Results - Stacked WS graphs for four applications sharing various cache sizes. Notice that the increases in performance across sizes do not always come from the same applications, affirming SLAM's ability to detect runtime behavior and is not dependent upon any prior knowledge of an application.

space to App2 and App3. This agility of operation demonstrates the robust nature of SLAM in terms of assessing the real-time behaviors of applications that are running.



(a) FS

(b) WS

Figure 5.7: Geometric Mean Results - Geometric means of FS and WS results across all 16 workloads for all cache sizes studied.

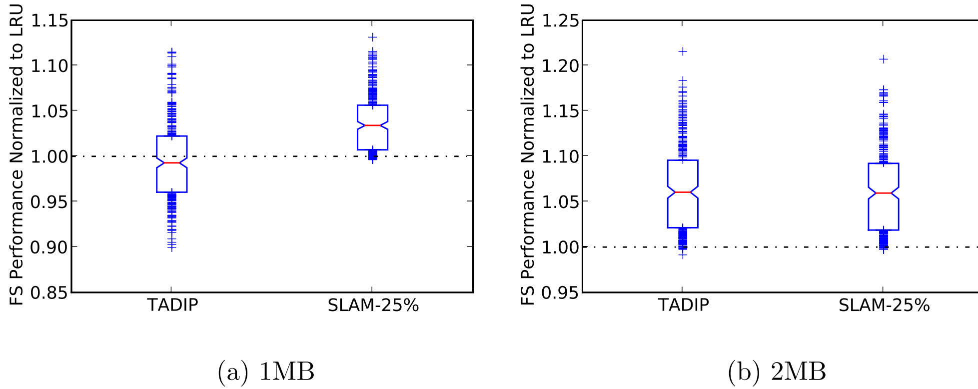
Figure 5.7 shows the geometric means of both FS and WS across the cache sizes examined above. It is quite interesting to note that despite the completely different approaches taken by TADIP and SLAM for identifying threads that should be throttled, on the whole performance is comparable.

5.3.2 Comprehensive Studies

In addition to the detailed studies described in Section 5.3.1, this set of experiments is meant to ensure that this approach is reasonable as a comprehensive solution. The only way to do this is to perform more comprehensive testing. These experiments reflect the performance of 330 workloads for 4 threads, 165 for 8 threads, and 28 for 16 threads. The methodology behind benchmark selection is detailed in Section 5.2.2.

Varying Cache Size

Figure 5.8 shows our results for SLAM with four threads sharing 1MB and 2MB of cache. As in the detailed studies above, SLAM outperforms TADIP when the cache size being shared is 1MB. We believe this is as a result of the set pinning mechanism



(a) 1MB

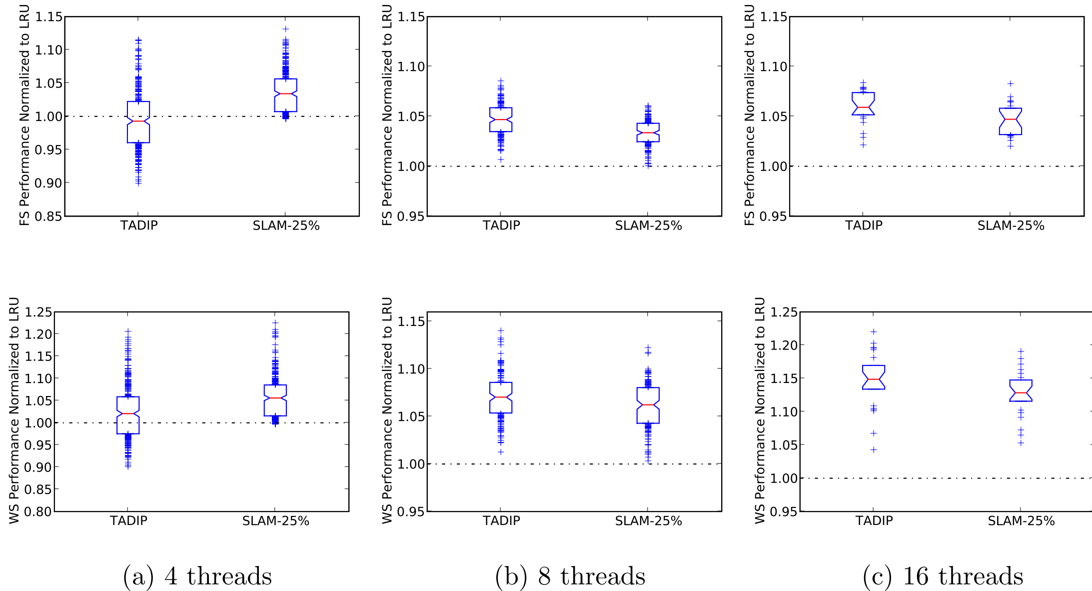
(b) 2MB

Figure 5.8: Varying Cache Size - FS performance for four applications sharing 1MB and 2MB. At 1MB, SLAM has both a better max improvement and average performance (13.1% and 3.4% vs. 11.4% and -0.7%, respectively), while at 2MB, the two behave comparably (21.5% max and 6.1% average for TADIP, and 20.7% and 6.0% average for SLAM).

in TADIP which requires 64 sets of the cache to be pinned to one policy per thread – 32 to standard LRU and 32 to the BIP policy. That means that at any given moment, $32 \times N$ sets of the cache are performing the "wrong" policy for their particular thread, which can be an burdensome overhead when the cache size being shared is not overly generous. Thus, if these techniques were ever to be applied to shared mid-level caches in a three-level hierarchy, SLAM is likely the better candidate.

Scalability

To ensure scalability, SLAM is tested under all 165 combinations of eight workloads sharing 2MB, as well as a small randomized subset of 16-thread workloads, taken from a larger pool of SPEC CPU2000 applications, sharing 4MB. The results are shown in Figure 5.9. SLAM is scalable not just in terms of hardware overhead, but in its ability to improve performance as well. TADIP and SLAM continue to be comparable, though the relative performance of TADIP again improves slightly with larger caches. It is possible that since the thresholds for MPAC and RIT were tuned for four threads sharing 1MB (see Section 5.1.3), SLAM would be more competitive



(a) 4 threads (b) 8 threads (c) 16 threads

Figure 5.9: Scalability Results - The results for FS are in the first row, WS in the second row. Again, despite the highly different approaches, the two schemes perform comparably well. For FS, SLAM outperforms TADIP at four threads, and is within 1.5% of TADIP in mean performance at 8 threads, and is within 1% at 16 threads.

using different thresholds.

5.3.3 QoS

Using the MPAC and RIT monitors presented in this dissertation, SLAM can easily be translated into a useful provider of differentiated QoS. Merely by adding the ability for software or a higher level QoS policy to set MPAC and RIT thresholds, SLAM can go from being a performance optimization to a framework for providing differentiated QoS based on prioritized efficiency thresholds.

One potential way to do this in a commercialized environment is to provide set service level guarantees. Table 5.2 lists a possible structure for providing differing levels of QoS. In this setup, higher service levels are guaranteed to never be throttled unless they are using over their equal share of insertions, while lower service levels are held to basic SLAM standards of $1/2N$. The primary point of variation here is the change in miss rate standards, ranging from a 10% threshold at the lowest

service level, and a 100% miss rate at the highest (essentially guaranteeing never to be throttled).

High/Low	Level	MPAC Threshold	RIT Threshold
Low	0	10%	$1/2N$
	1	15%	$1/2N$
	2	25%	$1/2N$
High	0	50%	$1/N$
	1	75%	$1/N$
	2	100%	$1/N$

Table 5.2: Service Level Thresholds - Potential structure for providing different QoS service levels in a commercial shared platform.

Figure 5.10 shows a simple proof of concept; a strawman case of four copies of SPECweb running on a shared 1MB cache, where one of the copies is considered high priority and the rest are low priority. All nine $\{High\} \times \{Low\}$ possibilities are shown. The blocks comprising each stack are the performance of each copy of the benchmark relative to standard SLAM for performance optimization. The high priority copy is designated by the x-axis label. The first immediately apparent phenomenon is that when the low priority copies are running L0, very high performance is accorded to the high priority thread, to the point that total performance exceeds basic SLAM (as evidenced by the stack height being above 4.0). It is clear here that reducing the priority of the low priority threads gives a big advantage to the high priority thread. This phenomenon is slightly reduced when the low priority copies are at service level L1, with the low priority threads improving slightly in performance, and the high priority threads reducing somewhat. When low priority is L2, however, performance is almost indistinguishable from basic SLAM, in large part because of the minor difference between L2 and H0.

When examining the changes that result from varying H0 to H2, however, there are hardly any changes apparent. This is because of the SPECweb benchmark—miss rates do not go above 50% unless the cache size allotted is extremely small, so for

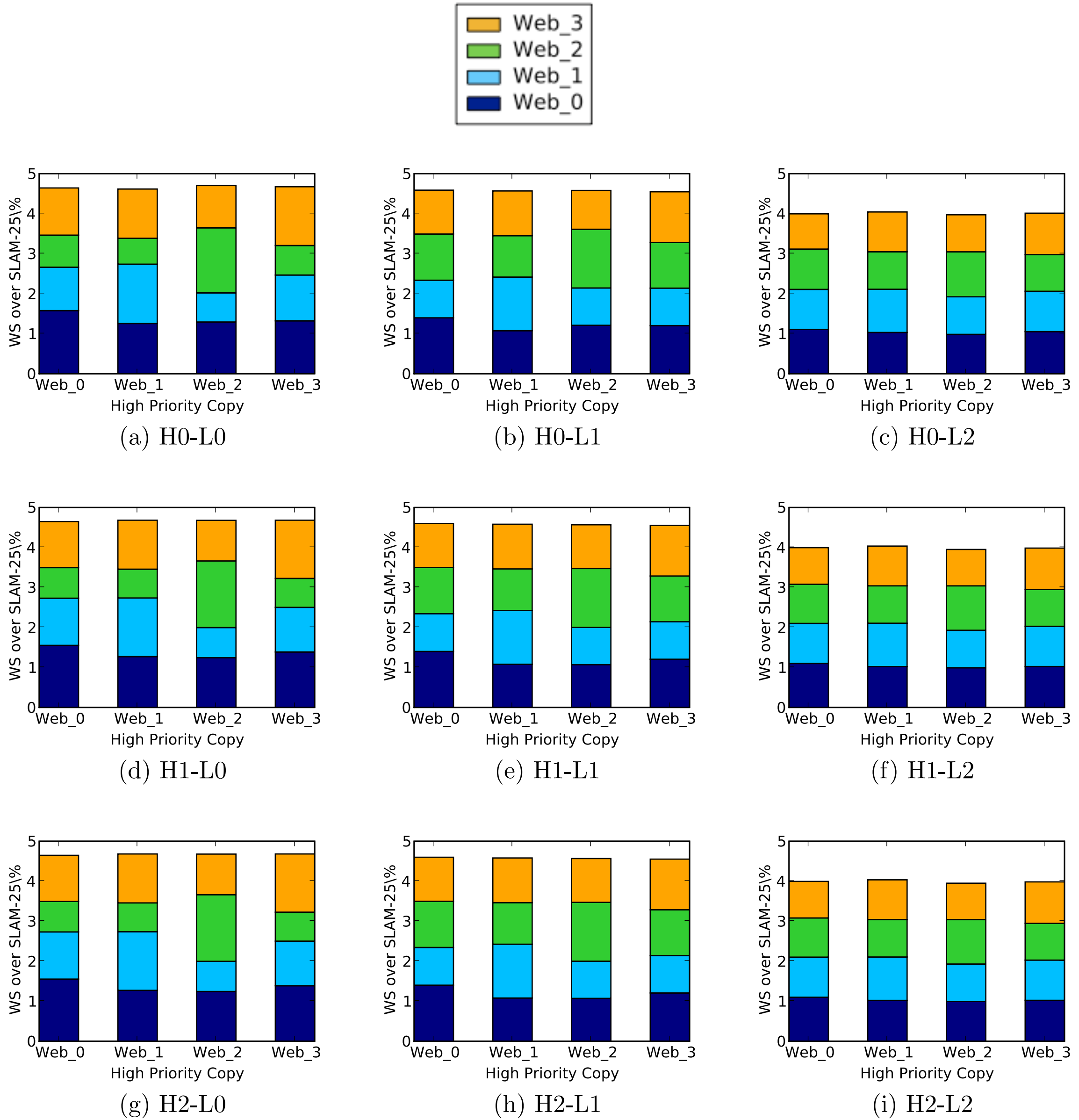


Figure 5.10: Four copies of SPECweb sharing a 1MB cache. Each stack represents a different copy as having high priority, as designated by the x-axis label. All nine combinations of High and Low service levels given by Table 5.2 are tested and shown here.

this case, all the high priority service levels are functionally equivalent. However, if the workload examined were different, such that miss rates tended to be very high, these high service levels would be differentiated, much the way there is differentiation in the low priority levels.

Note that all of the stacks in a single graph are not merely shifted copies of each

other because the copies are offset from each other by one million instructions so that all of the benchmarks are not running in lockstep.

In short, this figure demonstrates that SLAM is capable of providing differentiated service that can potentially lead directly to improved performance, in particular by reducing the priority of lower priority threads.

A more realistic demonstration of SLAM for QoS is shown in Figure 5.11. Pictured are four workload instances from the detailed studies of Section 5.3.1 where an inefficient thread was sacrificed in favor of another for the sake of higher overall performance. Here, those threads are labeled high priority with varying definitions of high priority.

The low priority threads are held at standard SLAM thresholds, *i.e.*, L2, while high priority threads range through H0 to H2. Performance for all schemes are shown relative to LRU, with the high priority thread on the bottom.

Across all cases, SLAM provides a non-trivial performance benefit over LRU, trading the performance of a low-efficiency thread for a higher one. When the high priority thread is given small amount of preferential treatment, Cases 1, 2, and 3, get all if not more of the sacrificed performance back. Case 0 needs even more preferential treatment to get its performance back, but at the cost of all the gains obtained by SLAM. The application in this case is *lbm* from Mix 13 sharing 2MB of cache, which has a miss rate of 60% and an insertion rate of 33% in the LRU case over the entire simulation. This is quite inefficient. So, depending on how important *lbm* really is, or what service levels its owner has paid for, a user or QoS policy can decide what thresholds to set. What is interesting here is that, unlike in the strawman case, increasing priority from H0 to H2 does in fact produce an effective improvement in performance for the high priority thread; in this case, it is because the high priority threads tend to have high miss rates and having MPAC thresholds varying above 50% produces noticeable variations in behavior. Conversely, varying the low priority

threads between L0 and L2 has little effect; plots for L0 and L1 are virtual replicas of Figure 5.11 and are not shown.

The upshot of these results is that a high priority thread could potentially achieve higher performance either by running alongside very low priority threads, or by having a high service level. Either way, SLAM can leverage MPAC and RIT to provide a toolbox for providing differentiated quality of service. While this toolbox cannot guarantee certain amounts of performance, it can guarantee certain degrees of preferential treatment.

It is worth noting that giving the sacrificed thread high priority can sometimes result in higher overall performance than with standard SLAM. Recall that SLAM is not a complex enforcement of some calculation of optimal — rather, it is just a series of local decisions which leverage the observation that threads which are inefficiently using the cache are probably hurting the ability of other threads with better utilization properties to use the cache. Given the simplicity of the SLAM scheme, it is not surprising that it does not provide optimal performance in every instance, but here, even when evidence of that is exposed, the differences are minimal.

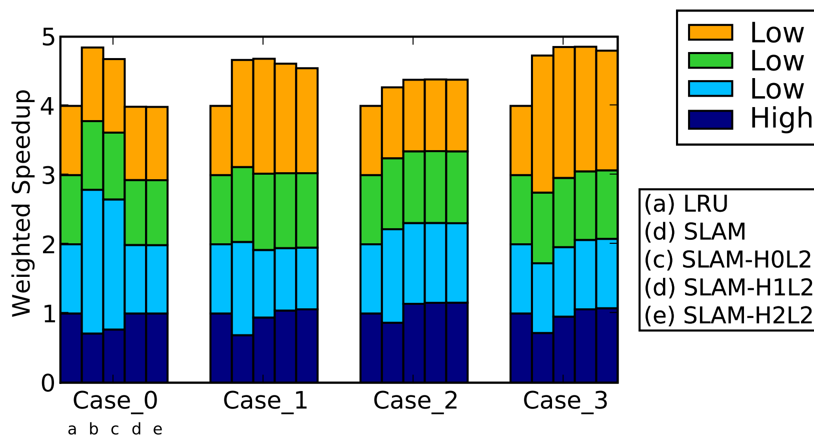


Figure 5.11: Threads whose performance was sacrificed in SLAM in favor of other threads are treated as high priority threads, under increasingly lenient standards, where SLAM-H2L2 indicates that the high-priority thread is *never* throttled. Increasing priority can bring the performance of the sacrificed thread up to LRU levels, with various amounts of sacrifice for overall performance.

In general, these studies show that leveraging MPAC and RIT in the SLAM framework can translate to flexible and meaningful QoS prioritization. SLAM can potentially be deployed in commercial virtualization environments, with different levels of service corresponding to different prioritization levels that can be easily explained to customers.

5.4 Conclusions

This chapter outlines a new direction in shared cache management, based on reactive throttling of inefficient threads rather than proactive resource allocation, whereby the criteria for detecting inefficiencies stem from empirical analysis with machine learning discussed in Chapter 4. I present an embodiment of this approach, Scalable Lightweight Adaptive Monitoring (SLAM), which makes thread-local throttling decisions based on very lightweight per-thread performance monitors. These two novel monitoring components – the Misses Per Access Counter (MPAC) and the Relative Insertion Tracker (RIT) – are simple and scalable to hundreds of thread contexts, and do not require modifications to a conventional cache structure.

Comprehensive simulations of hundreds of workloads across multiple cache sizes show that SLAM provides global performance improvements competitive with the best published approach. Unlike previous approaches, SLAM's intuitive approach of throttling inefficient threads also enables differentiated quality of service using the same hardware structures. A thread's cache priority can be modified simply by adjusting the threshold values used to classify it as inefficient, such that higher (lower) priority threads will be less (more) likely to be throttled.

In short, SLAM is a simple, flexible, low hardware cost, low design cost, scalable solution for not only providing performance optimization in a shared cache, but also for providing differentiated QoS by allowing differing standards for cache efficiency.

SLAM addresses the need for a scalable, feasible cache management scheme that can also meet the emerging needs for providing QoS in large-scale server platforms.

CHAPTER 6

Conclusions and Future Work

This dissertation focuses on shared cache management in the context of large-scale server class machines. The results is a prescription for how to approach the design of a high-performance shared cache for large-scale chip multiprocessors which deviates from much of the established literature. Cache designers should focus on reacting to and alleviating poor cache allocations, rather than explicitly searching for an optimal allocation during run time. The reasons for this are two-fold. First, studies in cache metrics from Chapter 3 reveal that there is no one perfect metric that encompasses all positive attributes in shared cache performance. Indeed, sometimes optimizing for one means poor performance in another. While some metrics, characterized by this thesis as "Utopian" in nature, balance fairness and throughput by measuring the harmonic mean of the performance of all sharers, optimizing for Utopianism can sometimes mask losses in aggregate performance up to 40%, or losses in fairness with differences in standard deviation up to 0.25. Since there is no one metric which is clearly superior, seeking an "optimal" partition during run-time seems futile. Second, using this reactive approach has much less inherent complexity than an algorithm that seeks optimal partitions at run-time.

In keeping with this philosophy, this dissertation uses established machine learning techniques to extract salient runtime characteristics which indicate that a shared

cache is suffering from poor allocation and requires attention. A fair amount of effort was required to produce a methodology that could provide useful results. This thesis discusses some of the lessons and pitfalls in using machine learning for research. For the purposes of this work, decision tree analysis is superior to ridge regression analysis.

The result of this machine learning study provides direction for how to build a simple, scalable, feasible cache framework for identifying poor cache sharing situations and how to mitigate these situations. This dissertation presents Scalable, Lightweight, Adaptive Management (SLAM) as a framework for managing shared cache. SLAM consists primarily of two performance measurement facilities that are a novel contribution of this work. The Misses Per Access Counter (MPAC) is a weighted history measure of per-hardware thread misses per access in the cache. The Relative Insertion Tracker (RIT), measures the relative rates of insertion into the cache from each contributing hardware thread. Together, identifying when a thread is using the cache inefficiently is a simple matter knowing when a thread's MPAC value exceeds a certain threshold (experimentally determined to be a value representing 25%), and when that same thread's RIT value exceeds a certain threshold (experimentally determined to be $1/2N$ on a system with N hardware threads). When a thread is detected to be inefficient, its cache capacity is throttled by placing its insertions to the cache in the least recently used position of a cache set with high probability, rather than the typical most recently used position, which has been demonstrated previously as an effective way to reduce cache occupancy.

The machine learning study presaged the number of false positives produced by this policy. These false positives can be mitigated by pitting the SLAM policy against the standard LRU replacement policy via dueling, a technique used to select the better of two policies during run-time. The resulting SLAM framework can provide significant performance gains over LRU performance. With respect to the Fair Speedup metric in comprehensive four-thread studies involving 330 distinct workloads, SLAM

provides an average improvement of 3.4% over LRU, and a maximum of 13.1%. Performance gains are on par with the best scalable solution in the literature.

The SLAM framework does not merely provide effective performance optimization. This dissertation makes the case that, as in the networking world, the concept of quality of service (QoS) in a shared resource should primarily focus on differentiation and preferential treatment, rather than explicit guarantees about resource usage or performance. As such, the SLAM framework is a natural provider for this sort of differentiated QoS. Threads can easily be prioritized based on more lenient efficiency standards, or by placing stricter efficiency standards on other threads. This can be done without adding new hardware on top of basic SLAM. Studies in Chapter 5 show that by altering the efficiency thresholds for individual hardware threads, SLAM can provide improvements in performance for thread designated as high priority.

Although SLAM is an effective technique on its own, the mechanisms and approach taken in this work can be fruitfully extended in several ways. First, SLAM itself can likely be improved by better tuning of thresholds and slightly more flexible throttling conditions. Additionally, revisiting the third-order effect as extracted by the ML studies in Chapter 4 could lead to even better detection of poor cache performance.

Also, the information provided by the MPAC and RIT, both individually and together, can very likely be profitably employed to improve performance in other parts of the system. For example, prioritizing cache miss requests belonging to the thread with the lowest RIT value would be a very low-cost way of improving fairness in cache and memory bandwidth allocation. Bitirgen *et al.* [8] have studied coordinated control of the memory subsytem using learned behavior; providing the memory controller with information of known value can reduce the complexity of the learning algorithm, or even obviate the use of one.

Exposing MPAC and RIT values upward to the operating system could be used to influence scheduling decisions within a single platform or thread migration decisions

across different nodes in a cloud computing environment. While the information provided by these mechanisms is similar to that obtainable via performance counters, MPAC and RIT automatically provide weighted historical averages of key event ratios for each thread, values that would require significant software overhead to maintain based on raw event counters.

Finally, SLAM could be useful in the context of a highly software threaded machine, where the number of software threads far exceed the number of hardware threads. Saving the limited state of SLAM upon a context switch could make for improved performance when a thread is switched back to a running state.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. *ACM-SE 42: Proceedings of the 42nd Annual Southeast Regional Conference*, pages 267–272, 2004.

[2] AMD. Key architectural features - AMD Phenom II processors. http://www.amd.com/us/products/desktop/processors/phenom-ii/Pages/phenom-ii-key-architectural-features.aspx.

[3] AMD. Quad-core AMD Opteron processor: Product brief. *AMD Website*. http://www.amd.com/us/products/server/processors/opteron/Pages/3rd-gen-server-product-brief.aspx.

[4] A. Aron, E. Aron, and E. J. Coups. *Statistics for Psychology.* Pearson Prentice Hall, 5th edition, 2009.

[5] B. Beckmann, M. Marty, and D. Wood. ASR: Adaptive selective replication for CMP caches. *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2006.

[6] L. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, Dec 1966.

[7] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52 – 60, Jul 2006.

[8] R. Bitirgen, E. Ipek, and J. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 318 – 329, Oct 2008.

[9] S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, and J. Rattner. Platform 2015: Intel processor and platform evolution for the next decade. *Technology@Intel Magazine*, 2005.

[10] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[11] J. Chang and G. Sohi. Cooperative caching for chip multiprocessors. *ISCA '06: Proceedings of the 33rd Annual International Symposium on Computer Architecture*, Jun 2006.

[12] J. Chang and G. Sohi. Cooperative cache partitioning for chip multiprocessors. *ICS '07: Proceedings of the 21st Annual International Conference on Supercomputing*, Jun 2007.

[13] Cisco Systems. *Internetworking Technologies Handbook*.

[14] P. Denning. The working set model for program behavior. *Communications of the ACM*, Dec 1968.

[15] P. Denning. The locality principle. *Communications of the ACM*, Dec 2005.

[16] H. Dybdahl and P. Stenstrom. An adaptive shared/private NUCA cache partitioning scheme for chip multiprocessors. *Proceedings of the 13th International Symposium on High Perfomance Computer Architecture*, Dec 2007.

[17] H. Dybdahl, P. Stenström, and L. Natvig. A cache-partition aware replacement policy for chip multiprocessors. In *In Proceedings of 13th International Conference of High Performance Computing (HiPC)*, 2006.

[18] H. Dybdahl, P. Stenström, and L. Natvig. An LRU-based replacement algorithm augmented with frequency of access in shared chip-multiprocessor caches. *MEDEA '06: Proceedings of the 2006 Workshop on Memory Performance: Dealing with Applications, Systems, and Architectures*, Sep 2006.

[19] R. Eickemeyer, M. Lipasti, T. Mullins, B. O'Krafka, H. Rosenberg, S. VanderWiel, P. Vitale, L. Whitley, and S. Kunkel. A performance methodology for commercial servers. *IBM Journal of Research and Development*, 44(6):22, Feb 2000.

[20] A. Fedorova and M. Seltzer. Improving performance isolation on chip multiprocessors via an operating system scheduler. *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 25 – 38, Aug 2007.

[21] B. Fox. Discrete optimization via marginal analysis. *Management Science*, 13(3):210–216, Dec 1966.

[22] F. Guo, H. Kannan, L. Zhao, R. Illikkal, R. Iyer, D. Newell, Y. Solihin, and C. Kozyrakis. From chaos to QoS: case studies in CMP resource management. *SIGARCH Computer Architecture News*, 35(1), Mar 2007.

[23] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2007.

[24] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, chapter 1. Springer, 2001.

[25] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, chapter 3. Springer, 2001.

[26] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, chapter 7. Springer, 2001.

[27] Intel. Intel Core i7 processor extreme edition: Product brief. *Intel Website.* http://download.intel.com/products/processor/corei7EE/extreme$_prod_brief.pdf$.

[28] Intel. Leading virtualization performance and energy efficiency in a multi-processor server: Xeon 7400 product brief. *Intel Website.* http://download.intel.com/products/processor/xeon/7400$_prodbrief.pdf$.

[29] E. İpek, S. McKee, R. Caruana, B. Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov 2006.

[30] R. Iyer. On modeling and analyzing cache hierarchies using CASPER. *MASCOTS 2003: 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems*, pages 182 – 187, Sep 2003.

[31] R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. *ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing*, Jun 2004.

[32] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Jun 2007.

[33] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, and J. Emer. Adaptive insertion policies for managing shared caches. *PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Oct 2008.

[34] P. Joseph, K. Vaswani, and M. Thazhuthaveetil. A predictive performance model for superscalar processors. *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2006.

[35] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, Sep 2004.

[36] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21 – 29, Mar 2005.

[37] J. D. Kron, B. Prumo, and G. H. Loh. Double-DIP: Augmenting DIP with adaptive promotion policies to manage shared L2 caches. *CMP-MSI: 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, page 9, May 2008.

[38] B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov 2006.

[39] W.-F. Lin, S. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. *HPCA '01: The 7th International Symposium on High-Performance Computer Architecture*, pages 301 – 312, Dec 2000.

[40] J. Liptay. Structural aspects of the System/360 Model 85, PartII: the cache. *Readings in Computer Architecture*, pages 373–379, Dec 2000.

[41] K. Luo, J. Gummaraju, and M. Franklin. Balancing thoughput and fairness in SMT processors. In *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*, pages 164–171, 2001.

[42] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. YALE: rapid prototyping for complex data mining tasks. *KDD '06: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug 2006.

[43] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[44] T. Mudge. Power: a first-class architectural design constraint. *Computer*, 34(4):52 – 58, Apr 2001.

[45] K. Nesbit, J. Laudon, and J. Smith. Virtual private caches. *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, Jun 2007.

[46] E. Perelman, G. Hamerly, M. Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for accurate and efficient simulation. *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Jun 2003.

[47] M. Qureshi, A. Jaleel, Y. Patt, S. Steely, and J. Emer. Adaptive insertion policies for high performance caching. *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, Dec 2007.

[48] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2006.

[49] M. K. Qureshi. Adaptive spill-receive for robust high-performance caching in cmps. *HPCA '09: Proceedings of the 15th Annual IEEE International Symposium on High Performance Computer Architecture*, pages 45 – 54, Jan 2009.

[50] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. *PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, Sep 2006.

[51] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *ASPLOS '02: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Dec 2002.

[52] A. Smith. Cache memories. *ACM Computing Surveys (CSUR)*, Dec 1982.

[53] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *ASPLOS '00: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Dec 2000.

[54] S. Srikantaiah, M. Kandemir, and M. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. *ASPLOS '08: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar 2008.

[55] H. Stone, J. Turek, J. Wolf, and I. Center. Optimal partitioning of cache memory. *Computers*, Dec 1992.

[56] F. Su, X. Shi, G. Liu, Y. Xia, and J.-K. Peir. Comparative evaluation of multicore cache occupancy strategies. *ICPADS '07: Proceedings of the 13th International Conference on Parallel and Distributed Systems*, 1:1 – 8, Nov 2007.

[57] G. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, Dec 2004.

[58] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic cache partitioning for simultaneous multithreading systems. *13th IASTED International Conference on Parallel and Distributed Computing Systems*, 2001.

[59] Sun Microsystems. UltraSPARC T2 processor brochure. *Available from Sun Website*, page 2, Aug 2007. http://www.sun.com/processors/UltraSPARC-T2/brochure.pdf.

[60] D. Thiebaut, H. Stone, and J. Wolf. Improving disk cache hit-ratios through cache partitioning. *Computers, IEEE Transactions on*, 41(6):665 – 676, Jun 1992.

[61] C. Waldspurger. Memory resource management in VMware ESX server. *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Dec 2002.

[62] S. Weisberg. *Applied Linear Regression*. John Wiley and Sons, Inc., third edition, 2005.

[63] D. Wood, M. Hill, and R. Kessler. A model for estimating trace-sample miss ratios. *SIGMETRICS '91: Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Apr 1991.

[64] Y. Xie and G. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, Jun 2009.

[65] T. Y. Yeh and G. Reinman. Fast and fair: Data-stream quality of service. *CASES '05: Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 237–248, 2005.

[66] L. Zhao, R. Iyer, M. Upton, and D. Newell. Towards hybrid last level caches for chip-multiprocessors. *SIGARCH Computer Architecture News*, 36(2), May 2008.