The Dissertation Committee for Moinuddin Khalil Ahmed Qureshi
certifies that this is the approved version of the following dissertation:

Adaptive Caching for High-Performance Memory Systems

Committee:

Yale N. Patt, Supervisor

Jacob Abraham

Derek Chiou

Philip Emma

Sanjay Patel

Emmett Witchel

Adaptive Caching for High-Performance Memory Systems

by

Moinuddin Khalil Ahmed Qureshi, B.E.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2007

Dedicated to Abba and Ammi.

Acknowledgments

This thesis is dedicated to my loving parents, Khalil Ahmed Qureshi and Hasina Qureshi, both of whom passed away during the course of my studies. Words are incapable of describing my feeling of gratitude for them. The value my father placed on education, even though he had limited access to it, is the single biggest reason in my completing this PhD. My mother was a constant source of love and encouragement. The strength and courage I derive from their memories has kept me going forward.

I am grateful for the patience, love, and unconditional support of my siblings: Shaheen, Mona, Amina, and Alauddin. I am especially grateful to my sister Mona for always being there for me when I needed support. Her unwavering faith in me and her taking care of things at home at the most delicate times allowed me to do my studies in the US.

I am indebted to my adviser, Yale Patt, for his influence on my life with both his teaching and guidance. His EE360N is responsible for much of what I know in computer architecture. His EE382N motivated me to pursue a PhD. Yale provided the right balance of freedom and guidance needed for my development as a researcher and as an individual. His teachings and principles will continue to influence my life for a long time.

My life in graduate school would have been barren had it not been for two members of Monga family: Vishal Monga and Archna Monga. More than friends they became my family away from home. Vishal was my roommate for the first four years. He helped me focus during the most difficult times and made my dark days seem brighter. I was deeply touched by his friendship, patience, and maturity. During the last year, I had the good fortune of knowing Archna. Archna taught me that it is possible to balance work and life. I am grateful for her friendship, understanding, discussions, and yummy parathas. I also thank other elderly members of the Monga family for their affection and blessing.

Adaptive Caching for High-Performance Memory Systems

Moinuddin Khalil Ahmed Qureshi, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Yale N. Patt

One of the major limiters to computer system performance has been the access to main memory, which is typically two orders of magnitude slower than the processor. To bridge this gap, modern processors already devote more than half of the on-chip transistors to the last-level cache. However, traditional cache designs – developed for small first-level caches – are inefficient for large caches. Therefore, cache misses are common which results in frequent memory accesses and reduced processor performance. The importance of cache management has become even more critical because of the increasing memory latency, increasing working sets of many emerging applications, and decreasing size of cache devoted to each core due to increased number of cores on a single chip. This dissertation focuses on analyzing some of the problems with managing large caches and proposing cost-effective solutions to improve their performance.

Different workloads and program phases have different locality characteristics that make them better suited to different replacement policies. This dissertation proposes hybrid replacement policy that can select from multiple replacement policies depending on which policy has the highest performance. To implement hybrid replacement with low-overhead,

it shows that cache behavior can be approximated by sampling few sets and proposes the concept of *Dynamic Set Sampling*.

The commonly used LRU replacement policy results in thrashing for memory-intensive workloads that have a working set bigger than the cache size. This dissertation shows that performance of memory-intensive workloads can be improved significantly by changing the recency position where the incoming line is inserted. The proposed mechanism reduces cache misses by 21% over LRU, is robust across a wide variety of workloads, incurs a total storage overhead of less than two bytes, and does not change the existing cache structure.

Modern systems try to service multiple cache misses in parallel. The variation in Memory Level Parallelism (MLP) causes some misses to be more costly on performance than other misses. This dissertation presents the first study on MLP-aware cache replacement and proposes to improve performance by eliminating some of the performance-critical isolated misses.

Finally, this dissertation also analyzes cache partitioning policies for shared caches in chip multi-processors. Traditional partitioning policies either divide the cache equally among all applications or use the LRU policy to do a demand based cache partitioning. This dissertation shows that performance can be improved if the shared cache is partitioned based on how much the application benefits from the cache, rather than on its demand for the cache. It proposes a novel low-overhead circuit that can dynamically monitor the utility of cache for any application. The proposed partitioning improves weighted-speedup by 11%, throughput by 17% and fairness by 11% on average compared to LRU.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Problem

Over the past two decades, processor speeds have increased at a much faster rate than DRAM speeds. Consequently, the number of processor cycles it takes to access main memory has also increased. Current high performance processors have memory access latency of well over 300 cycles, and trends [84] indicate that this number will only increase in the future. The growing disparity between processor speed and memory speed is popularly referred in the architecture community as the *Memory Wall* [86]. Main memory accesses affect processor performance adversely. Therefore, current processors use caches to reduce the number of memory accesses. A cache hit provides fast access to recently accessed data. However, if there is a cache miss at the last level cache, a memory access is initiated and the processor is stalled for hundreds of cycles [84][34]. Therefore, to sustain high performance, it is important to reduce cache misses.

The design of the first level cache is heavily constrained by access time. Furthermore, with out-of-order execution, current processors are able to tolerate some of the first level cache misses [35]. The stringent requirement of fast access time and the limited potential for improvement because of out-of-order execution has led to simpler designs for the first level cache. On the other hand, the design of the second level cache[1] is constrained more by the available on-chip transistors and less by the access time to the cache. Moreover,

---

[1]For simplicity, we assume a two level cache hierarchy throughout this proposal. However, the problems, discussion, and solutions can easily be extended to all the non-primary caches in a multi-level cache hierarchy.

the locality characteristics of the second level cache access stream are different from the first level cache access stream. However, current processors use a traditional management policies for the second level cache without paying attention to the locality characteristics of the access stream visible to the second level cache. With traditional designs, the second level cache is not used efficiently which leads to a large number of cache misses and lower performance. Performance can be improved with designs that are able to better exploit the locality characteristics and the design constraints visible to the second level cache.

Different workloads and program phases have different locality characteristics that make them better suited to different replacement policies. However, traditional cache designs decide the replacement policy at design time and use that policy for all applications and phases. Cache performance can be improved if the cache management can select from multiple replacement policies depending on which policy performs better for the given application or phase. This dissertation provides a practical framework to implement hybrid replacement that can choose the best performing policy at runtime.

Implementing hybrid replacement in a straight-forward manner requires tracking the information for competing replacement policies on a per-set basis using extra tags. The hardware overhead for extra tags for all sets can be prohibitively expensive. Small caches have few tens of sets, however, large caches typically have hundreds or thousands of sets. The hardware overhead for tracking information about replacement policies can be reduced by using the key insight that cache behavior can be approximated with high accuracy by sampling few sets. This mechanism, called *Dynamic Set Sampling* enables allows cost-effective optimization for several caching policies.

The commonly used LRU replacement policy causes thrashing for memory-intensive workloads that have a working set greater than the available cache size. In fact with traditional LRU policy, more than 60% of the lines installed in the second-level cache remain unused between insertion and eviction. Thus, most of the inserted lines occupy cache space

without ever contributing to cache hits. When the working set is larger than the available cache size, cache performance can be improved by retaining some fraction of the working set long enough that at least that fraction of the working set contributes to cache hits. This dissertation shows that performance of memory-intensive workloads can be improved significantly by changing the recency position where the incoming line is inserted.

Performance loss due to long-latency memory accesses can be reduced by servicing multiple memory accesses concurrently. The notion of generating and servicing long-latency cache misses in parallel is called Memory Level Parallelism (MLP). MLP is not uniform across cache misses – some misses occur in isolation while some occur in parallel with other misses. Isolated misses are more costly on performance than parallel misses. Unfortunately, traditional cache replacement algorithms are not aware of the disparity in performance loss that results from the variation in MLP among cache misses. Cache replacement, if made MLP-aware, can improve performance by reducing the number of performance-critical isolated misses. This dissertation proposes a framework for MLP-aware cache replacement by using a run-time technique to compute the MLP-based cost for each cache miss. It then describes a simple cache replacement mechanism that takes both MLP-based cost and recency into account.

Finally, this dissertation also investigates the problem of partitioning a shared cache between multiple concurrently executing applications. The commonly used LRU policy implicitly partitions a shared cache on a demand basis, giving more cache resources to the application that has a high demand and fewer cache resources to the application that has a low demand. However, a higher demand for cache resources does not always correlate with a higher performance from additional cache resources. It is beneficial for performance to invest cache resources in the application that benefits more from the cache resources rather than in the application that has more demand for the cache resources. This dissertation proposes *utility-based cache partitioning (UCP)*, a low-overhead, runtime mechanism that

3

partitions a shared cache between multiple applications depending on the reduction in cache misses that each application is likely to obtain for a given amount of cache resources. The proposed mechanism monitors each application at runtime using a novel, cost-effective, hardware circuit that requires less than 2kB of storage. The information collected by the monitoring circuits is used by a partitioning algorithm to decide the amount of cache resources allocated to each application.

1.2   Thesis Statement

As locality characteristics and design constraints of large caches are different from first-level caches, traditional cache designs – developed for small first-level caches – are inefficient for large caches. Simple and cost-effective changes to cache management can substantially improve the performance of large caches.

1.3   Contributions

This dissertation makes the following contributions:

1. This dissertation presents hybrid replacement policy that can select from multiple replacement policies depending on which policy has the highest performance. To implement hybrid replacement with low-overhead, it shows that cache behavior can be approximated by sampling few sets and proposes the concept of *Dynamic Set Sampling*.

2. This dissertation shows that performance of memory-intensive workloads can be improved significantly by changing the recency position where the incoming line is inserted. The proposed mechanism reduces cache misses by 21% over LRU, is robust across a wide variety of workloads, incurs a total storage overhead of less than

two bytes, and does not change the existing cache structure.

3. This dissertation presents the first study on MLP-aware cache replacement and proposes to improve performance by eliminating some of the performance-critical isolated misses. It describes a hardware mechanism to measure MLP-based cost at runtime and used this cost to drive a cost-sensitive replacement policy.

4. This dissertation shows that performance of shared caches can be improved if the shared cache is partitioned based on how much the competing application benefits from the cache, rather than on its demand for the cache. It proposes a novel low-overhead circuit that can dynamically monitor the utility of cache for any application. The proposed partitioning improves weighted-speedup by 11%, throughput by 17% and fairness by 11% on average compared to LRU. As optimal partitioning is NP hard, this dissertation also proposes a low time-complexity algorithm that is scalable to many cores and performs similar to searching through all the exponential number of possible partitions.

1.4   Dissertation Organization

This dissertation is divided into seven chapters. Related work in discussed in Chapter 2. Chapter 3 describes cost-effective hybrid replacement policies. Chapter 4 analyzes insertion policies for high performance caching. MLP-Aware cache replacement in proposed in Chapter 5. Chapter 6 discusses utility based partitioning of shared caches. Finally, Chapter 7 provides the summary and directions for future work.

Chapter 2

Related Work

Addressing the memory wall problem has been a hot topic of research in the computer architecture community for the past several years. Consequently, there have been several proposals for reducing the penalty of memory accesses. This chapter describes some of the work that relates to the memory wall in general and cache management in particular. Related work for the specific problems of cache management studied in this dissertation is discussed in detail in the corresponding chapters so that qualitative and quantitative comparison can be made with the proposed techniques.

2.1 Caches: Background and Terminology

Caching is one of the most fundamental concepts in computing. Processor caches were introduced as early as the mid sixties [83] to bridge the difference in speed between the processor and memory. Smith [71] did the initial study on performance of cache memory as a function of the three basic parameters of the cache: size, associativity and linesize. Hill [26] classified cache misses into the popular 3C model: compulsory misses, conflict misses, and capacity misses. Compulsory misses correspond to the number of cache lines in the trace, conflict misses are the misses that would be reduced by increasing the associativity of the cache to fully associative, and the remaining misses are capacity misses. This model however does not taking into account the variation in misses because of changing the replacement policy. Puzak [61] analyzed cache replacement algorithms for processor caches and proposed the *shadow directory* mechanism for improving cache replacement.

2.2 Related Work in Cache Organization

During the initial years, on-chip caches were small and determined the cycle time of the processor. To keep the access time small, the cache was typically configured as a direct-mapped structure. A substantial body of research has investigated caching optimizations to reduce conflict misses. Recently, researchers have also started to focus on cache organizations that increase the effective capacity of the cache. This section describes some of the cache organization related work that focuses on reducing conflict and capacity misses.

2.2.1 Reducing Conflict Misses

Memory accesses in general purpose applications are non-uniformly distributed across the sets in a cache [57] [37]. This non-uniformity creates a heavy demand on some sets, which can lead to conflict misses, while other sets remain underutilized. Substantial research effort has been put forth to address this problem for direct-mapped caches. Victim caches [33] are small, fully-associative buffers that provide limited additional associativity for heavily utilized entries in a direct-mapped cache. The hash-rehash cache [1], the adaptive group-associative cache [57], and the predictive sequential-associative cache [8] trade variable hit latency for increased associativity. With these schemes, if the first attempt to access the cache results in a miss, the hash function that maps addresses to sets is changed, and a new cache access is initiated. This process may be repeated multiple times until either the data is found or a miss is detected. These techniques were proposed for first level direct-mapped caches, and their effectiveness reduces as associativity increases due to the inherent performance benefit of increased associativity.

Reducing conflict misses for large secondary caches have also been studied in the literature. Hallnor et al. [25] proposed the Indirect Index Cache (IIC) as a mechanism to achieve full-associativity through software management. Qureshi et al [64] proposed the *Variable way set associative (V-Way)* cache to achieve the global replacement benefits of a

fully associative cache while maintaining the constant hit latency of a set-associative cache.

2.2.2 Reducing Capacity Misses

Capacity misses can be reduced by increasing the cache size. Several studies [87][4] have looked at compression techniques for increasing the effective capacity of the cache. The key idea in all the proposed compression schemes is that some values occur much more frequently than others and hence can be stored in few bits. It is desirable that the compression scheme has fast compression and decompression latency. If the performance loss from decompression is more than the capacity benefits obtained from compression, then compression can reduce performance. Adaptive cache compression was proposed by Alameldeen et al. [3] to perform compression only if it is likely to improve performance. An orthogonal approach to increase cache capacity is to filter unused words in the cache lines. The recently proposed *Line Distillation* technique [62] filters unused words in the cache lines once the lines have crossed a pre-defined position in the LRU stack. There are several predictor-based techniques [31][40][11][60] that provides spatial filtering.

2.3 Related Work in Improving Cache Management

2.3.1 Improving Cache Replacement

Current caches typically use either LRU or some approximation of LRU [73] as the replacement policy. An ideal replacement scheme can minimize the number of misses by choosing the victim that will be accessed the farthest in the future [7]. Although, such a scheme is impossible to build, it shows that there is significant room for improvement over the LRU replacement policy. Several proposals [50] [6] [70] [42] [54] have looked at improving cache replacement by taking into account both recency as well as frequency.

2.3.2 Related Work in Cache Bypassing

It is not beneficial to install a line that is never referenced while it is in the cache. McFarling[47] proposed dynamic exclusion to reduce conflict misses in a direct-mapped instruction cache. However, the proposed scheme is not easily applicable to data caches. Gonzalez et al. [24] proposed using a *locality prediction table* to bypass access patterns that are likely to pollute the cache. Their technique works well only for predictable access patterns, which are usually found in numeric code. Tyson et al. [81] looked at static and dynamic techniques to mark the load instructions as *Cacheable/Not Allocatable* (CNA). All the data references generated by the CNA instructions are not allocated in the cache. Rivers and Davidson [66] looked at reducing the conflict misses in a direct mapped cache by bypassing lines with low temporal locality into a small fully associative cache. Johnson [31] describes a technique to track the reuse behavior of cache lines by keeping the access information in a Macro Address Table. If the incoming line is likely to have less reuse than the line it will evict, then the incoming line is not installed in the cache. Their studies with direct-mapped caches show that cache bypassing can help performance by reducing both misses and bus traffic.

2.3.3 Related Work in Early Eviction

Another approach to address the problem of low locality lines is to evict them early. Wong et al. [85] proposed modified LRU policies for early eviction of lines with low temporal locality by marking some instructions as low temporal instructions. Wang et al. [82] looked at compiler techniques to help the replacement engine by tagging cache lines with *Evict-me* bits. Another area of research has been to predict the last touch to a cache line [41] [43]. After the last touch is encountered, the line can either be turned off [36] or be used to store prefetched data [41].

2.3.4 Cost-Sensitive Cache Management

Another area of research is to make the cache management aware of the variation in performance impact of cache misses. Srinivasan et al. [75] analyzed the criticality of load misses for out-of-order processors. Based on the criticality analysis, Srinivasan et al. also investigated criticality based caches [74] and concluded that the working set of critical loads is large, and therefore it is better to have locality based caching. Cost-sensitive replacement for on-chip caches was investigated by Jeong et al. [30]. They proposed variations of LRU that take *cost* (any numerical property associated with a cache block) into account. They evaluated their cost-sensitive policies for Non-Uniform Memory Access (NUMA) systems by taking the bank access latency (local-hit vs remote-hit) as the *cost* parameter. They showed that significant performance improvements are possible when there is huge variation in the *cost* of different cache blocks.

2.4 Reducing Misses with Prefetching

A cache miss can be avoided if the requested line is brought into the cache ahead of its use. Several proposals have investigated run-time prefetching techniques using specialized hardware [12] [17] [22] [32] [5] [53]. The central idea of these schemes is to store information about recent memory accesses that missed the cache and detect a pattern in the miss address stream. If the compiler can predict the address of the desired data before it is likely to be referenced, then it can insert software prefetches in the code [59]. Data prefetching is extremely beneficial when the access stream has either predictable pattern or the compiler can accurately predict the addresses of the data ahead of its use. While numerical programs contain regular access patterns that are easy to prefetch, integer programs have very irregular access patterns that are much more difficult to predict. If the prefetches generated by the prefetcher are not accessed then the prefetched lines can cause cache pollution and bandwidth contention, which can lead to reduced performance. Palacharla et

al. [56] investigated using the stream buffer instead of the secondary cache for streaming numerical programs that have a working set larger than the cache size.

2.5   Servicing Demand Misses in Parallel

Another method to reduce the performance impact of cache misses is to service the misses in parallel. The notion of generating and servicing multiple outstanding cache misses in parallel is called *Memory Level Parallelism* (MLP) [23]. Kroft [39] proposed lockup free caches to allow instruction processing under a cache miss. Out-of-order execution engines inherently improve MLP by continuing to execute instructions after a long-latency miss. Instruction processing stops only when the instruction window becomes full. If additional misses are encountered before the window becomes full, then these misses are serviced in parallel with the stalling miss. The effectiveness of an out-of-order engine's ability to increase MLP is limited by the instruction window size.

Runahead execution [51] overcomes the limitation posed by the instruction window size. When the instruction window becomes full due to a long-latency cache miss, a runahead execution engine removes the stalling instruction from the window and processes instructions speculatively such that long-latency cache misses and their dependents do not stall the window. Instruction processing continues speculatively with the sole aim of generating additional (useful) misses to be serviced in parallel with the stalling miss.

Chou et al. [16] analyzed the effectiveness of different microarchitectural techniques such as out-of-order execution, value prediction [89], and runahead execution on increasing MLP. They concluded that microarchitecture optimizations can have a profound impact on increasing MLP. MLP can also be improved at the compiler level. Read miss clustering [55] is a compiler technique in which the compiler reorders load instructions with predicable access patterns to improve memory parallelism.

11

Chapter 3

Hybrid Replacement via Dynamic Set Sampling

Different workloads and program phases have varying locality characteristics that make them better suited to different replacement policies. However, traditional cache designs decide the replacement policy at design time and use that policy for all applications and phases. Cache performance can be improved if the cache management can select from multiple replacement policies depending on which policy performs better for the given application or phase. This chapter provides a practical framework to implement hybrid replacement that can choose the best performing policy at runtime.

3.1 Motivation

Figure 3.1 compares the Misses Per Thousand Instructions (MPKI) for the baseline 1MB 16-way L2 cache for two replacement policies: Least Recently Used (LRU) and Least Frequently Used (LFU). The details about other parameters of the experiment are discussed in section 3.2. LFU replacement reduces MPKI by more than 10% compared to LRU for seven out of the fifteen benchmarks. Benchmarks such as art and galgel have less than 50% of the misses with LFU compared to LRU. However, LFU can substantially increase misses for LRU-friendly benchmarks such as equake, parser, mgrid, and swim. For example, LFU more than doubles the MPKI for parser and swim. Thus, neither of the two policies, LRU and LFU, perform well across all benchmarks. A mechanism that selects the best performing policy for each benchmark can substantially improve cache performance.
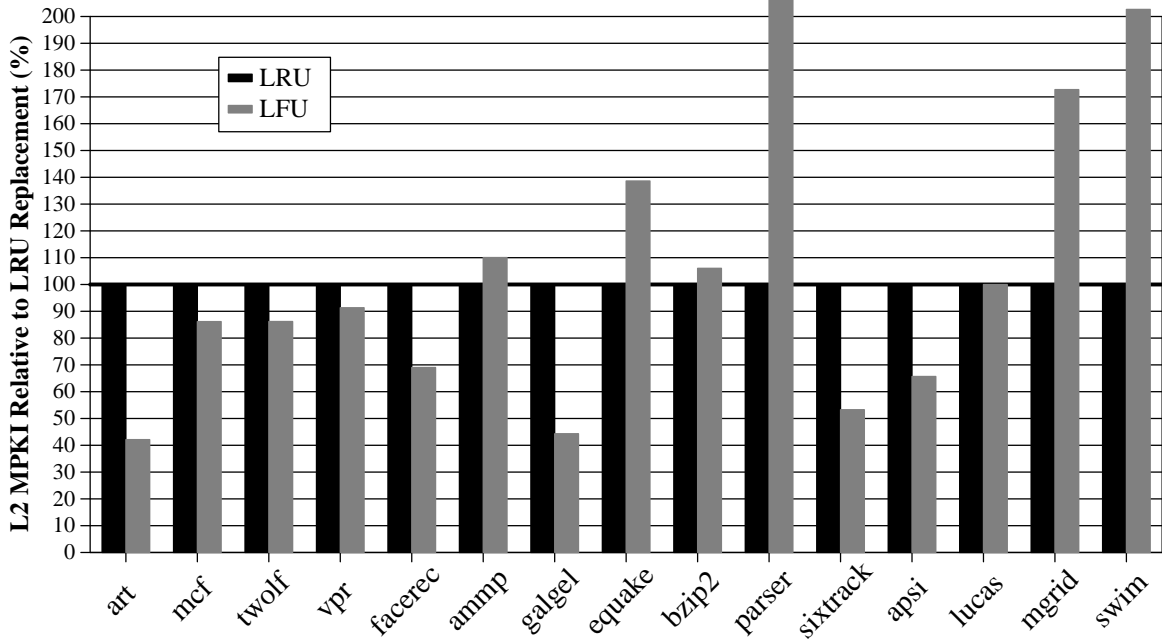
Figure 3.1: Comparison of replacement policies: LRU and LFU

3.2 Experimental Methodology

3.2.1 Configuration

Table 3.1 shows the parameters of the baseline configuration used in our studies. We use an in-house execution-driven simulator that models the alpha ISA. The processor core is 8-wide issue, out-of-order, with 128-entry reservation station. The 128-entry store buffer prevents the processor for stalling from store-misses unless the store buffer is full. Because our study deals with the memory system we model the memory system in detail. DRAM bank conflicts and bus queuing delays are modeled. The baseline L2 cache is 1MB in size and is organized as a 16-way set-associative structure. Unless stated otherwise, all caches use LRU policy for replacement decisions. For experiments with LFU replacement, the LFU policy is implemented by associating a five-bit frequency counter with each cache line. When a cache line is installed, the frequency counter associated with that line is

initialized to 0. The frequency counter is incremented at each access to the line. When the frequency counter of a line reaches its maximum value, the frequency counter of all the lines in that set is halved. On a cache miss, the line that has the lowest value of the frequency counter in the miss-causing set is identified as the victim. Ties for the lowest value of frequency counter are broken randomly. Unless stated otherwise, MPKI numbers are obtained using a trace-driven cache simulator to reduce simulation time.

Table 3.1: Baseline system configuration

Pipeline	8 wide, out-of-order, with 128 entry reservation station;
Branch Predictor	64 kB hybrid branch predictor with 4k-entry BTB minimum branch misprediction penalty of 15 cycles.
Instruction Cache	16kB, 64B line-size 4-way with LRU replacement, 2-cycle access.
Data Cache	16kB, 64B line-size,4-way with LRU replacement, 2-cycle access.
Unified L2 Cache	1MB, 64B line-size, 16-way with LRU replacement, 15-cycle hit, 32-entry MSHR, 128-entry store buffer.
Memory	32 DRAM banks; 400-cycle access latency; bank conflicts modeled; maximum 32 outstanding requests;
Bus	16B-wide split-transaction bus at 4:1 frequency ratio. queuing delays modeled

3.2.2 Benchmarks

We use SPEC CPU2000 benchmarks compiled for the Alpha ISA with the `-fast` optimizations and profiling feedback enabled. For each benchmark, a representative slice of 250M instructions was obtained with a tool we developed using the SimPoint [58] methodology. For all benchmarks, except apsi, the `reference` input set is used. For apsi, the `train` input set is used.

Because cache replacement does not affect the number of compulsory misses, benchmarks that have a high percentage of compulsory misses are unlikely to benefit from improvements in cache replacement algorithms. Therefore, detailed studies are performed

only for benchmarks for which approximately 50% or fewer misses are compulsory misses. Key results for the 11 SPEC benchmarks excluded from the detailed study will be shown in Appendix A. Table 3.2 shows the type, the fast-forward interval (FFWD), the number of L2 misses, and the percentage of compulsory misses for each benchmark.

Table 3.2: Benchmark summary (B = Billion)

Name	Type	FFWD	MPKI	Compulsory Misses
art	FP	18.25B	38.7	0.5%
mcf	INT	14.75B	136	1.8%
twolf	INT	30.75B	3.48	2.9%
vpr	INT	60B	2.16	4.3%
facerec	FP	111.75B	3.66	4.8%
ammp	FP	4.75B	2.83	5.0%
galgel	FP	14B	5.34	5.9%
equake	FP	26.25B	18.4	14.2%
bzip2	INT	2.25B	2.4	14.8%
parser	INT	66.25B	1.57	20.0%
sixtrack	FP	8.5B	0.42	20.7%
apsi	FP	3.25B	0.32	21.4%
lucas	FP	2.5B	16.2	41.6%
mgrid	FP	3.5B	7.73	46.6%
swim	FP	3.5B	23.0	50.4%

3.3   Hybrid Replacement

For some benchmarks LRU has fewer misses than LFU and for some LFU has fewer misses than LRU. We want a mechanism that can dynamically choose the replacement policy that has the fewest misses. A straightforward method of doing this is to implement both LFU and LRU in two additional tag directories (note that data lines are not required to estimate the performance of replacement policies) and to keep track of which of the two policies is doing better. The main tag directory of the cache can select the policy that is

giving the lowest number of cache misses. In fact, a similar technique of implementing multiple policies and dynamically choosing the best performing policy is well understood for hybrid branch predictors [49]. However, to our knowledge, no previous research has looked at dynamic selection of replacement policy by implementing multiple replacement schemes concurrently. Part of the reason is that the hardware overhead of implementing two or more additional tag directories, each the same size as the tag directory of the main cache, is expensive. To reduce this hardware overhead, we provide a novel, cost-effective solution that makes hybrid replacement practical. We explain our selection mechanism before describing the final cost-effective solution.

3.3.1   Tournament Selection of Replacement Policy

Let MTD be the main tag directory of the cache. For facilitating hybrid replacement, MTD is capable of implementing both LFU and LRU. MTD is appended with two Auxiliary Tag Directories (ATDs): ATD-LFU and ATD-LRU. Both ATD-LFU and ATD-LRU have the same associativity as MTD. ATD-LFU implements only the LFU policy, and ATD-LRU implements only the LRU policy. A saturating counter (PSEL) keeps track of which of the two ATDs is doing better. The access stream visible to MTD is also fed to both ATD-LFU and ATD-LRU. Both ATD-LFU and ATD-LRU compete and the output of PSEL is an indicator of which policy is doing better. The replacement policy to be used in MTD is chosen based on the value of PSEL. We call this mechanism Tournament Selection (TSEL). Figure 3.2 shows the operation of the TSEL mechanism for one set in the cache.

If a given access hits or misses in both ATD-LFU and ATD-LRU, neither policy is doing better than the other. Thus, PSEL remains unchanged. If an access misses in ATD-LFU but hits in ATD-LRU, LRU is doing better than LFU for that access. In this case, PSEL is decremented. Conversely, if an access misses in ATD-LRU but hits in ATD-LFU, LFU is doing better than LRU. Therefore, PSEL is incremented. Note that, only accesses

16

ATD–LFU	ATD–LRU	Action
HIT	HIT	PSEL unchanged
MISS	MISS	PSEL unchanged
MISS	HIT	Decrement PSEL
HIT	MISS	Increment PSEL

Saturating Counter

ATD–LFU   ATD–LRU

SET A → − PSEL + ← SET A

↓ MTD

SET A

If MSB of PSEL is 1,   MTD uses LFU
else MTD uses LRU
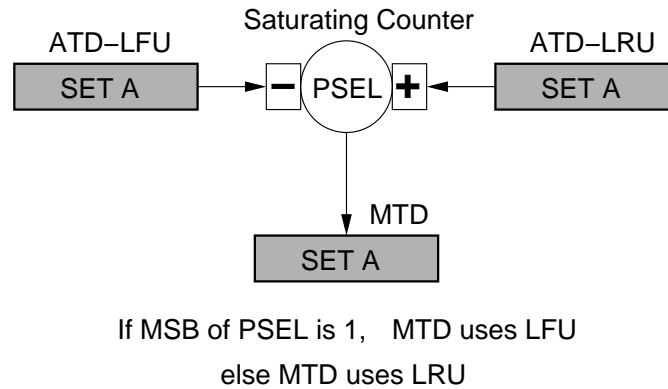
Figure 3.2: Tournament selection of replacement policies for a single set.

that result in a miss for MTD are serviced by the memory system. If an access results in a hit for MTD but a miss for either ATD-LFU or ATD-LRU, then it is not serviced by the memory system. Instead, the ATD that incurred the miss finds a replacement victim using its replacement policy and updates the tag field associated with the replacement victim. Unless stated otherwise, we use a 10-bit PSEL counter in our experiments. All PSEL updates are done using saturating arithmetic.

If LFU incurs fewer misses than LRU, then PSEL will be saturated towards its maximum value. Similarly, PSEL will be saturated towards zero if the opposite is true. If the most significant bit (MSB) of PSEL is 1, the output of PSEL indicates that LFU is doing better. Otherwise, the output of PSEL indicates that LRU is doing better.

17

A simple method of extending the TSEL mechanism for the entire cache is to have both ATD-LFU and ATD-LRU feed a single global PSEL counter. The output of the single PSEL decides the policy for *all* the sets in MTD. We call this mechanism TSEL-global. An example of the TSEL-global scheme is shown in Figure 3.3 for a cache that has eight sets.
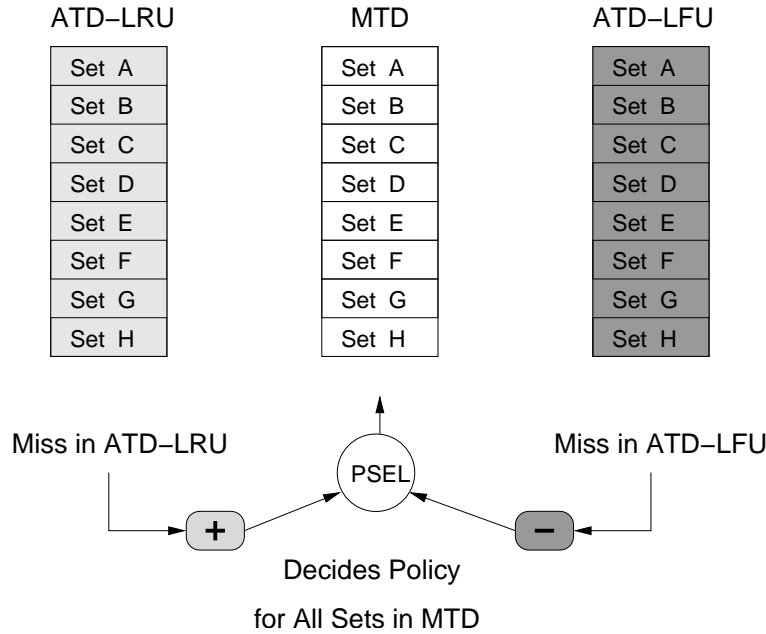
Figure 3.3: TSEL-global mechanism

3.3.2 Results for Tournament Selection

Figure 3.4 compares the MPKI of LRU, LFU, and the TSEL-global mechanism. In almost all cases TSEL-global provides a similar MPKI as the better of the two policies, LRU and LFU. For ammp, TSEL-global has better MPKI than either of the component policies. This happens because ammp has two phases during the program execution. LFU has fewer misses than LRU in the first phase and LRU has fewer misses than LFU in the second phase. With TSEL-global the cache is able to get the policy better suited to each

phase, thus outperforming each of the component policies. Although we have explained the TSEL-global mechanism with LRU and LFU as the component policies, the mechanism can be used to select between any two replacement policies.
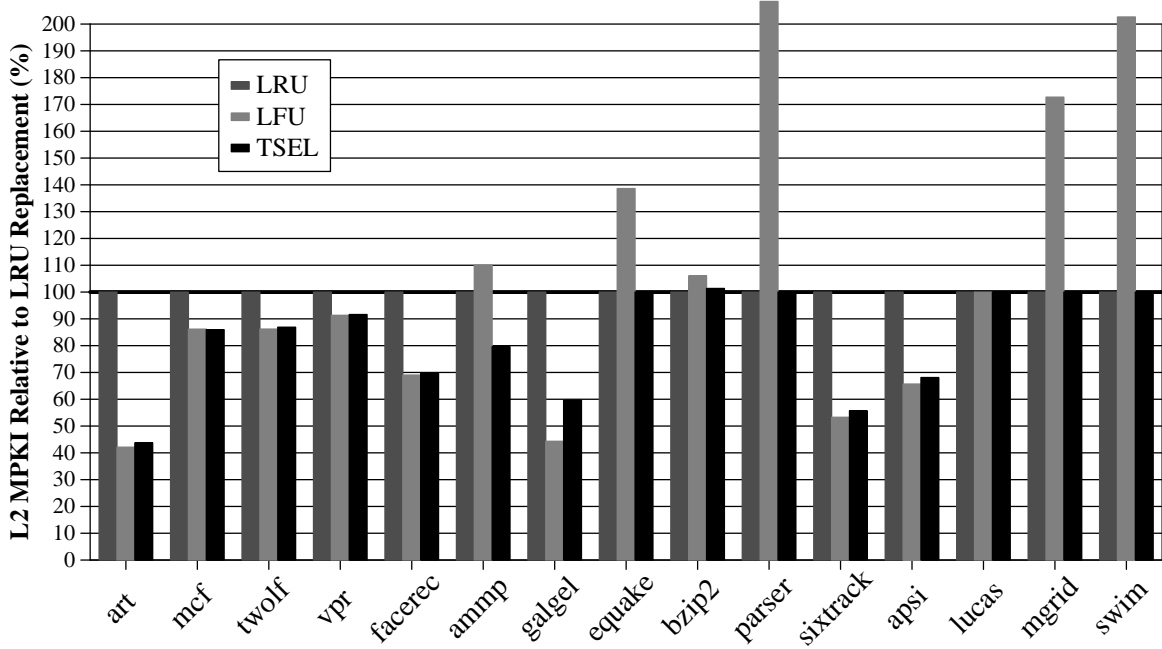
Figure 3.4: Comparison of replacement policies: LRU, LFU, and Selecting between LRU and LFU using TSEL-global.

3.4 Dynamic Set Sampling

Although TSEL-global can select between component policies, it requires two ATDs, each sized the same as MTD, which makes TSEL-global a high-overhead option. The key insight that allows us to reduce the number of ATD entries for TSEL-global is that it is not necessary to have all the sets participate in deciding the output of PSEL. If only a few sampled sets are allowed to decide the output of PSEL, then the TSEL-global mechanism

will still choose the best performing policy with a high probability. The sets that participate in updating PSEL are called *Leader Sets*. Figure 3.5 shows a TSEL-global mechanism with *Dynamic Set Sampling (DSS)*. Sets B, E, and G are the leader sets. These sets have ATD entries and are the only sets that update the PSEL counter. There are no ATD entries for the remaining sets.
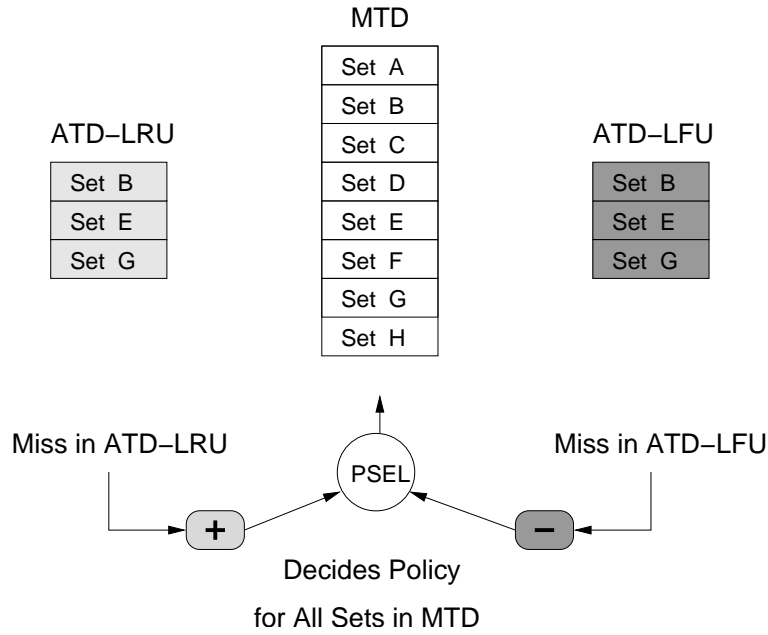
Figure 3.5: Reducing ATD overhead via Dynamic Set Sampling.

For the example in Figure 3.5, DSS reduces the number of ATD entries required for the TSEL-global mechanism to 3/8 of its original value. A natural question is: how many leader sets are sufficient to select the best performing replacement policy? We provide both analytical as well as empirical answers to this question.

3.4.1 Analytical Model for Dynamic Set Sampling

We make the simplifying assumption that all sets affect performance equally. Let $P(Best)$ be the probability that the best performing policy is selected by the sampling-based TSEL-global mechanism. Let there be N sets in the cache. Let p be the fraction of the sets that favor the best performing policy. Given that we have two policies, LRU and LFU, by definition $p \geq 0.5$. Thus, if only one set is selected at random from the cache as the leader set, then $P(Best) = p$.

If three sets ($N \gg 3$) are chosen at random from the cache as leader sets, then for the mechanism to correctly select the globally best performing policy, at least two of the three leader sets should favor the globally best performing policy. Thus, for three leader sets, $P(Best)$ is given by:

$$P(Best) = p^3 + 3 \cdot p^2 \cdot (1 - p) \tag{3.1}$$

In general, if k sets ($k \ll N$) are randomly selected from the cache as leader sets, then $P(Best)$ is given by:

$$P(Best) = \sum_{i=0}^{(k-1)/2} \binom{k}{i} \cdot p^{(k-i)} \cdot (1 - p)^i ... For\ odd\ k \tag{3.2}$$

$$P(Best) = \frac{1}{2} \cdot \binom{k}{k/2} \cdot p^{k/2} \cdot (1 - p)^{k/2} + \sum_{i=0}^{(-1+k/2)} \binom{k}{i} \cdot p^{(k-i)} \cdot (1 - p)^i ... For\ even\ k \tag{3.3}$$

Where $\binom{k}{i}$ refers to the number of combinations of i elements from a group of k elements ($k!/(i! \cdot (k-i)!)$). Figure 3.6 plots $P(Best)$ for different numbers of leader sets as p is varied. When there is a significant difference in the performance of the two replacement policies, the value of p is higher than 0.7. Therefore, from Figure 3.6 we can conclude that

a small number of leader sets (16-32) is sufficient to select the globally best-performing policy with a high ($> 95\%$) probability. This is an important result because it means that the baseline cache can have expensive ATD entries for only 16-32 sets (i.e., about 2% to 3% of all sets) instead of all the 1024 sets in the cache.
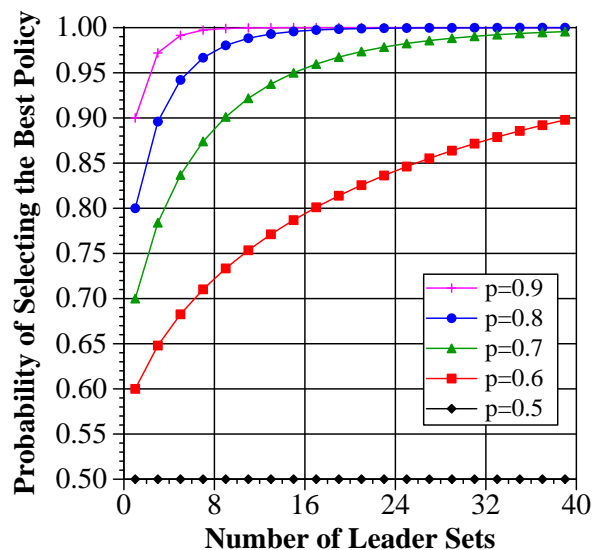
Figure 3.6: Analytical Bounds on Number of Leader Sets.

3.5 Sampling Based Adaptive Replacement

DSS makes it possible to choose the best performing policy with high probability even with very few sets in the ATD. Because the number of leader sets is small, the hardware overhead can be further reduced by embedding the functionality of one of the ATDs in MTD. Figure 3.7 shows such a sampling-based hybrid scheme, called *Sampling Based Adaptive Replacement (SBAR)*. The sets in MTD are logically divided into two categories: *Leader Sets* and *Follower Sets*. The leader sets in MTD use only the LRU policy for replacement and participate in updating the PSEL counter. The follower sets implement both

22

the LFU and the LRU policies for replacement and use the PSEL output to choose their replacement policy. The follower sets do not update the PSEL counter. There is only a single ATD, ATD-LRU. ATD-LRU implements only the LRU policy and has only sets corresponding to the leader sets. In Figure 3.7 Sets B, E, and G are leader sets and Sets A, C, D, F, and H are follower sets. Thus, the SBAR mechanism requires a single ATD with entries only corresponding to the leader sets.
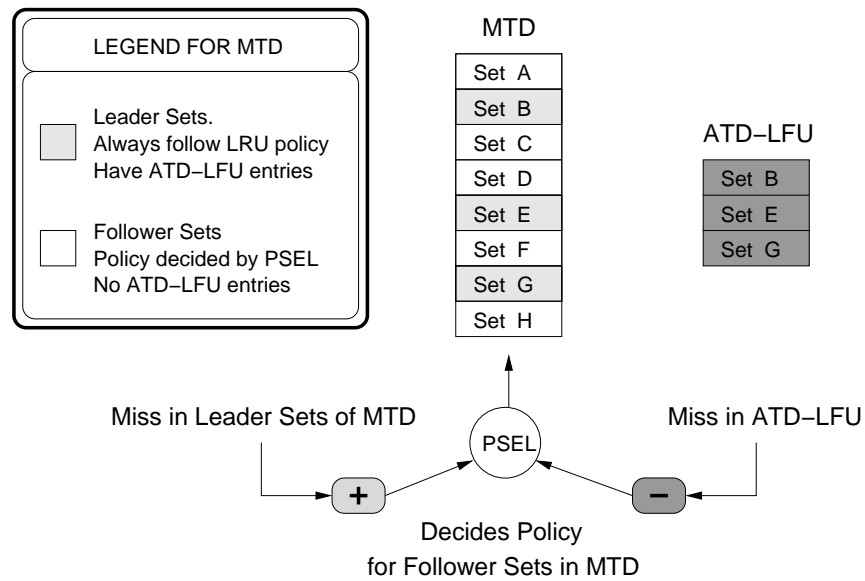
Figure 3.7: Sampling Based Adaptive Replacement

Although the example shows LRU policy implemented in leader sets of MTD, in general, any of the two policies can be implemented in the leader sets of the MTD and the state of the other policy can be tracked using the ATD. Figure 3.5 shows is a special case of the SBAR mechanism where both policies are tracked using the ATD. An alternative configuration where both policies are implemented in some small dedicated number of sets and using the better performing policy on the remaining sets. This option in explored by the in-cache dynamic set sampling mechanism in detail in the Chapter 4.

3.5.1 Leader Set Selection Mechanism

We now discuss a method to select leader sets. Let N be the number of sets in the cache and K be the number of leader sets (in our studies we restrict the number of leader sets to be a power of 2). We logically divide the cache into K equally-sized regions each containing N/K sets. We call each such region a *constituency*. One leader set is chosen from each constituency, either statically at design time or dynamically at runtime. A bit associated with each set then identifies whether the set is a leader set. We propose a leader set selection policy that obviates the need for marking the leader set in each constituency on a per-set basis. We call this policy the *simple-static* policy. It selects set 0 from constituency 0, set 1 from constituency 1, set 2 from constituency 2, and so on. For example, if K=32 and N=1024, the simple-static policy selects sets 0, 33, 66, 99,..., and 1023 as leader sets. For the leader sets, bits [9:5] of the cache index are identical to the bits [4:0] of the cache index, which means that the leader sets can easily be identified using a single five-bit comparator without any extra storage. Unless stated otherwise, SBAR uses the simple-static policy.

3.5.2 Hardware Cost of SBAR

The dynamic selection of SBAR comes at a small hardware overhead in terms of the ATD entries. Table 3.3 details the storage overhead of SBAR assuming a 40-bit physical address space and 32 leader sets. SBAR requires a storage overhead of 1920 bytes, which is less than 0.2% of the total area of the baseline L2 cache. In addition MTD needs storage for implementing the two component replacement policies.

Table 3.3: Storage overhead of SBAR.

Size of each ATD entry (1 valid bit + 24-bit tag + 5-bit LFU)	30 bits
Total number of ATD entries per leader set	16
ATD overhead per leader set (30 bits/way * 16 ways)	60 B
Total SBAR overhead (32 leader sets * 60 B/set)	1920 B
Area of baseline L2 cache (64kB tags + 1MB data)	1088 kB
Percentage increase in L2 area due to SBAR (1920B/1088kB)	**0.18%**

3.6 Results

3.6.1 Comparison of TSEL-global and SBAR
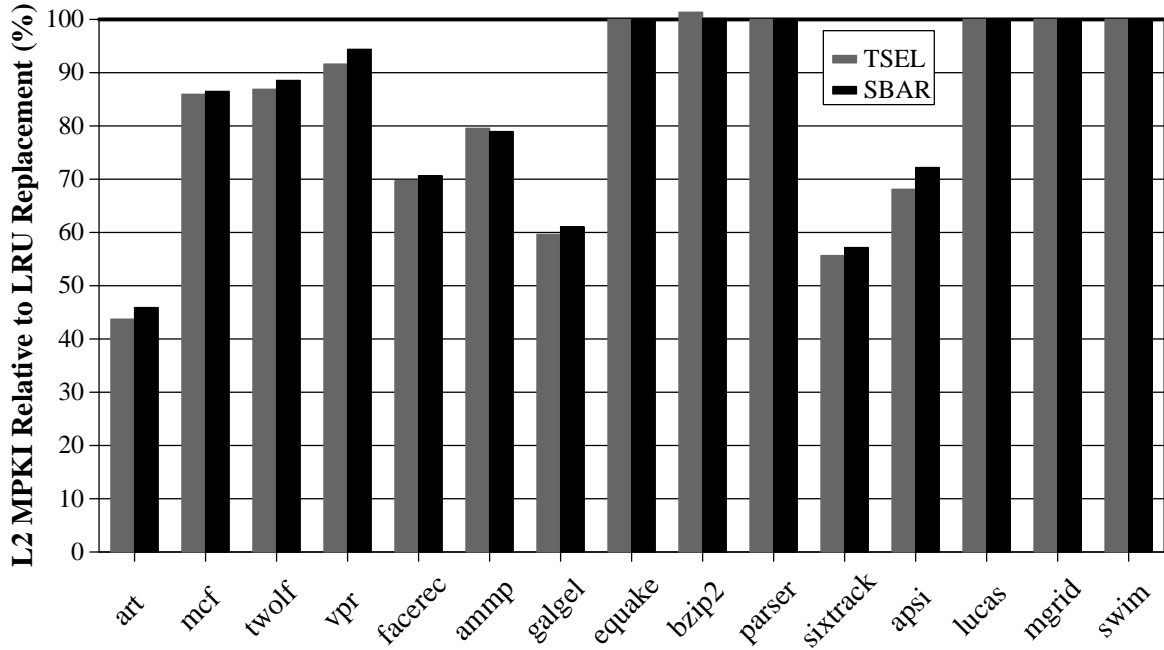
Figure 3.8: Comparison of TSEL-global and SBAR.

Figure 3.8 shows the MPKI of the TSEL-global mechanism and the SBAR mechanism relative to the baseline cache with LRU policy. Both TSEL-global and SBAR select between LRU and LFU. The reduction provided by both mechanisms is similar, except that SBAR has substantially less hardware overhead than TSEL-global as it requires 64x fewer ATD entries than TSEL-global. Thus DSS allows SBAR to implement the hybrid replacement mechanism in a cost-effective manner.

3.6.2 Effect of Number of Leader Sets on SBAR

We use 32 leader sets for implementing SBAR. This sections analyzes the sensitivity of varying the number of leader sets on the performance of SBAR. Figure 3.9 compares SBAR mechanism with 16, 32, and 64 leader sets with the TSEL-global mechanism. The PSEL counter is scaled appropriately for SBAR, using a 9-bit PSEL for 16 leader sets and a 11-bit PSEL for 64 leader sets. For two benchmarks, ammp and apsi, there is a significant difference between the MPKI of SBAR with 16 leader sets and TSEL-global indicating that 16 sets are not sufficient for SBAR to perform similar to TSEL-global. However, with 32 or more sets SBAR and TSEL have similar cache performance for all the benchmarks studied.
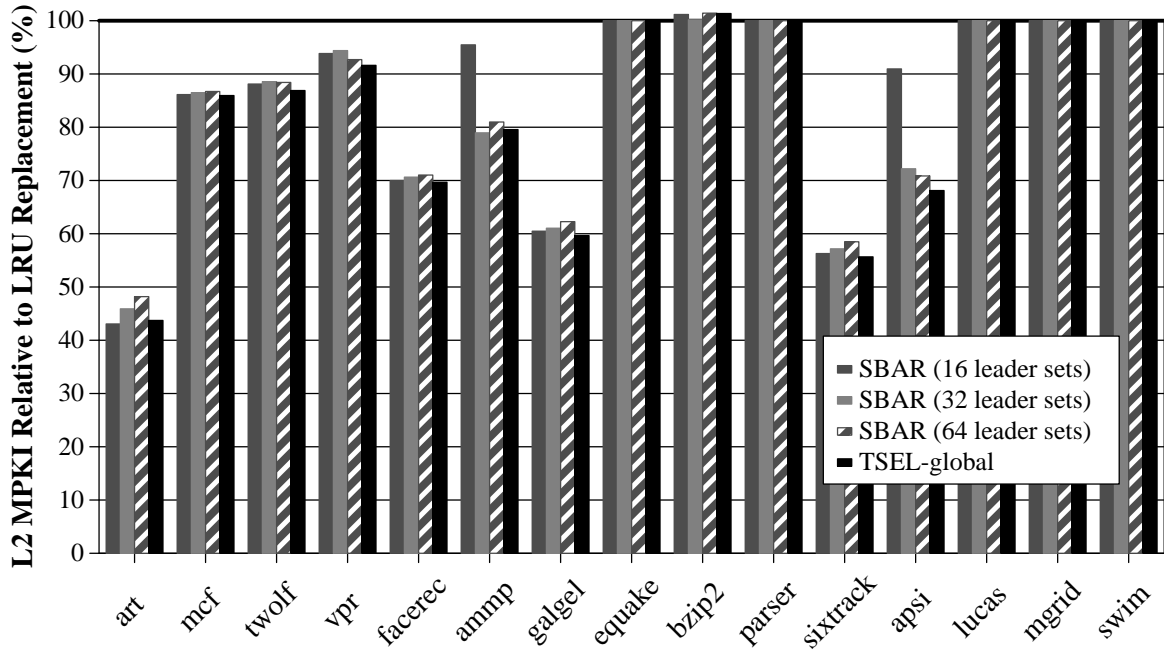
Figure 3.9: Effect of Number of Leader Sets on SBAR.

3.6.3   SBAR selection between LRU and Random replacement

The proposed SBAR mechanism can be used to select between any two replacement policies. For example, SBAR can be used to select between LRU and random (RND) replacement by implementing random replacement in the ATD. The PSEL counter would then be an indicator of which of the two policies, LRU and RND, is doing better. In our experiments with random replacement, we implement RND using the gnu c rand() function. Figure 3.10 compares the MPKI of three replacement schemes: LRU, RND, and the SBAR-based dynamic selection between LRU and RND. RND replacement reduces misses substantially for benchmarks art, facerec, ammp, galgel, sixtrack, and apsi while increasing misses for benchmarks by more than 20% for benchmarks twolf, vpr, bzip, and parser. SBAR based dynamic selection between LRU and RND has MPKI that is similar to the policy that incurs the fewest misses for the given benchmark.
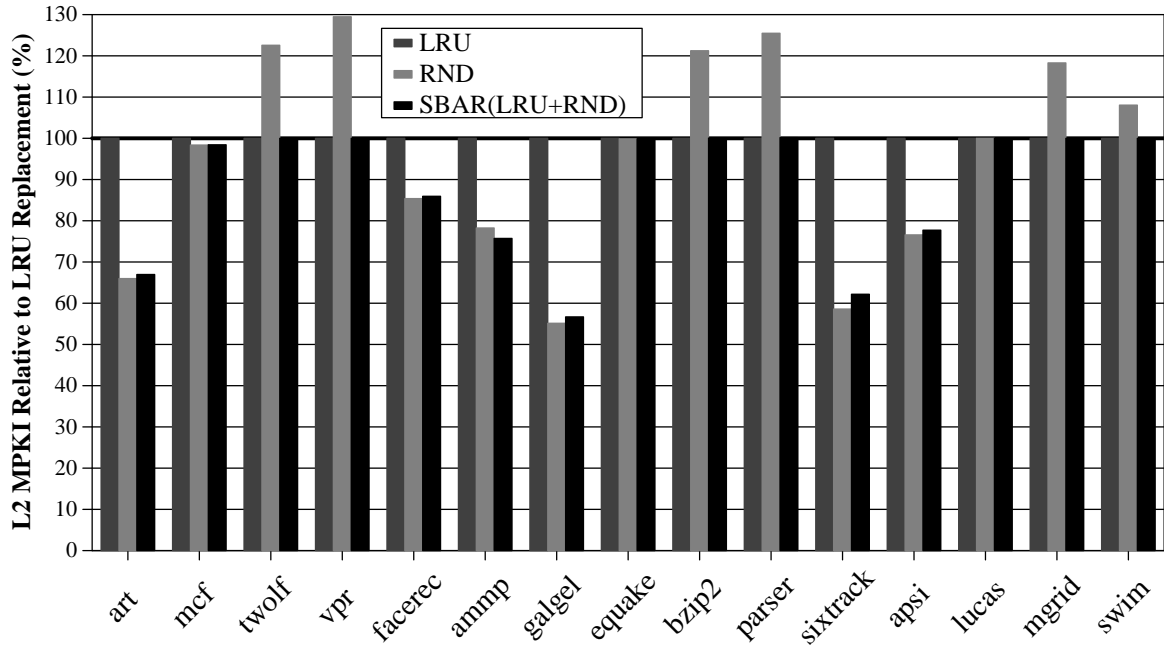
Figure 3.10: Comparisons of LRU, Random, and SBAR (LRU+RND)

3.7 Summary

This chapter proposed an implementable mechanism that can adaptively select between two replacement policies, depending on which policy is providing fewer misses at a given time during execution. Different replacement policies can perform better in different program phases, and therefore having an adaptive hybrid replacement policy can provide better performance than either of the constituent replacement policies.

We propose the Dynamic Set Sampling (DSS) mechanism that predicts the behavior of the whole cache by sampling the behavior of only a few sets in the cache. In our hybrid replacement scheme, DSS significantly reduces the hardware cost of predicting the performance impact of a replacement policy. In general, DSS is a basic building block that can enable cost-effective implementation of several other cache optimizations.

Chapter 4

Adaptive Insertion Policies

The commonly used LRU replacement policy is susceptible to thrashing for memory-intensive workloads that have a working set greater than the available cache size. For such applications, the majority of lines traverse from the MRU position to the LRU position without receiving any cache hits, thus, resulting in inefficient use of cache space. Cache performance can be improved if some fraction of the working set is retained in the cache so that at least that fraction of the working set can contribute to cache hits.

We show that simple changes to the *insertion policy* can significantly reduce cache misses for memory-intensive workloads. We propose the *LRU Insertion Policy (LIP)* which places the incoming line in the LRU position instead of the MRU position. LIP protects the cache from thrashing and results in close to optimal hit-rate for applications that have a cyclic reference pattern. We also propose the *Bimodal Insertion Policy (BIP)* as an enhancement of LIP that adapts to changes in the working set while maintaining the thrashing protection of LIP. We finally propose a *Dynamic Insertion Policy (DIP)* to choose between BIP and the traditional LRU policy depending on which policy incurs fewer misses. The proposed insertion policies do not require any change to the existing cache structure, are trivial to implement, and have a storage requirement of less than two bytes. DIP reduces the average MPKI of the baseline 1MB 16-way L2 cache by 21%, bridging two-thirds of the gap between LRU and OPT.

4.1 Introduction

The LRU replacement policy and its approximations have remained as the de-facto standard for replacement policy in on-chip caches over the last several decades. While the LRU policy has the advantage of good performance for high-locality workloads, it can have a pathological behavior for memory-intensive workloads that have a working set greater than the available cache size. There have been numerous proposals to improve the performance of LRU, however, many of these proposals incur a huge storage overhead, significant changes to existing design, and poor performance for LRU-friendly workloads. Every added structure and change to the existing design requires design effort, verification effort, and testing effort. Therefore, it is desirable that changes to the conventional replacement policy require minimal changes to the existing design, require no additional hardware structures, and perform well for a wide variety of applications. This chapter focuses on designing a cache replacement policy that performs well for both LRU-friendly and LRU-averse workloads while requiring negligible hardware overhead and changes.

We divide the problem of cache replacement into two parts: *victim selection policy* and *insertion policy*. The victim selection policy decides which line gets evicted for storing an incoming line, whereas, the insertion policy decides where in the replacement list is the incoming line placed. For example, the traditional LRU replacement policy inserts the incoming line in the MRU position, thus using the policy of *MRU Insertion*. Inserting the line in the MRU position gives the line a chance to obtain a hit while it traverses all the way from the MRU position to the LRU position. While this may be a good strategy for workloads whose working-set is smaller than the available cache size or for workloads that have high temporal locality, such an insertion policy causes thrashing for memory-intensive workloads that have a working set greater than the available cache size. We show that with the traditional LRU policy, more than 60% of the lines installed in the L2 cache remain unused between insertion and eviction. Thus, most of the inserted lines occupy

cache space without ever contributing to cache hits. When the working set is larger than the available cache size, cache performance can be improved by retaining some fraction of the working set long enough that at least that fraction of the working set contributes to cache hits. However, the traditional LRU policy offers no protection for retaining the cache lines longer than the cache capacity.

We show that simple changes to the insertion policy can significantly improve cache performance for memory-intensive workloads while requiring negligible hardware overhead. We propose the *LRU Insertion Policy (LIP)* which places *all* the incoming lines in the LRU position. These lines are promoted from the LRU position to the MRU position only if they get referenced while in the LRU position. LIP prevents thrashing for workloads whose working set is greater than the cache size and obtains near-optimal hit rates for workloads that have a cyclic access pattern. LIP can easily be implemented by avoiding the recency update at insertion.

LIP may retain the lines in the non-LRU position of the recency stack even if they cease to contribute to cache hits. Since LIP does not have an aging mechanism, it may not respond to changes in the working set of a given application. We propose the *Bimodal Insertion Policy (BIP)*, which is similar to LIP, except that BIP infrequently (with a low probability) places the incoming line in the MRU position. We show that BIP adapts to changes in the working set while retaining the thrashing protection advantages of LIP.

For LRU-friendly workloads that favor the traditional policy of MRU insertion, the changes to the insertion policy are detrimental to cache performance. We propose a *Dynamic Insertion Policy (DIP)* to choose between the traditional LRU policy and BIP depending on which policy incurs fewer misses. DIP requires runtime estimates of misses incurred by each of the competing policies. To implement DIP without requiring significant hardware overhead, we propose *In-Cache Dynamic Set Sampling (IDSS)*. IDSS dedicates a few sets of the cache to each of the two competing policies and uses the policy that performs

31

better on the *dedicated sets* for the remaining *follower sets*. We analyze both analytical as well as empirical bounds for the number of dedicated sets and show that as few as 32 to 64 dedicated sets are sufficient for IDSS to choose the best policy. An implementation of DIP using IDSS requires no extra storage other than a single saturating counter and performs similar to LRU for LRU-friendly workloads.

Insertion policies come into effect only during cache misses, therefore, changes to the insertion policy does not affect the access time of the cache. The proposed changes to the insertion policy are particularly attractive as they do not require *any* changes to the structure of an existing cache design, incur only a negligible amount of logic circuitry, and have a storage overhead of less than two bytes.

4.2   Motivation

Our study is focused on reducing L2 misses by managing the L2 cache efficiently. The access stream visible to the L2 cache has filtered temporal locality due to the hits in the first-level cache. The loss of temporal locality causes a significant percentage of L2 cache lines to remain unused. We refer to cache lines that are not referenced between insertion and eviction as *zero reuse lines*. Figure 4.1 shows that for the baseline 1MB 16-way LRU-managed L2 cache *more than half the lines installed in the cache are never reused before getting evicted*. Thus, the traditional LRU policy results in inefficient use of cache space as most of the lines installed occupy cache space without contributing to cache hits.

Zero reuse lines occur because of two reasons. First, the line has no temporal locality which means that the line is never re-referenced. It is not beneficial to insert such lines in the cache. Second, the line is re-referenced at a distance greater than the cache size, which causes the LRU policy to evict the line before it gets reused. Several studies have investigated bypassing [48][24][31][81] and early eviction [85][82] of lines with poor

32
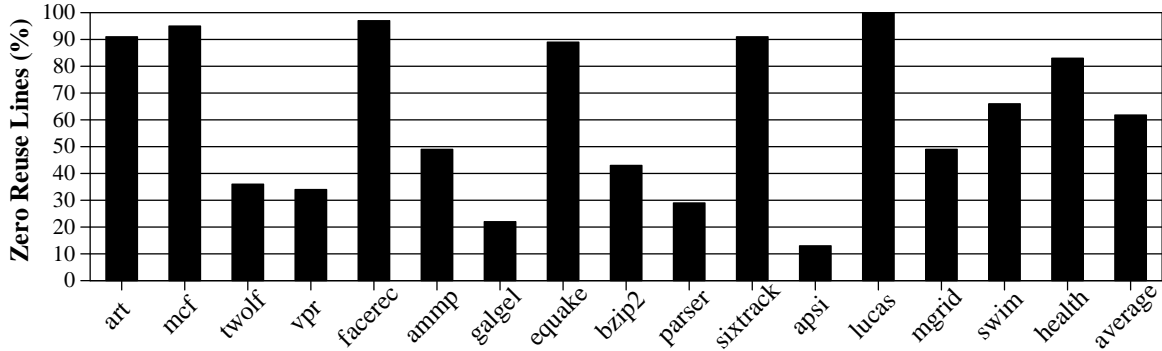
Figure 4.1: Percentage of Zero Reuse Lines for the Baseline 1MB 16-way L2 cache

temporal locality. However, temporal locality exploited by the cache is a function of both the replacement policy and the size of the working set relative to the available cache size. For example, if a workload frequently reuses a working set of 2 MB, and the available cache size is 1MB, then the LRU policy will cause all the installed lines to have poor temporal locality. In such a case, bypassing or early evicting all the lines in the working set will not improve cache performance. The optimal policy in such cases is to retain some fraction of the working set long-enough so that at least that fraction of the working set provides cache hits. However, the traditional LRU policy offers no protection for retaining the cache lines longer than the cache capacity.

For workloads with working set greater than the cache size, cache performance can be significantly improved if the cache can retain some fraction of the working set. To achieve this, we separate the replacement policy into two parts: *victim selection policy* and *insertion policy*. The victim selection policy decides which line gets evicted for storing an incoming line. The insertion policy decides where in the replacement list is the incoming line placed. We propose simple changes to the insertion policy that significantly improves cache performance of memory-intensive workloads while requiring negligible overhead.

33

4.3 Static Insertion Policies

The traditional LRU replacement policy inserts all the incoming lines in the MRU position. Inserting the line in the MRU position gives the line a chance to obtain a hit while it traverses all the way from the MRU position to the LRU position. While this may be a good strategy for workloads whose working set is smaller than the available cache size or for workloads that have a high temporal locality, such an insertion policy causes thrashing for memory-intensive workloads that have a working set greater than the available cache size. When the working set is greater than the available cache size, cache performance can be improved by retaining some fraction of the working set long enough that at least that fraction of the working set results in a cache hit.

For such workloads, we propose the *LRU Insertion Policy (LIP)*, which places *all* incoming lines in the LRU position. These lines are promoted from the LRU position to the MRU position only if they are reused while in the LRU position. LIP prevents thrashing for workloads that reuse a working set greater than the available cache size. To our knowledge this is the first study to investigate the insertion of *demand* lines in the LRU position. Earlier studies [21] have proposed to insert prefetched lines in the LRU position to reduce the pollution caused by inaccurate prefetching. However, they were targeting the problem of extraneous references generated by the prefetcher while our study is targeted towards the fundamental locality problem in memory reference streams. In their model, demand references were still placed in the MRU position leaving the cache vulnerable to thrashing under LRU replacement.

LIP may retain the lines in the non-LRU position of the recency stack even if they cease to be re-referenced. Since LIP does not have an aging mechanism, it may not respond to changes in the working set of the given application. We propose the *Bimodal Insertion Policy (BIP)* which is similar to LIP, except that it infrequently (with a low probability) places some incoming lines into the MRU position. BIP is regulated by a parameter, *over-*

ride probability (ϵ), which controls the percentage of incoming lines that are placed in the MRU position. Both traditional LRU policy and LIP can be viewed as a special case of BIP with $\epsilon = 1$ and $\epsilon = 0$ respectively. In Section 4.3.1 we show that for small values of ϵ, BIP can adapt to changes in working set while retaining the thrashing protection of LIP.

4.3.1 Analysis with Cyclic Reference Model

To analyze workloads that cause thrashing with the LRU policy, we use a theoretical model of cyclic references. A similar model has been used earlier by McFarling [48] for modeling conflict misses in a direct-mapped instruction cache. Let a_i denote the address of a cache line. Let $(a_1 \cdots a_T)$ denote a temporal sequence of references a_1, a_2, ..., a_T. A temporal sequence that repeats for N times is represented as $(a_1 \cdots a_T)^N$.

Let there be an access pattern in which $(a_1 \cdots a_T)^N$ is followed by $(b_1 \cdots b_T)^N$. We analyze the behavior of this pattern for a fully associative cache that contains space for storing $K(K < T)$ lines. We assume that the parameter ϵ in BIP is small, and that both sequences in the access pattern repeat many times ($N >> T$ and $N >> K/\epsilon$). Table 3 compares the hit-rate of LRU, OPT, LIP, and BIP for this access pattern.

Table 4.1: Hit Rate for LRU, OPT, LIP, and BIP under Cyclic Reference Model

Policy	$(a_1 \cdots a_T)^N$	$(b_1 \cdots b_T)^N$
LRU	0	0
OPT	$(K-1)/(T-1)$	$(K-1)/(T-1)$
LIP	$(K-1)/T$	0
BIP	$(K-1-\epsilon \cdot [T-K])/T \approx (K-1)/T$	$\approx (K-1-\epsilon \cdot [T-K])/T \approx (K-1)/T$

As the cache size is less than T, LRU causes thrashing and results in zero hits for both sequences. The optimal policy is to retain any K out of the T lines of the cyclic reference so that those K lines receive hits. For both sequence, OPT obtains a hit rate

35

$(K-1)/(T-1)$ [72]. After the cache is warmed up, LIP evicts the most recently installed line and achieves a hit-rate of $(K-1)/T$ for the first sequence. However, LIP never allows any element of the second sequence to enter the non-LRU position of the cache, thus, causing zero hits for the second sequence.

In each iteration, BIP inserts approximately $\epsilon \cdot (T - K)$ lines in the MRU position which means a hit-rate of $(K-1-\epsilon \cdot [T-K])/T$. As the value of ϵ is small, BIP obtains a hit-rate of approximately $(K-1)/T$, which is similar to the hit-rate of LIP. However, BIP probabilistically allows the lines of any sequence to enter the MRU position. Therefore, when the sequence changes from the first to the second, all the lines in the cache belong to the second sequence after K/ϵ misses. For large N, the transition time from the first sequence to the second sequence is small, and the hit-rate of BIP is approximately equal to $(K-1)/T$. Thus, for small values of ϵ, BIP can respond to changes in the working set while retaining the thrashing protection of LIP.

4.3.2   Case Studies of Memory-Intensive Thrashing Workloads

In addition to the SPEC benchmarks, we also used the health benchmark from the olden suite for evaluations in this chapter. The working set of the health benchmark increases with time, which results in thrashing with the LRU policy for the later parts of program execution. We ran the health benchmark to completion. The MPKI for the baseline L2 cache for health is 61.7 with 0.7% of the misses as compulsory misses.

We analyze LIP and BIP in detail using three memory-intensive benchmarks: mcf, art, and health. These benchmarks incur the highest MPKI for the SPEC INT, SPEC FP, and olden benchmark suite respectively. The LRU policy results in thrashing as the working set of these benchmarks is greater than the baseline 1MB cache. For all experiments in this section a value of $\epsilon = 1/32$ is used.

4.3.2.1 The mcf benchmark:

```
while (arcin)                Causes 84% of all L2 misses
{                             (28% by each instruction)
    tail = arcin->tail;

    if(tail->time + arcin->org_cost > latest)
    {
        arcin= (arc_t *) tail->mark;
        continue;
    }
    ...
    arcin= (arc_t *) tail->mark
}
```
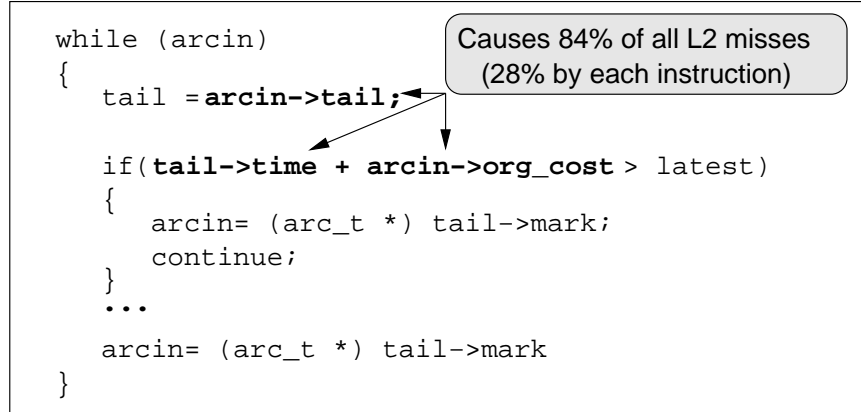
Figure 4.2: Miss-causing instructions from the mcf benchmark

Figure 4.2 shows the code structure from the `implicit.c` file of the mcf benchmark with the three load instructions that are responsible for 84% of the total L2 misses for the baseline cache. The kernel of mcf can be approximated as linked-list traversals of a data structure whose size is approximately 3.5MB. Figure 4.3 shows the MPKI for mcf when the cache size is varied under the LRU policy. The MPKI reduces only marginally till 3.5MB and then the first *"knee"* of the MPKI curve occurs. LRU results in thrashing for the baseline 1MB cache and almost all the inserted lines are evicted before they can be reused. Both LIP and BIP retain around 1MB out of the 3.5MB working set resulting in hits for at least that fraction of the working set. For the baseline 1MB cache, LRU incurs an MPKI of 136, both LIP and BIP incur an MPKI of 115 (17% reduction over LRU), and OPT incurs an MPKI of 101 (26% reduction over LRU). Thus, both LIP and BIP bridge two-thirds of the gap between LRU and OPT without requiring extra storage.

37
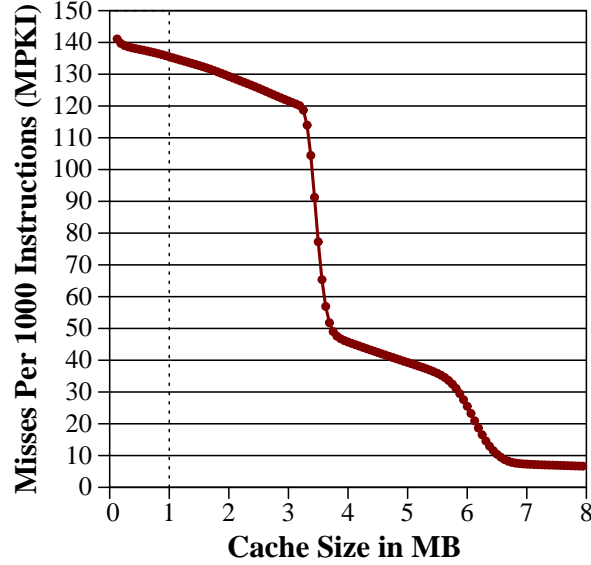
Figure 4.3: MPKI vs. cache size for mcf

4.3.2.2 The art benchmark:

Figure 4.4 shows the code snippet from the `scanner.c` file of the art benchmark containing the two load instructions that are responsible for 80% of all the misses for the baseline cache. The first load instruction traverses an array of type `f1_layer`. The class of `f1_layer` defines it as a `neuron` containing seven elements of type `double` and one element of type pointer to `double`. Thus, the size of each each object of type `f1_layer` is 64B. For ref-1 input set, `numf1s=100000`, therefore, the total size of the array of `f1_layer` is $64B * 10K = 640KB$. The second load instruction traverses a two dimensional array of type `double`. The total size of this array is $8B * 11 * 10K = 880KB$. Thus, the size of the working set of the kernel is approximately 1.5MB.

Figure 4.5 shows the MPKI of art for varying cache size under LRU replacement. LRU is oblivious to the *"knee"* around 1.5MB and causes thrashing for the baseline 1MB cache. Both LIP and BIP prevent thrashing by retaining a significant fraction of the working set in the cache. For the baseline 1MB cache, LRU incurs an MPKI of 38.7, LIP incurs an

38

```
numf1s = lwidth*lheight;  // = 100*100 for ref input set
numf2s = numObjects+1;    // = 10+1 for ref input set
 ...
for (tj=spot;tj<numf2s;tj++)
{
   Y[tj].y = 0;              ┌─────────────────────────────┐
                             │ Causes 41% of all L2 misses │
   if( !Y[tj].reset )        └─────────────────────────────┘
   for (ti=0;ti<numf1s;ti++)           │
   Y[tj].y += f1_layer[ti].P * bus[ti][tj}
                           ┌─────────────────────────────┐
}                          │ Causes 39% of all L2 misses │
                           └─────────────────────────────┘
```
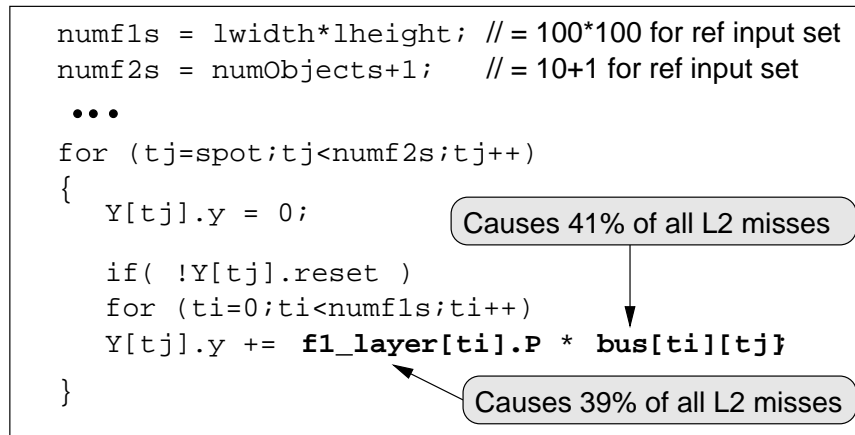
Figure 4.4: Miss-causing instructions from the art benchmark
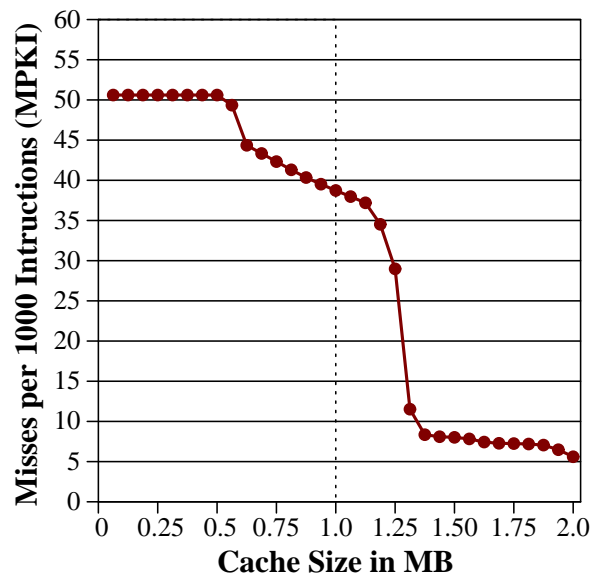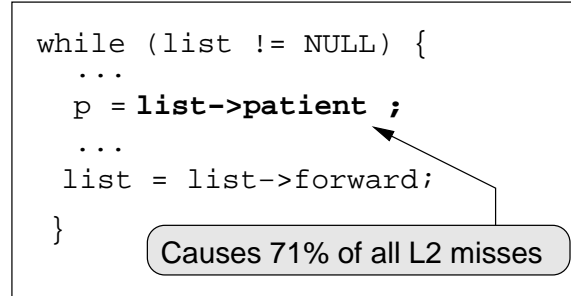
Figure 4.5: MPKI vs. cache size for art

MPKI of 23.6 (39% reduction over LRU), BIP incurs an MPKI of 18 (54% reduction over LRU), and OPT incurs an MPKI of 12.8 (67% reduction over LRU). Both LIP and BIP are closer to OPT. The adaptation in BIP results in much lower MPKI with BIP than with LIP.

39

4.3.2.3   The health benchmark:

```
while (list != NULL) {
  ...
  p = list->patient ;
  ...
  list = list->forward;
}
                  Causes 71% of all L2 misses
```

Figure 4.6: Miss-causing instruction from the health benchmark

Figure 4.6 shows a code snippet from the `health.c` file. It contains the pointer de-referencing load instruction that is responsible for more than 70% of the misses for the baseline cache. The health benchmark can be approximated as a micro kernel that performs linked list traversals with frequent insertions and deletions. The size of the linked-list data structure increases dynamically with program execution. Thus, the memory reference stream can be approximated as a cyclic reference sequence for which the period increases with time. To show the dynamic change in the size of the working set, we split the benchmark execution into four parts (of approximately 50M instructions each). Figure 4.7 shows the MPKI of each of the four phases of execution of health as the cache size is varied under the LRU policy. During the first phase, the size of the working set is less than the baseline 1MB cache so the LRU policy works well. However, in the other three phases, the size of the working set is greater than 1MB, which causes thrashing with LRU. For the full execution of health, LRU incurs an MPKI of 61.7, LIP incurs an MPKI of 38 (38.5% reduction over LRU), BIP incurs an MPKI of 39.5 (36% reduction over LRU), and OPT incurs an MPKI of 34 (45% reduction over LRU).
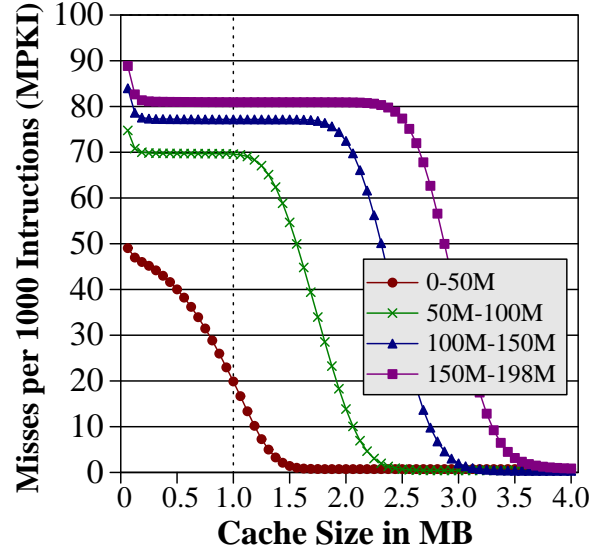
Figure 4.7: MPKI vs. cache size for health

4.3.3 Case Study of a Memory-Intensive LRU-Friendly Workload

For workloads that cause thrashing with LRU, both LIP and BIP reduce cache misses significantly. However, some workloads inherently favor the traditional policy of inserting the incoming line at the MRU position. In such cases, changing the insertion policy can hurt cache performance. An example of such a workload is the swim benchmark from the SPEC FP suite. Swim performs matrix multiplies in its kernel. The first *"knee"* of the matrix multiplication occurs at $\frac{1}{2}$ MB while the second *"knee"* occurs at a cache size greater than 64 MB. Figure 4.8 shows the MPKI for swim as the cache size is increased from $\frac{1}{8}$ MB to 64 MB under LRU replacement. There is a huge reduction in MPKI as the cache size is increased from $\frac{1}{8}$ MB to $\frac{1}{2}$ MB. However, subsequent increase in cache size till 64 MB does not have a significant impact on MPKI. For the baseline cache, the MPKI with both LRU and OPT are similar indicating that there is no scope for reducing misses over the LRU policy. In fact, changes to the insertion policy can only reduce the hits obtained

41

from the middle of the LRU stack for the baseline 1 MB cache. Therefore, both LIP and BIP increase MPKI significantly over the LRU policy. For the baseline cache, LRU incurs an MPKI of 23, LIP incurs an MPKI of 46.5, BIP incurs an MPKI of 44.3, and OPT incurs an MPKI of 22.8.
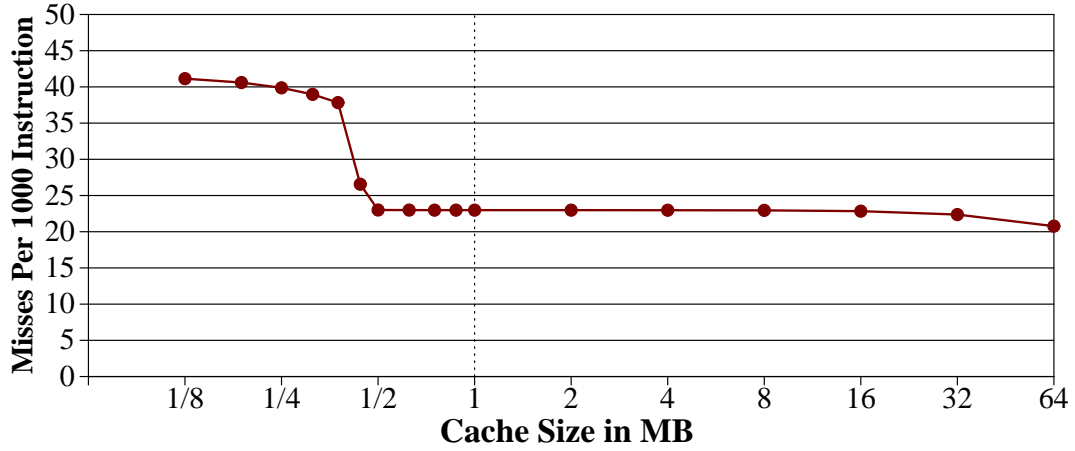
Figure 4.8: MPKI vs. cache size for swim (Note: horizontal axis is in log scale)

4.3.4 Results

Figure 4.9 shows the reduction in MPKI with the two proposed insertion policies, LIP and BIP, over the baseline LRU replacement policy. For BIP, we show results for $\epsilon = 1/64$, $\epsilon = 1/32$, and $\epsilon = 1/16$ which mean every 64th, 32nd, or 16th miss is inserted in the MRU position respectively.[1]

---

[1]In our studies, we restrict the value of ϵ to 1/power-of-two. To implement BIP, a pseudo-random number generator is required. If there is no pseudo-random number available then an n-bit free running counter can be used to implement a 1-out-of-2^n policy ($n = log_2(1/\epsilon)$). The n-bit counter is incremented on every cache miss. BIP inserts the incoming line in the MRU position only if the value of this n-bit counter is zero. We experimented with both gnu c rand function as well as the 1-out-of-2^n policy and obtained similar results.
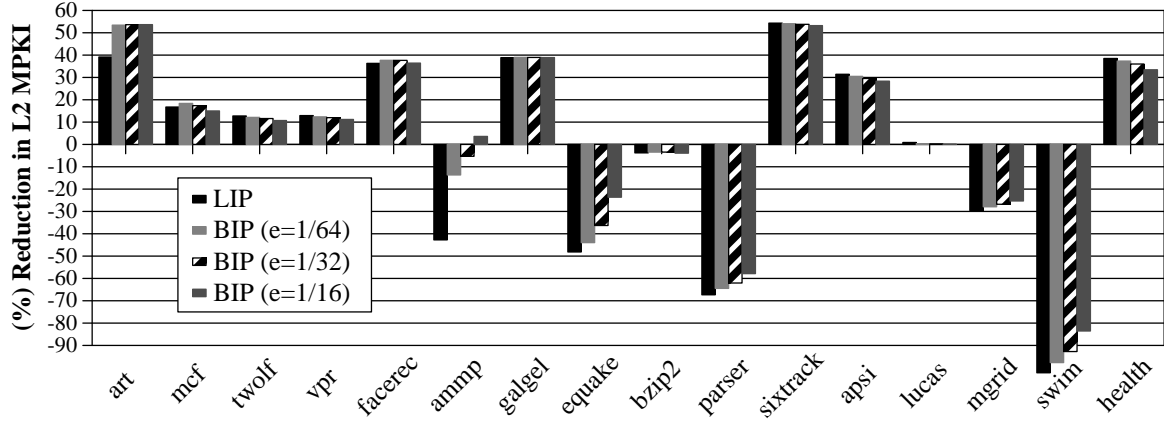
Figure 4.9: Comparison of Static Insertion Policies

The thrashing protection of LIP and BIP reduces MPKI by 10% or more for nine out of the sixteen benchmarks. BIP has better MPKI reduction than LIP for art and ammp because it can adapt to changes in the working set of the application. For most applications that benefit from BIP, the amount of benefit is not sensitive to the value of ϵ. For benchmarks equake, parser, bzip2 and swim both LIP and BIP increase the MPKI considerably. This occurs because these workloads either have an LRU friendly access pattern, or the knee of the MPKI curve is less than the cache size and there is no significant benefit from increasing the cache size. For the insertion policy to be useful for a wide variety of workloads, we need a mechanism that can select between the traditional LRU policy and BIP depending on which incurs fewer misses. The next section describes a cost-effective run-time mechanism to choose between LRU and BIP. For the remainder of the chapter we use a value of $\epsilon = 1/32$ for all experiments with BIP.

4.4   Dynamic Insertion Policy

For some applications BIP has fewer misses than LRU and for some LRU has fewer misses than BIP. We want a mechanism that can choose the insertion policy that has the fewest misses for the application. We propose a mechanism that dynamically estimates the number of misses incurred by the two competing insertion policies and selects the policy that incurs the fewest misses. We call this mechanism *Dynamic Insertion Policy (DIP)*. A straightforward method of implementing DIP is to implement both LRU and BIP in two extra tag directories (data lines are not required to estimate the misses incurred by an insertion policy) and keep track of which of the two policies is doing better. The main tag directory of the cache can then use the policy that incurs the fewest misses. Since this implementation of DIP gathers information globally for all the sets, and enforces a uniform policy for all the sets, we call it *DIP-Global*.

4.4.1   The DIP-Global Mechanism

Figure 4.10(a) demonstrates the working of DIP-Global for a cache containing sixteen sets. Let MTD be the main tag directory of the cache. The two competing policies, LRU and BIP, are each implemented in a separate Auxiliary Tag Directory (ATD). ATD-LRU uses the traditional LRU policy and ATD-BIP uses BIP. Both ATD-LRU and ATD-BIP have the same associativity as the MTD. The access stream visible to MTD is also applied to both ATD-LRU and ATD-BIP. A saturating counter, which we call *Policy Selector (PSEL)*, keeps track of which of the two ATDs incurs fewer misses. All operations on PSEL are done using saturating arithmetic. A miss in ATD-LRU increments PSEL and a miss in ATD-BIP decrements PSEL. The Most Significant Bit (MSB) of PSEL is thus an indicator of which of the two policies incurs fewer misses. If MSB of PSEL is 1, MTD uses BIP, otherwise MTD uses LRU.

44

ATD–LRU   MTD   ATD–BIP

Set 0 through Set 15

Miss in ATD–LRU   PSEL   Miss in ATD–BIP

Decides Policy
for All Sets in MTD

(a)

MTD

Set 0 through Set 15

LEGEND

Sets dedicated to LRU

Sets dedicated to BIP

Follower Sets.
Policy decided by PSEL

Decides Policy for
Follower Sets

Miss in a Set
Dedicated to LRU   PSEL   Miss in a Set
Dedicated to BIP

(b)
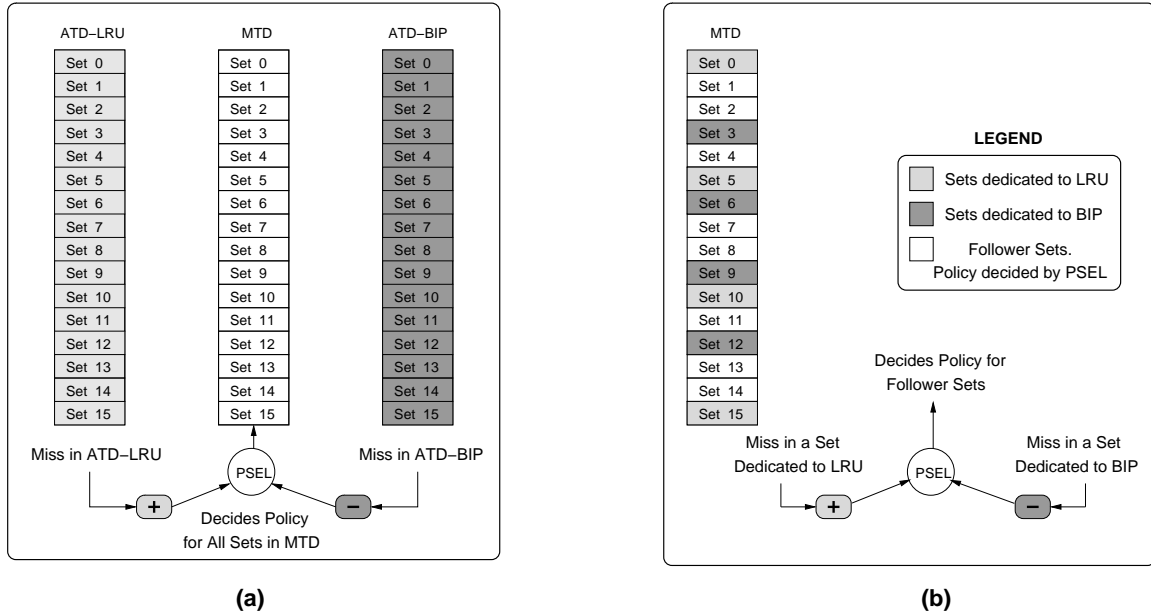
Figure 4.10: Implementations of Dynamic Insertion Policy: (a) DIP-Global (b) DIP-IDSS

4.4.2   The DIP-IDSS Mechanism

The DIP-Global mechanism requires a substantial hardware overhead of two ex-
tra tag directories. The hardware overhead of comparing two policies can be significantly
reduced by using *Dynamic Set Sampling (DSS)*. The key insight in DSS is that the cache
behavior can be approximated with a high probability by sampling few sets in the cache.
Thus, DSS can significantly reduce the number of ATD entries in DIP-Global from thou-
sand(s) of sets to about 32 sets.

Although DSS significantly reduces the storage required in implementing the ATD
(to around 2kB), it still requires building the separate ATD structure. Thus, implementing
DIP will still incur the design, verification, and testing overhead of building the separate
ATD structure. We propose *In-cache Dynamic Set Sampling (IDSS)*, which obviates the
need for a separate ATD structure. IDSS dedicates few sets of the cache to each of the two

45

competing policies. The policy that incurs fewer misses on the *dedicated sets* is used for the remaining *follower sets*. An implementation of DIP that uses IDSS is called *DIP-IDSS*.

Figure 4.10(b) demonstrates the working of DIP-IDSS on a cache containing sixteen sets. Sets 0, 5, 10, and 15 are dedicated to the LRU policy, and Sets 3, 6, 9, and 12 are dedicated to the BIP policy. The remaining sets are follower sets. A miss incurred in the sets dedicated to LRU increments PSEL, whereas, a miss incurred in the sets dedicated to BIP decrements PSEL. If the MSB of PSEL is 0, the follower sets use the LRU policy; otherwise the follower sets use BIP. Note that IDSS does not require any separate storage structure other than a single saturating counter.

DIP-IDSS compares the number of misses across different sets for two competing policies. However, the number of misses incurred by even a single policy varies across different sets in the cache. A natural question is how does the per-set variation in misses of the component policies affect the dynamic selection of IDSS? Also, how many dedicated sets are required for DIP-IDSS to approximate DIP-Global with a high probability? In Section 4.4.3, we derive analytical bounds[2] for DIP-IDSS as a function of both the number of dedicated sets and the per-set variation in misses of the component policies. In Section 4.4.5 we compare the misses incurred by DIP-IDSS and DIP-Global.

4.4.3 Analytical Model for IDSS

Let there be N sets in the cache. Let IDSS be used to choose between two policies $P1$ and $P2$. When policy P1 is implemented on all the sets in the cache, the average

---

[2]Chapter 3 used a Bernoulli model to derive the bounds for DSS. However, because there was a separate ATD, the model was comparing the two policies by implementing both policies for a few sampled set. That analytical model does not consider the per-set variation in misses incurred by the component policies. However, in the present case, IDSS compares the component policies by implementing them on different sets in the cache. Therefore, the analytical model of IDSS must take into account the per-set variation in misses incurred by the component policies. Therefore, the bounds derived in Chapter 3 are not applicable to IDSS.

number of misses per set is μ_1 with standard deviation σ_1. Similarly, when policy P2 is implemented on all the sets in the cache, the average number of misses per set is μ_2 with standard deviation σ_2. Let Δ denote the difference in average misses $|\mu_1 - \mu_2|$ and σ denote the combined standard deviation $\sqrt{\sigma_1^2 + \sigma_2^2}$.

Let n sets be randomly selected from the cache to estimate the misses with policy P1 and another group of n sets be randomly selected to estimate the misses with policy P2. We assume that the number of dedicated sets n is sufficiently large such that by the *central limit theorem* [68] the sampling distribution can be approximated as a Gaussian distribution. We also assume that n is sufficiently small compared to the total number of sets in the cache (N) so that removing the n sets does not significantly change the mean and standard deviation of the remaining $(N - n)$ sets. To derive the bounds for IDSS we use the following well-established results [68] for sampling distribution: *If the distribution of two independent random variables have the means μ_a and μ_b and the standard deviation σ_a and σ_b, then the distribution of their sum (or difference) has the mean $\mu_a + \mu_b$ (or $\mu_a - \mu_b$) and the standard deviation $\sqrt{\sigma_a^2 + \sigma_b^2}$.*

Let $sum1$ be the total number of misses for the n sets dedicated to policy P1. Then, by central limit theorem, $sum1$ can be approximated as a Gaussian random variable with mean μ_{sum1} and standard deviation σ_{sum1}, given by:

$$\mu_{sum1} = n \cdot \mu_1 \tag{4.1}$$

$$\sigma_{sum1} = \sqrt{\sum \sigma_1^2} = \sqrt{n} \cdot \sigma_1, \tag{4.2}$$

Similarly, let $sum2$ be the total number of misses for the n sets dedicated to policy P2. Then, $sum2$ can also be approximated as a Gaussian random variable with mean μ_{sum2} and standard deviation σ_{sum2}, given by:

$$\mu_{sum2} = n \cdot \mu_2 \tag{4.3}$$

47

$$\sigma_{sum2} = \sqrt{\sum \sigma_2^2} = \sqrt{n} \cdot \sigma_2, \tag{4.4}$$

PSEL tracks the difference in $sum1$ and $sum2$ and selects the policy that has fewer misses on the sampled sets. Let θ denote the difference in the value of the two sums, i.e. $\theta = sum1 - sum2$. Because $sum1$ and $sum2$ are Gaussian random variables, θ is also a Gaussian random variable with mean μ_θ and standard deviation σ_θ given by:

$$\mu_\theta = \mu_{sum1} - \mu_{sum2} = n \cdot \mu_1 - n \cdot \mu_2 = n \cdot (\mu_1 - \mu_2) \tag{4.5}$$

$$\sigma_\theta = \sqrt{\sigma_{sum1}^2 + \sigma_{sum2}^2} = \sqrt{n \cdot \sigma_1^2 + n \cdot \sigma_2^2} = \sqrt{n} \cdot \sqrt{\sigma_1^2 + \sigma_2^2} = \sqrt{n} \cdot \sigma, where \, \sigma = \sqrt{\sigma_1^2 + \sigma_2^2}$$
$$\tag{4.6}$$

Let policy P2 have fewer misses than policy P1, i.e. $\mu_1 > \mu_2$. Then, for IDSS to select the best policy, $\theta > 0$. If $P(Best)$ is the probability that IDSS selects the best policy, then $P(Best)$ can be written as:

$$P(Best) = P(\theta > 0) = P(\frac{(\theta - \mu_\theta)}{\sigma_\theta} > \frac{(0 - \mu_\theta)}{\sigma_\theta}) \tag{4.7}$$

$$P(Best) = P(Z > \frac{-n \cdot (\mu_1 - \mu_2)}{\sqrt{n} \cdot \sigma}), where \, Z = \frac{(\theta - \mu_\theta)}{\sigma_\theta} \, is \, the \, Standard \, Gaussian \, Variable$$
$$\tag{4.8}$$

$$P(Best) = 1 - P(Z > \frac{n \cdot (\mu_1 - \mu_2)}{\sqrt{n} \cdot \sigma}), \, as \, P(Z > -z) = 1 - P(Z > z) \tag{4.9}$$

$$P(Best) = 1 - P(Z > \sqrt{n} \cdot \frac{\Delta}{\sigma}), \quad where \, \Delta = |\mu_1 - \mu_2| \, , given \, \mu_1 > \mu_2 \tag{4.10}$$

$$P(Best) = 1 - P(Z > \sqrt{n} \cdot r), \quad where \quad r = \frac{\Delta}{\sigma} \qquad (4.11)$$
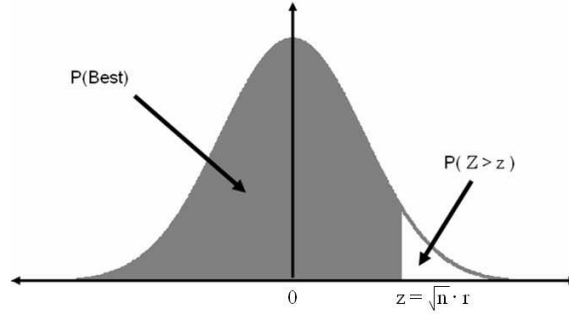
Figure 4.11: P(Best) from Gaussian Curve

Z is the standard Gaussian variable for which the value of $P(Z > z)$ can be obtained using standard statistical tables (see Figure 4.11). Equation 11 can be used to compute P(Best) for any two policies. For example, if for policy P1, $\mu_1 = 100$ and $\sigma_1 = 12$ and for policy P2, $\mu_2 = 94$ and $\sigma_2 = 16$. Then, $\Delta = 6$, $\sigma = 20$ and $r = 0.3$. For n=32, $P(Best) = 1 - P(Z > \sqrt{32} \cdot 0.3) = 1 - P(Z > 1.7) = 96\%$.

Figure 4.12 shows the variation in P(Best) as the number of dedicated sets is changed for different values of the r metric. The r metric is a function of workload, cache organization, and the relative difference between the two policies. For most of the benchmarks studied, the r-metric for the two policies LRU and BIP is more than 0.2 indicating that 32-64 sampled sets are sufficient for IDSS to select the best policy with a high probability. Thus, DIP-IDSS can be implemented by dedicating about 32 to 64 sets to each of the two policies, LRU and BIP, and using the winning policy (of the dedicated sets) for the remaining sets.

49
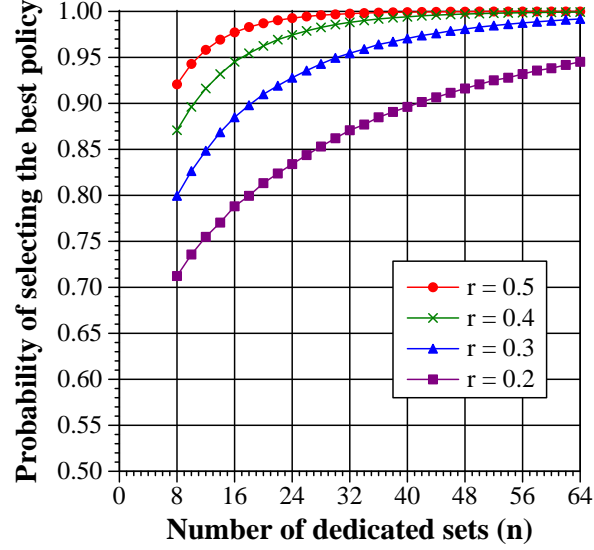
Figure 4.12: Analytical Bounds for IDSS

4.4.4 Dedicated Set Selection Policy

The dedicated set for each of the competing policies can be selected statically at design time or dynamically at runtime. In this section we describe our method of selecting the dedicated sets. Let N be the number of sets in the cache and K be the number of sets dedicated to each policy (in our studies we restrict the number of dedicated sets to powers of 2). We logically divide the cache into K equally-sized regions each containing N/K sets. Each such region is called a *constituency* [63]. One set is dedicated from each constituency to each of the competing policies. Two bits associated with each set can then identify the set as either a follower set or a dedicated set to one of two competing policies.

We employ a dedicated set selection policy that obviates the need for marking the leader set in each constituency on a per-set basis. We call this policy the *complement-select* policy. For a cache with N sets, the set index consists of $log_2(N)$ bits out of which the most significant $log_2(K)$ bits identify the constituency and the remaining $log_2(N/K)$

bits identify the *offset* from the first set in the constituency. The complement-select policy dedicates to LRU all the sets for which the constituency identifying bits are equal to the offset bits. Similarly, it dedicates to BIP all the sets for which the complement of the offset equals the constituency identifying bits. Thus for the baseline cache with 1024 sets, if 32 sets are to be dedicated to both LRU and BIP, then complement-select dedicates Set 0 and every 33rd set to LRU, and Set 31 and every 31st set to BIP. The sets dedicated to LRU can be identified using a five bit comparator for the bits [4:0] to bits [9:5] of the set index. Similarly, the sets dedicated to BIP can be identified using another five bit comparator that compares the complement of bits [4:0] of the set index to bits [9:5] of the set index. Unless stated otherwise, the default implementation of DIP is DIP-IDSS with 32 dedicated sets using the complement-select policy[3] and a 10-bit[4] PSEL counter.

4.4.5 Results

Figure 4.13 shows reduction in MPKI with BIP, DIP-Global, and DIP-IDSS with 32 and 64 dedicated sets. The bar labeled *amean* is the reduction in arithmetic mean MPKI measured over all the sixteen benchmarks. DIP-Global retains the MPKI reduction of BIP while eliminating the significant MPKI increase of BIP on benchmarks equake, parser, mgrid, and swim. With DIP-Global, no benchmark incurs an MPKI increase of more than 2% over LRU. However, DIP-Global requires a significant hardware overhead of about 128kB. DIP-IDSS obviates this hardware overhead while obtaining an MPKI reduction similar to DIP-Global for all benchmarks, except twolf. As the number of dedicated sets increase from 32 to 64, the probability of selecting the best policy increases, therefore DIP-

---

[3]We also experimented with a rand-dynamic policy which randomly dedicates one set from each constituency to each of the two policies LRU and BIP. We invoke rand-dynamic once every 5M retired instructions. The MPKI with both rand-dynamics and complement-select are similar. However, rand-dynamic incurs the hardware overhead of bits for identifying the dedicated sets which are not required for complement-select.

[4]For experiments of DIP-IDSS in which 64 sets are dedicated to each policy, we use a 11-bit PSEL counter.
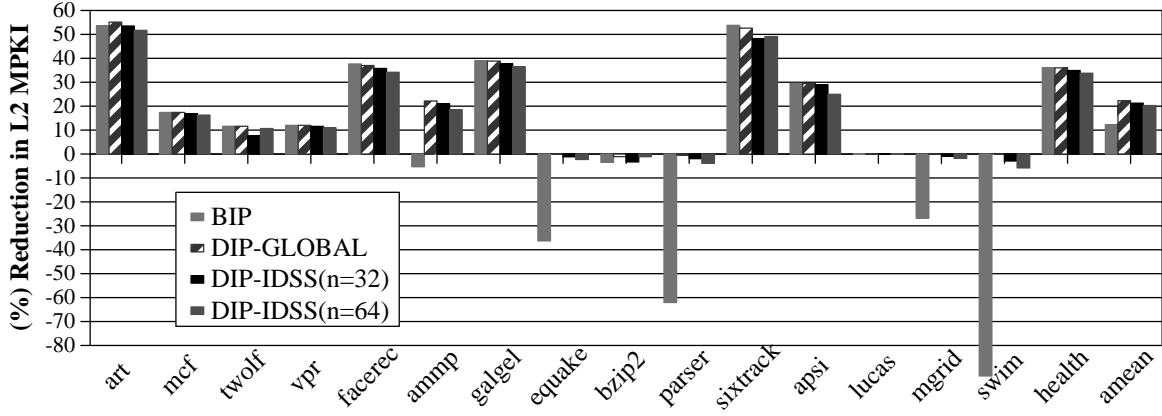
Figure 4.13: Comparison of Dynamic Insertion Policies

IDSS with 64 dedicated sets behaves similar to DIP-Global for twolf. However, having a large number of dedicated sets also means that a higher fraction (n/N) of sets always use BIP, even if BIP increases MPKI. This causes the MPKI of swim to increase by 5% with 64 dedicated sets. For ammp, DIP reduces MPKI by 20% even though BIP increases MPKI. This happens because in one phase LRU has fewer misses and in the other phase BIP has fewer misses. With DIP, the cache uses the policy best suited to each phase and hence a better MPKI than each of the component policies. We discuss the dynamic adaptation of DIP in more detail in Section 4.4.6. On average, DIP-Global reduces average MPKI by 22.3%, DIP-IDSS (with 32 dedicated set) reduces average MPKI by 21.3%, and DIP-IDSS (with 64 dedicated set) reduces average MPKI by 20.3%.

4.4.6 Dynamic Adaptation of DIP to Application Behavior

DIP can adapt to different applications as well as different phases of the same application. DIP uses the PSEL counter to select between the component policies. For a 10-bit PSEL counter, a value of 512 or more indicates that DIP uses BIP, otherwise DIP uses LRU. Figure 4.14 shows the value of the 10-bit PSEL counter over the course of execution

52

for the benchmarks mcf, art, health, swim, and ammp. We sample the value of the PSEL counter once every 1M instructions. The horizontal axis denotes the number of instructions retired (in millions) and the vertical axis represents the value of the PSEL counter.
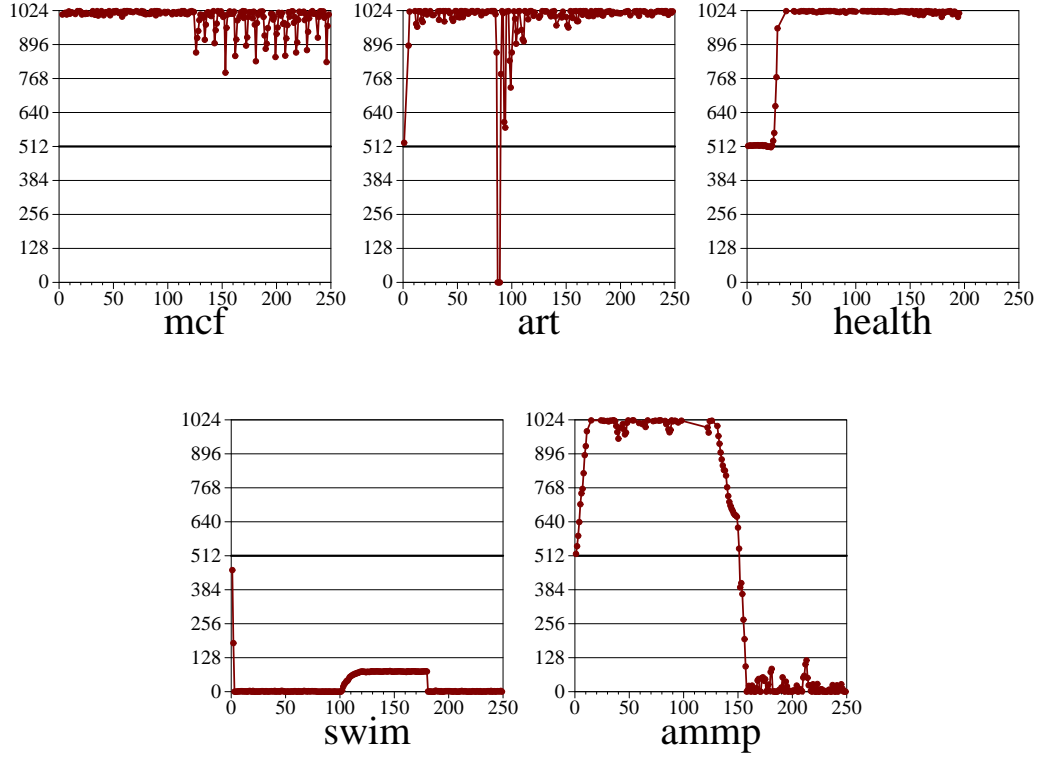
Figure 4.14: PSEL value during benchmark execution (horizontal axis denotes the number of instruction in Millions)

For mcf and art, the DIP mechanism almost always uses BIP. For health, the working set during the initial part of the program execution fits in the baseline cache and either policy works well. However, as the dataset increases during program execution, it exceeds the size of the baseline cache and LRU causes thrashing. As BIP would have fewer misses than LRU, the PSEL value reaches toward positive saturation and DIP selects BIP. For the

LRU friendly benchmark swim, the PSEL value is almost always towards negative saturation, so DIP selects LRU. Ammp has two phases of execution: in the first phase LRU is better and in the second phase BIP is better. With DIP, the policy best suited to each phase is selected; therefore, DIP has better MPKI than either of the component policies standalone.

4.5 Analysis

4.5.1 Varying the Cache Size

We vary the cache size from 1 MB to 8 MB and keep the associativity constant at 16-way. Figure 4.15 shows the MPKI of both LRU and DIP for four cache sizes: 1MB, 2MB, 4MB, and 8MB. The MPKI values are shown relative to the baseline 1MB LRU-managed cache. The bar labeled *avg* represents the arithmetic mean MPKI measured over all the sixteen benchmarks. As mcf has a large value of MPKI, the average without mcf, *avgNomcf*, is also shown.
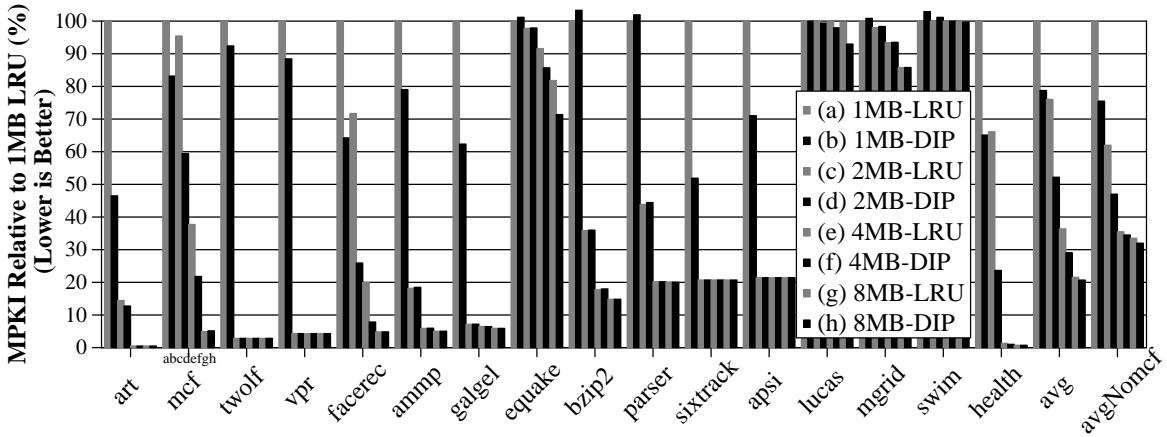
Figure 4.15: Comparison of LRU and DIP for different cache size

54

DIP reduces MPKI more than doubling the size of the baseline 1MB cache for benchmarks mcf, facerec, and health. DIP continues to reduce misses for most benchmarks that benefit from increased capacity. The working set of some benchmarks, e.g. vpr and twolf, fit in a 2MB cache. Therefore, neither LRU nor DIP reduces MPKI when the cache size is increased. Overall, DIP significantly reduces average MPKI over LRU for all cache sizes till 4MB.

4.5.2 Bypassing Instead of Inserting at LRU Position

DIP uses BIP which inserts most of the incoming lines in the LRU position. If such a line is accessed in the LRU position, only then is it updated to the MRU position. Another reasonable design point is to bypass the incoming line instead of inserting it in the LRU position. A DIP policy that employs BIP which bypasses the incoming line when the incoming line is to be placed in the LRU position is called *DIP-Bypass*. Figure 4.16 shows the MPKI reduction of DIP and DIP-Bypass over the baseline LRU policy.
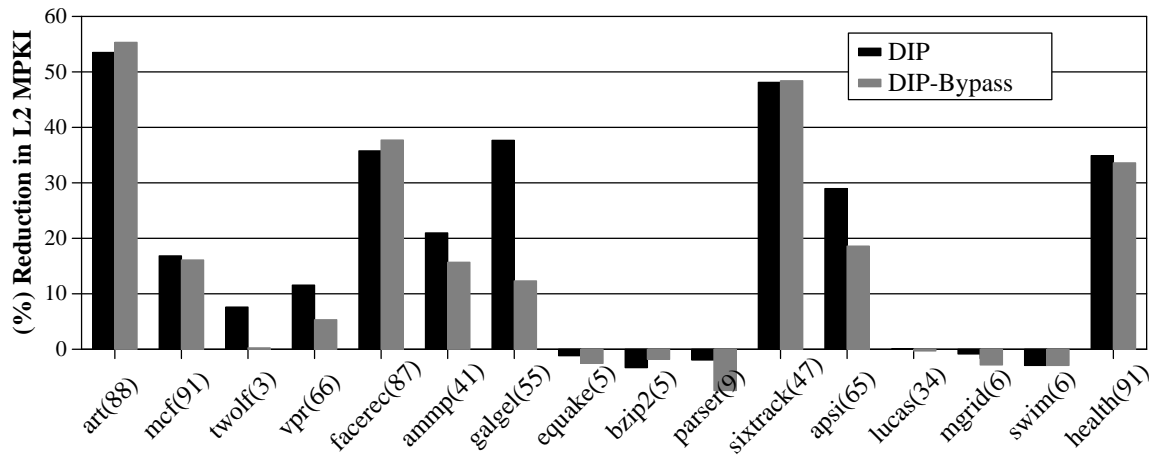
Figure 4.16: Effect of Bypassing on DIP (The number shows the percentage of misses bypassed by DIP-Bypass).

For all benchmarks, except art, facerec, and sixtrack, DIP reduces MPKI more than

DIP-Bypass. This happens because DIP promotes the line installed in the LRU position to the MRU position if the line is reused, thus increasing the useful lines in the non-LRU positions. On the other hand, DIP-Bypass has the advantage of power savings as it avoids the operation of inserting the line in the cache. The percentage of misses that are bypassed by DIP-Bypass are shown in Figure 4.16 by a number associated with each benchmark name. Thus, the proposed insertion policies can be used to reduce misses, cache power or both.

4.5.3   Impact on System Performance

To evaluate the effect of DIP on the overall processor performance, we use an in-house execution-driven simulator based on the Alpha ISA. The relevant parameters of our model are given in Table 5. Figure 6.10 shows the performance improvement measured in instructions per cycle (IPC) between the baseline system and the same system with DIP. The bar labeled *gmean* is the geometric mean of the individual IPC improvements seen by each benchmark. The system with DIP outperforms the baseline by an average of 9.3%. DIP increases the IPC of benchmarks art, mcf, facerec, and health by more than 15%.
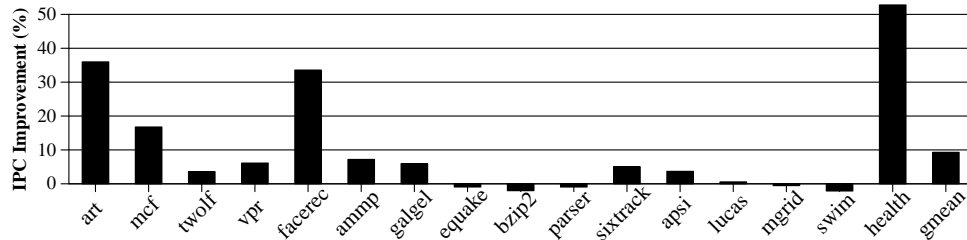
Figure 4.17: IPC improvement with DIP

4.5.4   Estimation of Hardware Overhead and Design Changes

The proposed insertion policies (LIP, BIP, and DIP) require negligible hardware overhead and design changes. LIP inserts all incoming lines in the LRU position, which

56

can easily be implemented by not performing the update to the MRU position that occurs on cache insertion.[5] BIP is similar to LIP, except that it infrequently inserts an incoming line into the MRU position. To control the rate of MRU insertion in BIP, we use a five-bit counter (BIPCTR). BIPCTR is incremented on every cache miss. BIP inserts the incoming line in the MRU position only if the BIPCTR is zero. Thus, BIP incurs a storage overhead of 5 bits. DIP requires storage for the 10-bit saturating counter (PSEL). The complement-select policy avoids extra storage for identifying the dedicated sets.
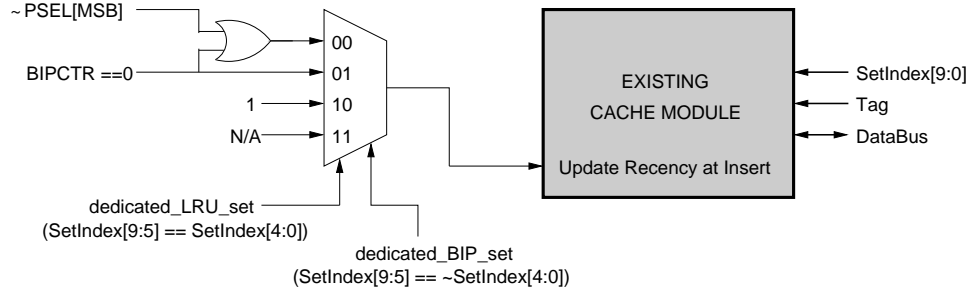
Figure 4.18: Hardware changes for implementing DIP

Figure 4.18 shows the design changes incurred in implementing DIP. The implementation requires a total storage overhead of 15 bits (5-bit BIPCTR + 10-bit PSEL) and negligible logic overhead. A particularly attractive aspect of DIP is that it does not require extra bits in the tag-store entry, thus avoiding changes to the existing structure of the cache. The absence of extra structures also means that DIP does not incur power and complexity overheads. As DIP does not add any logic to the cache access path, the access time of the cache remains unaffected.

---

[5]LIP, BIP, and DIP do not rely on true LRU which makes them amenable to the LRU approximations widely used in current on-chip caches.

4.5.5 Interaction with Prefetching

We simulate a PC-based stride prefetcher for the baseline machine to analyze the impact of prefetching on the proposed DIP mechanism. Figure 4.19 shows the reduction in MPKI for the baseline machine with no prefetching for three configurations. First, the DIP mechanism without prefetching. Second, the baseline LRU policy with prefetching enabled. Finally, the DIP mechanism with prefetching enabled. To reduce the pollution caused by prefetching, we always insert the prefetches in the LRU position as suggested by Lin et al. [21]. Prefetches are promoted from the LRU position to the MRU position on a hit.

For benchmarks such as art and mcf, prefetching reduces misses for the baseline machine. However, when DIP and prefetching are combined, the reduction is more than either scheme standalone. For benchmarks such as equake, bzip2 and parser, DIP alone does not affect the number of misses while prefetching can reduce the number of misses significantly. When DIP and prefetching are combined in such scenarios, DIP retains the effectiveness of prefetching. Thus DIP and prefetching can both be used to reduce misses.

4.6 Related Work

This section summarizes the work that most closely relates to the techniques proposed in this chapter, distinguishing our work where appropriate.

4.6.1 Alternative Cache Replacement Policies

The problem of thrashing can be mitigated with replacement schemes that are resistant to thrashing. If the working set of an application is only slightly greater than the available cache size, then even a naive scheme such as random replacement can have fewer misses than LRU. However, the effectiveness of random replacement at reducing misses
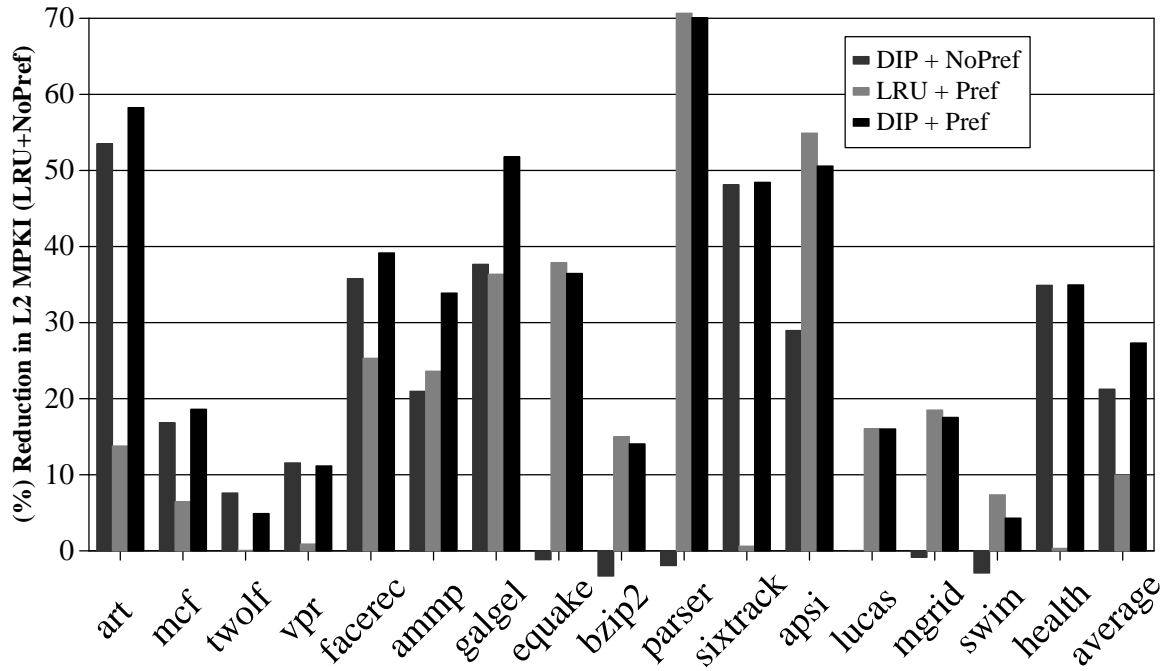
Figure 4.19: Interaction of Insertion Policy with Prefetching

significantly reduces as the size of the working set increases. For the baseline cache random replacement reduces MPKI for the thrashing workloads: art by 34%, mcf by 1.6%, facerec by 14.4%, and health by 16.9%, whereas, DIP reduces MPKI for art by 54%, mcf by 17%, facerec by 36% and health by 35%. Several proposals [67][42][54][25][64] have looked at including frequency (reuse count) information for improving cache replacement.

4.6.2 Related Work in Hybrid Replacement

For workloads that cause thrashing with LRU, both random and frequency-based replacement schemes have fewer misses than LRU. However, these schemes significantly increase the misses for LRU-friendly workloads. Recent studies have investigated hybrid replacement schemes that dynamically select from two or more competing replacement policies. Examples of hybrid replacement schemes include *Sampling-Based Adaptive Re-*

59

placement (SBAR) [63] and *Adaptive Cache (AC)* [77]. The problem with hybrid replacement is that it requires tracking the replacement information for each of the competing policies. For example, if the two policies are LRU and LFU (Least Frequently Used), then each tag-entry in the baseline cache needs to be appended with frequency counters ($\geq$ 5-bits each) which must be updated on each access. Also, the dynamic selection requires extra structures (2kB for SBAR and 34kB for AC) which consume hardware and power. DIP outperforms the best performing hybrid-replacement (See Table 4.2) while obviating the design changes, hardware overhead, power overhead, and complexity of hybrid replacement. In fact, DIP bridges two-third of the gap between LRU and OPT while requiring less than two bytes of extra hardware.

Table 4.2: Comparison of Replacement Policies

Policy	%Reduction in MPKI over LRU	Hardware Overhead
SBAR (LRU+Rand)	8.9	2 kB
AC (LRU+Rand)	9.2	34 kB
SBAR (LRU+LFU)	14.7	12 kB
AC (LRU+LFU)	15.8	44 kB
DIP	**21.3**	**2 B**
Belady's OPT	32.2	N/A

4.6.3 Related Work in Paging Domain

We also analyze some of the related replacement studies from the paging domain. Early Eviction LRU (EELRU) [69] tracks the hits obtained from each recency position for a larger sized cache. If there are significantly more hits from the recency positions larger than the cache size, EELRU changes the eviction point of the resident pages. For the studies

reported in [69], EELRU tracked 2.5 times as many pages as in physical memory. We analyzed EELRU for our workloads with 2.5 times the tag-store entries. EELRU reduces the average MPKI by 13.8% while incurring a storage overhead of 168kB, compared to DIP which reduces average MPKI by 22% while incurring a storage overhead of 2B.

A recent proposal, Adaptive Replacement Cache (ARC) [50], maintains two lists: *recency list* and *frequency list*. The recency list contains pages that were touched only once while resident, whereas the frequency list contains pages that were touched at least twice. ARC dynamically tunes the number of pages devoted to each list. We simulated ARC for our workloads and found that ARC reduces the average MPKI by 5.64% while requiring 64kB storage.

4.6.4   Related Work in Cache Bypassing and Early Eviction

Several studies have investigated cache bypassing and early eviction. McFarling[48] proposed dynamic exclusion to reduce conflict misses in a direct-mapped instruction cache. Gonzalez et al. [24] proposed using a *locality prediction table* to bypass access patterns that are likely to pollute the cache. Johnson [31] used reuse counters with a *macro address table* to bypass lines with low reuse. Several proposals [81] [85][82] exist for bypassing or early eviction of lines brought by instructions with low locality. Another area of research has been to predict the last touch to a cache line [41] [43]. After the predicted last touch, the line can either be turned off [36] or be used to store prefetched data [41].

However, *locality*, *liveness* and *last touch* are a function of both the replacement policy and the relative size of the working set to the available cache size. For example, if a cyclic reference pattern with a working set size slightly greater than the available cache size is applied to a LRU-managed cache, all the inserted lines will have poor locality, will be dead as soon as they are installed, and will have their last touch at insertion. The solution in such a case is neither to bypass all the lines nor to evict them early, but to retain some

fraction of the working set so that it provides cache hit. DIP retains some fraction of the working set for longer than LRU, thus obtaining hits for at least those lines.

4.6.5 Related Work in Prefetching

Lin et al. [21] propose to reduce cache pollution due to aggressive prefetching by inserting prefetched lines in the LRU position. However, their changes to the insertion policy are geared towards solving the problem of inaccuracy of prefetchers. With their proposal, demand lines are still inserted in the MRU position making the cache susceptible to thrashing by demand references. Our proposal, LIP, inserts *all* the incoming line in the LRU position thus protecting against thrashing by targeting the fundamental locality problem that exists in memory reference streams. As shown in Section 4.5.5, our work can be combined with Lin et al.'s work to protect the cache from both thrashing as well as prefetcher pollution.

4.7 Summary

The commonly used LRU replacement policy performs poorly for memory-intensive workloads that reuse a working set greater than the available cache size. The LRU policy inserts a line and evicts it before it is likely to be reused causing a majority of the lines in the cache to have zero reuse. In such cases, retaining some fraction of the working set would provide hits for at least that fraction of the working set. We separate the problem of replacement into two parts: *victim selection policy* and *insertion policy*. Victim selection deals with which line gets evicted to install the incoming line. The insertion policy deals with where on the replacement stack the incoming line is placed when installing it in the cache. This chapter show that simple changes to the insertion policy can significantly improve the cache performance of memory-intensive workloads, and make the following contributions:

1. We propose the LRU Insertion Policy (LIP) which inserts all the incoming lines in the LRU position of the recency stack. We show that LIP can protect against thrashing and yields close to optimal hit-rate for applications with cyclic reference pattern.

2. We propose the Bimodal Insertion Policy (BIP) as an enhancement to LIP that allows for aging and adapting to changes in the working set of an application. BIP infrequently inserts an incoming line in the MRU position, which allows it to respond to changes in the working set while retaining the thrashing protection of LIP.

3. We propose a Dynamic Insertion Policy (DIP) that dynamically chooses between BIP and traditional LRU replacement. DIP uses BIP for workloads that benefit from BIP while retaining traditional LRU for workloads that are LRU-friendly and incur increased misses with BIP.

4. We propose In-cache Dynamic Set Sampling (IDSS) to implement cost-effective dynamic selection between competing policies. IDSS dedicates a small percentage of sets in the cache to each of the two component policies and chooses the policy that has fewer misses on the dedicated set for the remaining follower sets. IDSS does not require any additional storage, except for a single saturating counter.

Chapter 5

MLP-Aware Cache Replacement

5.1 Introduction

Currently, processors are supported by large on-chip caches that try to provide faster access to recently-accessed data. Unfortunately, when there is a miss at the largest on-chip cache, instruction processing stalls after a few cycles [34], and the processing resources remain idle for hundreds of cycles [84]. The inability to process instructions in parallel with long-latency cache misses results in substantial performance loss. One way to reduce this performance loss is to process the cache misses in parallel.[1] Techniques such as non-blocking caches [39], out-of-order execution with large instruction windows, runahead execution [20][51], and prefetching improve performance by parallelizing long-latency memory operations. The notion of generating and servicing multiple outstanding cache misses in parallel is called *Memory Level Parallelism* (MLP) [23].

5.1.1 Not All Misses are Created Equal

Servicing misses in parallel reduces the number of times the processor has to stall due to a given number of long-latency memory accesses. However, MLP is not uniform across all memory accesses in a program. Some misses occur in isolation (e.g., misses due

---

[1]Multiple concurrent misses to the same cache block are treated as a single miss. *Parallel miss* refers to a miss that is serviced while there is at least one more miss outstanding. *Isolated miss* refers to a miss that is not serviced concurrently with any other miss.

to pointer-chasing loads), whereas some misses occur in parallel with other misses (e.g., misses due to array accesses). The performance loss resulting from a cache miss is reduced when multiple cache misses are serviced in parallel because the idle cycles waiting for memory get amortized over all the concurrent misses. Isolated misses hurt performance the most because the processor is stalled to service just a single miss. The non-uniformity in MLP and the resultant non-uniformity in the performance impact of cache misses opens up an opportunity for cache replacement policies that can take advantage of the variation in MLP. Cache replacement, if made MLP-aware, can save isolated (relatively more costly) misses instead of parallel (relatively less costly) misses.

Unfortunately, traditional cache replacement algorithms are not aware of the disparity in performance loss that results from the variation in MLP among cache misses. Traditional replacement schemes try to reduce the absolute number of misses with the implicit assumption that reduction in misses correlates with reduction in memory related stall cycles. However, due to the variation in MLP, the number of misses may or may not correlate directly with the number of memory related stall cycles. We demonstrate how ignoring MLP information in replacement decisions hurts performance with the following example. Figure 5.1(a) shows a loop containing 11 memory references. There are no other memory access instructions in the loop and the loop iterates many times.

Let K (K > 4) be the size of the instruction window of the processor on which the loop is executed. Points A, B, C, D, and E each represent an interval of at least K instructions. Between point A and point B, accesses to blocks P1, P2, P3, and P4 occur in the instruction window at the same time. If these accesses result in multiple misses then those misses are serviced in parallel, stalling the processor only once for the multiple parallel misses. Similarly, accesses between point B and point C will lead to parallel misses if there is more than one miss, stalling the processor only once for all the multiple parallel misses. Conversely, accesses to block S1, S2, or S3 result in isolated misses and the processor will

(a) An access pattern. A miss to an S block (S1,S2,S3) results in an Isolated miss. A miss to a P block (P1,P2,P3,P4) can be serviced in parallel with misses to other P blocks.

(b) Execution timeline for one iteration with Belady's OPT replacement

(c) Execution timeline for one iteration with MLP−aware replacement
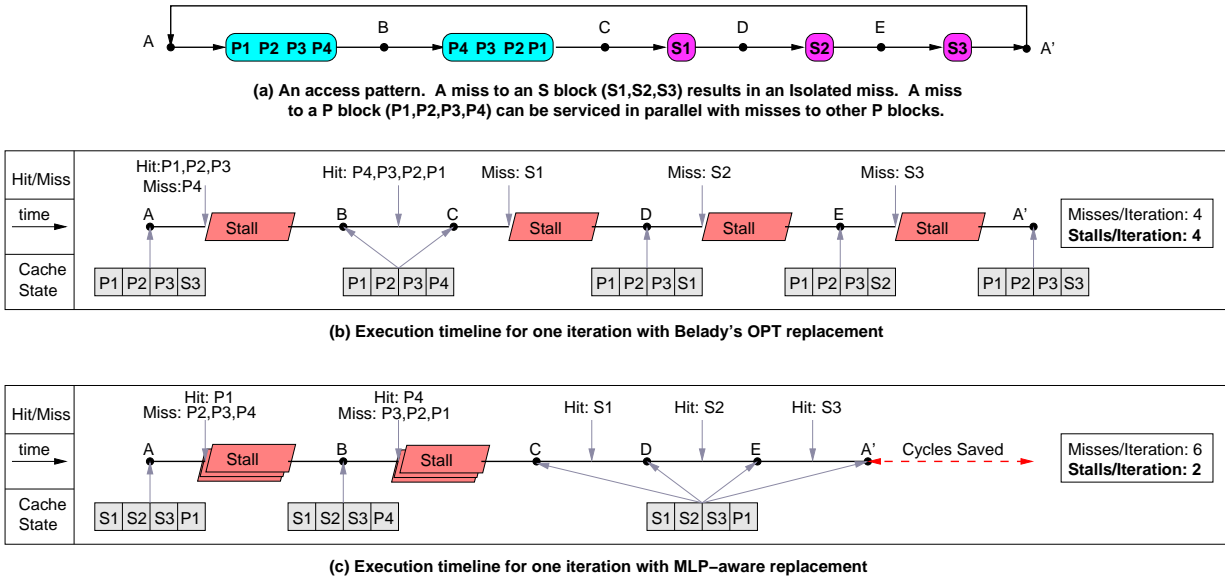
Figure 5.1: The drawback of not including MLP information in replacement decisions.

be stalled once for each such miss. We analyze the behavior of this access stream for a fully-associative cache that has space for four cache blocks, assuming the processor has already executed the first iteration of the loop.

First, consider a replacement scheme which tries to minimize the absolute number of misses, without taking MLP information into account. Belady's OPT [7] provides a theoretical minimum for the number of misses by evicting a block that is accessed furthest in the future. Figure 5.1(b) shows the behavior of Belady's OPT for the given access stream. At point B, blocks P1, P2, P3, and P4 were accessed in the immediate past and will be accessed again in the immediate future. Therefore, the cache contains blocks P1, P2, P3, and P4 at point B. This results in hits for the next accesses to blocks P4, P3, P2, and P1, and misses for the next accesses to blocks S1, S2, and S3. To guarantee the minimum number of misses, Belady's OPT evicts P4 to store S1, S1 to store S2, and S2 to store S3. Since the misses to S1, S2, and S3 are isolated misses, the processor incurs three long-latency stalls

66

between points C and A'. At point A, the cache contains P1, P2, P3, and S3 which results in a miss for P4, stalling the processor one more time. Thus, for each iteration of the loop, Belady's OPT causes four misses (S1, S2, S3, and P4) and four long-latency stalls.

Second, consider a simple MLP-aware policy, which tries to reduce the number of isolated misses. This policy keeps in cache the blocks that lead to isolated misses (S1, S2, S3) rather than the blocks that lead to parallel misses (P1, P2, P3, P4). Such a policy evicts the least-recently used P-block from the cache. However, if there is no P-block in the cache, then it evicts the least-recently used S-block. Figure 5.1(c) shows the behavior of such an MLP-aware policy for the given access stream. The cache has space for four blocks and the loop contains only 3 S-blocks (S1, S2, and S3). Therefore, the MLP-aware policy never evicts an S-block at any point in the loop. After the first loop iteration, each access to S1, S2, and S3 results in a hit. At point A, the cache contains S1, S2, S3, and P1. From point A to B, the access to P1 hits in the cache, and the accesses to P2, P3, and P4 miss in the cache. However, these misses are serviced in parallel, therefore the processor incurs only one long-latency stall for these three misses. The cache evicts P1 to store P2, P2 to store P3, and P3 to store P4. So, at point B, the cache contains S1, S2, S3, and P4. Between point B and point C, the access to block P4 hits in the cache, while accesses to blocks P3, P2, and P1 miss in the cache. These three misses are again serviced in parallel, which results in one long-latency stall. Thus, for each loop iteration, the MLP-aware policy causes six misses ([P2, P3, P4] and [P3, P2, P1]) and only two long-latency stalls.

Note that Belady's OPT uses oracle information, whereas the MLP-aware scheme uses only information that is available to the microarchitecture. Whether a miss is serviced in parallel with other misses can easily be detected in the memory system, and the MLP-aware replacement scheme uses this information to make replacement decisions. For the given example, even with the benefit of an oracle, Belady's OPT incurs twice as many long-

67

latency stalls compared to a simple MLP-aware policy.[2] This simple example demonstrates that it is important to incorporate MLP information into replacement decisions.

5.1.2 Contributions

Based on the observation that the aim of a cache replacement policy is to reduce memory related stalls, rather than to reduce the raw number of misses, this chapter proposes MLP-aware cache replacement and make the following contributions:

1. As a first step to enable MLP-aware cache replacement, we propose a run-time algorithm that can compute MLP-based cost for in-flight misses.

2. We show that, for most benchmarks, the MLP-based cost repeats for consecutive misses to individual cache blocks. Thus, the last-time MLP-based cost can be used as a predictor for the next-time MLP-based cost.

3. We propose a simple replacement policy called the Linear (LIN) policy which takes both recency and MLP-based cost into account to implement a practical MLP-aware cache replacement scheme. Evaluation with the SPEC CPU2000 benchmarks shows performance improvement of up to 23% with the LIN policy.

4. The LIN policy does not perform well for benchmarks in which the MLP-based cost differs significantly for consecutive misses to an individual cache block. We propose a cost-effective hybrid replacement policy to select between LIN and LRU, depending on which policy results in the least number of memory related stall cycles.

---

[2]We use Belady's OPT in the example only to emphasize that the concept of reducing the number of misses and making the replacement scheme MLP-aware are independent. However, Belady's OPT is impossible to implement because it requires knowledge of the future. Therefore, we will use LRU as the baseline replacement policy for the remainder of this paper. For the LRU policy, each iteration of the loop shown in Figure 5.1 causes six misses ([P2, P3, P4], S1, S2, S3) and four long-latency stalls.

5.2   Background

Out-of-order execution inherently improves MLP by continuing to execute instructions after a long-latency miss. Instruction processing stops only when the instruction window becomes full. If additional misses are encountered before the window becomes full, then these misses are serviced in parallel with the stalling miss. The analytical model of out-of-order superscalar processors proposed by Karkhanis and Smith [35] provides fundamental insight into how parallelism in L2 misses can reduce the cycles per instruction incurred due to L2 misses.

The effectiveness of an out-of-order engine's ability to increase MLP is limited by the instruction window size. Several proposals [51][2][18][88] have looked at the problem of scaling the instruction window for out-of-order processors. Chou et al. [16] analyzed the effectiveness of different microarchitectural techniques such as out-of-order execution, value prediction, and runahead execution on increasing MLP. They concluded that microarchitecture optimizations can have a profound impact on increasing MLP. They also formally defined instantaneous MLP as *the number of useful long-latency off-chip accesses outstanding when there is at least one such access outstanding*. MLP can also be improved at the compiler level. Read miss clustering [55] is a compiler technique in which the compiler reorders load instructions with predictable access patterns to improve memory parallelism.

All of the techniques described thus far try to improve MLP by overlapping long-latency memory operations. MLP is not uniform across all memory accesses in a program though. While some of the misses are parallelized, many misses still occur in isolation. It makes sense to make this variation in MLP visible to the cache replacement algorithm. Cache replacement, if made MLP-aware, can increase performance by reducing the number of isolated misses at the expense of parallel misses. To our knowledge no previous research has looked at including MLP information in replacement decisions. Srinivasan et

al. [75][74] analyzed the criticality of load misses for out-of-order processors. But, criticality and MLP are two different properties. Criticality, as defined in [75], is determined by how long instruction processing continues after a load miss, whereas, MLP is determined by how many additional misses are encountered while servicing a miss.

Cost-sensitive replacement policies for on-chip caches were investigated by Jeong and Dubois [29][30]. They proposed variations of LRU that take *cost* (any numerical property associated with a cache block) into account. In general, any cost-sensitive replacement scheme, including the ones proposed in [30], can be used for implementing an MLP-aware replacement policy. However, to use any cost-sensitive replacement scheme, we first need to define the *cost* of each cache block based on the MLP with which it was serviced. As the first step to enable MLP-aware cache replacement, we introduce a run-time technique to compute MLP-based cost.

5.3 Computing MLP-Based Cost

For current instruction window sizes, instruction processing stalls shortly after a long-latency miss occurs. The number of cycles for which a miss stalls the processor can be approximated by the number of cycles that the miss spends waiting to get serviced. For parallel misses, the stall cycles can be divided equally among all concurrent misses.

5.3.1 Algorithm

The information about the number of in-flight misses and the number of cycles a miss is waiting to get serviced can easily be tracked by the MSHR (Miss Status Holding Register). Each miss is allocated an MSHR entry before a request to service that miss is sent to memory [39]. To compute the MLP-based cost, we add a field *mlp_cost* to each MSHR entry. Algorithm 1 describes the calculation of MLP-based cost of a cache miss.

70

Algorithm 1 Calculate MLP-based cost for cache misses

---
init_mlp_cost(miss):   /* when miss enters MSHR */
  $miss.mlp\_cost = 0$
update_mlp_cost( ):   /* called every cycle */
  $N \Leftarrow$ Number of outstanding demand misses in MSHR
  for each demand miss in the MSHR
    $miss.mlp\_cost \mathrel{+}= (1/N)$

When a miss is allocated an MSHR entry, the mlp_cost field associated with that entry is initialized to 0. We count instruction accesses, load accesses, and store accesses that miss in the largest on-chip cache as demand misses. All misses are treated on correct path until they are confirmed to be on the wrong path. Misses on the wrong path are not counted as demand misses. Each cycle, the mlp_cost of all demand misses in the MSHR is incremented by the amount *1/(Number of outstanding demand misses in MSHR)*.[3,4] When a miss is serviced, the mlp_cost field in the MSHR represents the MLP-based cost of that miss. Henceforth, we will use mlp-cost to denote MLP-based cost.

5.3.2   Distribution of mlp-cost

Figure 5.2 shows the distribution of mlp-cost for 14 SPEC benchmarks measured on an eight-wide issue, out-of-order processor with a 128-entry instruction window. An isolated miss takes 444 cycles (400-cycle bank access + 44-cycle bus delay) to get serviced. The vertical axis represents the percentage of all misses and the horizontal axis corresponds

---

[3] The number of adders required for the proposed algorithm is equal to the number of MSHR entries. However, for the baseline machine with 32 MSHR entries, time sharing four adders among the 32 entries has only a negligible effect on the absolute value of the MLP-based cost. For all our experiments, we assume that the MSHR contains only four adders for calculating the MLP-based cost. If more than four MSHR entries are valid, then the adders are time-shared between all the valid entries using a simple round-robin scheme.

[4] We also experimented by increasing the mlp_cost only during cycles when there is a full window stall. However, we did not find any significant difference in the relative value of mlp_cost or the performance improvement provided by our proposed replacement scheme. Therefore, for simplicity, we assume that the mlp_cost is updated every cycle.
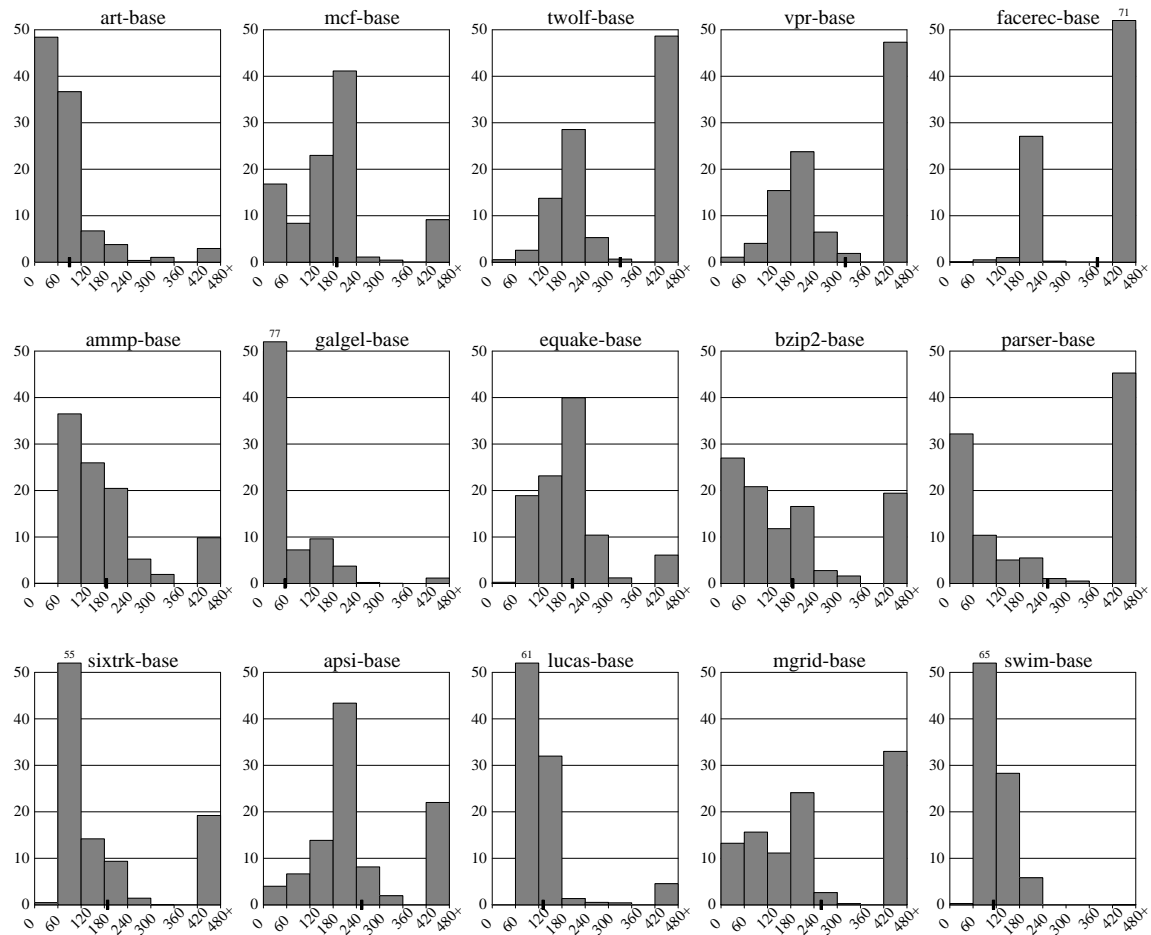
Figure 5.2: Distribution of mlp-cost. The horizontal axis represents the value of mlp-cost in cycles and the vertical axis represents the percentage of total misses. The dot on the horizontal axis represents the average value of mlp-cost.

to different values of mlp-cost. The graph is plotted with 60-cycle intervals, with the leftmost bar representing the percentage of misses that had a value of $0 \leq$ mlp-cost < 60 cycles. The rightmost bar represents the percentage of all misses that had an mlp-cost of more than 420 cycles. All isolated misses (and some parallel misses that are serialized because of DRAM bank conflicts) are accounted for in the right-most bar.

For each benchmark, the average value of mlp-cost is much less than 444 cycles (number of cycles needed to serve an isolated miss). For art, more than 85% of the misses have an mlp-cost of less than 120 cycles indicating a high parallelism in misses. For mcf, about 40% of the misses have an mlp-cost between 180 and 240 cycles, which corresponds to two misses in parallel. Mcf also has about 9% of its misses as isolated misses. Facerec has two distinct peaks, one for the misses that occur in isolation and the other for the misses that occur with a parallelism of two. Twolf, vpr, facerec, and parser have a high percentage of isolated misses and hence the peak for the rightmost bar. The results for all of these benchmarks clearly indicate that there exists non-uniformity in mlp-cost which can be exploited by MLP-aware cache replacement. The objective of MLP-aware cache replacement is to reduce the number of isolated (i.e., relatively more costly) misses without substantially increasing the total number of misses. mlp-cost can serve as a useful metric in designing an MLP-aware replacement scheme. However, for the decision based on mlp-cost to be meaningful, we need a mechanism to predict the future mlp-cost of a miss given the current mlp-cost of a miss. For example, a miss that happens in isolation once can happen in parallel with other misses the next time, leading to significant variation in the mlp-cost for the miss. If mlp-cost is not predictable for a cache block, the information provided by the mlp-cost metric is not useful. The next section examines the predictability of mlp-cost.

5.3.3   Predictability of the mlp-cost metric

One way to predict the future mlp-cost value of a block is to use the current mlp-cost value of that block. The usefulness of this scheme can be evaluated by measuring the difference between the mlp-cost for successive misses to a cache block. We call the absolute difference in the value of mlp-cost for successive misses to a cache block as *delta*. For example, let cache block A have mlp-cost values of {444 cycles, 80 cycles, 80 cycles,

220 cycles} for the four misses it had in the program. Then, the first delta for block A is 364 ($\|444 - 80\|$) cycles, the second delta for block A is 0 ($\|80 - 80\|$) cycles, and the third delta for block A is 140 ($\|80 - 220\|$) cycles. To measure delta, we do an off-line analysis of all the misses in the program. Table 5.1 shows the distribution of delta. A small delta value means that mlp-cost does not significantly change between successive misses to a given cache block.

Table 5.1: Repeatability of mlp-cost. The first three columns after the benchmark name represent the percentage of deltas that were between 0-59 cycles, 60-119 cycles, and more than 120 cycles respectively. The last column represents the average value of delta.

Benchmark	delta < 60	60 ≤ delta < 120	delta ≥ 120	Average delta
art	86%	7%	7%	30
mcf	86%	7%	7%	21
twolf	52%	12%	36%	98
vpr	50%	14%	36%	100
facerec	96%	0%	4%	9
ammp	82%	10%	8%	32
galgel	71%	9%	20%	82
equake	78%	12%	10%	40
bzip2	43%	15%	42%	126
parser	43%	5%	52%	190
apsi	85%	5%	10%	28
sixtrack	100%	0%	0%	1
lucas	84%	6%	10%	52
mgrid	18%	16%	66%	187
swim	80%	16%	4%	41

For all the benchmarks, except bzip2, parser, and mgrid, the majority of the delta values are less than 60 cycles. The average delta value is also fairly low, which means that the next-time mlp-cost for a cache block remains fairly close to the current mlp-cost. Thus, the current mlp-cost can be used as a predictor of the next mlp-cost of the same block in MLP-aware cache replacement. We describe our experimental methodology before discussing the design and implementation of an MLP-aware cache replacement scheme based on these observations.

5.4 The Design of an MLP-Aware Cache Replacement Scheme

Figure 5.3 shows the microarchitecture design for MLP-aware cache replacement. The added structures are shaded. The cost calculation logic (CCL) contains the hardware implementation of Algorithm 1. It computes mlp-cost for all demand misses. When a miss gets serviced, the mlp-cost of the miss is stored in the tag-store entry of the corresponding cache block. For replacement, the cache invokes the Cost Aware Replacement Engine (CARE) to find the replacement victim. CARE can consist of any generic cost-sensitive scheme [30][52]. We evaluate MLP-aware cache replacement using both an existing as well as a novel cost-sensitive replacement scheme.
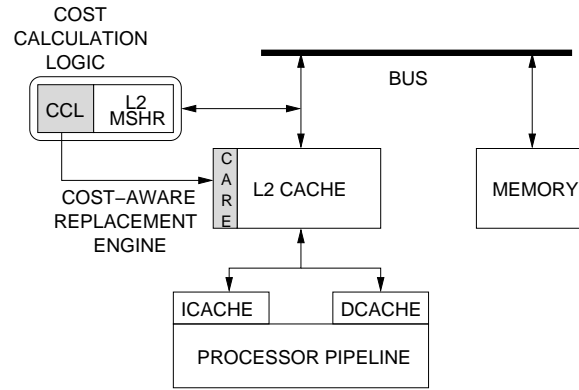
Figure 5.3: Microarchitecture for MLP-aware cache replacement (Figure not to scale).

Before discussing the details of the MLP-aware replacement scheme, it is useful to note that the exact value of mlp-cost is not necessary for replacement decisions. In a real implementation, to limit the storage overhead, the value of mlp-cost can be quantized to a few bits and the quantized value would be stored in the tag-store. We consider one such quantization scheme. It converts the value of mlp-cost into a 3-bit quantized value, according to the intervals shown in Table 5.2. Henceforth, we use $cost_q$ to denote the quantized value of mlp-cost.

Table 5.2: Quantization of mlp-cost

Computed value of mlp-cost	Quantized value of mlp-cost
0-59 cycles	0
60-119 cycles	1
120-179 cycles	2
180-239 cycles	3
240-299 cycles	4
300-359 cycles	5
360-419 cycles	6
420+ cycles	7

5.4.1 The Linear (LIN) Policy

The baseline replacement policy is LRU. The replacement function of LRU selects the candidate cache block with the least recency. Let $Victim_{LRU}$ be the victim selected by LRU and $R(i)$ be the recency value (highest value denotes the MRU and lowest value denotes LRU) of block i. Then, the victim of the LRU policy can be written as:

$$Victim_{LRU} = \arg\min_i \{R(i)\} \tag{5.1}$$

We want a policy that takes into account both $cost_q$ and recency. We propose a replacement policy that employs a linear function of recency and $cost_q$. We call this policy the Linear (LIN) policy. The replacement function of LIN can be summarized as follows: Let $Victim_{LIN}$ be the victim selected by the LIN policy, $R(i)$ be the recency value of block i, and $cost_q(i)$ be the quantized cost of block i, then the victim of the LIN policy can be written as:

$$Victim_{LIN} = \arg\min_i \{R(i) + \lambda \cdot cost_q(i)\} \tag{5.2}$$

76

The parameter λ determines the importance of cost_q in choosing the replacement victim. In case of a tie for the minimum value of $\{R + \lambda \cdot \mathsf{cost_q}\}$, the candidate with the smallest recency value is selected. Note that LRU is a special case of the LIN policy with $\lambda = 0$. With a high λ value, the LIN policy tries to retain recent cache blocks that have high $\mathsf{mlp\text{-}cost}$. For our experiments, we used the position in the LRU stack as the recency value (e.g. for a 16-way cache, $R(MRU) = 15$ and $R(LRU) = 0$). Since cost_q is quantized into three bits, its range is from 0 to 7. Unless stated otherwise, we use $\lambda = 4$ in all our experiments.

5.4.2 Results for the LIN Policy

Figure 5.4 shows the performance impact of the LIN policy for different values of λ. The effect of the LIN policy is more pronounced as the value of λ is increased from 1 to 4. With λ=4, the LIN policy provides a significant IPC improvement for art, mcf, vpr, galgel, and sixtrack. In contrast, it degrades performance for bzip2, parser, and mgrid. These benchmarks have high average delta values (refer to Table 5.1), so the replacement decisions based on $\mathsf{mlp\text{-}cost}$ hurts performance. LIN can improve performance by reducing the number of isolated misses, or by reducing the total number of misses, or both. We analyze the LIN policy further by comparing the $\mathsf{mlp\text{-}cost}$ distribution of the LIN policy with the $\mathsf{mlp\text{-}cost}$ distribution of the baseline.

Figure 5.5 shows the $\mathsf{mlp\text{-}cost}$ distribution for both the baseline and the LIN policy. The inset contains information about the change in the number of misses and the change in IPC due to LIN. For mcf, almost all the isolated misses are eliminated by LIN. For twolf, although the total number of misses increases by 7%, IPC increases by 1.5%. A similar trend of increase in misses accompanied by increase in IPC is observed for ammp and equake. For these benchmarks, the IPC improvement is coming from reducing the number of misses with high $\mathsf{mlp\text{-}cost}$ even if this translates into a slightly-increased total number

77
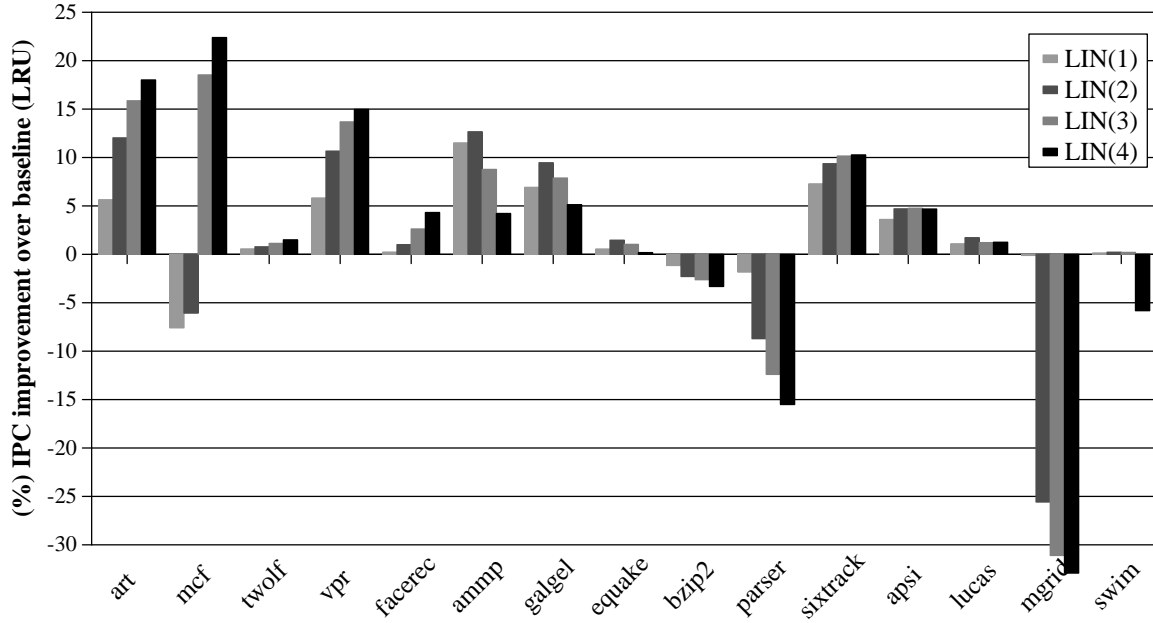
Figure 5.4: IPC improvement with LIN (λ) as λ is varied.

of misses. For all benchmarks, except art and galgel, the distribution of mlp-cost is skewed towards the left (i.e. lower mlp-cost) for the LIN policy when compared to the baseline. This indicates that LIN -successfully- has a bias towards reducing the proportion of high mlp-cost misses.

For art, galgel, and sixtrack, LIN reduces the total number of misses by more than 30%. This happens for applications that have very large data working-sets with low temporal locality, causing LRU to perform poorly [56][85]. The LIN policy automatically provides filtering for access streams with low temporal locality by at least keeping some of the high mlp-cost blocks in the cache, when LRU could have potentially caused thrashing. The large reduction in the number of misses for art and galgel reduces the parallelism with which the remaining misses get serviced. Hence, for both art and galgel, the average mlp-cost with the LIN policy is slightly higher than for the baseline.
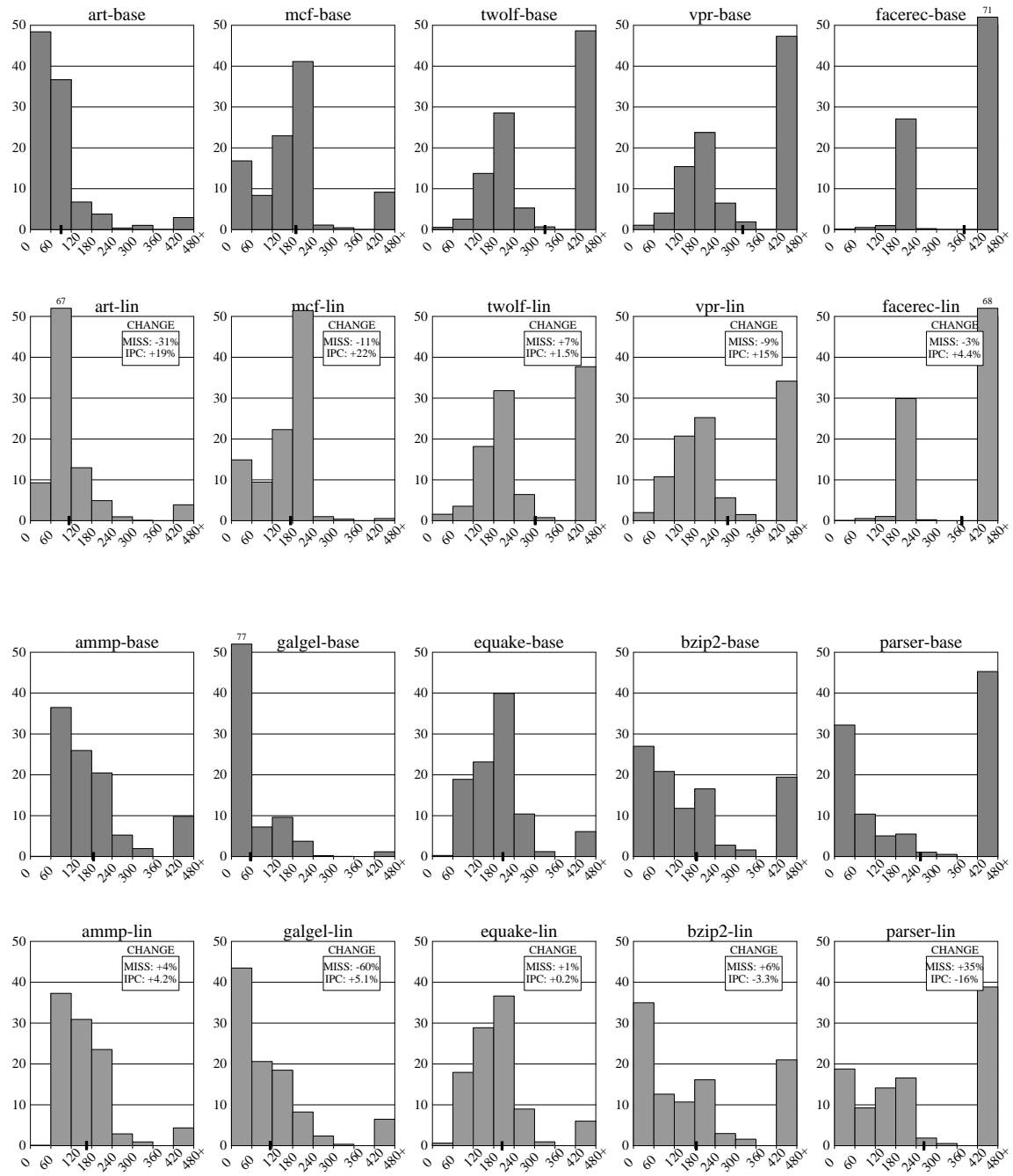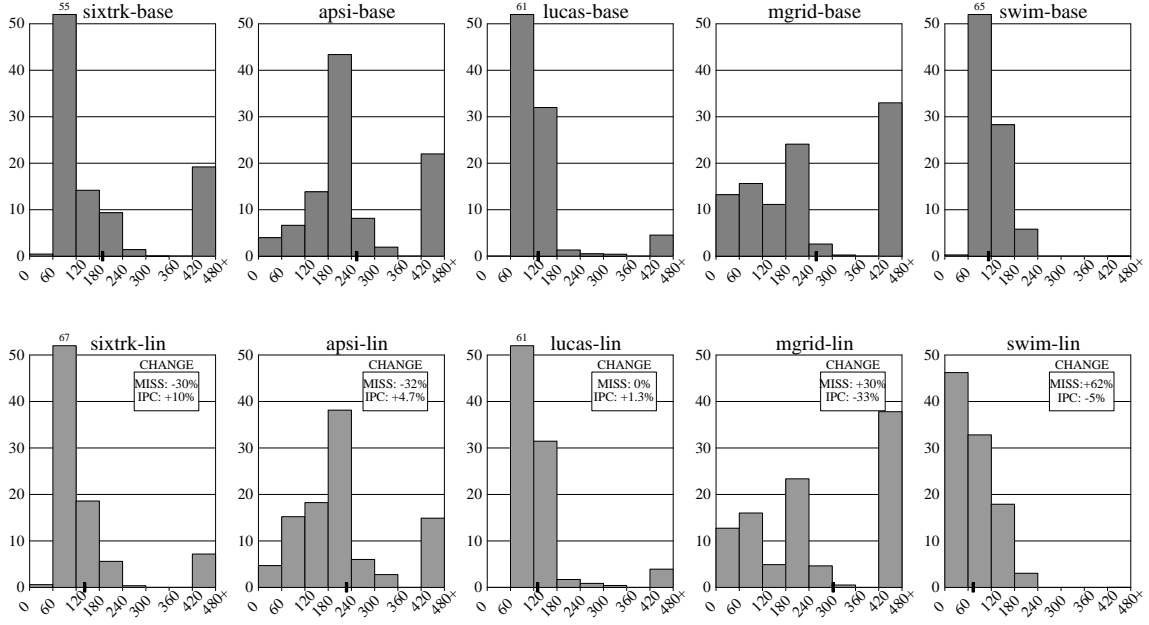
Figure 5.5: Distribution of mlp-cost for baseline and LIN ($\lambda = 4$). The horizontal axis represents the value of mlp-cost in cycles and the vertical axis represents the percentage of all misses. The dot on the horizontal axis represents the average value of mlp-cost. The insets in the graphs contain information about the change in the number of misses and IPC with the use of the LIN policy.

The LIN policy tries to retain recent cache blocks that have high mlp-cost values. The implicit assumption is that the blocks that had high mlp-cost at the time they were brought in the cache will continue to have high mlp-cost the next time they need to be fetched. Therefore, the LIN policy performs poorly for benchmarks in which current mlp-cost is not a good indicator of the next-time mlp-cost. Examples of such benchmarks are bzip2 (average delta = 126 cycles), parser (average delta = 190 cycles), and mgrid (average delta = 187 cycles). For these benchmarks, the number of misses increases significantly with the LIN policy. For the LIN policy to be useful for a wide variety of applications, we need a feedback mechanism that can limit the performance degradation caused by LIN. This can be done by dynamically choosing between the baseline LRU policy and the LIN policy depending on which policy is doing better. The next section presents a novel, low-overhead adaptation scheme that provides such a capability.

80

5.5 Cost-Sensitive Hybrid Replacement

LIN performs better on some benchmarks and LRU performs better on some benchmarks. We want a mechanism that can dynamically choose the replacement policy that provides higher performance, or equivalently fewer memory related stall cycles. The SBAR policy proposed in Chapter 3 can be modified to choose the component policy that incurs the minimum aggregate cost of the miss. The next section describes cost-sensitive tournament selection which can be used with SBAR to select between LIN and LRU.

5.5.1 Cost-Sensitive Tournament Selection of Replacement Policy

Let MTD be the main tag directory of the cache. For facilitating hybrid replacement, MTD is capable of implementing both LIN and LRU. MTD is appended with two Auxiliary Tag Directories (ATDs): ATD-LIN and ATD-LRU. ATD-LIN implements only the LIN policy, and ATD-LRU implements only the LRU policy. A saturating counter (PSEL) keeps track of which of the two ATDs is doing better. The access stream visible to MTD is also fed to both ATD-LIN and ATD-LRU. Both ATD-LIN and ATD-LRU compete and the output of PSEL is an indicator of which policy is doing better. The replacement policy to be used in MTD is chosen based on the output of PSEL. Figure 5.6 shows the operation of cost-sensitive tournament selection (TSEL) mechanism for one set in the cache.

If a given access hits or misses in both ATD-LIN and ATD-LRU, neither policy is doing better than the other. Thus, PSEL remains unchanged. If an access misses in ATD-LIN but hits in ATD-LRU, LRU is doing better than LIN for that access. In this case, PSEL is decremented by a value equal to the cost_q of the miss (a 3-bit value) incurred by ATD-LIN. Conversely, if an access misses in ATD-LRU but hits in ATD-LIN, LIN is doing better than LRU. Therefore, PSEL is incremented by a value equal to the cost_q of the miss incurred by ATD-LRU. Unless stated otherwise, we use a 6-bit PSEL counter in our experiments. All PSEL updates are done using saturating arithmetic.

ATD–LIN	ATD–LRU	Action
HIT	HIT	PSEL unchanged
MISS	MISS	PSEL unchanged
MISS	HIT	Decrement PSEL by cost $_q$ of Miss in ATD–LIN
HIT	MISS	Increment PSEL by cost $_q$ of Miss in ATD–LRU
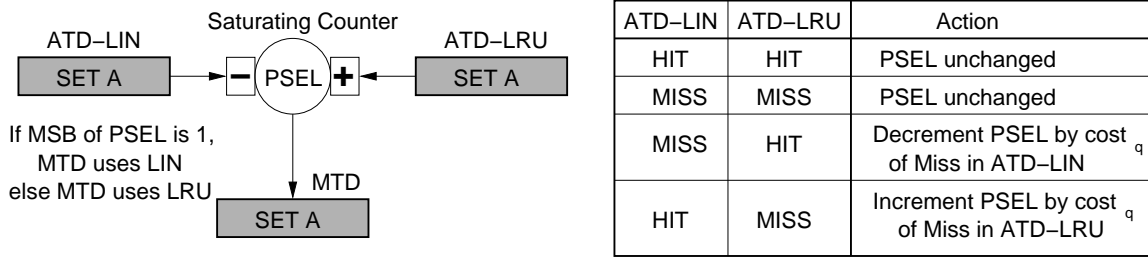
Figure 5.6: Cost-sensitive Tournament Selection for a single set.

Only accesses that result in a miss for MTD are serviced by the memory system. If an access results in a hit for MTD but a miss for either ATD-LIN or ATD-LRU, then it is not serviced by the memory system. Instead, the ATD that incurred the miss finds a replacement victim using its replacement policy. The tag field associated with the replacement victim of the ATD is updated. The value of $cost_q$ associated with the block is obtained from the corresponding tag-directory entry in MTD.

If LIN reduces memory related stall cycles more than LRU, then PSEL will be saturated towards its maximum value. Similarly, PSEL will be saturated towards zero if the opposite is true. If the most significant bit (MSB) of PSEL is 1, the output of PSEL indicates that LIN is doing better. Otherwise, the output of PSEL indicates that LRU is doing better. Note that PSEL is incremented or decremented by $cost_q$ instead of by 1, which results in selection based on the cumulative value of MLP-based cost of misses (i.e., $\sum cost_q$), rather than the raw number of misses. This is an important factor in the TSEL mechanism that allows TSEL to select the policy that results in the smallest number of stall cycles, rather than the smallest number of misses. If the value of $cost_q$ is constant or random, then the adaptation mechanism automatically degenerates to selecting the policy that results in the smallest number of misses.

5.5.2 Sampling Based Adaptive Replacement

The cost-sensitive tournament selection can be extended to the whole cache by using the SBAR mechanism of Chapter 3. Figure 5.7 shows the cost-sensitive SBAR selection between LIN and LRU for a cache containing eight sets. The sets in MTD are logically divided into two categories: *Leader Sets* and *Follower Sets*. The leader sets in MTD use only the LIN policy for replacement and participate in updating the PSEL counter. The follower sets implement both the LIN and the LRU policies for replacement and use the PSEL output to choose their replacement policy. The follower sets do not update the PSEL counter. There is only a single ATD, ATD-LRU. ATD-LRU implements only the LRU policy and has only sets corresponding to the leader sets. The leader sets are chosen using the simple static policy described in Chapter 3.
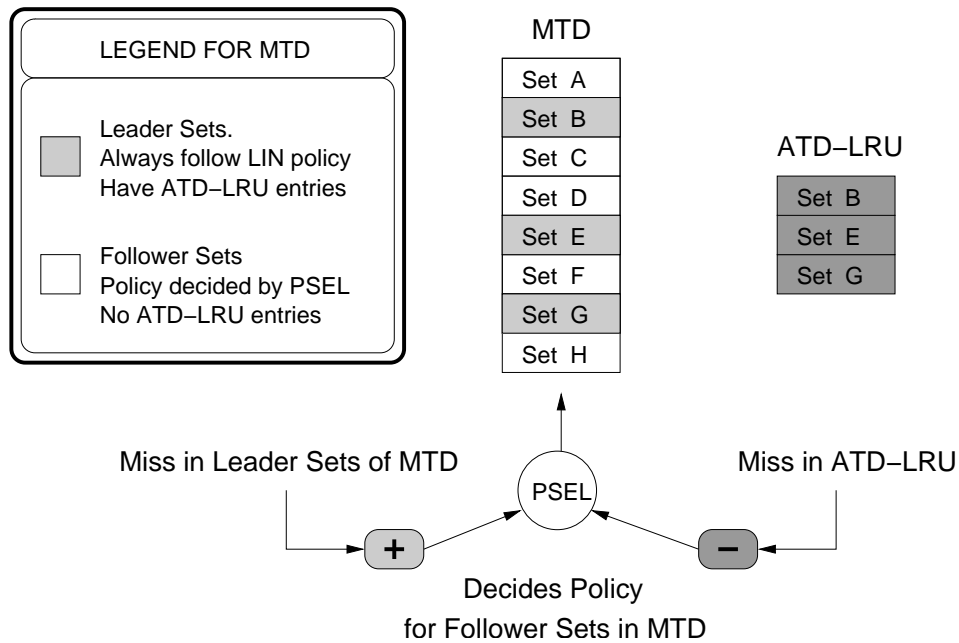
Figure 5.7: Cost-sensitive SBAR selection of LIN vs. LRU for a cache that has eight sets.

5.5.3   Results for the SBAR Mechanism

Figure 5.8 shows the IPC improvement over the baseline configuration when the SBAR mechanism is used to dynamically choose between LRU and LIN. For comparison, the IPC improvement provided by the LIN policy is also shown.
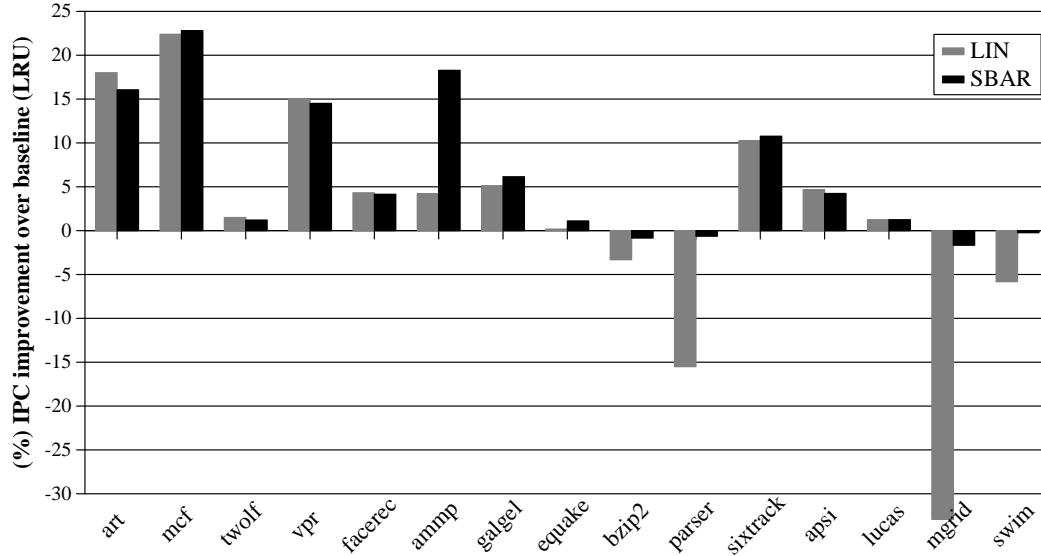
Figure 5.8: IPC improvement with the SBAR mechanism.

For art, mcf, vpr, facerec, sixtrack, and apsi, SBAR maintains the IPC improvement provided by LIN. The most important contribution of SBAR is that it eliminates the performance degradation caused by LIN on bzip2, parser, mgrid, and swim. For these benchmarks, the PSEL in the SBAR mechanism is almost always biased towards LRU. The marginal performance loss in these three benchmarks is because the leader sets in MTD still use only LIN as their replacement policy. For ammp and galgel, the SBAR policy does better than either LIN or LRU alone. This happens because in some phases of the program LIN does better, while in others LRU does better. With SBAR, the cache is able to select the policy better suited for each phase, thereby allowing it to outperform either policy implemented alone. In Section 5.6.1, we analyze the ability of SBAR to adapt to varying program phases using ammp as a case study.

5.5.4 Effect of Leader Set Selection Policies and Different Number of Leader Sets

To analyze the effect of leader set selection policies, we introduce a runtime policy, *rand-runtime*. Rand-runtime randomly selects one set from each constituency as the leader set. In our experiments, we invoke rand-runtime once every 25M instructions and mark the sets chosen by rand-runtime as leader sets for the next 25M instructions. Figure 5.9 shows the performance improvement for the SBAR policy with the simple-static policy and the rand-runtime policy for 8, 16, and 32 leader sets.
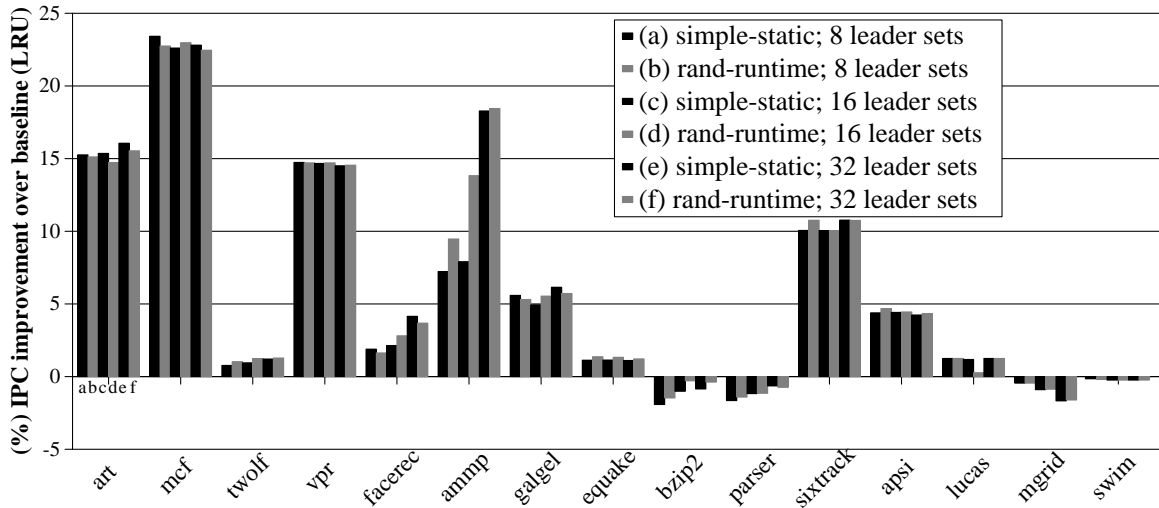
Figure 5.9: Performance impact of SBAR for different leader set selection policies and different number of leader sets.

For all benchmarks, except ammp, the IPC improvement of SBAR is relatively insensitive to both the leader set selection policy and the number of leader sets. In most benchmarks, one replacement policy does overwhelmingly better than the other. This causes almost all the sets in the cache to favor one policy. Hence, even as few as eight leader sets are sufficient, and the simple-static policy works well. For ammp, the rand-runtime policy performs better than the simple-static policy when the number of leader sets

is 16 or smaller. This is because ammp has widely-varying demand across different cache sets, which is better handled by the random selection of the rand-runtime policy than the rigid static selection of the simple-static policy. However, when the number of leader sets increases to 32, the effect of the set selection policy is less pronounced, and there is hardly any performance difference between the two set selection policies. Due to its simplicity, we use the simple-static policy with 32 leader sets as default in all our SBAR experiments.

We also compared SBAR to TSEL-global and found that, except for ammp, the IPC increase provided by SBAR is within 1% of the TSEL-global policy (we use a seven-bit PSEL for TSEL-global). For ammp, TSEL-global improves IPC by 20.3% while SBAR improves IPC by 18.3%. However, SBAR requires 64 times fewer ATD entries than TSEL-global, making it a much more practical solution.

5.6   Analysis

5.6.1   Ammp: A Case Study for Dynamic Adaptation of SBAR

For ammp, SBAR improves IPC by 18.3% over the baseline LRU policy while the LIN policy improves IPC by only 4.2%. This difference in IPC improvement between SBAR and LIN is because ammp has two distinct phases: in one phase LIN performs better than LRU and in the other LRU performs better than LIN. To view this time-varying phase behavior, we collected statistics from the cache every 10M retired instructions during simulation. Figure 5.10(a) shows the average $cost_q$ per miss, Figure 5.10(b) shows the misses per 1000 retired instructions, and Figure 5.10(c) shows the IPC for three different policies: LRU, LIN, and SBAR over time during the simulation runs.
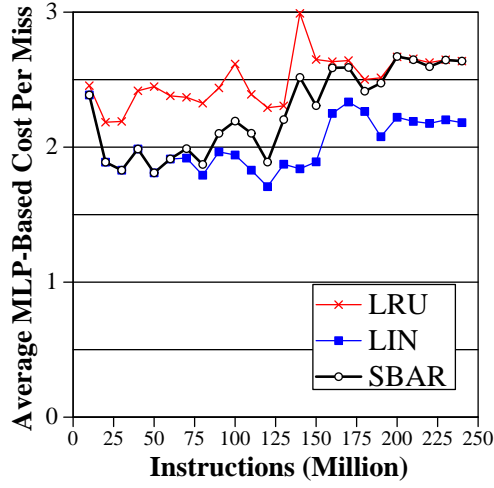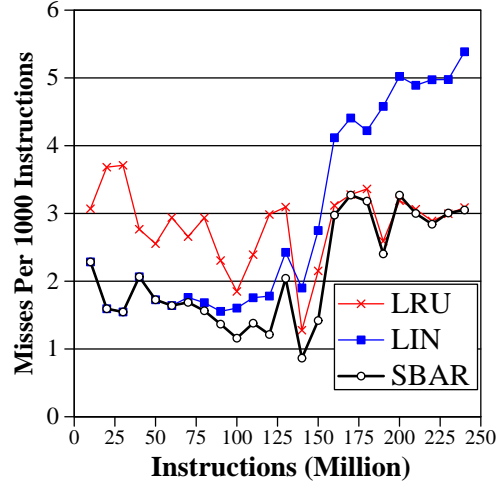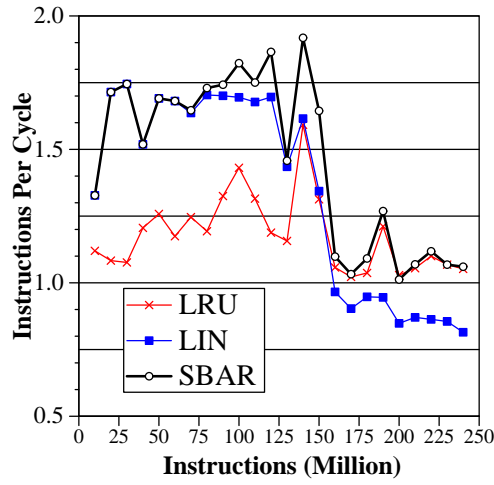
Figure 5.10: Comparison of LRU, LIN, and SBAR for the ammp benchmark in terms of: (a) the average cost of misses, (b) the number of misses per 1000 instructions, and (c) IPC.

As expected, LIN results in lower cost_q per miss than LRU throughout the whole simulation, indicating that the LIN policy is successful at reducing the cost_q of misses. However, this reduction can come at the expense of significantly increasing the raw number of misses, which may negatively impact the IPC. Until 150M instructions, this is not a problem: LIN has both lower cost_q per miss and fewer misses than LRU. Therefore, the IPC with LIN is much better than the IPC with LRU for the first 150M instructions. However, after 150M instructions, LIN has significantly more misses than LRU, which reduces the IPC for the LIN policy compared to LRU. With SBAR, the cache dynamically adapts and uses the policy that is best suited for each phase: LIN until 150M instructions and LRU after 150M instructions. Therefore, SBAR provides higher performance than both LIN and LRU.

5.6.2 Hardware Cost of MLP-Aware Replacement

The performance improvement of MLP-aware replacement comes at a small hardware overhead. For each entry in the MSHR, an additional bits are required to store mlp-cost. We assume that each MSHR entry stores the mlp-cost in a 9.5 fixed point format, where 9 bits are used to encode the integer part and 5 bits are used to encode the fractional part, thus 14 bits are required per each MSHR entry to compute the mlp-cost. Also, cost_q is stored in each tag-store entry in the cache, increasing the size of each tag-store entry by three bits. If SBAR is used to adaptively choose between LRU and LIN, then additional storage is required for the ATD entries. The hardware overhead of the ATD is 1856 bytes, which is less than 0.2% of the total area of the baseline L2 cache.

5.6.3 MLP-Aware Replacement using Existing Cost-Sensitive Replacement Policy

We proposed the SBAR mechanism to implement a MLP-aware cache replacement policy. However, the central idea of this chapter, MLP-aware cache replacement, is not lim-

ited in implementation to the proposed SBAR mechanism. Our framework for MLP-aware cache replacement makes even existing cost-sensitive replacement policies applicable to the MLP domain. As an example, we use Adaptive Cost-Sensitive LRU (ACL) [30] to implement an MLP-aware replacement policy. ACL was proposed for cost-sensitive replacement in Non-Uniform Memory Access (NUMA) systems and used the memory access latency as the *cost* parameter. Similarly, MLP information about a cache block can also be used as the *cost* parameter in ACL. Figure 5.11 shows the performance improvement of an MLP-aware replacement scheme implemented using ACL. For comparison, the results for SBAR are also shown.
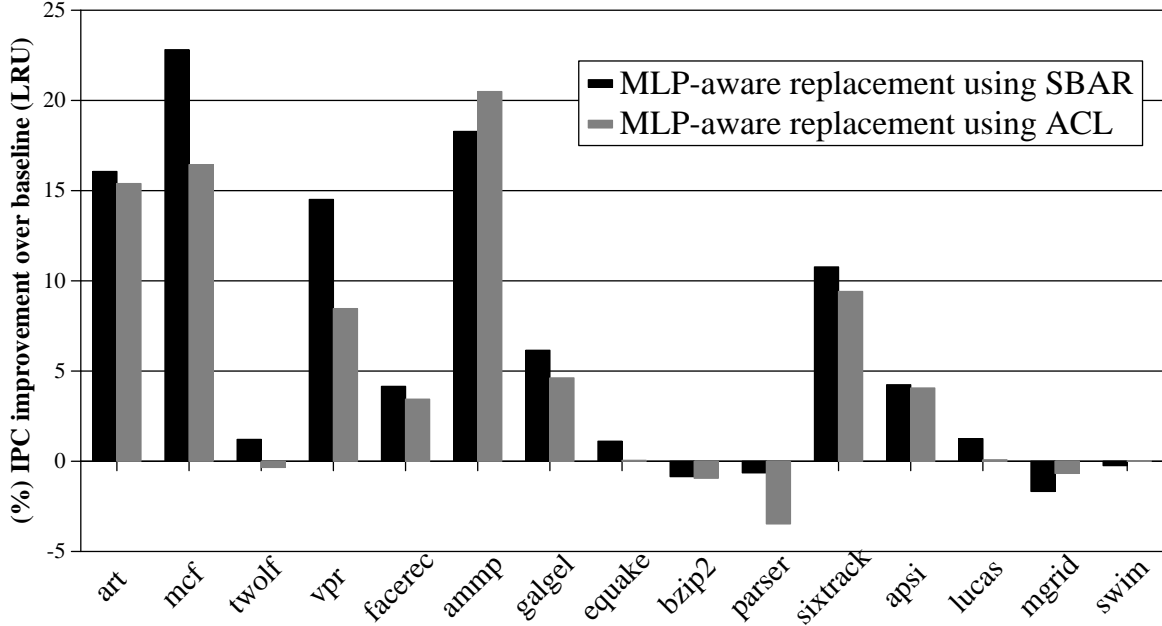
Figure 5.11: MLP-aware replacement using different cost-sensitive policies.

MLP-aware replacement improves performance for both implementations: ACL and SBAR, indicating that MLP-aware replacement works with both existing (ACL) and proposed (SBAR) cost-sensitive polices. However, SBAR has higher performance and substantially lower hardware overhead than ACL , which makes SBAR a much more favorable

candidate for implementing MLP-aware cache replacement. The cost-sensitive policy employed by ACL requires a shadow directory on a per-set basis. For the baseline 16-way cache, ACL needs a 15-way shadow directory [30]. Assuming a 40-bit physical address space, each entry in the shadow directory needs four bytes of storage (24-bit tag + 1 valid bit + 4 LRU bits + 3 cost bits = 4B). Thus, the total overhead of the shadow directory is 60kB (4B/entry * 15 entries/set * 1024 sets = 60 kB). Comparatively, the overhead of SBAR is only 1856B (see Table 4), which is 33 times smaller than the overhead of ACL. Because ACL requires shadow directory information on a per-set basis, it is not straightforward to use dynamic set sampling to reduce the storage overhead of ACL.

5.7 Summary

Memory Level Parallelism (MLP) varies across different misses of an application, causing some misses to be more costly on performance than others. The non-uniformity in the performance impact of cache misses can be exposed to the cache replacement policy so that it can improve performance by reducing the number of costly misses. Based on this observation, we propose MLP-aware cache replacement. We present a run-time technique to compute the MLP-based cost for each cache block. This cost metric is used to drive cost-sensitive cache replacement policies. We also extend the Sampling Based Adaptive Replacement (SBAR) to dynamically choose between an MLP-aware replacement policy (LIN) and a traditional (LRU) replacement policy, depending on which one is providing fewer memory related stalls.

Chapter 6

Utility Based Partitioning of Shared Caches

This chapter investigates the problem of partitioning a shared cache between multiple concurrently executing applications. The commonly used LRU policy implicitly partitions a shared cache on a demand basis, giving more cache resources to the application that has a high demand and fewer cache resources to the application that has a low demand. However, a higher demand for cache resources does not always correlate with a higher performance from additional cache resources. It is beneficial for performance to invest cache resources in the application that benefits more from the cache resources rather than in the application that has more demand for the cache resources.

This chapter proposes *Utility-Based Cache Partitioning (UCP)*, a low-overhead, runtime mechanism that partitions a shared cache between multiple applications depending on the reduction in cache misses that each application is likely to obtain for a given amount of cache resources. The proposed mechanism monitors each application at runtime using a novel, cost-effective, hardware circuit that requires less than 2kB of storage. The information collected by the monitoring circuits is used by a partitioning algorithm to decide the amount of cache resources allocated to each application. Our evaluation, with 20 multiprogrammed workloads, shows that UCP improves performance of a dual-core system by up to 23% and on average 11% over LRU-based cache partitioning.

91

6.1 Introduction

Modern processors contain multiple cores which enables them to concurrently execute multiple applications (or threads) on a single chip. As the number of cores on a chip increases, the pressure on the memory system to sustain the memory requirements of all the concurrently executing applications (or threads) increases. One of the keys to obtaining high performance from multicore architectures is to manage the largest level on-chip cache efficiently so that off-chip accesses are reduced. This chapter investigates the problem of partitioning the shared largest-level on-chip cache among multiple competing applications.

Traditional design for on-chip cache uses the LRU (or an approximation of LRU) policy for replacement decisions. The LRU policy implicitly partitions a shared cache among the competing applications on a demand[1] basis, giving more cache resources to the application that has a high demand and fewer cache resources to the application that has a low demand. However, the benefit (in terms of reduction in misses or improvement in performance) that an application gets from cache resources may not directly correlate with its demand for cache resource. For example, a streaming application can access a large number of unique cache blocks but these blocks are unlikely to be reused again if the working set of the application is greater than the cache size. Although such an application has a high demand, devoting a large amount of cache will not improve its performance. Thus, it makes sense to partition the cache based on how much the application is likely to benefit from the cache rather than the application's demand for the cache.

---

[1]Demand is determined by the number of unique cache blocks accessed in a given interval [19]. Consider two applications A and B sharing a fully-associative cache containing N blocks. Then with LRU replacement, the number of cache blocks that each application receives is decided by the number of unique blocks accessed by each application in the last N unique accesses to the cache. If U_A is the number of unique blocks accessed by application A in the last N unique accesses to the cache, then application A will receive U_A cache blocks out of the N blocks in the cache.
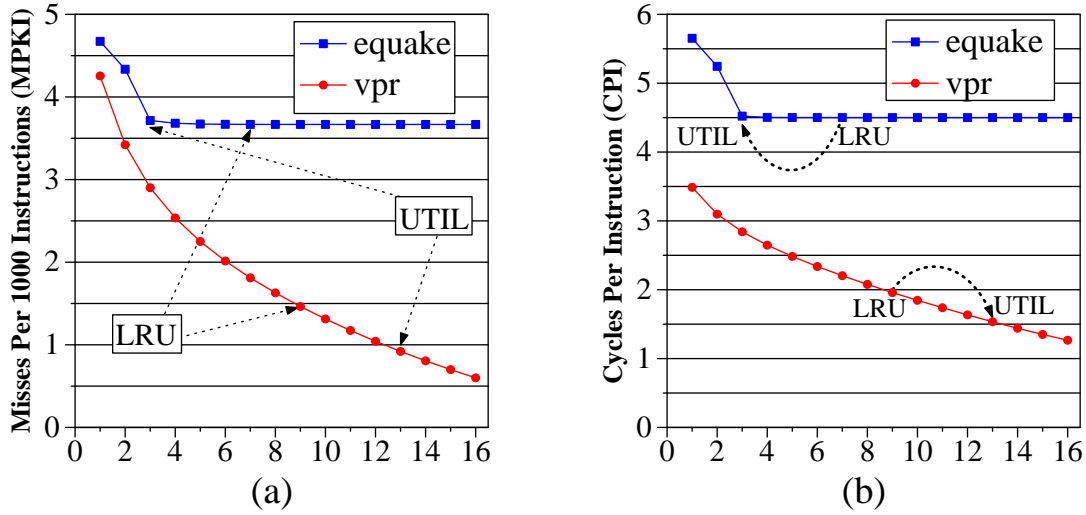
Figure 6.1: A case for utility based cache partitioning : (a) MPKI and (b) CPI as cache size is varied when vpr and equake are executed separately. The horizontal axis shows the number of ways allocated from a 16-way 1MB cache (remaining ways are turned off).

We explain the problem with LRU-based partitioning with a numerical example. Figure 6.1(a) shows the number of misses for two SPEC benchmarks, vpr and equake, as the cache size is varied, when each one is run separately. We vary the cache size by changing the number of ways and keeping the number of sets constant. The baseline L2 cache in our experiments is 16-way, 1MB in size and contains 1024 sets (other parameters of the experiment are described in Section 6.4). For vpr, the number of misses reduce monotonically as the cache size is increased from 1 way to 16 ways. For equake, the number of misses decrease as the number of allocated ways increase from 1 to 3, but increasing the cache size by more than 3 ways does not decrease misses. Thus, equake has no benefit or *utility* for cache resources in excess of three ways.

When vpr and equake are run together on a dual-core system, sharing the baseline 1MB 16-way cache, the LRU policy allocates, on average, 7 ways to equake and 9 ways

to vpr. If cache partitioning was based on utility (UTIL) of cache resources, then equake would get only 3 ways and vpr would get the remaining 13 ways. Decreasing the cache resources devoted to equake from 7 ways to 3 ways does not increase its misses but increasing the cache resources devoted to vpr from 9 ways to 13 ways reduces its misses. As shown in Figure 6.1(b), partitioning the cache based on utility information can potentially reduce the CPI of vpr from 2 to 1.5 without affecting the CPI of equake, improving the overall performance of the dual-core system.

To partition the cache based on application's utility for the cache resource, we propose *Utility-Based Cache Partitioning (UCP)*. An important component of UCP is the monitoring circuits that can obtain the information about utility of cache resource for all the competing applications at runtime. For the UCP scheme to be practical, it is important that the utility monitoring (UMON) circuits are not hardware-intensive or power-hungry. Section 6.3 describes a novel, low-overhead, UMON circuit that requires a storage overhead of only 1920B (less than 0.2% area of the baseline 1MB cache). The information collected by UMON is used by a partitioning algorithm to decide the amount of cache allocated to each competing application. Our evaluation in Section 6.5 shows that UCP outperforms LRU, improving the performance of a dual-core system by up to 23%, and on average 11%.

The number of possible partitions increases exponentially with the number of applications sharing the cache. It becomes impractical for the partitioning algorithm to find the best partition by evaluating every possible partition, when a large number of applications share a highly associative cache. In Section 6.6, we propose the *Lookahead Algorithm* as a scalable alternative to evaluating all the possible partitions for partitioning decisions.

6.2 Motivation and Background

Caches improve performance by reducing the number of main memory accesses. Thus, the utility of cache resources for an application can be directly correlated to the change in the number of misses or improvement in performance of the application when the cache size is varied. Figure 6.2, Figure 6.3, and Figure 6.4 shows the misses and CPI for some of the SPEC benchmarks as a function of cache size. The utility of cache resource varies widely across applications. The applications are classified into three categories based on how much each of them benefits as the cache size is increased from 1 way to 16 ways (keeping the number of sets constant).
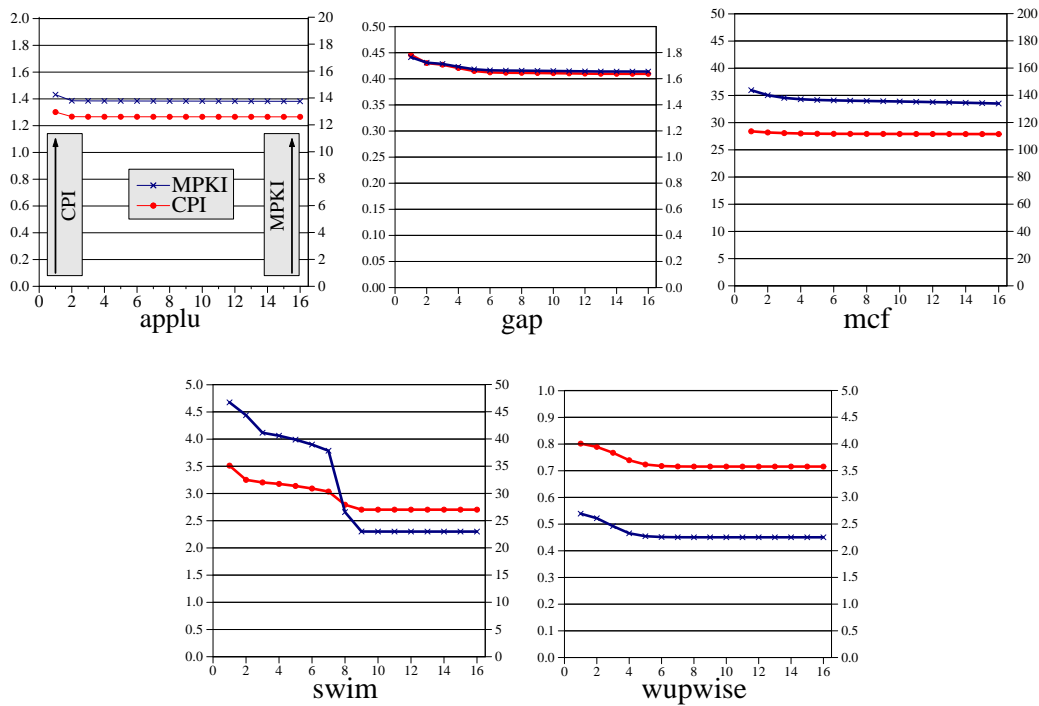
Figure 6.2: MPKI and CPI for Low Utility Benchmarks. The horizontal axis shows the number of ways allocated from a 16-way 1MB cache (the remaining ways are turned off).

Figure 6.2 contains benchmarks that do not benefit significantly as the cache size is increased from 1 way to 16 ways. We say such applications have *low* utility. These benchmarks either have large fraction of compulsory misses (e.g. gap) or have a data set larger than the cache size[2] (e.g. mcf).
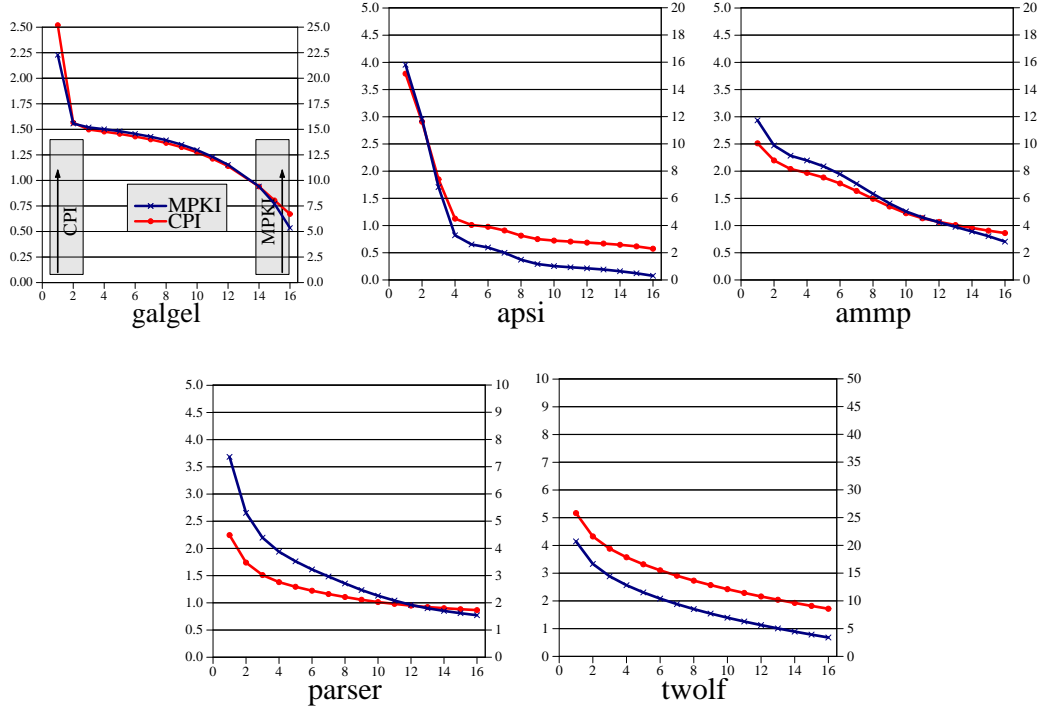
Figure 6.3: MPKI and CPI for High Utility Benchmarks. The horizontal axis shows the number of ways allocated from a 16-way 1MB cache (the remaining ways are turned off).

Benchmarks shown in Figure 6.3 continue to benefit as the cache size is increased from 1 way to 16 ways. We say such applications have *high* utility. Benchmarks in Figure 6.4 benefit significantly as the cache size is increased from 1 way to 8 ways. These

---

[2]Applications with low utility can show a large reduction in misses when the cache size is increased such that the dataset fits in the cache. For example, Figure 6.14 shows that the MPKI of art does not decrease when the cache size is increased from 1 way to 8 ways (0.5MB). However, increasing the size to 24 ways (1.5MB) reduces MPKI by a factor of 5. In such cases, the curve of MPKI vs. cache size resembles a step function.

benchmarks have a small working set that fits in a small size cache, therefore, giving them more than 8 ways does not significantly improve their performance. We say such applications have *saturating* utility.
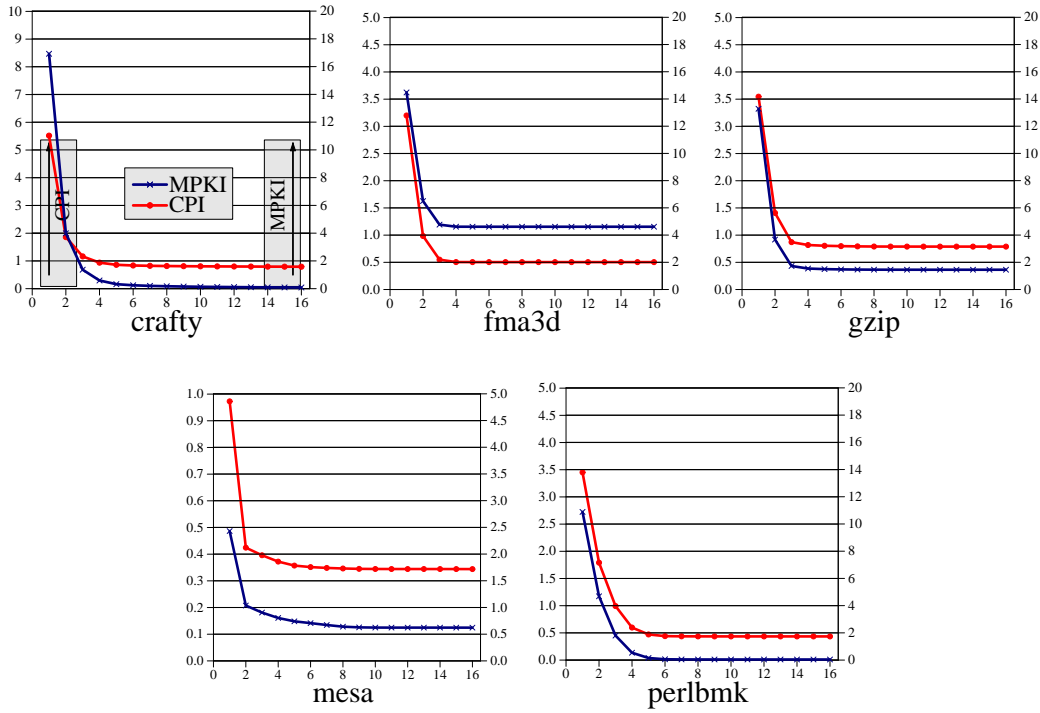
Figure 6.4: MPKI and CPI for Saturating Utility Benchmarks. The horizontal axis shows the number of ways allocated from a 16-way 1MB cache (the remaining ways are turned off).

If two applications having low utility (e.g. mcf and applu) are executed together, then their performance is not sensitive to the amount of cache available to each application. Similarly, when two applications of saturating utility are executed together, then the cache can support the working set of both applications. However, when an application with saturating utility is run with an application with low utility then the cache may not hold the working set of the application with saturating utility. Similarly, when an application with high utility is run with any other application, its performance is highly sensitive to

the amount of cache available to it. In such cases, it is important to partition the cache judiciously by taking utility information into account.

Figure 6.2, Figure 6.3 and Figure 6.4 shows that in most cases,[3] reduction in misses correlates with reduction in CPI. Thus, we can use the information about reduction in misses to make cache partitioning decisions. To include utility information in partitioning decisions, we provide a quantitative definition of utility for cache resources for a given application. Since cache is allocated only on a way basis in our studies, we define utility on a way granularity. If $miss_a$ and $miss_b$ are the number of misses that an application incurs when it receives a and b ways respectively $(a < b)$, then the utility (U_a^b) of increasing the number of ways from a to b is:

$$U_a^b = miss_a - miss_b \qquad (6.1)$$

Section 6.3 describes cost-effective monitoring circuits that can estimate the utility (U) information for an application at run-time, along with the framework, the partitioning algorithm, and the replacement scheme for UCP.

6.3 Utility-Based Cache Partitioning

6.3.1 Framework

Figure 6.5 shows the framework to support UCP between two applications that execute together on a dual-core system. One of the two applications execute on CORE1 and the other on CORE2. Each core is assigned a utility monitoring (UMON) circuit that tracks the utility information of the application executing on it. The UMON circuit is separated

---

[3]When eight ways are allocated to swim, it sees a huge reduction in misses. However, this reduction in misses does not translate into a substantial reduction in CPI. This happens because a set of accesses with high memory-level parallelism (MLP) now fits in the cache which reduces the average MLP and increases the average mlp-based cost of each miss.

from the shared cache, which allows the UMON circuit to obtain utility information about an application for all the ways in the cache, independent of the contention from the application executing on the other core. The partitioning algorithm uses the information collected by the UMON to decide the number of ways to allocate to each core. The replacement engine of the shared cache is augmented to support the partitions allocated by the partitioning algorithm.
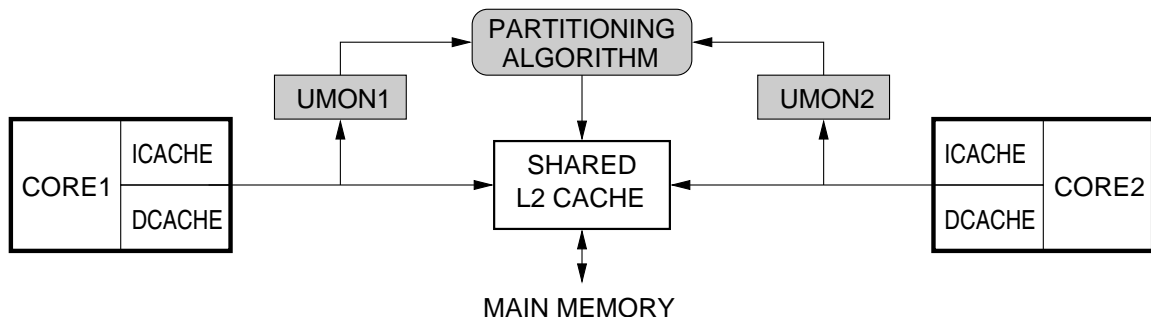
Figure 6.5: Framework for Utility-Based Cache Partitioning. Newly added structures are shaded.

6.3.2 Utility Monitors (UMON)

Monitoring the utility information of an application requires a mechanism that tracks the number of misses for all possible number of ways. To compute the utility information for the baseline 16-way cache, the monitoring circuit is required to track misses for all the sixteen cases, ranging from when only 1 way is allocated to the application to when all 16 ways are allocated to the application. A straight-forward, but expensive, method to obtain this information is to have sixteen tag directories, each having the same number of sets as the shared cache, but each having a different number of ways ranging from 1 way to 16 ways (note that data lines are not required to estimate hit-miss information). Although this scheme can track utility information for any replacement scheme implemented in the

shared cache, the hardware overhead of multiple directories makes this scheme impractical. Fortunately, the baseline LRU policy obeys the stack property [46], which means that an access that hits in a LRU managed cache containing N ways is guaranteed to also hit if the cache had more than N ways (the number of sets being constant). This means even with a single tag directory containing sixteen ways, it is possible to compute the hit-miss information about all the cases when the cache contains from one way through sixteen ways. To see how the stack algorithm provides utility information, consider the example of a four way set-associative cache shown in Figure 6.6(a).
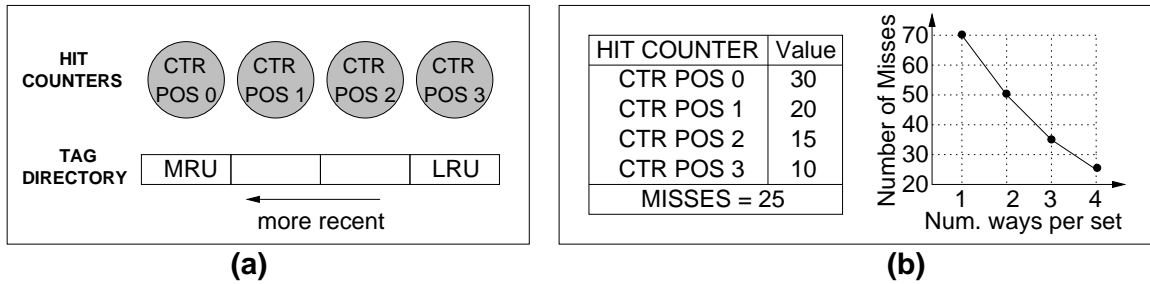
Figure 6.6: Tracking utility information using stack property: (a) Hit counters for each recency position (b) Obtaining utility information from hit counters using stack property.

Each set has four counters for obtaining the hit counts for each of the four recency positions ranging from MRU to LRU. The position next to MRU in the recency position is referred as *position 1* and the next position as *position 2*. If a cache access results in a hit, the counter corresponding to the hit-causing recency position is incremented. The counters then represent the number of misses saved by each recency position. Figure 6.6(b) shows an example in which out of the 100 accesses to the cache, 25 miss, 30 hit in MRU, 20 hit in position 1, 15 hit in position 2, and the remaining 10 hit in the LRU position. Then, if the cache size is reduced from four ways to three ways, the misses increase from 25 to 35. Further reducing the cache size to two ways, increases the number of misses to 50. And

with only one way the cache incurs 70 misses. Thus, given information about misses in a cache that has a large number of ways, it is possible to obtain the information about misses for a cache with smaller number of ways.

The UMON circuit tracks the utility of each way using an Auxiliary Tag Directory (ATD) and hit counters. The ATD has the same associativity as the main tag directory of the shared cache and uses the LRU policy for replacement decisions. Figure 6.7 (a) shows a UMON that contains the hit counters for each set in the cache. We call this organization as *UMON-local*. Although UMON-local can perform partitioning on a per-set basis, it requires a huge overhead because of the extra tag entry and hit counter for each line in the cache. The hardware overhead of the hit counters in UMON-local can be reduced by having one group of hit counters for all the sets in the cache. This configuration, shown in Figure 6.7(b), is called *UMON-global*. UMON-global enforces a uniform partition for all the sets in the cache. Compared to UMON-local, UMON-global reduces the number of hit counters required to implement UMON by a factor of number of sets in the cache. However, the number of tag entries required to implement UMON still remains equal to the number of lines in the cache.

6.3.3 Reducing Storage Overhead Using DSS

The number of UMON circuits in the system is equal to the number of cores. For the UMON circuit to be practical, it is important that it requires low hardware overhead. UMON-global requires an extra tag entry for each line in the cache. If each tag entry is 4 bytes then the UMON overhead per cache line is 8 bytes for a two-core system and 16 bytes for a four-core system. Considering that the baseline cache is 64 byte in size, the overhead of UMON-global is still substantial. To reduce the overhead of UMON, we use the *Dynamic Set Sampling (DSS)* concept proposed in Chapter 3. The key idea behind DSS is that the behavior of the cache can be approximated by sampling only a few sets. We can
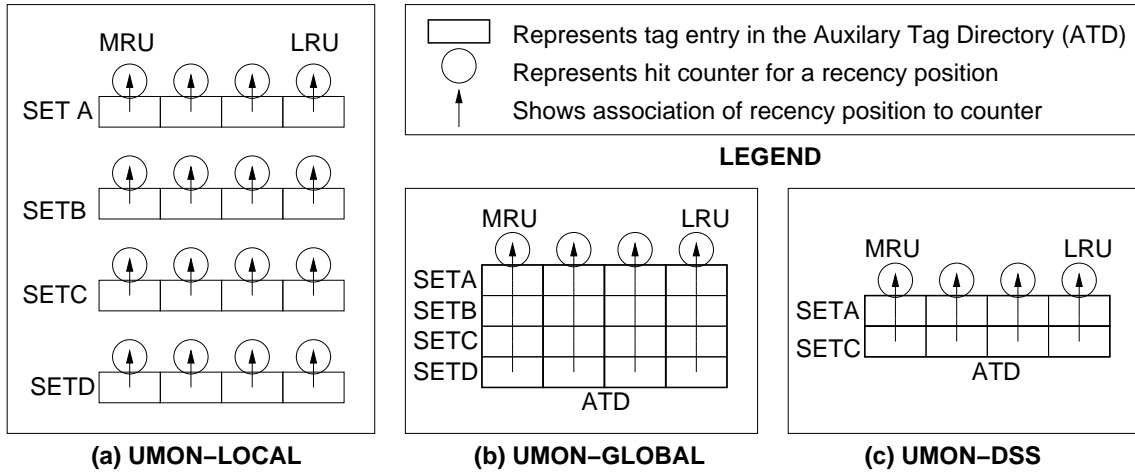
Figure 6.7: Utility Monitors: (a) UMON-local (b) UMON-global (c) UMON implemented with Dynamic Set Sampling (DSS).

use DSS to approximate the hit counter information of UMON-global by sampling few sets in the cache. Figure 6.7(c) shows the UMON circuit with Dynamic Set Sampling (UMON-DSS). The ATD in UMON-DSS contains ATD entries only for two sets A and C instead of all the four sets in the cache. An important question is that how many sampled sets are required for UMON-DSS to approximate the performance of UMON-global? We derive analytical bounds[4] for UMON-DSS in the next section and in Section 6.5.5, we compare the performance of UMON-DSS with UMON-global.

---

[4]DSS was used in Chapter 3 to choose between two replacement policies. Thus, it was used to approximated a global decision which had a binary value (one of the two replacement policy) by using the binary decisions obtained on the sampled sets. We are interested in approximating the global partitioning decision which is a discrete value (how many ways to allocate) by using the hit counter information of the sampled sets. Therefore the bounds derived in Chapter 3 are not applicable to the proposed mechanism.

6.3.4 Analytical Model for Dynamic Set Sampling

Let there be two applications A and B competing for a cache containing S sets. Let $a(i)$ denote the number of ways that application A receives for a given set i, if the partitioning is done on a per-set basis. Then if $a(i)$ does not vary across sets then even with a single set UMON-DSS can approximate UMON-global. However, $a(i)$ may vary across sets. The number of ways allocated to application A by UMON-global (u_g) can be approximated as the average of all $a(i)$, assuming that UMON-global gives equal importance to all the sets. Thus,

$$u_g = \sum_{i=1}^{S} a(i)/S \qquad (6.2)$$

Let n be the number of randomly selected sets sampled by UMON-DSS. Let u_s be the number of ways allocated to application A by UMON-DSS. We are interested in bounding the value of $|u_s - u_g|$ to some threshold ϵ. If σ^2 is the variance in the values of $a(i)$ across all the sets, then by Chebyshev's inequality [68]:

$$P(|u_s - u_g| \geq \epsilon) \leq \sigma^2/(n \cdot \epsilon^2) \qquad (6.3)$$

For bounding u_s to within one way of u_g, $\epsilon = 1$.

$$P(u_s \ is \ at \ least \ one \ way \ from \ u_g) \leq \sigma^2/n \qquad (6.4)$$

$$P(u_s \ is \ within \ one \ way \ from \ u_g) > 1 - (\sigma^2/n) \qquad (6.5)$$

As Chebyshev's inequality considers only variance without making any assumption about the distribution of the data, the bounds obtained from Chebyshev's inequality are pessimistic[5][68]. Figure 6.8 shows the lower bound provided by Chebyshev's inequality as the number of sampled sets is varied, for different values of variance. For most of

---

[5]In general, much tighter bounds can be obtained if the mean and the distribution of the sampled data are known [68].
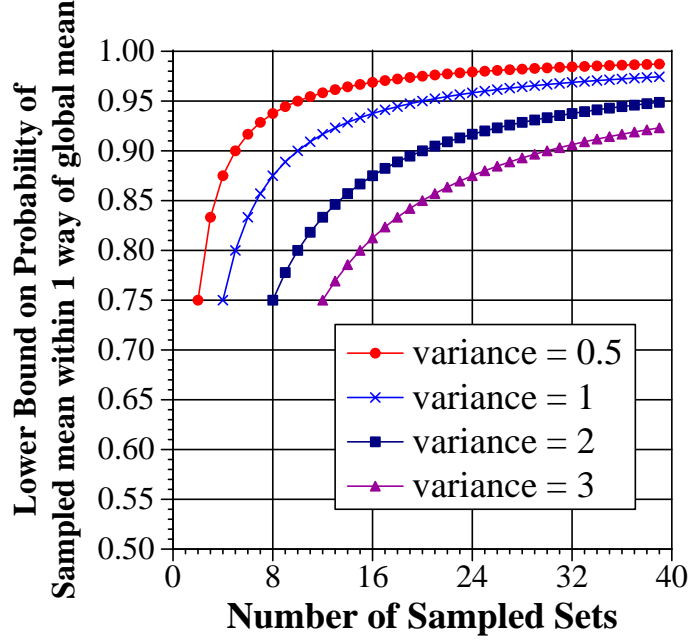
Figure 6.8: Bounds on Number of Sampled Sets

the workloads studied, the value of variance (σ^2) is less than 3, indicating that even with the pessimistic bounds, as few as 32 sets are sufficient for UMON-DSS to approximate UMON-global. We compare UMON-DSS to UMON-global in Section 6.5.5. Unless stated otherwise, we use 32 sets for UMON-DSS. The sampled sets for UMON-DSS are chosen using the simple static policy [63], which means set 0 and every 33rd set is selected. For the remainder of the chapter UMON by default means UMON-DSS.

6.3.5 Partitioning Algorithm

The partitioning algorithm reads the hit counters from all the UMON circuits of each of the competing applications. The partitioning algorithm tries to minimize the total number of misses incurred by all the applications. The utility information in the hit counters directly correlates with the reduction in misses for a given application when given a

fixed number of ways. Thus, reducing the most number of misses is equivalent to maximizing the combined utility. If A and B are two applications with utility functions *UA* and *UB* respectively, then for partitioning decisions, the combined utility (U_{tot}) of A and B is computed for all possible partitions for the baseline 16-way cache:

$$U_{tot}(a) = UA_1^i + UB_1^{(16-i)} \ ... \ For \ i = \ 1 \ to \ (16-1) \qquad (6.6)$$

The partition that gives the maximum value for U_{tot} is selected. In our studies, we guarantee that the partitioning algorithm gives at least one way to each application. We invoke the partitioning algorithm once every five million cycles (a design choice based on simulation results). After each partitioning interval, the hit counters in all UMONs are halved. This allows the UMON to retain past information while giving importance to recent information.

6.3.6 Changes to Replacement Policy

To incorporate the decisions made by the partitioning algorithm, the baseline LRU policy is augmented to enable way partitioning [13][78][28]. To implement way partitioning, we add a bit to the tag-store entry of each block to identify the core which installed the block in the cache. On a cache miss, the replacement engine counts the number of cache blocks that belong to the miss-causing application in the set. If this number is less than the number of blocks allocated to the application, then the LRU block among all the blocks that do not belong to the application is evicted. Otherwise, the LRU block among all the blocks of the miss-causing application is evicted.

If the number of ways allocated to an application is increased by the partitioning algorithm, then these added ways are consumed by the application only on cache misses. This gradual change of partitions allows the cache to retain the cache blocks till they are required by the application that is allocated the cache space.

6.4 Experimental Methodology

6.4.1 Multicore System Configuration

Table 6.1 shows the parameters of the baseline configuration used in our experiments. We use an in-house simulator that models the alpha ISA. The processor core is 8-wide issue, out-of-order, with 128-entry reservation station. The first-level instruction cache and data cache are private to the processor core. The processor parameters are kept constant in our study. This allows us to use a fast event-driven processor model to reduce simulation time. Because our study deals with the memory system we model the memory system in detail. DRAM bank conflicts and bus queuing delays are modeled. The baseline L2 cache is shared among all the processor cores and uses LRU replacement. Thus, the L2 cache gets partitioned among all the competing cores on a demand basis.

Table 6.1: Multicore System Configuration.

Processor core	8 wide, out-of-order, with 128 entry reservation station; 64 kB hybrid branch predictor with 4k-entry BTB minimum branch misprediction penalty of 15 cycles.
	L1 Icache and Dcache :16kB, 64B line-size, 4-way, LRU. The L1 caches are private to each core.
Unified Shared L2 Cache	1MB, 64B line-size, 16-way with LRU replacement, 15-cycle hit, 32-entry MSHR, 128-entry store buffer. L2 cache is shared among all the cores
Memory	32 DRAM banks; 400-cycle access latency; bank conflicts modeled; maximum 32 outstanding requests
Bus	16B-wide split-transaction bus at 4:1 frequency ratio. queueing delays modeled

6.4.2 Multicore Performance Metrics

There are several metrics to quantify the performance of a system in which multiple applications execute concurrently. We discuss the three metrics commonly used in the

literature: weighted speedup, sum of IPCs, and harmonic mean of normalized IPCs. Let IPC_i be the IPC of the ith application when it concurrently executes with other applications and $SingleIPC_i$ be the IPC of the same application when it executes in isolation. Then, for a system in which N threads execute concurrently, the three metrics are given by:

$$Weighted\ Speedup = \sum(IPC_i/SingleIPC_i) \tag{6.7}$$

$$IPC_{sum} = \sum IPC_i \tag{6.8}$$

$$IPC_{norm\_hmean} = N/\sum(SingleIPC_i/IPC_i) \tag{6.9}$$

The *Weighted Speedup* metric indicates reduction in execution time. The IPC_{sum} metric indicates the throughput of the system but it can be unfair to a low IPC application. The IPC_{norm_hmean} metric balances both fairness and performance [45]. We will use *Weighted Speedup* as the metric for quantifying the performance of multicore configurations throughout the chapter. Evaluation with the IPC_{sum} and IPC_{norm_hmean} metric will also be discussed for some of the key results in the chapter.

6.4.3 Multi-programmed Workloads

We use benchmarks from the SPEC CPU2000 suite for our studies. A representative slice of 250M instructions is obtained for each benchmark using a tool that we developed using the SimPoint methodology [58]. Two separate benchmarks are combined to form one multiprogrammed workload that can be run on a dual-core system. To include a wide variety of multiprogrammed workload in our study, we classify the multiprogrammed workloads into five categories. Workloads with *Weighted Speedup* for the baseline configuration between 1 and 1.2 are classified as *Type A*, between 1.2 and 1.4 as *Type B*, between 1.4 and 1.6 as *Type C*, between 1.6 and 1.8 as *Type D*, and between 1.8 and 2 as *Type E*. A suite containing 20 workloads is created by using four workloads from each of the five categories.

Simulation for a dual-core system is continued until both benchmarks in the multiprogrammed workload execute at least 250M instructions each. If a benchmark finishes the stipulated 250M instruction before the other benchmark finishes 250M instruction, it is restarted so that the two benchmarks continue to compete for the L2 cache throughout the simulation. Table 6.2 shows the classification based on baseline weighted speedup (BaseWS), Misses Per 1000 Instruction (MPKI) and Cycles Per Instruction (CPI) for the baseline dual-core configuration for all the 20 workloads. The benchmark names for ammp (amp), swim (swm), perlbmk (perl), and wupwise (wup) are abbreviated.

Table 6.2: Multi-programmed Workload Summary

Category (BaseWS)	Workload Bmk1-Bmk2	MPKI Bmk1	MPKI Bmk2	CPI Bmk1	CPI Bmk2
TYPE A (1.0-1.2)	galgel-vpr	11.84	8.41	1.25	2.55
	galgel-twolf	11.46	11.44	1.20	3.51
	amp-galgel	6.91	10.62	1.74	1.21
	apsi-galgel	3.08	10.82	1.14	1.19
TYPE B (1.2-1.4)	twolf-vpr	8.76	6.22	2.81	2.06
	apsi-twolf	2.05	7.51	0.93	2.61
	amp-art	6.73	43.73	1.72	4.90
	apsi-art	2.91	43.12	1.12	4.76
TYPE C (1.4-1.6)	apsi-swm	2.71	22.98	1.05	2.89
	amp-applu	6.71	13.76	1.69	1.28
	swm-twolf	22.98	10.64	2.73	3.26
	art-parser	42.75	3.48	4.52	1.33
TYPE D (1.6-1.8)	equake-vpr	18.33	5.74	4.57	1.97
	vpr-wup	5.40	2.25	1.89	0.72
	gzip-twolf	1.61	5.36	0.84	2.17
	art-crafty	41.10	0.63	4.33	0.96
TYPE E (1.8-2.0)	fma3d-swm	4.62	23.53	0.51	2.94
	mcf-applu	134	13.76	28.5	1.27
	gap-mesa	1.66	0.62	0.41	0.35
	crafty-perl	0.14	0.04	0.81	0.44

6.5 Results and Analysis

6.5.1 Performance on Weighted Speedup Metric

We compare the performance of UCP to two partitioning schemes: LRU and Half-and-Half. The Half-and-Half scheme statically partitions the cache equally among the two competing applications. The disadvantage of the Half-and-Half scheme over LRU is that it cannot change the partition in response to the varying demands of competing applications. However, it also has the advantage of performance isolation, which means that the performance of an application does not degrade substantially when it executes concurrently with a badly behaving application. Figure 6.9 shows the weighted speedup of the three partitioning schemes. The bar labeled *gmean* represents the geometric mean of the individual weighted-speedup of all the 20 workloads.
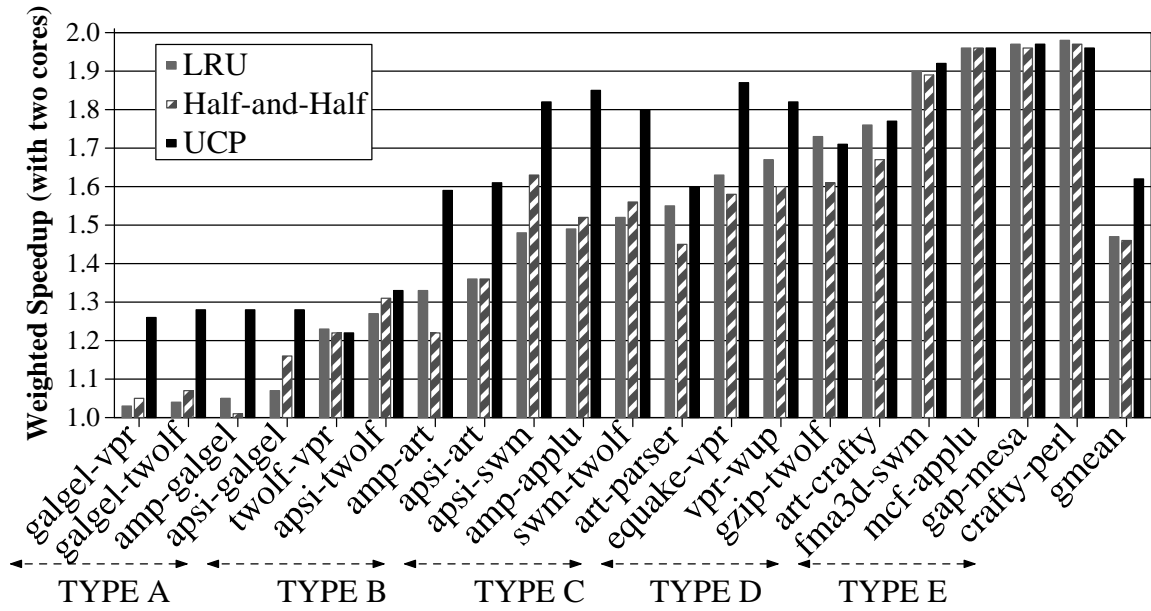
Figure 6.9: Performance of LRU, Half-and-Half, and UCP.

LRU performs better than Half-and-Half for some workloads and for some Half-and-Half performs better than LRU. For most workloads, UCP outperforms the best-performing scheme out of the other two schemes. On average, UCP improves performance by 10.96% over the baseline LRU policy, increasing the geometric mean weighted speedup from 1.46 to 1.62.

The Type A category contains workloads where both benchmarks in the workload have high utility and high demand for the L2 cache. Therefore, the baseline LRU policy has a value of weighed speedup that is almost half of the ideal value of 2. Partitioning the cache based on utility, rather than demand, improves performance noticeably. For example, UCP increases the weighted speedup for the workload galgel-twolf from 1.04 to 1.28.

The Type C category contains workloads where one benchmark has high utility and the other has low utility. In such cases, UCP allocates most of the cache resource to the application with high utility, thus improving the overall performance. For example, for amp-applu, UCP allocates 14 or more ways out of the 16 ways to amp, improving the weighted speedup from 1.49 to 1.83.

When both benchmarks in the workload have low utility (e.g. mcf-applu), the performance of each benchmark in the workload is not sensitive to the amount of cache available to it, so the weighted speedup is close to ideal. Similarly, if the cache can accommodate the working set of both benchmarks in the workload, the weighted speedup for that workload is close to ideal. Such workloads are included in the Type E category. As the weighted speedup of these workloads is close to the ideal, UCP does not change performance significantly.

For twolf-vpr, crafty-perl, and gzip-twolf , UCP reduces performance marginally compared to LRU. This happens because UCP allocates partitions once every partition interval (5M cycles in our experiments), so it is unable to respond to the phase changes that occur at a finer granularity. On the other hand, LRU can respond to such fine-grained

change in behavior of the applications by changing the partitions potentially at every access. The LRU policy also has the advantage of doing the partitioning on a per-set basis depending on the demand on the individual set. On the other hand, the proposed UCP policy globally allocates a uniform partition for all the sets in the cache, sacrificing fine-grained, per-set, control for reduced overhead.

6.5.2 Performance on Throughput Metric

Figure 6.10 compares the performance of the baseline LRU policy to the proposed UCP policy for the throughput metric, $IPCsum$. To show the change in the IPC of the individual benchmark of each workload, the graph is drawn as a stacked bar graph. The IPC of the first benchmark that appears in the name of the workload is labeled as *IPC-Benchmark1*. The IPC of the other benchmark is labeled as *IPC-Benchmark2*. For example, for the workload galgel-vpr, IPC-Benchmark1 shows the IPC of galgel and IPC-Benchmark2 shows the IPC of vpr. The bar labeled *hmean* represents the harmonic mean of the IPC_{sum} of all the 20 workloads.

For 15 out of the 20 workloads, UCP improves the IPC_{sum} compared to the LRU policy. UCP can improve performance by improving the IPC of one benchmark in the workload without affecting the IPC of the other benchmark in the workload. Examples of such workloads are apsi-swm and equake-vpr. UCP can also improve the aggregate IPC by marginally reducing the IPC of one benchmark and significantly improving the IPC for the other benchmark. Examples include apsi-galgel and amp-art. For the IPC_{sum} metric, UCP reduces performance on two workloads, gzip-twolf and crafty-perl. On average, UCP improves the performance on the throughput metric by 16.8%, increasing the harmonic mean IPC_{sum} of the system from 1.21 to 1.41.
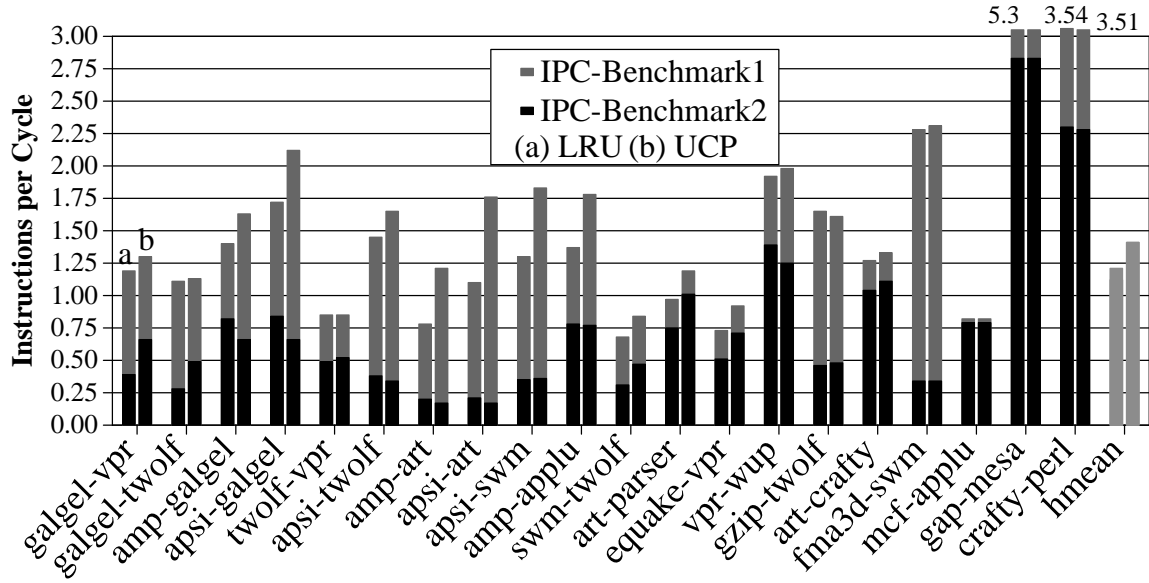
Figure 6.10: LRU (left bar) vs. UCP (right bar) on throughput metric.

6.5.3 Evaluation on Fairness Metric

A dynamic partitioning mechanism may improve the overall performance of the system at the expense of severely degrading the performance of one of the applications. The harmonic mean of the normalized IPCs is shown to consider both fairness and performance [45]. Figure 6.11 shows the performance of LRU, Half-and-Half, and UCP for this metric. The bar labeled gmean is the geometric mean over all the 20 workloads. UCP improves the average on this metric by 11% increasing the gmean from 0.71 to 0.79. Note that more improvement in this metric can be obtained by modifying the partitioning algorithm to directly favor fairness.

112
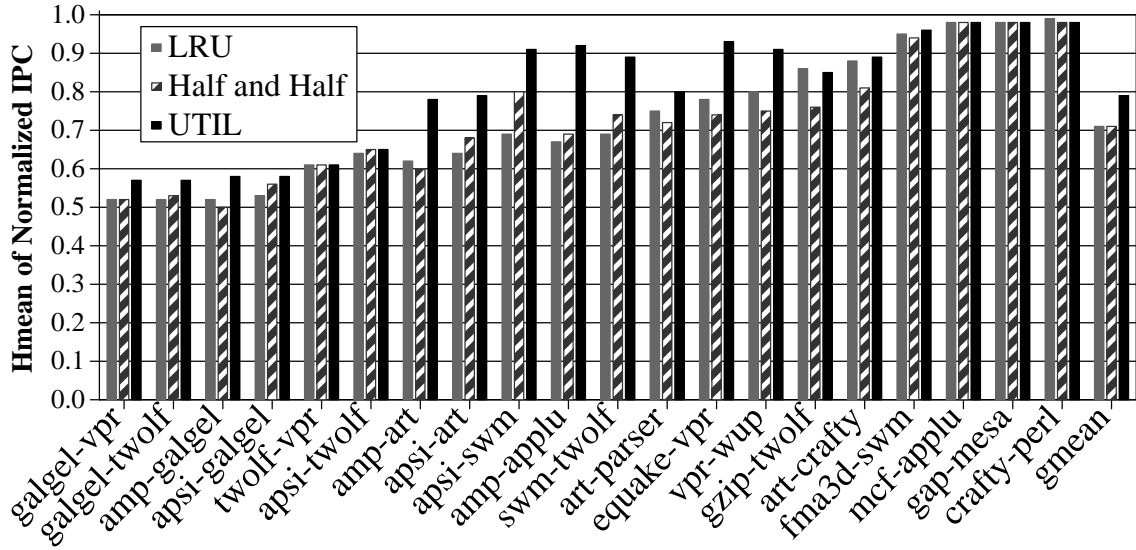
Figure 6.11: LRU, Half-and-Half, and UCP on fairness metric.

6.5.4 Phase-Based Adaptation of UCP

The utility of cache resources for an application can vary over time. The dynamic partitioning of UCP allows it to adapt to the time-varying phase behavior of the competing applications. The variation in utility for cache resources of an application may not correlate with its variation in demand for cache resources. We analyze the time varying phase behavior of the workload swim-twolf by comparing UCP and LRU to a partitioning scheme that statically allocates a fixed number of ways to each competing application. Figure 6.12(a) shows the MPKI of swim for the static partitioning scheme as the number of ways devoted to swim is varied.

With static partitioning, devoting less than 9 ways to swim increases its MPKI considerably. When swim and twolf are executed together, the baseline LRU policy allocates,

113
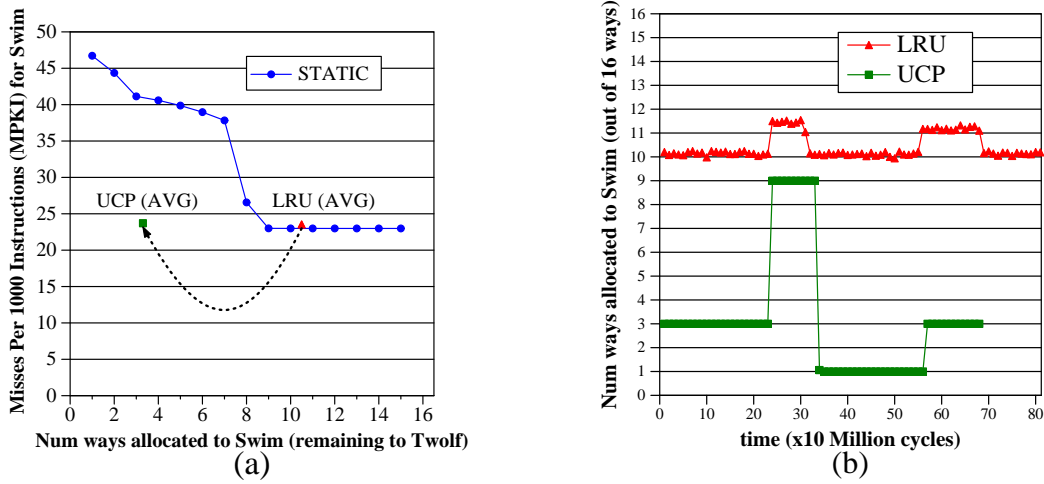
Figure 6.12: UCP vs. Static Partitioning (a) Variation in MPKI as the number of ways allocated to swim is changed statically. (b) The average number of ways dynamically allocated to swim when it is executed with twolf by the LRU policy and UCP policy.

on average[6], 10.5 ways to swim, whereas, the UCP policy allocates, on average, only 3.3 ways to swim. However, the MPKI of swim with the UCP policy (23.7) remains similar that with the LRU policy (22.98). This happens because UCP allocates ways to swim only in phases when the allocated ways are likely to reduce the misses. Figure 6.12(b) shows the number of ways allocated to swim over time by LRU and UCP. LRU consistently allocates 10 or more ways to swim throughout the simulation. UCP allocates 9 ways to swim only between 230M and 320M cycles of simulation, and three or fewer ways otherwise. As swim receives cache resources in the phase when not having them would increase MPKI considerably, the number of misses for swim does not increase compared to the LRU policy. Reducing the average number of ways of swim from 10.5 to 3.3 allows twolf to have 12.7 ways instead of 5.5 ways. This reduces the MPKI of twolf from 10.64 to 5.18.

---

[6]The average number of ways allocated to an application by the LRU policy is measured by sampling the cache every 2M cycles. The number of lines present in the cache for the given application is counted and this number is divided by the number of sets in the cache.

6.5.5   Effect of Varying the Number of Sampled Sets

We use 32 sets for each of the UMON circuit in the default UCP configuration. This section analyzes the sensitivity of the UCP mechanism to the number of sampled sets in the UMON. Figure 6.13 compares the performance of four UCP configurations: the first samples 8 sets, the second samples 16 sets, the third is the default UCP configuration with 32 sampled sets, and the last is the UMON-global configuration which contains all the sets.
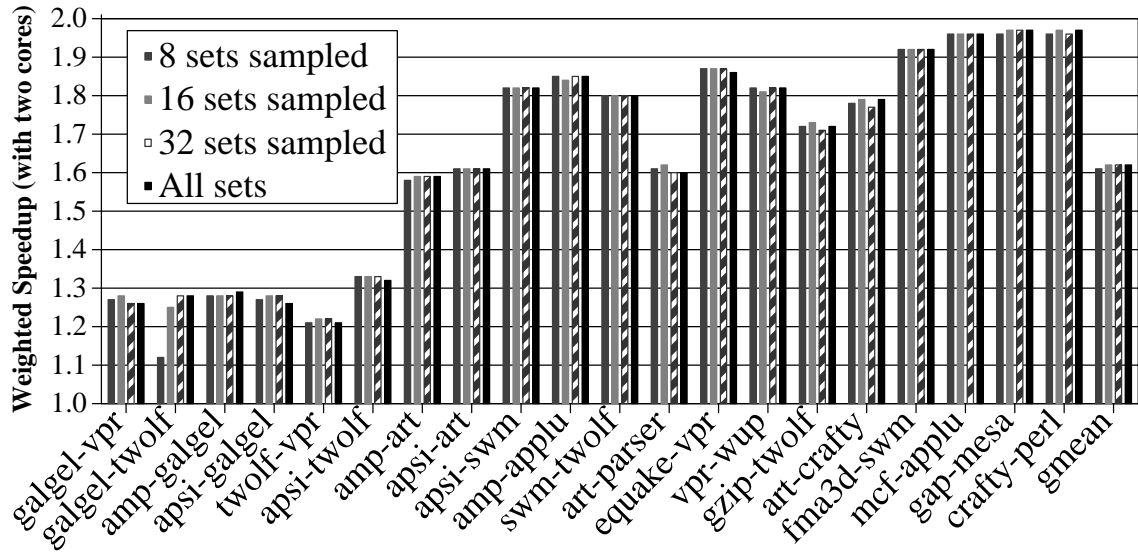
Figure 6.13: Effect of Number of Sampled Sets on UCP.

For all workloads, the default UCP configuration with 32 sampled sets performs similar to UMON-global (All sets). The performance of the workload galgel-twolf reduces if the number of sampled sets is reduced to 8. For other workloads, the performance of UCP is relatively insensitive to the number of sampled sets (for sampled sets ≥ 8). This is consistent with the lower bounds derived with the analytical model presented in Section 6.3.4. This result is particularly useful result as it means that default UCP configuration with only 32 sets performs similar to the UMON-global configuration without requiring the huge hardware overhead associated with the UMON-global configuration. This reduced overhead makes the UCP scheme practical. The next section quantifies the hardware overhead.

115

6.5.6 Hardware Overhead of UCP

The major source of hardware overhead of UCP is the UMON circuit. Table 6.3 details the storage overhead of UMON containing 32 sampled sets, assuming a 40-bit physical address space. Each UMON requires 1920 B of storage overhead (less than 0.2% of the area of the baseline 1MB cache), indicating that for the baseline dual-core configuration UCP requires less than 0.4% of storage overhead for implementing the UMON circuits. The low overhead for UMON means that the UCP scheme is cost-effective even if the number of core increases (e.g. UMON overhead of less than 1% with four cores). The storage overhead of UMON can further be reduced by using partial tags in the ATD. In addition to the storage bits, each UMON also contains an adder for incrementing the hit counters and a shifter to halve the hit counters after each partitioning interval.

Table 6.3: Storage Overhead of a UMON circuit with 32 Sets

Size of each ATD entry (1 valid bit + 24-bit tag + 4-bit LRU)	29 bits
Total number of ATD entries per sampled set (1/way * 16)	16
ATD overhead per sampled set (29 bits/way * 16 ways)	58 B
Total ATD overhead (32 sampled sets * 58 B/set)	1856 B
Overhead of hit counters (16 counters * 4B each)	64 B
Total UMON overhead (1856B + 64B)	1920 B
Area of baseline L2 cache (64kB tags + 1MB data)	1088 kB
% increase in L2 area due to 1 UMON (1920B/1088kB)	**0.17%**

Implementing way-partitioning on a dual-core system requires a bit in each tag-store entry to identify which of the two cores installed the line in the cache. The partitioning algorithm contains a comparator circuit and requires negligible storage. Note that none of the structures or operations required by UCP is in the critical path, resource-intensive, complex, or power hungry.

6.6 Scalable Partitioning Algorithm

We assumed that the partitioning algorithm is able to find the partition of maximum utility by computing the combined utility of all the applications for every possible partition. This is not a problem when there are only two applications, as an *N-way* cache can be way-partitioned among two applications in only *N+1* ways. However, the number of possible partitions increases exponentially as the number of competing applications, making it impractical to evaluate every possible partition. For example, a 32-way cache can be shared by four applications in $6,545$ ways, and by 8 applications in $15,380,937$ ways. Finding an optimal solution to the partitioning problem has been shown to be NP-hard [65]. In this section we develop a partitioning algorithm that has a worst-case time complexity of $N^2/2$.

6.6.1 Background

Our algorithm is derived from the greedy algorithm proposed in [76]. The *greedy algorithm* is shown in Algorithm 2. In each iteration, one block[7] is assigned to the application that has the maximum utility for that block. The iteration continues till all the blocks are assigned. This algorithm is shown to be optimal if the utility curves for all the competing applications are convex [76]. However, when the utility curves are non-convex, the greedy algorithm can have pathological behavior. Figure 6.14 shows example of two benchmarks, art and galgel, that has non-convex utility curve. Art shows no reduction in misses until it is assigned at least 8 blocks and after that it shows huge reduction in misses. As the greedy algorithm considers the gain from only the immediate one block it will not assign any blocks to art (unless the utility of that block for even the other application is zero). To address this shortcoming of the greedy algorithm, Suh et. al [78] propose to also invoke

---

[7]We use the term blocks instead of ways because the greedy algorithm was used in [76] to decide the number of cache blocks that each application receives in a fully associative cache. However, the explanation can also be thought of as assigning ways in a set-associative cache.

the greedy algorithm for each combination of the non-convex points of all applications. However, the number of times the greedy algorithm is invoked increases with the number of combinations on non-convex points of all the applications. Figure 6.14 shows that an application (galgel) can have as many as 15 non convex points, indicating that the number of combinations of non-convex points of all the competing applications can be very large. To avoid the time complexity, [78] suggests that the greedy algorithm be invoked only for some number of randomly chosen combination of non-convex points. However, for a given number of trials, the likelihood that randomization will yield the optimum partition reduces as the number of combinations increase.

Algorithm 2 Greedy Algorithm

---

balance = N   /* Num blocks to be allocated */
allocations[i] = 0 for each competing application i
   while(balance) **do**:
     foreach application i, **do**: /* get utility for next 1 block */
       alloc = allocations[i]
       Unext[i] = get_util_value(i, alloc, alloc+1)
     winner = application with maximum value of Unext
     allocations[winner]++
     balance = balance-1
 return allocations

get_util_value(p, a, b):
  U = change in misses for application p when the number
  of blocks assigned to it increases from a to b
  return U
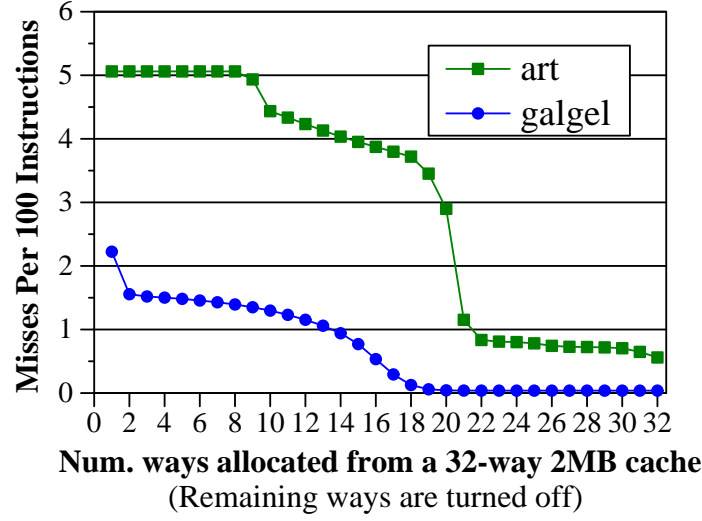
Figure 6.14: Benchmarks with non-convex utility curves

6.6.2 The Lookahead Algorithm

We define *marginal utility (MU)* as the utility U per unit cache resource. If $miss_a$ and $miss_b$ are the number of misses that an application incurs when it receives a and b blocks respectively, then the marginal utility, MU_a^b of increasing the blocks from a to b is defined as:

$$MU_a^b = (miss_a - miss_b)/(b - a) \; = \; U_a^b/(b - a) \qquad (6.10)$$

The basic problem with the greedy algorithm is that it considers the marginal utility of only the immediate block, and thus fails to see potentially high gains after the first block if there is no gain from the first block. If the algorithm could also take into account the gains from far ahead, then it could make better partitioning decisions. We propose the *Lookahead Algorithm*, which considers the marginal utility for all possible number of blocks that the application can receive. The pseudo code for the Lookahead algorithm is shown in Algorithm 3.

Algorithm 3 Lookahead Algorithm

---

balance = N   /* Num blocks to be allocated */
allocations[i] = 0 for each competing application i
   while(balance) **do**:
      foreach application i, **do**: /* get max marginal utility */
         alloc = allocations[i]
         max_mu[i] = get_max_mu(i, alloc, balance)
         blocks_req[i] = min blocks to get max_mu[i] for i
      winner = application with maximum value of max_mu
      allocations[winner] += blocks_req[winner]
      balance − = blocks_req[winner]
return allocations

get_max_mu(p, alloc, balance):
   max_mu = 0
   for(ii=1; ii<=balance; ii++) **do**:
      mu = get_mu_value(p, alloc, alloc+ii)
      if(mu > max_mu) max_mu = mu
   return max_mu

get_mu_value(p, a, b):
  U = change in misses for application p when the number
  of blocks assigned to it increases from a to b
  return U/(b-a)


In each iteration, the maximum marginal utility *(max_mu)* and the minimum number of blocks at which the max_mu occurs is calculated for each application. The application with highest value for max_mu is assigned the number of blocks it needs to obtain max_mu. Ties for highest value of max_mu are broken arbitrarily. The iterations are repeated until all blocks are assigned. The lookahead algorithm can assign a different number of blocks in each iteration and is guaranteed to terminate as at least one block is assigned in each iteration. For applications with convex utility function, the maximum value of marginal utility occurs for the first block. Therefore, if all the applications have convex utility function, then the lookahead algorithm behaves identical to the greedy algorithm, which is proved to be optimal for convex functions.

The step for obtaining the value of max_mu for each of the application is executed in parallel by the UMON circuits. Calculating the max_mu for an application if it could get up to N blocks takes N operations of add-divide-compare each. As the blocks are allocated, the number of blocks that an application can receive in an iteration reduces. In the worst case only one block is allocated in every iteration. Then, even in the worst case, the time required for the lookahead algorithm to allocate N blocks is: $N + (N - 1) + (N - 2) + ... + 1 = N(N - 1)/2 \approx N^2/2$ operations. In our studies, cache is assigned on a way granularity instead of a block granularity. Therefore, the value of N is equal to the associativity of cache. Thus, for partitioning a 32-way cache the lookahead algorithm will require a maximum time of 512 operations (recall that we perform partitioning once every 5M cycles). In our experiments, we ensure that both the greedy algorithm and the lookahead algorithm allocates at least one way to each of the competing applications.

121

6.6.3   Result for Partitioning Algorithms

We evaluate the partitioning algorithms on a quad-core system in which four applications share a 2MB 32-way cache. As there are four cores, the ideal value for weighted speedup is 4. Figure 6.15 shows the weighted speedup for the LRU policy, and the UCP policy with the three partitioning algorithms - greedy, lookahead, and EvalAll. The *EvalAll* algorithm evaluates all the possible partitions to find the best partition. The greedy algorithm works well when all the benchmarks in the workload have convex utility curves (mix1) or when the cache is big enough to support the working set of majority of the benchmarks in the workload (mix2). However, for workloads that contain benchmarks with non convex utility curves (mix3 and mix4), the greedy algorithm does not perform as well as the EvalAll algorithm. The lookahead algorithm performs similar to the EvalAll algorithm without requiring the associated time complexity.
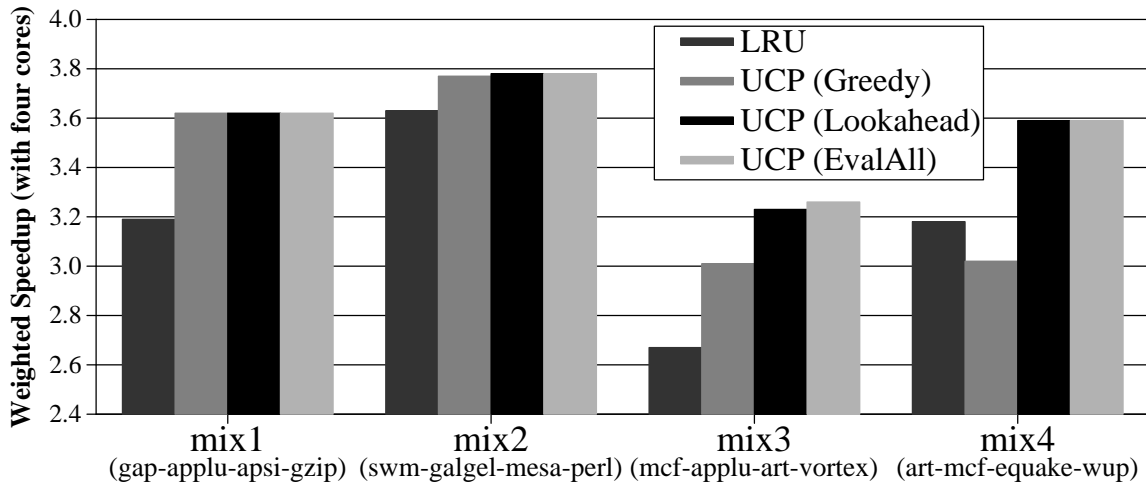
Figure 6.15: Comparison of Partitioning Algorithms

122

6.7 Related Work

6.7.1 Related Work in Cache Partitioning

Stone et al. [76] investigated optimal (static) partitioning of cache memory between two or more applications when the information about change in misses for varying cache size is available for each of the competing application. However, such information is hard to obtain statically for all applications as it may depend on the input set of the application. The objective of our study is to dynamically partition the cache by computing this information at runtime. Moreover, as shown in Section 6.5.4, dynamic partitioning can adapt to the time-varying phase behavior of the competing applications, which makes it possible for dynamic partitioning to out perform even the best static partitioning.

Dynamic partitioning of shared cache was first investigated by Suh et al. [79][78]. [78] describes a low-overhead scheme that uses recency position of the hits for the lines in the cache to estimate the utility of the cache for each application. However, obtaining the utility information from main cache has the following shortcomings: (1) The number of lines in each set for which the utility information can be obtained for a given application is also dependent on the other application. (2) The recency position at which the application gets a hit is also affected by the other application, which means that the utility information computed for an application is dependent on (and polluted by) the concurrently executing application. UCP avoids these problems by separating the monitoring circuit from the main cache so that the utility information of the application is independent of other concurrently executing applications. Figure 6.16 compares UCP to a scheme that uses in-cache information for estimating utility information. The in-cache scheme provides 4% average improvement compared to the 11% average improvement of UCP. Thus, seperating the monitoring circuit from the main cache is important to obtain high performance from dynamic partitioning. However, doing this by having extra tags for each cache line incurs prohibitive hardware overhead. Our proposal makes it practical to compute the utility infor-

mation for an application, independent of other competing applications, without requiring huge hardware overhead.
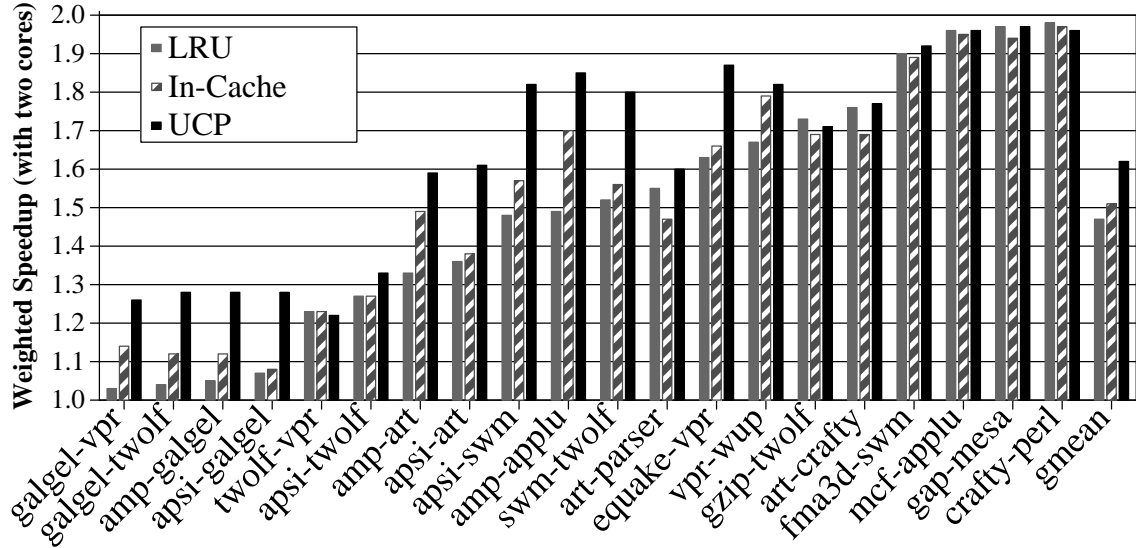
Figure 6.16: UCP vs. an In-cache monitoring scheme.

Mechanisms for enabling Quality of Service (QoS) in multicore and multithreaded architectures are discussed in [28]. It emphasizes that factors such as priority, locality, and latency sensitivity should be considered in dividing the cache among competing applications. It also describes different mechanisms to facilitate static and dynamic partitioning of cache between competing applications. However, the design of intelligent partitioning policies to use these mechanisms is left as an open research topic.

Recently, Hsu et al. [27] studied different policies, including a utilitarian policy, for partitioning a shared cache between competing applications. However, they analyzed these policies using best offline parameters and mechanisms for obtaining these parameters at runtime is left for future work.

6.7.2 Related Work in Cache Organization

Liu et al. [44] investigated cache organizations for CMPs. They proposed *Shared Processor-Based Split L2* Caches, in which the number of private banks allocated to each competing application is decided statically using profile information. However, it may be impractical to profile all the applications that execute concurrently. Our mechanism avoids profiling by computing the utility information at run-time using cost-effective UMONs.

Recent proposals [14][10] have looked at dynamic mechanisms to obtain the hit latency of a private cache while approximating the capacity benefits of a shared cache. Our work differs from these in that it focuses on increasing the capacity benefits of a shared cache. It can be combined with these proposals to obtain both improved capacity and improved latency from a cache organization.

6.7.3 Related Work in Memory Allocation

In the operating systems domain, Zhou et al. [90] looked at page allocation for competing applications using *miss ratio curve*. The objective of both their study and our study is the same, however, their study deals with the allocation of physical memory, which is fully associative, whereas, our study deals with the allocation of on-chip caches, which are set-associative. The hardware solution proposed in [90] stores an extra tag entry for each page in a separate hardware structure for each competing application. While this may be cost-effective in paging domain (approximately 4B per 4kB page), keeping multiple tags for each cache line for on-chip caches is hardware-intensive and power-hungry. For example, if four applications share a cache and each tag-entry is 4B, then the storage required per cache line is 16B (which is a 25% overhead for a 64B cache line), rendering the scheme impractical for on-chip caches. Fortunately, on-chip caches are set-associative which makes them amenable to dynamic set sampling (DSS). Our mechanism uses DSS to propose a cost-effective partitioning framework which requires less than 1% storage overhead.

6.7.4 Related work in SMT

Several proposals [80][9][15] have looked at policies for dynamic partitioning of processor resources such as reorder buffer entries, execution bandwidth, and physical register file between the applications that concurrently execute on an SMT processor. However, none of these proposals discuss the problem of partitioning the last-level cache among the competing applications. Although, we evaluated UCP for CMP processors, ideas presented in this chapter are also applicable for SMT processors.

6.8 Summary

Traditional designs for a shared cache use LRU replacement which partitions the shared cache among competing applications on a demand basis. The application that accesses more unique lines in a given interval gets more cache than an application that accesses fewer unique lines in that interval. However, the benefit (reduction in misses) that applications get for a given amount of cache resources may not correlate with the demand. This chapter proposes *Utility-Based Cache Partitioning (UCP)* to divide the cache among competing applications based on the benefit (utility) of cache resource for each application and makes the following contributions:

1. It proposes a low hardware overhead, utility monitoring circuit to estimate the utility of the cache resources for each application. Evaluation with 20 multiprogrammed workloads shows that UCP outperforms LRU on dual-core system by up to 23% and on average 11%, while requiring less than 1% storage overhead.

2. It proposes the *Lookahead Algorithm*, as a scalable alternative to evaluating every possible partition for partitioning decisions when there are a large number of applications sharing a highly associative cache.

126

Chapter 7

Conclusions and Future Work

7.1 Conclusions

To bridge the gap between processor speed and memory speed, modern processors devote majority of the on-chip transistors to the last level cache. However, traditional cache designs – developed for small first-level caches – are inefficient for large caches. Therefore, cache misses are common which results in frequent memory accesses and reduced processor performance. The importance of cache management has become even more critical because of the increasing memory latency, increasing working sets of many emerging applications, and decreasing size of cache devoted to each core due to increased number of cores on a single chip. This dissertation focuses on analyzing some of the problems with managing large caches and proposing cost-effective solutions to improve their performance.

Chapter 3 proposes a cost-effective hybrid replacement mechanism that can select from multiple replacement policies depending on which policy incurs the fewest cache misses. This technique exploits the fact that different workloads and program phases have locality characteristics that make them better suited to different replacement policies. Thus, a mechanism that selects the best performing policy at runtime substantially improve cache performance. To implement hybrid replacement with low-overhead, it shows that cache behavior can be approximated by sampling few sets and proposes the concept of *Dynamic Set Sampling (DSS)*.

Chapter 4 focuses on improving the cache insertion policy. The commonly used LRU replacement policy results in thrashing for memory-intensive workloads that have a working set bigger than the cache size. This dissertation shows that performance of memory-intensive workloads can be improved significantly by changing the recency position where the incoming line is inserted. The proposed mechanism reduces cache misses by 21% over LRU, is robust across a wide variety of workloads, incurs an overhead of less than two bytes, and does not change the existing cache structure.

Chapter 5 targets the variation in performance impact of misses to propose parallelism-aware cache replacement. Modern systems try to service multiple cache misses in parallel. The variation in Memory Level Parallelism (MLP) causes some misses to be more costly on performance than other misses. This dissertation presents the first study on MLP-aware cache replacement and proposes to improve performance by eliminating some of the performance-critical isolated misses. It also presents a framework that can compute the mlp-based cost at runtime and uses this cost to drive a cost-sensitive replacement policy.

Chapter 6 analyzes cache partitioning policies for shared caches in chip multi-processors. Traditional partitioning policies either divide the cache equally among all applications or use the LRU policy to do a demand based cache partitioning. This dissertation shows that performance can be improved if the shared cache is partitioned based on how much the application benefits from the cache, rather than on its demand for the cache. It proposes a novel low-overhead circuit that can dynamically monitor the utility of cache for any application. The proposed partitioning improves weighted-speedup by 11%, throughput by 17% and fairness by 11% on average compared to LRU. This dissertation also proposes a low time-complexity algorithm that is scalable to many cores and performs similar to searching through all the exponential number of possible partitions.

128

7.2   Future Work

7.2.1   Applications of Dynamic Set Sampling

In the era of CMPs, L2 caches become a scarce resource. Therefore, it is important to intelligently manage the cache space among competing applications. However, to be implementable and useful, such management must be cost-effective. Because DSS can approximate the cache behavior by sampling only a few sets in the cache, it provides a cost-effective framework for cache management decisions. The idea of dynamic set sampling can also be used for other cache related optimizations, such as dynamically tuning the parameters of a given replacement policy, reducing the hardware overhead of an expensive replacement policy, or reducing the pollution caused by prefetching mechanisms.

7.2.2   Region-Aware Cache Management

This dissertation took an approach of making cache decisions globally. For example, choosing a single replacement policy for the entire cache, or allocating uniform number of ways devoted to an application across all the sets in a shared cache. Thus, it does not take into account the variation in locality that exists between different regions of memory and across different instructions. For example, the locality characteristics and MLP of data fetched by loads that access arrays is very different from that fetched by pointer-chasing loads. Thus, it may be possible to improve the cache management by doing a per-instruction or per-region control rather than a global control. However, the primary challenge in such a scheme would be to gather information and do enforcement in a cost-effective manner as the number of regions increase.

7.2.3   Prefetching-Aware Cache Management

This dissertation studied cache management for the demand stream. An important area of future work in cache management is how to partition the cache between demand

and prefetch streams. Heuristics such as prefetcher accuracy, reuse of demand lines, and bandwidth contention can be used to limit the prefetcher-based pollution while maximizing the latency hiding advantage of the prefetching mechanism. Also, the performance impact of cache lines that are easily prefetch-able is very different from cache lines that are hard to prefetch. Thus, cache performance can be greatly improved if hard-to-prefetch lines are given more cache space than easier-to-prefetch lines. The framework presented in Chapter 5 can be extended to implement such a prefetching-aware cache management scheme.

7.2.4 MLP-Aware Microarchitecture and Memory System

Although we proposed MLP-awareness for cache replacement, the concept of MLP-awareness can be extended to design an MLP-aware memory system. For example, MLP-aware prefetchers can focus on prefetching costly misses. The idea of MLP-awareness is also applicable to processor design. For example, an MLP-aware fetch policy can allow SMT processors to exploit the maximum MLP from a cache-miss-causing thread.

7.2.5 Extensions of Cache Partitioning

We considered the problem of cache partitioning among the demand streams of competing applications. The Utility Monitoring circuit (UMON) can be extended to compute utility information for prefetched data, which can help in partitioning the cache among multiple demand and prefetch streams. The UMON circuits can also be modified to compute the CPI information for a given application, which can help in providing performance or fairness guarantees for an application [28][38]. We investigated UCP only for multiprogrammed workloads. For multithreaded workloads, UCP can take into account both the variation in the utility of private and shared data, as well as the variation in the utility of private data of competing threads. Also, the insertion policies proposed in Chapter 4 can be extended to high-performance cache partitioning at a significantly low hardware overhead.

130

Appendix

Appendix 1

Proposed Techniques on Remaining SPEC Benchmarks

This appendix analyzes the effect of the optimizations described in this dissertation on the eleven SPEC CPU2000 benchmarks that were left out from detailed studies. Table 1.1 shows the fraction of misses that are compulsory misses for these benchmarks. For all benchmarks the majority of the misses are compulsory misses. As improving cache replacement cannot reduce the number of compulsory misses, there is little scope for reducing misses with the caching optimizations proposed in Chapter 3, 4, and 5.

Table 1.1: Compulsory misses for the remaining SPEC benchmarks

gcc	vortex	applu	wupwise	mesa	crafty	gzip	fma3d	gap	perlbmk	eon
51.2%	53.7%	62.9%	83.0%	83.6%	88.8%	97.2%	99.3%	99.9%	100%	100%

1.1 Hybrid Replacement via Dynamic Set Sampling

Table 1.2 shows the MPKI with four replacement policies: LRU, LFU, the TSEL selection between LRU and LFU, and the SBAR-based selection between LRU and LFU. For all benchmarks the performance of all the four policies is similar.

Table 1.2: MPKI with Hybrid Replacement on Remaining SPEC Benchmarks

Policy	gcc	vortex	applu	wupwise	mesa	crafty	gzip	fma3d	gap	perlbmk	eon
LRU	0.35	0.71	13.75	2.25	0.62	0.09	1.45	4.61	1.65	0.04	0.01
LFU	0.40	0.74	13.99	2.53	0.67	0.09	1.46	4.60	1.67	0.04	0.01
TSEL	0.35	0.71	13.71	2.25	0.62	0.09	1.46	4.61	1.65	0.04	0.01
SBAR	0.35	0.71	13.71	2.25	0.62	0.09	1.46	4.61	1.65	0.04	0.01

1.2 Adaptive Insertion Policies

Table 1.3 shows the MPKI with the baseline LRU policy and the DIP policy proposed in Chapter 4. For all benchmarks, the MPKI with both LRU and DIP are similar.

Table 1.3: MPKI with LRU and DIP on Remaining SPEC Benchmarks

Policy	gcc	vortex	applu	wupwise	mesa	crafty	gzip	fma3d	gap	perlbmk	eon
LRU	0.35	0.71	13.75	2.25	0.62	0.09	1.45	4.61	1.65	0.04	0.01
DIP	0.35	0.72	13.76	2.26	0.63	0.09	1.46	4.61	1.66	0.04	0.01

1.3 MLP-Aware Cache Replacement

Table 1.4 shows the IPC with the baseline LRU policy, the cost-sensitive policy LIN and the SBAR based cost-sensitive selection between LRU and LIN. For all benchmarks, except wupwise, the performance of the the three policies is similar. For wupwise, LIN reduces performance by 6% compared to LRU while SBAR performs similar to the baseline.

Table 1.4: IPC with LRU, LIN, and SBAR on Remaining SPEC Benchmarks

Policy	gcc	vortex	applu	wupwise	mesa	crafty	gzip	fma3d	gap	perlbmk	eon
LRU	0.56	1.78	0.79	1.40	2.91	1.26	1.27	1.99	2.44	2.30	1.91
LIN	0.56	1.76	0.81	1.30	2.90	1.26	1.26	2.00	2.42	2.30	1.91
SBAR	0.56	1.77	0.79	1.40	2.91	1.26	1.27	2.00	2.44	2.30	1.91

Bibliography

[1] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache performance of operating systems and multiprogramming. In *ACM Transactions on Computer Systems, 6*, pages 393–431, November 1988.

[2] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 423–434, 2003.

[3] Alaa R. Alameldeen and David A. Wood. Adaptive cache compression for high-performance processors. In *ISCA-31*, page 212, 2004.

[4] Alaa R. Alameldeen and David A. Wood. Frequent pattern compression: A significance-based compression scheme for L2 caches. Technical Report 1500, Computer Sciences Department, University of Wisconsin - Madison, 2004.

[5] Jean-Loup Baer and Tien-Fu Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.*, 44(5):609–623, 1995.

[6] S. Bansal and D. Modha. CAR: Clock with adaptive replacement. In *in Proceedings of the USENIX Conference on File and Storage Technologies(FAST)*, pages 187–200, March 2004.

[7] L A Belady. A study of replacement algorithms for a virtual-storage computer. In *IBM Systems journal*, pages 78–101, 1966.

[8] Brad Calder, Dirk Grunwald, and Joel Emer. Predictive sequential associative cache. In *Proceedings of the Second IEEE International Symposium on High Performance Computer Architecture*, pages 244–253, 1996.

[9] Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and Enrique Fernandez. Dynamically controlled resource allocation in SMT processors. In *Proceedings of the 37th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 171–182, 2004.

[10] Jichuan Chang and Gurinar S. Sohi. Cooperative caching for chip multiprocessors. In *Proceedings of the 33nd Annual International Symposium on Computer Architecture*, pages 264–276, 2006.

[11] Chi F. Chen, Se-Hyun Yang, Babak Falsafi, and Andreas Moshovos. Accurate and complexity-effective spatial pattern prediction. In *HPCA-10*, page 276, 2004.

[12] William Y. Chen, Roger A. Bringmann, Scott A. Mahlke, Richard E. Hank, and James E. Sicolo. An efficient architecture for loop based data prefetching. In *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 92–101, 1992.

[13] D.T. Chiou. *Extending the reach of microprocessors: column and curious caching.* PhD thesis, Massachusetts Institute of Technology, 1999.

[14] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in CMPs. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 357–368, 2005.

[15] Seungryul Choi and Donald Yeung. Learning-based SMT processor resource distribution via hill-climbing. In *Proceedings of the 33nd Annual International Symposium on Computer Architecture*, 2006.

135

[16] Yuan Chou, Brian Fahs, and Santosh Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[17] Robert Cooksey. *Content-Sensitive Data Prefetching*. PhD thesis, University of Colorado, Boulder, 2002.

[18] Adrian Cristal et al. Kilo-instruction processors: Overcoming the memory wall. *IEEE Micro*, 25(3):48–57, May 2005.

[19] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[20] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 1997 International Conference on Supercomputing*, pages 68–75, 1997.

[21] Wi fen Lin et al. Reducing dram latencies with an integrated memory hierarchy design. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 301–312, 2001.

[22] W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 102–110, 1992.

[23] A. Glew. MLP yes! ILP no! In *Wild and Crazy Ideas Session, 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[24] Antonio Gonzalez, Carlos Aliagas, and Mateo Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 338–347, 1995.

[25] Erik G. Hallnor and Steven K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 107–116, 2000.

[26] Mark Donald Hill. *Aspects of cache memory and instruction buffer performance*. PhD thesis, 1987.

[27] Lisa R. Hsu et al. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *PACT-15*, 2006.

[28] Ravi Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of the 18th International Conference on Supercomputing*, pages 257–266, 2004.

[29] Jaeheon Jeong and Michel Dubois. Optimal replacements in caches with two miss costs. In *SPAA '99: Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 155–164, 1999.

[30] Jaeheon Jeong and Michel Dubois. Cost-sensitive cache replacement algorithms. In *Proceedings of the Ninth IEEE International Symposium on High Performance Computer Architecture*, 2003.

[31] Teresa L. Johnson. *Run-time adaptive cache management*. PhD thesis, University of Illinois, Urbana, IL, May 1998.

[32] Doug Joseph and Dirk Grunwald. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, 1997.

[33] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990.

[34] Tejas Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *Second Annual Workshop on Memory Performance Issues*, 2002.

[35] Tejas S. Karkhanis and James E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[36] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 240–251, 2001.

[37] Mazen Kharbutli, Keith Irwin, Yan Solihin, and Jaejin Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *Proceedings of the Tenth IEEE International Symposium on High Performance Computer Architecture*, pages 288–299, 2004.

[38] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[39] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, 1981.

[40] Sanjeev Kumar and Chris Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *ISCA-25*, pages 357–368, 1998.

[41] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 144–154, 2001.

[42] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong-Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Measurement and Modeling of Computer Systems*, pages 134–143, 1999.

[43] W. Lin and S. Reinhardt. Predicting last-touch references under optimal replacement. In *Technical Report CSE-TR-447-02, University of Michigan*, 2002.

[44] Chun Liu, Anand Sivasubramaniam, and Mahmut Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. In *Proceedings of the Tenth IEEE International Symposium on High Performance Computer Architecture*, page 176, 2004.

[45] Kun Luo et al. Balancing throughput and fairness in SMT processors. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2001.

[46] R. L. Mattson et al. Evaluation techniques in storage hierarchies. *IBM Journal of Research and Development*, 9:78–117, 1970.

[47] Scott McFarling. Cache replacement with dynamic exclusion. Technical Report TN-22, Digital Western Research Laboratory, November 1991.

[48] Scott McFarling. Cache replacement with dynamic exclusion. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 191–200, 1992.

[49] Scott McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[50] Nimrod Megiddo and Dharmendra Modha. ARC: A low overhead self tuning replacement cache. In *USENIX File and Storage Technologies*, 2003.

[51] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the Ninth IEEE International Symposium on High Performance Computer Architecture*, pages 129–140, 2003.

[52] N. Young. The K-server dual and loose competetiveness for paging. *Algorithmica*, 11(2), 1994.

[53] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 96, 2004.

[54] Victor F. Nicola, Asit Dan, and Daniel M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *SIGMETRICS '92/PERFORMANCE '92: Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 35–46, 1992.

[55] Vijay S. Pai and Sarita Adve. Code transformations to improve memory parallelism. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 147–155, 1999.

[56] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, 1994.

[57] Jih-Kwon Peir, Yongjoon Lee, and Windsor W. Hsu. Capturing dynamic memory reference behavior with adaptive cache topology. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 134–143, 1998.

[58] Erez Perelman et al. Using SimPoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):318–319, 2003.

[59] Allan Kennedy Porterfield. *Software methods for improvement of cache performance on supercomputer applications*. PhD thesis, Rice University, 1989.

[60] Prateek Pujara and Aneesh Aggarwal. Increasing the cache efficiency by eliminating noise. In *HPCA-12*, 2006.

[61] T. R. Puzak. *Analysis of cache replacement algorithms*. PhD thesis, Univ. of Mass., ECE Dept., Amherst, MA., 1985.

[62] Moinuddin Qureshi, M. Aater Suleman, and Yale N. Patt. Line distillation: Increasing cache capacity by filtering unused words in cache lines. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007.

[63] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for MLP-aware cache replacement. In *Proceedings of the 33nd Annual International Symposium on Computer Architecture*, 2006.

[64] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. The V-Way Cache: Demand Based Associativity via Global Replacement. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 544–555, 2005.

[65] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for QoS management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, page 298, 1997.

[66] J.A. Rivers and E.G. Davidson. Reducing conflicts in direct-mapped caches with temporality-based design. In *International Conference on Parallel Processing*, pages 93–103, 1996.

[67] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *SIGMETRICS '90: Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 134–142, 1990.

[68] Sheldon Ross. *A First Course in Probability*. Pearson Prentice Hall, 7 edition, 2006.

[69] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. The EELRU adaptive replacement algorithm. *Performance Evaluation*, 53(2):93–123, 2003.

[70] A J Smith. Sequentiality and prefetching in database systems. *ACM Transaction on Database Systems*, 3(3):223–247, September 1978.

[71] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[72] James E. Smith and James R. Goodman. A study of instruction cache organizations and replacement policies. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, pages 132–137, Los Alamitos, CA, USA, 1983. IEEE Computer Society Press.

[73] Kimming So and Rudolph N. Rechtschaffen. Cache operations by mru change. *IEEE Trans. on Computers*, C-37(6), June 1988.

[74] Srikanth T. Srinivasan, Roy Dz-Ching Ju, Alvin R. Lebeck, and Chris Wilkerson. Locality vs. criticality. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[75] Srikanth T. Srinivasan and Alvin R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[76] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers.*, 41(9):1054–1068, 1992.

[77] R. Subramanian, Y. Smaragdakis, and Gabriel Loh. Adaptive caches: Effective shaping of cache behavior to workloads. In *Proceedings of the 39th Annual ACM/IEEE International Symposium on Microarchitecture*, 2006.

[78] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1):7–26, 2004.

[79] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the Tenth IEEE International Symposium on High Performance Computer Architecture*, page 117, 2002.

143

[80] Dean M. Tullsen and Jeffery A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 318–327, 2001.

[81] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A modified approach to data cache management. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 93–103, 1995.

[82] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. Using the compiler to improve cache replacement decisions. In *PACT '02: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, page 199, 2002.

[83] M. V. Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, 14(2):270–271, 1965.

[84] Maurice V. Wilkes. The memory gap and the future of high performance memories. *ACM Computer Architecture News*, 29(1):2–7, March 2001.

[85] Wayne A. Wong and Jean-Loup Baer. Modified LRU policies for improving second-level cache behavior. In *Proceedings of the Sixth IEEE International Symposium on High Performance Computer Architecture*, pages 49–60, 2000.

[86] Wm. Wulf and Sally McKee. Hitting the memory wall: Implications of the obvious. *ACM Computer Architecture News*, 23(1):20–24, March 1995.

[87] Jun Yang, Youtao Zhang, and Rajiv Gupta. Frequent value compression in data caches. In *MICRO-33*, pages 258–265, 2000.

[88] Huiyang Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 231–242, 2005.

[89] Huiyang Zhou and Thomas M. Conte. Enhancing memory level parallelism via recovery-free value prediction. In *Proceedings of the 17th International Conference on Supercomputing*, pages 326–335, 2003.

[90] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 177–188, 2004.

Vita

Moinuddin Qureshi, the son of Khalil Ahmed Qureshi and Hasina Qureshi, was born in Kalyan, India on October 2, 1978. He received the Bachelor of Engineering degree in Electronics from the University of Bombay in 2000. The following year he worked as an IC design engineer at the Texas Instruments research and development center in Bangalore, India. He entered the Ph.D. program in 2001 at the University of Texas at Austin where he began working with his Ph.D. adviser Dr. Yale N. Patt. He received the Master of Science degree in Electrical Engineering in 2003.

While in graduate school, he served as a teaching assistant for five semesters. He also had several summer internships at IBM and Intel. He has published papers in The International Symposium on Computer Architecture (ISCA-32, ISCA-33 and ISCA-34), The International Symposium on Microarchitecture (MICRO-39), The International Symposium on High Performance Computer Architecture (HPCA-13), and The International Conference on Dependable Systems and Networks (DSN). His graduate studies were supported in part by a Ph.D. Fellowship from IBM during the academic years 2003-2006.

Permanent address: 2128 VP Street, Apt B1-502
Camp, Pune 411001 India

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.

146