Fall 12-14-2012

Spatiotemporal Capacity Management for the Last Level Caches of Chip Multiprocessors

Dongyuan Zhan
University of Nebraska-Lincoln, dyzhan@gmail.com

SPATIOTEMPORAL CAPACITY MANAGEMENT FOR THE LAST LEVEL CACHES OF CHIP MULTIPROCESSORS

by

Dongyuan Zhan

A DISSERTATION

Presented to the Faculty of

The Graduate School at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Engineering

Under the Supervision of Professors Hong Jiang and Sharad C. Seth

Lincoln, Nebraska

December, 2012

SPATIOTEMPORAL CAPACITY MANAGEMENT FOR THE LAST LEVEL CACHES OF CHIP MULTIPROCESSORS

Dongyuan Zhan, Ph. D.

University of Nebraska, 2012

Advisers: Hong Jiang and Sharad C. Seth

Judicious management of on-chip last-level caches (LLC) is critical to alleviating the memory wall of chip multiprocessors (CMP). Although there already exist many LLC management proposals, belonging to either the spatial or temporal dimension, they fail to capture and utilize the inherent interplays between the two dimensions in capacity management. Therefore, this dissertation is targeted at exploring and exploiting the spatiotemporal interactions in LLC capacity management to improve CMPs' performance. Based on this general idea, we address four specific research problems in the dissertation.

For the private LLC organization, prior-art proposals can improve the efficacy of inter-core cooperative caching at the coarse-grained application level. However, they are still suboptimal because they are unable to take advantage of the diverse capacity demands at the fine-grained set level. We introduce the SNUG LLC design that exploits the set-level non-uniformity of capacity demands and thus further improves performance.

Still for the private LLC management, we notice that neither spatial nor temporal LLC management schemes, working separately as in prior work, can deliver robust performance under various circumstances due to set-level non-uniform capacity demands. We propose a novel adaptive scheme, called STEM, to solve the problem by interactively managing both spatial and temporal dimensions of capacity demands at the set level.

For the shared LLC organization, existing proposals try to improve either locality or utility for heterogeneous workloads. But we find that none of them can deliver consistently the best performance under a variety of workloads due to applications'

diverse locality and utility features. To address the problem, we present the CLU LLC design that co-optimizes the locality & utility of co-scheduled threads and thus adapts to more diverse workloads than the prior-arts.

To make a cache management strategy practical for industry, we will need to cut the overhead of the re-reference prediction value (RRPV). We observe that delicately-tuned replacement policies rooted in single-bit RRPVs can closely approximate the performance of their correspondents with $\log associativity$-bit RRPVs. Therefore, we propose a novel practical shared LLC design, called COOP, which entails a 1-bit RRPV per cacheline and a lightweight monitor per core for locality & utility co-optimization. At a considerably low storage cost, COOP achieves higher performance than the two recent practical replacement policies that rely on 2-bit RRPVs but are oriented towards locality optimization only.

ACKNOWLEDGMENTS

First and foremost, this dissertation is dedicated to my beloved parents Yonglong Zhan and Lilong Peng. It is their constant love, high expectations and inspiring encouragement that motivate me to work diligently and smartly on my doctoral program. They always set a good example for me with their courage, determination and resilience, which endows me with the will, strengths and skills to endure any hardships. I have every faith in the healthy, peaceful and happy life of us in the future!

I am really grateful to my two advisers, Dr. Hong Jiang and Dr. Sharad C. Seth, for their dedicated mentoring of my Ph.D. study and research in the past five years. They are both great tutors. Professionally, I learned a lot of knowledge, skills and research methodologies from them. Dr. Hong Jiang is quite a sharp-minded person. He taught me a great many about logic, argument and reasoning, as well as how to discover a research niche. Dr. Seth has a very solid mathematical background. From him, I learned how rigorous research progression can be made. Personally, they are good people with respectable personalities. I gradually learned from them how to live a harmonious and balanced life with research being the career focus.

I heartily thank Dr. Witawas Srisa-an, Dr. Lisong Xu and Dr. Sina Balkir for serving on my Ph.D. supervisory committee. I appreciate their constructive suggestions and comments on my dissertation work. In particular, I have been working closely with Dr. Srisa-an on collaborative research for over a year. His insights into programming languages, runtime systems and operating systems vastly broadened my horizons beyond my own computer architecture research.

My colleagues at the ADSL lab are highly supportive guys. Lei Tian played a vital role in helping with the cold start of my research during the second year of my program. Jianjun Du helped me a great deal overcome hard times of my life in 2009. Other group

members also provide kind and generous support for my daily life and work. I really feel like that we are united as brothers and sisters.

I am indebted to my sincere local friends, Allan F. Gossman and Joyce K. Latrom, for their consistent efforts to care about my well-being. Throughout the five years, I spent every New Year's Eve at their condo, attended many fantastic social events with them and could get immediate help from them when I was in difficulty. They always treat me like family. This is why I feel so joyful with them and value our friendship so much.

I owe thanks to Junwei Zhang, Wei Chen and Jian Kang for their help when I first settled down in Lincoln. Chaorong Wu has been my roommate since 2008. He is humble, optimistic and helpful. I learned a whole bunch from him how to be a good man.

Finally, I would also like to thank the Holland Computing Center for the wonderful technical support. My Ph.D. program was financially supported by the research assistantship from my advisers' grants sponsored by NSF and Intel and the teaching assistantship from the Department of Computer Science & Engineering at the University of Nebraska - Lincoln.

Dongyuan Zhan, Ph.D.
Lincoln, Nebraska
December 14, 2012

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Chip multiprocessors (CMPs) have become the *de facto* design paradigm for high-performance processors. Driven by a new corollary of Moore's Law [1] predicting that the CMP core count will double every 18 months, major chip manufacturers have been making steady progress in promoting the CMP technology to a many-core scale, which is evidenced by Tilera's recent announcement of the world's first 100-core processor TILE-Gx100^{TM} [2].

This trend of aggressive core count growth, however, is now threatened by obstacles imposed by certain performance-critical components that are less scalable. One such obstacle is the *memory wall* [3] that limits CMPs' performance with both long latency and limited bandwidth. Historically, the operating frequency of a processor was scaled much faster than that of main memory (e.g., DRAM) because of their distinct structural and electronic features [4]. Although CMPs' frequency scaling has already slowed down due to power and thermal constraints, the previous unbalanced frequency scaling has already resulted in a huge speed disparity between processors and DRAM, rendering off-chip memory accesses one or two orders of magnitude slower than on-chip cache references. At the same time, the aggressive core count growth is hitting the upper bound

of available memory bandwidth [5]. This is because the increase of on-chip cache capacity still lags behind the growth of on-chip core counts [6], leading to shrunk cache capacity per core and in turn more off-chip memory requests. If the available off-chip memory bandwidth cannot sustain the rate at which memory requests are generated, cores will be forced to degrade their performance until the rate of memory requests matches the available memory bandwidth, which defeats the purpose of yielding additional throughput performance by incorporating more cores.

Since the memory wall is a fundamental problem in computer architecture, there has been a large body of research literature on alleviating the impact of the memory wall from different angles, which can generally be classified into the following four categories:

- making efficient use of on-chip caches, especially the *last level caches* (LLC) that have the largest capacity among all cache levels, to minimize off-chip memory requests [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17];
- hiding memory latency by prefetching [18, 19, 20, 21, 22, 23, 24];
- overlapping memory latency by exploiting *memory level parallelism* (MLP) [25, 26];
- architecting 3D stacked memory or optical I/Os to reduce the wire delay and increase the bandwidth between processors and main memory [27, 28, 29, 30];

However, the four categories of techniques have very different optimization objectives and are thus largely orthogonal to one another. In this dissertation, I mainly focus on optimizing the management of CMPs' LLCs that plays an irreplaceable role in minimizing accesses to main memory, as this topic offers a rich space for solutions that have not yet been fully explored. The emphasis on minimizing off-chip memory requests was also highlighted during a recent panel discussion about the challenges facing *exascale computing* [31] in which applications work on huge datasets.

1.1 LLC Capacity Management Problems

The capacity management of CMP LLCs depends on the specific LLC organization. As will be detailed in Chapter 2, there are two basic types of LLC organizations in CMPs, private and shared. In the private LLC organization, the entire LLC capacity is partitioned into slices, and each processing core has only but exclusive access to its local LLC slice. The private LLC organization can benefit cache references with localized and minimal access latency. But it suffers from the weakness of fixed and limited space that is accessible to a core. Conversely, in the shared LLC organization, the entire LLC space can be accessed by every core. Despite its large aggregate capacity, the shared LLC organization usually places data in the slices distant from a requesting core due to the distributed nature of large cache space, leading to long access delays for the remote data. However, it is worth mentioning that, because of the complementary advantages and disadvantages of the two basic LLC organizations, there is little consensus about which one is superior in the research community. This dissertation will not get involved with the "private versus shared" debate, but rather make contributions to the better performance of both LLC organizations.

There are two major concerns about the capacity management of private LLCs. First, when different applications/threads [1] are co-scheduled on a CMP, they typically have distinct capacity demands from each other. The private LLC organization, however, statically allocates the same amount of exclusive space to each thread. As a result, some threads may need more capacity than what their private LLCs can provide, while others have underutilized LLC space. To overcome the barrier that prevents capacity sharing in the private LLC organization, Chang *et al.* [32] propose the framework of *inter-core cooperative caching* (CC) by allowing "private" LLCs to use each other as victim caches.

[1] In this dissertation, the terms "application" and "thread" are interchangeable unless they are specifically set apart.

But the original CC framework can unnecessarily favor the streaming-like applications in cooperative caching just because they have a large number of victim blocks, contributing little to overall performance. A recent proposal named the *dynamic spill-receive* (DSR) paradigm [33] partially addresses the problem by prioritizing the applications that have higher performance benefits from extra LLC space in obtaining cooperative capacity. But as we demonstrate in Chapter 3, DSR is still suboptimal.

Second, two fundamental issues arise with respect to the capacity management of intra-core private LLCs. On the one hand, the commonly-adopted *least recently used* (LRU) replacement policy can lead to LLC thrashing if a running application's temporal locality is inferior. In this regard, existing statistics about memory-intensive applications [7] indicate that over 60% blocks contribute no cache hits between their insertion into and eviction off the LLC which is managed by LRU. Thus, alternative replacement policies like the *dynamic insertion policy* (DIP) [7] and the *pseudo-LIFO policy* (PeLIFO) [8] are proposed to adapt LLC replacement algorithms to workloads' specific locality features, as opposed to LRU's consistent preference for excellent temporal locality. On the other hand, capacity demands are non-uniformly distributed across different LLC sets, while the conventional LLC design statically provisions all LLC sets with the same amount of space. Therefore, new LLC designs such as the *variable-way cache* (V-Way) [9] and the *set balancing cache* (SBC) [10] attempt to redistribute cache space among distinct LLC sets to meet individual sets' capacity needs. We will illustrate in Chapter 4 that, despite being tackled separately in prior studies, the two issues are actually interleaved with each other inherently. The interplay between the two aspects has a significant impact on the LLC performance, rendering prior separate solutions unable to perform robustly under a variety of workloads.

In terms of the shared LLC organization, although it allows processing cores to freely share its large aggregate capacity, it is still necessary to regulate the sharing

so as to minimize the inter-core interference and maximize overall performance. The management has been explored in two directions in the research community: (*i*) a locality-oriented approach, such as the *thread-aware dynamic insertion policy* (TADIP) [14] and the *next-use cache* (NUcache) [15], differentiates the locality features of co-scheduled threads (or different memory instructions) and coordinates replacement algorithms for them; (*ii*) a utility-oriented approach, including the *utility-based cache partitioning* (UCP) [12] and the *promotion/insertion pseudo partitioning* (PIPP) [13], explicitly partitions the shared space among concurrent threads based on an estimate of how efficiently each thread can generate cache hits with an amount of LLC capacity. The two types of approaches, however, have different working principles and thus distinct performance comfort zones. Neither of them, working separately as suggested in their proposals, is able to consistently deliver the best performance under all circumstances. Chapter 5 will show that, by interactively co-optimizing the locality and utility of co-scheduled threads, an adaptive solution can well bridge the gap between the existing two types of approaches.

In the cache structure, a *re-reference prediction value* (RRPV) field is included in every cache line. Its function can be explained as follows. If no invalid lines are present upon a cache replacement, the block with the largest RRPV in the set will be selected as the victim block, since it is predicted to be re-referenced furthest in the future. To strictly sort all lines in a set, there have to be $\log Associativity$ bits in a RRPV field. But this overhead is considered prohibitive and impractical for shared LLCs because of their large associativity. For instance, if the LLC associativity is 32, each line needs to have 5 bits in its RRPV field. To reduce the overhead, the industry typically devotes fewer bits to the RRPV to partially sort the cache lines in a set. Recently, two proposals, the *thread-aware dynamic re-reference interval prediction* (TA-DRRIP) [16] and the *signature-based hit predictor* (SHiP) [17], have significantly cut down the cost by relying on 2-bit RRPVs in spite of

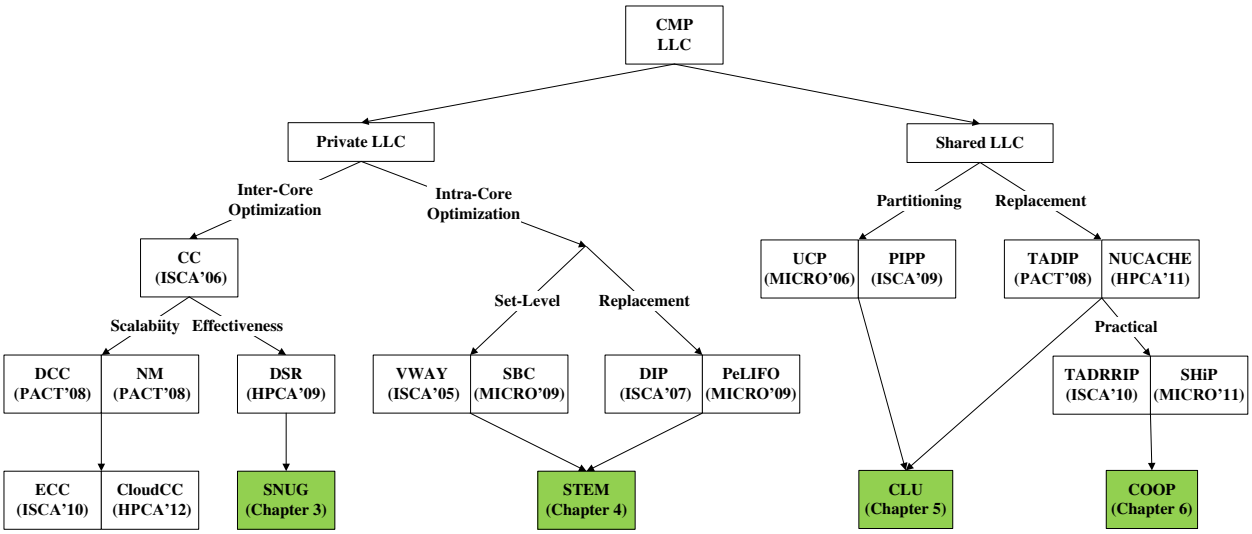


Figure 1.1: Dissertation Overview

the LLC associativity. But the two proposals are oriented towards locality optimization only, missing the performance improvement opportunities that are uniquely provided by utility optimization. Therefore, locality and utility co-optimization is still indispensable to achieving higher performance, as will be revealed in Chapter 6. We will also prove that a 1-bit RRPV field is sufficient to implement locality and utility co-optimization with the support of extra lightweight monitors, thus further reducing the hardware overhead.

1.2 Contributions of the Dissertation

As illustrated in Figure 1.1, by addressing the aforementioned four specific problems, we strive to make the following contributions in this dissertation.

- In Chapter 3, we derive a set of mathematical models to define the set-level capacity demand, characterize its non-uniform distribution in real-world applications and design a scheme called the *Set-level Non-Uniformity identifier and Grouper* (SNUG) [34] that utilizes the non-uniformity to enhance inter-core cooperative caching.

SNUG detects the capacity demand of a set with a per-set shadow tag array and saturating counter and groups cross-core peer sets for spilling and receiving in an flexible way by using an index-bit flipping scheme. Our SNUG cache design improves the CMP throughput by up to 22.3% and with an average of 13.9% over the baseline configuration, outperforming the prior-art DSR scheme that only achieves an improvement by up to 14.5% and 8.4% on average.

- In Chapter 4, we provide a unique taxonomy that categorizes prior studies on intra-core LLC management into either the spatial dimension or the temporal dimension. Further, our workload characterization indicates that none of prior-art spatial or temporal schemes can adapt to the diverse capacity needs that applications have in the two dimensions. Therefore, we propose the *SpatioTEMporal capacity management* (STEM) [35] solution to concurrently manage programs' spatial and temporal capacity demands. STEM works by interactively pairing off peer sets for intra-core cooperative caching and determining replacement algorithms for individual sets. STEM improves the performance metrics of MPKI (misses per 1k instruction), AMAT (average memory access time) and CPI (cycles per instruction) over LRU by 21.4%, 13.5% and 6.3% respectively, better than the improvements obtained by prior-art spatial and temporal approaches, at a manageable HW storage cost of only 3.1%.

- In Chapter 5, for the management of shared LLCs, we sort out prior work by differentiating locality-oriented versus utility-oriented mechanisms. By comparing the two types of approaches qualitatively and quantitatively, we observe that locality-oriented and utility-oriented approaches have distinct performance comfort zones and neither is consistently the best. To address this issue, we develop a novel management framework that is able to *Co-optimize Locality and Utility* (CLU) [36]

for shared LLCs. The key idea of CLU is to derive a composite hit curve from monitored LRU and BIP hit curves for each thread and leverage it to interactively determine capacity partitioning and replacement algorithms for all co-scheduled threads. CLU improves the throughput by 24.3%, 45.3% and 43.3% for our simulated dual-core, quad-core and eight-core systems (with 0.26%, 0.27% and 0.53% storage overhead) respectively, outperforming the existing locality-oriented and utility-oriented schemes.

- In Chapter 6, we show how to utilize minimal-overhead hardware to achieve high performance in practical shared LLC management. We argue that most of prior studies on shared LLC management discourage the industry to adopt them due to the prohibitive log *Associativity*-bit RRPV cost per cache line. While the two most-recent 2-bit RRPV proposals reduce the overhead for smart replacement policies that are oriented towards locality optimization, we demonstrate that they cannot always supersede cache partitioning in that they are unable to achieve certain performance benefits provided by locality optimization. We also observe that carefully-tuned replacement policies rooted in single-bit RRPVs can closely approximate the performance of their correspondents which are based on log *Associativity*-bit RRPVs. Therefore, we propose to leverage single-bit RRPVs to perform locality & utility co-optimization (COOP) [37] with the support of additional lightweight locality & utility monitors. COOP offers significant throughput improvement over LRU by on-average 7.67%, at a cost of 17.74KB RRPV overhead that is only 55.4% of LRU's.

1.3 Organization of the Thesis

The rest of the dissertation is organized as follows. In Chapter 2, we introduce the background of on-chip cache hierarchies, structures and organizations, and survey relevant

research concerns as well as the prior-art work. As briefly summarized in Section 1.2, from Chapter 3 to Chapter 6, we present our solutions to the four identified research problems. Chapter 7 outlines the wider implications and potential directions for future research from both architecture's and systems' perspectives. Finally, Chapter 8 concludes the entire dissertation.

Chapter 2

Background & Literature Review

In this chapter, we briefly introduce the essential background of processor caches at the architecture level. We also summarize existing cache studies in a coherent framework, delineate their pros and cons and highlight potential areas for research. Our intention is to present the landscape of research on CMP LLC management so that readers can have a big picture before reading subsequent chapters. Hence, we do not go into intricate details that can be found in the references to the original papers or reports.

2.1 On-Chip Cache Hierarchy

The design principle of on-chip caches is to exploit both *temporal* and *spatial* localities of running programs. Specifically, temporal locality is referred to as accesses to the same memory location that occur close together in time, while spatial locality means accesses to nearby memory locations in successive memory references [38]. Similar to the memory hierarchy in a computer system [39], on-chip caches are also built in hierarchy to balance the conflicting goals of speed and capacity among the multiple levels. The higher-level caches provide faster accesses, while the lower-level ones have larger capacity for data

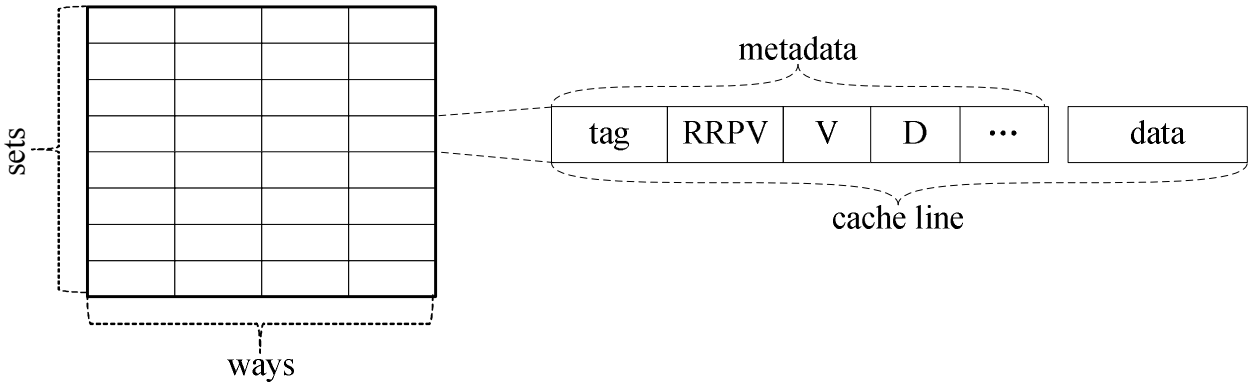


Figure 2.1: Cache Structure

retention. L1 data and instruction caches are at the highest level in the hierarchy. They are connected with the instruction fetch unit and the load/store queue of a processing core respectively, and both of them are exclusively used by the attached core. L1 caches typically have small sizes and low associativity (see the concept in Section 2.2) so that their access delays are as low as only 1 or 2 cycles. A miss in an L1 data or instruction cache will initiate a request to the L2 cache. If there are only two levels of caches on-chip, the L2 cache is the *last level cache* (LLC) that provides as much capacity as several megabytes (e.g., 4MB in AMD's A6-Series processors [40]) with 10-20 cycles of access latency. The last level cache can adopt either a private or a shared organization, as will be detailed in Section 2.3. In this thesis, for simplicity and without loss of generality, the L2 cache is always assumed to be the last level cache unless more cache levels are specifically mentioned.

2.2 Cache Structures

As illustrated in Figure 2.1, an on-chip cache is typically structured in three tiers internally. The 1st tier is the cache entity itself to which the upper-level memory hierarchy components send requests. E.g., an L2 cache receives references from L1 caches. In

the 2nd tier, a cache is organized in sets. The cache accesses are mapped to individual cache sets based on the requested addresses. In practice, for the simplicity of address decoding, the integer MOD function is widely adopted with the modulo base equal to the number of cache sets (typically an integral power of 2). Then, the references whose target addresses have the same congruence relation will be mapped to the same cache set and thus form a working set. The 3rd tier is the cache line, which is the basic unit of resource management in cache. All cache lines assigned to the same set will be used to host the member blocks of the corresponding working set, and the number of lines in a set is defined as the set's *associativity*. Also for simplicity, all cache sets have the same static associativity in a typical cache design, and we usually call the cache A-way set associative, where A is the static associativity. In particular, if A is equal to 1, the cache is called a *direct-mapped* cache; if the number of sets is equal to 1, the cache is called a *fully-associative* cache.

Also depicted in Figure 2.1, a cache line typically consists of two parts, *data* and *metadata*. To exploit the *spatial locality*, the data part typically contains tens of bytes of information, and the length is termed the *line size*. A commonly-adopted line size in AMD's and Intel's processors is 64 bytes. To look for certain content in the cache, the cache controller needs to decompose the corresponding address to find out which set it is located in and what it is tagged as in the set, as illustrated in Figure 2.2 and formalized in Equation 2.1. The metadata part includes such fields as a *tag*, a *valid bit*, a *dirty bit*, *coherence bits* and a *re-reference prediction value* (RRPV) [16]. The tag field differentiates the blocks located in the same cache set. The *valid* bit indicates whether or not this cache line contains valid information. The *dirty* bit tells if the cache line's content is modified and thus different from the copy in the next-level memory component, and the bit can be absent in an instruction cache or a data cache with the *write-through* policy [39]. The *coherence bits* are utilized to maintain cache coherence (see Section 2.8) when necessary.

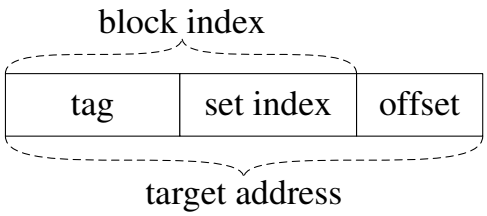


Figure 2.2: Ordinary Address Decoding

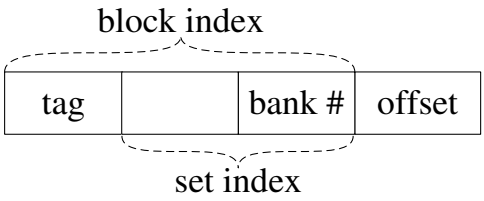
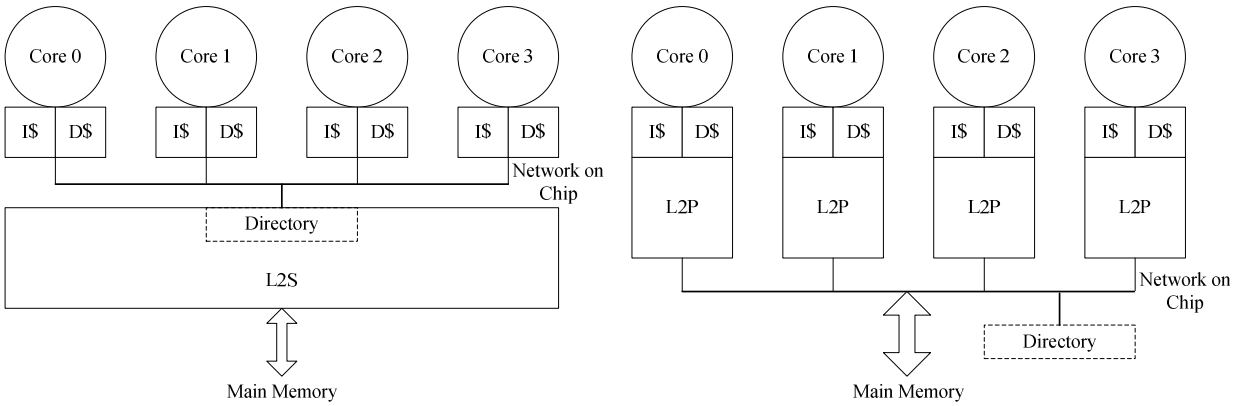


Figure 2.3: Banked Shared LLC Address Decoding

The RRPV field is used to predict how soon the cache line will be accessed again.

$$\begin{aligned}
&\text{(block index)} = \lfloor \frac{\text{(target address)}}{\text{(line size)}} \rfloor \\
&\text{(set index)} = \text{(block index) MOD (the number of sets in cache)} \\
&\text{tag} = \lfloor \frac{\text{(block index)}}{\text{(the number of sets in cache)}} \rfloor
\end{aligned} \tag{2.1}$$

Upon a cache access, the requested address is decoded to get the target set and the tag value, as shown in Figure 2.2. The cache controller selects the target set, activates all lines of the set, and compares the tag fields of the valid blocks with the tag value being searched. If the block is found, it is called a *cache hit*; otherwise, it is a *cache miss*. Upon a cache miss, the cache controller needs to select a victim block and replace it with the requested block to be fetched from the lower-level memory component. The cache controller will search the target set to find an invalid block (if there are any) or a block with a certain RRPV for eviction. The selection process is carried out by a *replacement policy* that decides on which block in the set needs to be evicted to make room for an incoming block. For instance, the *least-recently-used* replacement policy always searches and selects an invalid block or the one that was referenced longest ago (the line with the maximum RRPV in the target set) as the victim block. For a valid victim line in a cache with the *write-back* policy [39], if it is dirty, its content will need to be written back to the



(a) Shared LLC Organization (b) Private LLC Organization

Figure 2.4: Last Level Cache Organizations

next-level memory component.

2.3 Private vs. Shared LLCs

As the core count keeps increasing, efficient management of CMPs' LLCs has two implications. On the one hand, the capacity of the LLC should be fully utilized to retain as much data on-chip as possible for accesses in the short term. On the other hand, design and manufacturing considerations dictate that the large on-chip LLC is fragmented into slices and distributed on a chip, giving rise to *non-uniform cache access* (NUCA) time in the LLC (see Section 2.7). In essence, the NUCA time implies that the latency of a cache access depends on the communication distance between the requesting core and the slice where the cache block is located.

There are two basic LLC organizations, *shared* and *private* LLCs (demonstrated in Figure 2.4 (a) and (b) respectively), targeting at supporting capacity sharing and latency reduction respectively. As shown in Figure 2.3 and Figure 2.4(a), shared LLCs employ address-interleaved banking (namely, using the lower bits of a block's set index to determine its home LLC slice) to evenly distribute blocks to different LLC slices, which

provides a natural way of capacity sharing for cores. But the shared LLC organization can incur excessive remote accesses that are penalized by non-local cache access overhead and impose much pressure on the *on-chip interconnects* (NoC). To reduce LLC hit latency in the shared organization, *replication*-based approaches and *alternative data layout* strategies (e.g., by means of page coloring, see Section 2.9) are introduced to promote the data proximity for requesting core(s). In terms of capacity utilization, although the shared organization can provide a large aggregate capacity to all cores, provisioning the capacity according to individual cores'/threads' needs so as to yield the best throughput or fairness performance is beyond its scope. Therefore, different spatial capacity partitioning schemes and various advanced replacement policies have been proposed to improve the capacity utilization in the shared LLC organization. As shown in Figure 2.4(b), in the private LLC organization, a core places requested blocks in an adjacent LLC slice exclusively used by the core, incurring local access delays only. The intra-core private LLC capacity management is targeted at enabling each core to make the best use of its own LLC space. But the limited capacity accessible to a core can result in more off-chip requests if a running thread's capacity demand exceeds what its private LLC can provide. To tackle this problem, the inter-core private LLC management allows different cores to utilize each other's "private" LLCs as victim caches, overcoming the barrier of capacity sharing among different cores' own LLCs.

2.4 CMP LLC Capacity Management

LLC management has been studied extensively since the uniprocessor era. In this section, we briefly review the past research on private and shared cache management that is most relevant to this dissertation (a necessary taxonomy of the existing cache management mechanisms is shown in Figure 2.5).

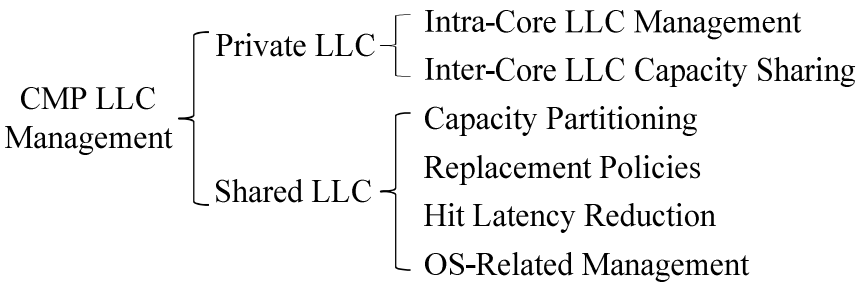


Figure 2.5: A Taxonomy of CMP LLC Management Mechanisms

2.4.1 Basic Replacement Algorithms

Before summarizing existing studies on LLC management, we supplement the background of an oracle replacement algorithm, the *Belady's optimal algorithm*. We also discuss two baseline replacement algorithms, the *least recently used* (LRU) and the *not recently used* (NRU) replacement policies, as most existing work attempts to gain the advantages and overcome the disadvantages of these two algorithms.

The Belady's optimal algorithm [41] assumes the availability of future information and utilizes it to make replacement decisions. It evicts the block that will be referenced furthest in the future. The algorithm is provably optimal, meaning that it can always lead to the lowest cache miss rates. However, due to the unrealistic assumption of knowing future information, the algorithm is not implementable in real computer systems but typically used for offline analysis.

The *least recently used* (LRU) replacement policy assumes that recency is a good predicator of future behavior. In other words, if a block is not accessed for the longest time, it would presumably be referenced furthest in the future. Thus, LRU seeks to approximate the performance of the *Belady's optimal algorithm* by victimizing the block that was last referenced the longest time ago. It mimics a so-called *LRU stack* [42] in which blocks are ordered by recency in a set. To maintain the recency order, LRU entails $\log A$-bit RRPVs in an A-way set associative cache. The block with a 0-valued RRPV,

which is called the *most-recently-used* (MRU) block, is considered to be at the very top of the LRU stack; in contrast, the one with a $(A-1)$-valued RRPV, which is named the *least-recently-used* (LRU) block, is regarded to be at the bottom of the LRU stack. As a result of the strict recency ordering, the LRU replacement policy features a unique mathematical trait, the *LRU stack property* [42], stipulating that the blocks which would be in an A-way associative cache should be subsumed by those that would be in an $(A+1)$-way associative cache. The LRU replacement policy is provably able to favor applications with good temporal locality, but it works poorly otherwise [7]. In addition, since $\log A$-bit RRPVs are indispensable in the LRU replacement policy, the overhead is much higher compared to its approximations like the NRU replacement policy.

The *not recently used* (NRU) replacement policy entails just 1-bit RRPVs in an A-way set associative cache. It maintains a partial recency order for the blocks of a set by classifying them into recently-used (with 0-valued RRPVs) and not-recently-used (with 1-valued RRPVs) groups. Upon a cache hit, NRU updates the block's RRPV bit to 0. In order to select a victim block for eviction followed by a cache miss, there are two possible cases: (*i*) if there are any blocks with 1-valued RRPVs in the target set, the first such block found by scanning the set will be selected as the replacement candidate; (*ii*) otherwise, NRU flips all blocks' RRPVs to 1 and then repeats process (*i*) to find the replacement candidate. Upon a cache fill, the RRPV of the newly-inserted block is set to 0. According to a variety of studies [43, 16], the NRU replacement policy can closely (99.52%) approximate the performance of the LRU replacement policy with just $\frac{1}{\log A}$ of LRU's RRPV overhead. However, NRU lacks the stack property that LRU features.

2.4.2 Intra-Core Private LLC Management

Research on intra-core private LLC management dates back to the uniprocessor era. Generally speaking, the work in this aspect can be classified into two categories.

Temporal LLC Management: Temporal LLC management is referred to as replacement policies that determine how the capacity of an LLC set is temporally shared among the competing blocks of a working set mapped to the LLC set, when the LLC set cannot retain all of them. In [7], Qureshi *et al.* have shown that the commonly-used LRU replacement policy performs well if a running application has excellent temporal locality but thrashes the LLC space otherwise. In their *dynamic insertion policy* (DIP) proposal, two small groups of sample LLC sets are dedicated to LRU and the *bimodal insertion policy* (BIP) respectively for performance dueling, and the winning policy is adopted for other non-sample sets to either exploit temporal locality or prevent thrashing. Another recent work, the *pseudo LIFO* (PeLIFO) [8], takes advantage of a fill-stack to rank the blocks of a set according to the *last in first out* (LIFO) order. Upon replacement, PeLIFO typically does not victimize the block with the lowest LIFO rank, but instead learns the most preferred eviction position close to the top of the fill-stack that can lead to the best performance. In the *cache bursts* study, Liu *et al.* [44] propose to trigger the prediction of whether a block is dead as long as the block is moved off the *most recently used* (MRU) position. Once identified, dead blocks can be replaced much earlier to make room for incoming ones.

Set-Level Spatial LLC Management: *Set-level spatial LLC management* is defined as space allocation schemes that dynamically determine how the overall capacity of an LLC is spatially partitioned among LLC sets that are hosting different working sets. It has been noticed that there exists a non-uniform distribution of accesses to different LLC sets in many applications. The *skewed associativity* cache [45] and the *prime-based set indexing*

scheme [46] are the early work that diffuses accesses to LLC sets in a more balanced way. The recent proposal ZCache [11] leverages *cuckoo hashing* [47] that has provably better randomization traits than *skewed associativity* to scatter blocks among LLC sets. In [9], the *variable way* (V-Way) cache is an LLC design that has twice as many tag entries as data lines. The association between a tag entry and a data line needs to be dynamically established by using a pair of front and backward pointers, via which different LLC sets can have various numbers of ways. In another study, the *set balancing cache* (SBC) [10], it is observed that the difference between the miss and the hit counts of a set, which is defined as the saturation level, varies from set to set in the LLC. The SBC scheme pairs off two sets exhibiting complementary saturation levels and enables the saturated set to place victim blocks in the other's space.

2.4.3 Inter-Core Private LLC Management

For CMPs, the private LLC organization offers fast hit latency at the cost of limited local capacity. Thus, the main disadvantage of private LLCs lies in the rigid constraint that a core can only access its private LLC and cannot share cache capacity with other cores. To overcome this barrier, Chang *et al.* [32] propose the concept of CMP *cooperative caching* (CC) to enable capacity sharing among "private" LLCs by allowing them to utilize the capacity of each other as victim caches. That is, if a core needs to replace a native block from its original LLC slice, instead of evicting it off chip, the block is spilled to another core's LLC slice. Upon the next access, the core can directly get the block from the peer and thus avoid off-chip access penalties. But in their proposal, cooperative caching is evoked whenever a block is evicted from its own private LLC slice, which implicitly favors the applications/threads with higher LLC miss counts. However, a higher miss count does not necessarily reflect the applications/threads' real capacity demands. For

instance, a streaming application can have excessive cache misses, but it will occupy much extra capacity without any performance contributions in CC.

To overcome this shortcoming, Qureshi [33] has recently proposed the *dynamic spill-receive* (DSR) paradigm to regulate block spilling and receiving in response to different applications' cache resource needs. In the DSR paradigm, applications are classified into two categories: *taker* applications that can have their performance improved with additional cache capacity and *giver* applications that can contribute part of their cache capacity to others with little performance degradation. DSR enables taker applications' caches to spill victim blocks to those of giver applications but not vice versa, and thus produces a better performance than the original CC scheme. Besides the aforementioned efficacy problem, the original CC proposal also suffers from poor scalability due to its centralized coherence engine design [32]. Herrero *et al.* [48] propose the *distributed cooperative caching* (DCC) by utilizing the existing distributed coherence directory to eliminate the bottleneck imposed by the centralized coherence engine. Eisley *et al.* [49] leverage the NoC infrastructure to provide heuristics for cooperative caching so that victim blocks can be deterministically migrated towards an appropriate destination. Herrero *et al.* [50] have recently proposed the *elastic cooperative caching* (ECC) mechanism to combine the strengths of both DCC and DSR in a unified framework for large-scale CMPs. In [51], Lee *et al.* propose a new hardware design called CloudCache to minimize off-chip traffic, reduce remote cache accesses and hide the latency of remote directory references for many-core CMPs.

2.4.4 Shared LLC Management

For CMPs with the shared LLC organization, the capacity management is oriented towards either locality or utility optimization in the literature.

Locality-Oriented Capacity Management: As LRU is ineffective in handling workloads with inferior locality, alternative replacement policies have been proposed to adapt management decisions to workloads' specific locality characteristics, by means of sophisticated block insertion, aging, promotion and victimization. The *thread-aware dynamic insertion policy* (TADIP) [14] is a subtle CMP extension of the high-performance *dynamic insertion policy* (DIP) originally designed for uniprocessors (see Section 2.4.2). TADIP identifies the locality of individual threads through set-sampling and dueling, and then coordinates locality optimization for all of the co-scheduled threads under feedback control. Recently, Manikantan *et al.* [15] have found that the distribution of next-use distances of the blocks, which are brought into the LLC by the same load instruction, can reflect the temporal locality of the instruction's memory references. In [15], the next-use distance of a block is defined as the number of intervening misses to an LLC set between the block's eviction from the set and its next reference. The larger number of blocks that feature short next-use distances, the better temporal locality the corresponding instruction has. This motivates their design of a new temporal capacity management scheme called NUcache that enables selected load instructions with top-k temporal locality to have their cache blocks stay longer in the SLLC. Based on LRU-managed set samples and dead-block prediction tables, the *sampling dead block prediction* (SDBP) scheme [52] learns which memory instructions (identified by their PC signatures) tend to access cache blocks that immediately become "dead", victimizes the blocks touched by those PCs prior to default replacement candidates and bypasses predicted dead-on-fill blocks.

However, TADIP, SDBP and NUcache are deemed to be not suitable for practical CMP designs. This is because they are all based on the assumption of log A-bit RRPVs, where A is the set associativity. But the $log A$-bit RRPV overhead is considered to be prohibitive according to industry standards [43]. For example, in a 16-way 4MB cache, the 4-bit RRPVs account for a 32KB storage overhead, equal to a typical size of the L1

cache. Recently, Jaleel *et al.* [16] have proposed a high-performance practical replacement policy called RRIP (an acronym for *re-reference interval prediction*). With 2 bits per line for re-reference interval prediction, a block can have any of the three different categories of re-reference intervals: near, long and distant. RRIP always predicts a long re-reference interval for incoming blocks in effort to prevent the cache pollution due to a subset of incoming blocks being dead-on-fill. Additionally, the bimodal variant of RRIP (called BRRIP) can prevent thrashing by predicting a distant (or a long) re-reference interval for an incoming block with a high (or a complementarily low) probability. TA-DRRIP is a thread-aware extension of RRIP to CMPs with SLLCs by coordinating either RRIP or BRRIP for individual threads under set-dueling and feedback control. Still rooted in the 2-bit RRPV substrate, SHiP [17] assigns either a distant or a long re-reference interval to an incoming block depending on whether or not it is predicted to be dead-on-fill. Specifically, SHiP leverages a history table and sample sets to dynamically learn which memory instructions (identified by their PC signatures) tend to insert dead-on-fill blocks, and predicts a distant re-reference interval for new blocks if they are inserted by those PCs or predicts a long interval otherwise.

Utility-Oriented Capacity Management: The commonly used LRU policy implicitly divides the SLLC capacity among competing threads on a miss-driven basis, which is also ineffective in that a thread may occupy much capacity by bringing into the cache a number of missed blocks but without re-referencing them. SLLC capacity partitioning is targeted at allocating LLC resources to threads on a *utility*, *fairness* or *quality-of-service* (QoS) basis. Here, *utility* is defined as a thread's ability to reduce cache misses with a certain amount of cache capacity. *Utility-based cache partitioning* (UCP) employs a light-weight *utility monitor* (UMON) based on set sampling and the *LRU stack property* (see Section 2.4.1) to dynamically estimate the efficiency/utility of allocating a certain number of SLLC

ways to each thread. Based on the estimation, UCP always favors the threads with the highest space utility in capacity partitioning. Another proposal *promotion/insertion pseudo-partitioning* (PIPP) [13] also adopts the UMON idea for utility estimation, but implicitly partitions the capacity by relying on a combination of insertion and promotion policies.

There are also some studies focusing on QoS metrics other than cache performance. [53] proposes fairness-based cache partitioning so that all threads can receive equal slowdowns compared to the cases in which each of them monopolizes the SLLC. Nesbits *et al.* [54] introduce the notion of *virtual private caches* by means of SLLC partitioning such that the NoC bandwidth and cache capacity are fairly provisioned to satisfy some QoS requirement. Zhou *et al.* [55] argue that the SLLC miss count, a commonly used fairness metric in the literature, is inadequate for QoS consideration and propose to take into account the measures of both miss count and miss penalty of each thread when deciding SLLC partitioning.

2.5 Inclusiveness vs. Exclusiveness

Independent of cache organizations, another key design consideration for a CMP is whether inclusion or exclusion should be enforced across its cache levels. Then, a cache hierarchy can be categorized into three classes: (1) inclusive: if the content of upper-level caches is strictly a subset of the lower-level cache; (2) exclusive: if the upper- and the lower-level caches have no content in common at any time; (3) non-inclusive: if it is neither inclusive nor exclusive.

Although inclusion can greatly simplify cache coherence protocols [56], inclusion is prone to the weakness of bringing a large number of duplicated blocks across different cache levels, reducing effective cache capacity. The inclusion property also requires that,

if a block is evicted from the lower-level cache (e.g., LLC), all copies in the upper-level caches have to be invalidated, even though those copies are "hot" at the upper level. Recently, Jaleel *et al.* [57] have proposed to convey the temporal locality information from L1 caches to the shared L2 cache and utilize it to guide L2 replacement, so that the copy of "hot" L1 blocks can be retained longer in L2.

For the exclusive cache hierarchy, effective cache capacity is maximized as a result of the uniqueness of data stored across cache levels. In an exclusive LLC, a cache line is allocated and deallocated upon an inner-level eviction and an LLC hit respectively [58]. Therefore, exclusive LLCs may exhaust on-chip bandwidth to support frequent insertions of cache lines that are evicted from the inner level [59]. Worse still, the deallocation upon an hit will make the exclusive LLC lose track of the locality information of the corresponding block for replacement policies. To tackle this issue, Gaur *et al.* [58] propose to leverage the number of trips made by a block between inner- and last-level caches and the hit count contributed by the block during its residency in the inner-level cache to guide bypass and insertion algorithms for exclusive LLCs.

The non-inclusive cache hierarchy tries to take the middle way between the inclusive and the exclusive ones, but it cannot achieve the best sides of the two. For example, the data redundancy in non-inclusive caches is still high, thereby reducing performance when a workload does not have access to enough effective capacity [59]. Based on the observation that both non-inclusion and exclusion rely on similar hardware support, Sim *et al.* [59] propose a dynamic mechanism called FLEXclusion that is able to adapt a cache hierarchy to either exclusion to maximize effective capacity or non-inclusion to minimize NoC traffic, depending on the workload characteristics.

2.6 Prefetching

Cache prefetching is a memory latency hiding technique that speculatively brings data to higher cache levels (e.g., from main memory to on-chip LLC) in anticipation of future requests for it. Prefetching can be realized in software or hardware. Software-based prefetching takes advantage of compile-time information to instrument the original program with extra prefetch instructions. Hardware-based prefetching detects memory access patterns on the fly and preload data accordingly.

Generally speaking, there are four types of memory reference patterns in real-world applications, *sequential*, *stride*, *linked* and *irregular* [18]. The sequential reference pattern means that consecutive blocks following the block being accessed will also be referenced shortly. For instance, streaming applications typically exhibit this kind of pattern. The stride reference pattern implies that a memory access sequence spans multiple blocks with a fixed distance/step in addresses. This pattern is common in programs that access a certain field in an array of objects successively. In many high-level languages (e.g., C) with the notion of pointers, the data pointed by one address can be used to compute a subsequent address, which essentially forms a linked reference pattern. If a memory access sequence does not fall into any of the three aforementioned categories, it is considered to be an irregular reference pattern.

The research community has mainly focused on two topics about prefetching, of which one is how to improve its accuracy and the other is how to determine and control the ratio, priority and timeliness of prefetch requests for optimal performance. Lin *et al.* [19] devise an approach of adjusting the issue rate of prefetch requests to maximize the DRAM row-buffer hit ratio, as well as always inserting prefetched blocks into the LRU positions of LLCs to minimize prefetch-induced cache pollution. Zhuang *et al.* [22] propose to use a history table that is based on either PC or memory addresses to

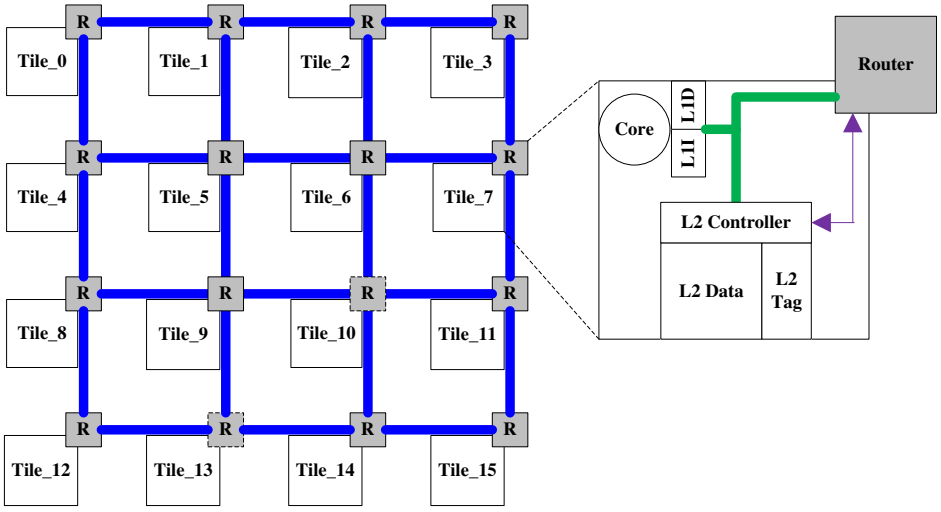


Figure 2.6: Tiled CMP Architecture

filter out useless prefetch requests. Srinath *et al.* [21] design a sophisticated framework based on feedback control, in which prefetch accuracy, timeliness and cache pollution are quantified and leveraged to minimize the negative effects due to prefetching. Ebrahimi *et al.* [23] propose a hierarchical coordination mechanism to maximize the benefits of prefetching on individual cores while minimizing inter-core cache interference that is originated from prefetching. Wu *et al.* [24] have recently proposed a prefetching-aware LLC replacement policy that is able to adapt cache insertion and promotion strategies by differentiating prefetch and demand requests.

2.7 Non-Uniform Cache Architecture

As the VLSI feature size keeps shrinking, more and more LLC capacity will be incorporated into CMPs. The large LLC has to get decomposed into a number of slices and be physically distributed across the die, rendering uniform access latency difficult and impractical. This gives rise to the *non-uniform cache architecture* [60], in which the access latency for an LLC slice is a function of its size and the communication distance between

the slice and the requesting core. As a result, cores can get faster access to adjacent slices and slower access to distant ones. Many researchers and chip designers believe that a *tiled architecture* is able to well support the increase in and the physical distribution of LLC capacity on the die [61]. In the tiled architecture, as depicted in Figure 2.6, a core is coupled with an LLC slice in each tile, and tiles communicate with each other via the NoC. The tiled architecture is attractive from both design and manufacturing perspectives, because its modularity and easy scalability to high core counts are suitable for many-core CMPs.

In the context of *non-uniform cache architecture* (NUCA) [60], the shared LLC organization can incur higher hit delays, because most blocks being accessed by a core reside in non-local SLLC slices. *Victim replication* (VM) [62] reduces SLLC hit latency by retaining/replicating each core's L1 victim blocks in its local SLLC slice if a victim's home slice is non-local. Noticing that blind replication would diminish the effective SLLC capacity and thereby hurt performance, the *adaptive selective replication* (ASR) [63] mechanism selectively replicates shared read-only data and adaptively modulates the replication level to minimize hit latency without an obvious increase in cache misses.

2.8 Cache Coherence

In CMPs, the same datum may have several copies in different cores' private caches because of data sharing. *Cache coherence* is a correctness-critical mechanism that maintains the logical consistency of these copies upon a update to any of them. To maintain cache coherence, a *coherence protocol* defines how private caches behave in the event of read, and more importantly, write requests. There are such protocols as MESI, MOESI, etc., with different complexity and performance features [64].

Generally speaking, existing coherence protocols can be classified into two categories

according to how they are realized in hardware, *snoop-based* and *directory-based*. Snoop-based protocols typically utilize a centralized "snoopy" bus to carry out coherence actions. Whenever a core needs to write to a piece of shared data, the write attempt will be broadcast to other cores via the NoC (typically a snoopy bus). Then, other cores having the same data will invalidate (or update) their local copies. A snoop coherence protocol, however, is not scalable to a large number of cores. This is because the centralized bus can be saturated and become a severe bottleneck if excessive coherence messages from a number of cores all go through it. Another obstacle preventing broadcast from being effective and scalable is its extremely low power efficiency. Since a snoop-based protocol does not provide the exact sharer information for common data, all cores sniffing on the bus have to look up the tag arrays of their private caches upon the arrival of a coherence message, even though many of them may not have a copy of the data at all. Therefore, the more cores are snooping on the bus, the more power is likely to be unnecessarily consumed.

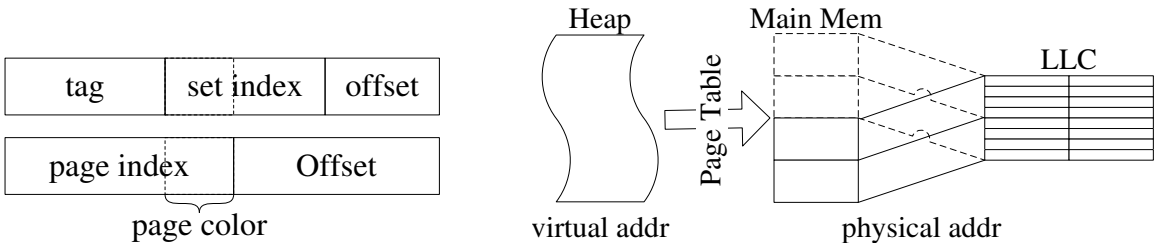
In contrast, a directory-based protocol is more scalable to a large number of cores, since a directory maintains exact sharer information that can guide precise coherence actions. The coherence directory is typically organized as a set-associative structure that has two fields in every entry, namely, the *coherence state field* and the *sharer information field*. In large-scale CMPs, the coherence directory is also banked and distributed. The definition of a coherence state depends on the implemented protocol. For instance, an MOESI protocol can have five possible states: *modified*, *owned*, *exclusive*, *shared* and *invalid*. With respect to how the sharer information is maintained, the simplest design is the *full-bit sharer vector*, which has exactly N bits to track all of the N cores on a CMP. If a core needs to write to a shared data copy, the write request will first be sent to the directory. Since the directory knows the exact sharer information by looking up the vector, the coherence messages (either invalidation or update) can be directly sent to those relevant

sharers only.

In the literature, there are a plenty of studies on coherence issues dating back to 1980s, 1990s and early 2000s, which established the foundations of coherence problems and solutions for multiprocessor systems. [65] presents a comprehensive survey of the related work prior to 2000. Recently, directory coherence issues for future many-core CMPs have received noticeable attention. Proximity coherence [66] improves performance by enabling neighboring cores' private L1 caches to directly obtain shared data from each other and autonomously maintain coherence among themselves without frequently accessing the coherence directory. [67] devises a content-aware coherence directory that essentially identifies and consolidates identical sharer vectors so as to reduce the redundancy and overhead of maintaining sharer information. G. Kurian *et al.* [68] design a new coherence directory for future 1024-core CMPs overlaid with an on-chip optical NoC by exploiting the high-bandwidth and low-latency features of the optical NoC. Kelm *et al.* [69] propose to use off-chip memory to store evicted sharer lists and later directly bring the lists into directories when they are needed again. Cuesta *et al.* [70] propose to avoid tracking private blocks that do not need coherence and decrease the contention on directory caches with the help of operating systems. Sanchez *et al.* [71] propose to take advantage of *cuckoo hashing* to assign a variable number of tags to maintain sharer information for different blocks according to their sharing degrees.

2.9 OS-Guided Cache Management

Page coloring [72] is an OS-guided approach that flexibly maps a virtual page to a region of consecutive sets in a physically-indexed cache. The memory management unit of an OS dedicates the upper bits of a physical address to the physical page number. When the address is applied to data lookups in a physically-indexed cache, the set-index field in the



(a) The Page Color Field in a Physical Address

(b) The Virtual-to-Physical Address Translation & Memory-to-Cache Mapping

Figure 2.7: Page Coloring

middle of the physical address is used to determine which cache set the data is located in. As depicted in Figure 2.7(a), there exist several overlapped bits between the physical page number and the cache set index, which is termed as a *page color*. Physical pages with an identical color will be mapped to a cache region which consists of sets with the same color value, implying that the mapping between physical pages and physically-indexed cache sets is fixed. But since the OS can manipulate a page table to determine which physical frame (e.g., 4KB in Linux) a virtual page is located in, the OS is thus able to flexibly map a virtual page to any cache region, as illustrated in Figure 2.7(b).

First proposed in [73], by using the lower bits of a block's physical page index rather than the block index to determine its home LLC slice, OS-guided SLLC management is shown to have a direct and flexible impact on CMP's SLLC hit latency and sharing degrees. Lin *et al.* [74] propose to partition the SLLC capacity to optimize throughput, fairness or QoS for CMPs by allocating page colors to different cores. Soares *et al.* [75] propose the *run-time operating system cache-filtering service* (ROCS) that designates a small LLC cache region which a physical page can fit in as a pollute buffer. Pages exhibiting high miss rates are recolored and remapped to the pollute buffer, preventing them from polluting others that exhibit high hit rates. Awasthi *et al.* [76] propose to utilize shadow address bits in migrating shared pages to optimal locations. Chaudhuri [77] devises a

HW mechanism to support the decision-making on when and where to migrate an entire page so as to amortize performance overhead. A recent LLC design called the *reactive NUCA* (R-NUCA) [78] classifies LLC accesses into distinct categories and adapts data placement, replication and migration to individual access categories via page coloring. Ding *et al.* [79] make the observation that the file system dataset has inferior temporal locality compared to the virtual memory dataset. Therefore, they propose the *selected region mapping buffer* (SRM-Buffer) design, which segregates the file system dataset and designates a small LLC region to accommodate this part of data by page coloring, so that the virtual memory dataset can be allocated with most LLC capacity and free from interference.

Chapter 3

Exploiting Set-Level Non-Uniformity of Capacity Demands to Enhance CMP Cooperative Caching

3.1 Problem Definition

As chip multiprocessors (CMP) are becoming predominant in processor manufacturing, computer architects are challenged to design CMPs in a way that fully exploits the performance potentials of multiple cores. One of the key research issues is to reduce the high cost of off-chip memory accesses, which are generally determined by the three factors: access latency, bandwidth and the number of off-chip accesses. While there are techniques such as 3D memory stacking [80], prefetching [81] and optical I/Os [82] that can help reduce (or hide) the long latency and increase the bandwidth of DRAM accesses, on-chip last level caches (LLC) play an irreplaceable role in reducing the number of DRAM accesses by keeping as much data as possible on-chip for future references, which necessitates a very effective management of CMP LLCs.

Recently, some researchers [48, 49] have advocated the private LLC (L2P) cache organization in future cache designs, since in comparison with the shared LLC (L2S) cache organization, L2P is experimentally found to have lower access latency, lower requirements for on-chip interconnects, better performance isolation and easier support for resource management. However, due to the limited cache capacity accessible to each core, the miss rate of L2P can be higher than L2S when a core's cache resource demand exceeds its local private L2 capacity.

To tackle the problem, Chang and Sohi [32] propose the mechanism of *cooperative caching* (CC) to allow capacity sharing among different "private" L2 caches by enabling each cache to utilize the capacity of others as victim caches. But in their proposal, cooperative caching is performed regardless of the performance implication: whenever a block is evicted from its own private cache, cooperative caching attempts to retain the block in one of the peer L2 caches, whether or not spilling the block to a peer cache will help the overall performance. For instance, a streaming application can actually always prevail in cooperative caching since it continuously replaces cache blocks; but having its victim blocks cooperatively cached will not be beneficial at all. Instead, retaining its victim blocks can adversely hurt other L2 caches' performance, since cooperative caching comes at the cost of occupying other caches' capacity.

To overcome the shortcoming of cooperative caching, Qureshi [33] has recently proposed the *dynamic spill receive* (DSR) paradigm to regulate block spilling and receiving in response to different applications' cache resource requirements. In the DSR paradigm, applications are classified into two categories: *taker* and *giver* applications. Taker applications can have their performance improved with additional cache capacity, while giver applications can contribute part of their cache capacity to others with little performance degradation. When taker and giver applications are co-scheduled on a CMP, taker applications' L2 caches can spill victim blocks to those of giver applications, but

not vice versa. While this approach is shown to improve the overall performance when the non-uniformity of cache resource demands explicitly exists at the application level, it becomes less effective for workloads of which such non-uniformity is manifested at a finer granularity as demonstrated in this chapter.

The objectives of this chapter are to prove the existence of non-uniform capacity demands at the cache-set level and thereby to exploit this non-uniformity to further enhance the effectiveness of cooperative caching. The key insight of this chapter is that differentiating the cache resource demands only at the application level is insufficient for enhancing cooperative caching when the performance-sensitive non-uniformity of capacity demands does not surface to the application level but instead exists at the cache-set level. This chapter then presents a novel L2 cache design, called the *Set-level Non-Uniformity identifier and Grouper* (or SNUG), which identifies and flexibly groups cache sets with complementary capacity demands for cooperative caching. Evaluation results show that the SNUG cache design can significantly boost the effectiveness of cooperative caching with manageable hardware overhead.

3.2 Research Motivations

Previous studies [33, 14] have revealed that applications have diverse requirements for cache resources. They try to utilize the application-level difference in resource demands to optimize the utilization of CMP L2 caches for multi-programmed workloads. Distinct from previous work, however, we take further steps to evidence the existence of non-uniform capacity demands at the cache-set level. To accomplish this goal, we need to first develop a group of mathematical models that accurately quantify a cache set's capacity requirement. With these models, we can characterize the set-level non-uniformity of capacity demands. Finally, we argue that this fine-grained non-uniformity can be utilized

Table 3.1: Glossary of Notation and Terms Used

Symbol	Annotation
N	the total number of sets in an L2 cache
A	#blocks owned by a set, namely the associativity, where $0 \leq A < \infty$
S	the index of a set, $0 \leq S \leq N-1$
I	a fine-grained sampling interval for workload characterization
$miss\_count(S, I, A)$	the number of misses on set S with A blocks during sampling interval I
$hit\_count(S, I, A)$	the number of hits on set S with A blocks during sampling interval I
$blocks\_required(S, I)$	the number of blocks required by set S during sampling interval I
$A_{threshold}$	a value of associativity that is large enough to approximate ∞
$A_{baseline}$	the associativity (integral power of 2) of the baseline private L2 cache
M	the number of buckets/sub-ranges within $[1, A_{threshold}]$
$bucket_j$	the j^{th} bucket, which is the subrange $[\frac{(j-1)\cdot A_{threshold}}{M} + 1, \frac{j\cdot A_{threshold}}{M}]$, where $1 \leq j \leq M$
$MF(S, I, bucket_j)$	a membership function used to indicate if the number of blocks required by set S is categorized into the j^{th} bucket during interval I
$size\_bucket_j(I)$	the size of the j^{th} bucket during interval I

to further optimize inter-core cooperative caching, achieving better performance than the application-level approaches.

3.2.1 Quantification of Set-Level Capacity Demands

We start with defining the notation and the terms in Table 3.1.

3.2.1.1 Modeling Set-level Capacity Demands

Since a cache set can be treated as an array of blocks, under a fixed block size, we can use the number of blocks in a set to measure the number of cache resources possessed by the set. Intuitively, if a set has enough blocks during a specific time interval, there will be no capacity or conflict misses on the set, because these two kinds of misses happen only when the set's resources are limited. Therefore, if we denote the capacity demand of a particular set during a certain time interval as $blocks\_required(S, I)$, where S is the index of the set and I is the time interval that we are interested in, we can define it as the

minimum number of blocks required to resolve all capacity and conflict misses for the set during the interval.

We introduce another function, $miss\_count(S, I, A)$, which means the number of misses on set S during interval I when S has A blocks. Under the LRU replacement policy that has the stack property [42], the following relationship is always true: $miss\_count(S, I, 0) \geq miss\_count(S, I, 1) \geq \cdots \geq miss\_count(S, I, \infty)$. From this property, we can also infer that $miss\_count(S, I, A)$ is monotonically non-increasing for given S and I when only A increases. Ideally, if set S could get an infinite number of blocks ($A = \infty$) during interval I, there would be no capacity or conflict misses on the set. At the other extreme, if set S had no blocks at all ($A = 0$), all accesses to the set during interval I would miss. Consequently, $miss\_count(S, I, \infty)$ is equal to the number of compulsory misses on set S during interval I, while $miss\_count(S, I, 0)$ is equivalent to the number of accesses to set S during interval I.

If set S's capacity demand is satisfied during interval I, which means that set S gets as many blocks as $blocks\_required(S, I)$, only compulsory misses can happen to set S. Thus, we give a quantitative definition of $blocks\_required(S, I)$ in Equation 3.1.

$$\begin{aligned} &blocks\_required(S, I) = \min A \\ &\text{s.t. } miss\_count(S, I, A) - miss\_count(S, I, \infty) = 0 \end{aligned} \tag{3.1}$$

Since it is impractical to measure $miss\_count(S, I, \infty)$ when the set associativity A is ∞, and also because the function $miss\_count(S, I, A)$ is monotonically non-increasing for given S and I when only A increases, we can use a finite number $A_{threshold}$ that is large enough to approximate ∞. Then, we can use Equation 3.2 to quantify the capacity

demand of a set.

$$\begin{aligned} &blocks\_required(S, I) = \min A \\ &\text{s.t. } miss\_count(S, I, A) - miss\_count(S, I, A_{threshold}) = 0 \end{aligned} \tag{3.2}$$

Alternatively, since $miss\_count(S, I, 0)$ is equivalent to the number of accesses to set S during interval I, the total number of hits on set S during interval I when the set has A blocks (denoted as $hit\_count(S, I, A)$) can be expressed as $hit\_count(S, I, A) = miss\_count(S, I, 0) - miss\_count(S, I, A)$. Therefore, Equation 3.2 can be reformulated as Equation 3.3:

$$\begin{aligned} &blocks\_required(S, I) = \min A \\ &\text{s.t. } hit\_count(S, I, A) - hit\_count(S, I, A_{threshold}) = 0 \end{aligned} \tag{3.3}$$

Practically, Equation 3.3 is more convenient than Equation 3.2, because it is much easier to locate a position in the LRU stack when an access to a set is a hit [12]. Equivalently, $hit\_count(S, I, A)$ is actually the total number of hits on the LRU positions that are smaller than or equal to A on set S during interval I.

3.2.1.2 Characterizing Set-Level Non-Uniformity of Capacity Demands

From the aforementioned analysis, we can infer that $blocks\_required(S, I)$ is in the integer range $[1, A_{threshold}]$. Without loss of accuracy, we divide the integer range $[1, A_{threshold}]$ into M sub-ranges (a.k.a., buckets) of equal length $bucket_1, bucket_2, \cdots, bucket_M$, where $bucket_j = [\frac{(j-1)\cdot A_{threshold})}{M} + 1, \frac{j\cdot A_{threshold}}{M}]$ for $1 \leq j \leq M$. Then, for a given interval I, set S is said to be categorized into $bucket_j$ if and only if the value of $blocks\_required(S, I)$ is in the integer range $[\frac{(j-1)\cdot A_{threshold}}{M} + 1, \frac{j\cdot A_{threshold}}{M}]$. Further, because any two adjacent buckets have no intersection, the value $blocks\_required(S, I)$ will be in one and only one bucket's

range. Therefore, we can differentiate two cache sets in terms of their individual capacity demands if their $blocks\_required(S, I)$ values belong to different buckets. Here, we restrict both $A_{threshold}$ and M to be an integral power of 2.

To identify if set S is categorized into the j^{th} bucket during interval I, we can define a membership function $MF(S, I, bucket_j)$ to indicate if set S has a capacity demand that is in the range of $bucket_j$ during interval I, which is formulated in Equation 3.4:

$$MF(S, I, bucket_j) = \begin{cases} 1, & \text{if } blocks\_required(S, I) \in bucket_j \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

For all of the N sets in an L2 cache, we are interested in how many sets are categorized into each one of the M buckets during sampling interval I, because any two sets that are categorized into distinct buckets will show different set-level capacity demands. Here, we normalize the number of sets that are categorized into the j^{th} bucket during interval I by the total number of sets N, define it as the size of the bucket for that interval, and denote the value as $size\_bucket_j(I)$. The formal definition of $size\_bucket_j(I)$ is shown in Equation 3.5.

$$size\_bucket_j(I) = \frac{\sum_{S=0}^{N-1} MF(S, I, bucket_j)}{N}, \text{ where } 1 \leq j \leq M \tag{3.5}$$

In summary, we can characterize the set-level non-uniformity of capacity demands for all of the N sets in an L2 cache by using Equation 3.5.

3.2.1.3 Methodology of Characterization

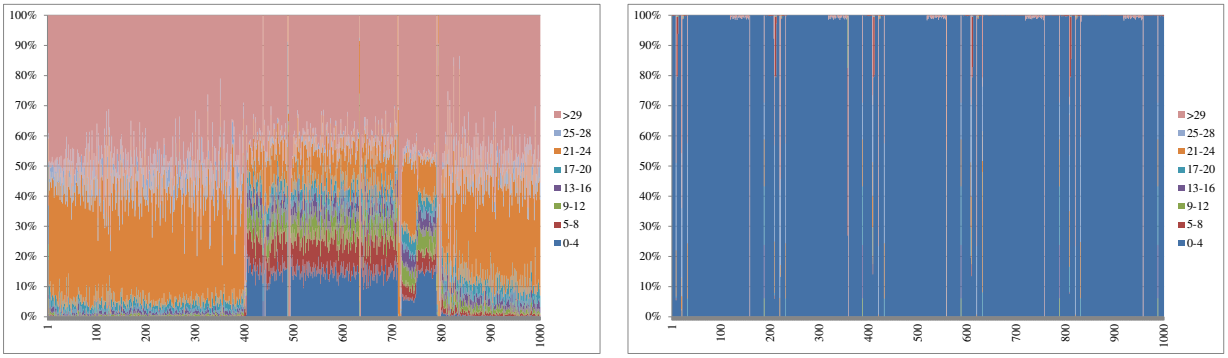
We experiment on all of the 26 SPEC CPU 2000 benchmarks [83] using the *sim-cache* tool of Simplescalar [84], and analyze the set-level capacity demands distributions of their L2 caches. The configurations of L1 and L2 caches are listed in Table 3.2. Specifically, there are 1024 sets in the L2 cache ($N = 1024$). All of the benchmarks are executed with the reference data inputs. For each benchmark, we fast forward the execution by 6 billion cycles and then simulate the caches until 1000 sampling intervals of which each contains 100K L2 accesses are encountered. Therefore, the variable I is in the range [1,1000]. Within sampling interval I, for an L2 set S, we track the number of hits on set S at each LRU position A that is smaller than or equal to $A_{threshold}$, and then find the minimum A (a.k.a. $blocks\_required(S, I)$) such that $hit\_count(S, I, A) = hit\_count(S, I, A_{threshold})$, where $A_{threshold}$ is assumed to be the double of $A_{baseline}$ ($A_{baseline} = 16$) in this chapter.

We further divide the entire range $[0, A_{threshold}]$ into 8 buckets $\{[0, 4], [5, 8], \cdots, [29, 32]\}$. Then, for all of the 1024 sets and 1000 sampling intervals, we can obtain the normalized size of each bucket, $size\_bucket_j(I)$ for $1 \leq j \leq 8$, which is actually the distribution of set-level capacity demands for all of the L2 sets during the entire sampling period.

3.2.1.4 Characterization Summary

To summarize, we find that among the 26 SPEC CPU 2000 benchmarks, there are 7 applications (*ammp*, *apsi*, *galgel*, *gcc*, *parser*, *twolf*, *vortex*) that show strong set-level non-uniformity of resource demands. Figure 3.1 [1] illustrates the distributions of set-level capacity demands for two applications, between which *vortex* shows strong set-level non-uniformity of capacity demands but *applu* does not. In Figure 3.1, the x-axis shows 1000 sampling intervals each of which contains 100K L2 accesses, while the y-axis

[1] Another set of similar figures is Figure 4.2.



(a) *vortex* (b) *applu*

Figure 3.1: Distribution of Set-level Capacity Demands

represents the distribution breakdowns for the 8 buckets corresponding to the 8 legends $\{[0,4],[5,8],\cdots,[29,32]\}$.

For instance, although *vortex* has been shown to benefit from additional cache resources at the application level in previous research [85], Figure 3.1(a) clearly indicates that there exists significant set-level non-uniformity of capacity demands for *vortex*. Specifically, from sampling interval 405 to about 792, about 15% sets require only 0-4 blocks, about 9% sets require 5-8 blocks, and over 7% sets require 9-12 blocks. In contrast, for the streaming application *applu* shown in Figure 3.1(b), almost all sets require only 0-4 blocks during the entire sampling period.

3.3 The SNUG Architecture

SNUG is designed to exploit the fine-grained set-level non-uniformity of capacity demands to enhance the performance of cooperative caching. It aims to accomplish two specific goals: (*i*) identifying the capacity demand of each L2 set, and (*ii*) grouping peer sets (from different cores) that have complementary set-level capacity demands for flexible cooperative caching.

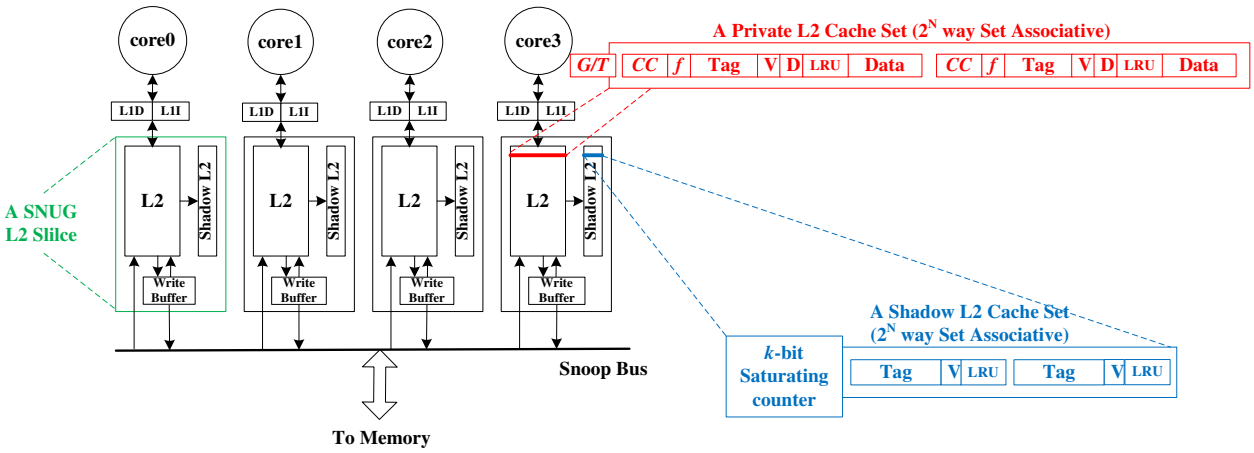


Figure 3.2: SNUG L2 Cache on a Quad-Core CMP

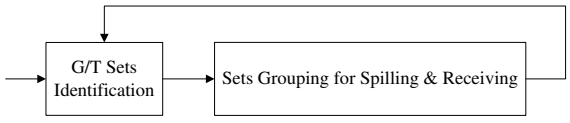


Figure 3.3: G/T Sets Identifying and Grouping Stages

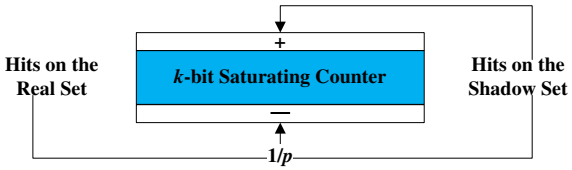


Figure 3.4: Operations on a Saturating Counter

Figure 3.2 illustrates a high-level view of a quad-core CMP with SNUG. Each core has a split private L1 instruction/data cache and a SNUG L2 slice that is a "private" cache capable of cooperative caching, a shadow L2 cache that is used to monitor the set-level capacity demands in the "private" L2 cache, and an L2 write-back buffer that frees the private L2 cache from write-back stalls and supports direct data read from the write buffer [86]. Within a SNUG L2 slice, the shadow L2 cache has the same number of sets as the L2 cache, and a one-to-one correspondence is maintained between two sets that have the same index in the L2 cache and its shadow cache. As Figure 3.2 shows, the block of a shadow set has all of the usual fields as an ordinary L2 block except for the data field. In addition, there is a per-set saturating counter associated with each shadow set. The design and working principles of a shadow L2 set will be elaborated in Section 3.3.1, and a detailed overhead analysis of this organization appears in Section 3.4.4.

During the program execution, the SNUG operation alternates between two stages, as shown in Figure 3.3. The first stage is used to identify the status of each L2 set as either a giver (G) or a taker (T) set using the per-set capacity demand monitor. Then, at the beginning of the second stage, the dynamic information about L2 sets is utilized to regroup them for spilling and receiving. Each two-stage cycle defines a sampling period: Stage I determines the G/T status of each L2 set after a sampling epoch of 5 million cycles; then Stage II follows for 100 million cycles until the start of the next sampling period, with a novel *index-bit flipping* scheme utilizing the G/T information in grouping sets for cooperative caching during Stage II. Typically, Stage I is much shorter than Stage II, and the total time of the two is shorter than a program phase during which the program exhibits relatively stable set-level capacity demands.

3.3.1 Identifying Giver and Taker Sets

In this part, we first explain the structures of the SNUG L2 sets and the shadow sets (shown in Figure 3.2) and how they are updated. Then we describe a HW scheme for measuring the set-level capacity demands and identifying the giver/taker status of each set based on the measurement.

3.3.1.1 The Structures of "Private" & "Shadow" L2 Sets

In an L2 cache, shown in Figure 3.2, besides the typical fields such as tag, valid, dirty, LRU and data, each cache line is augmented with a CC bit that indicates whether this cache line is owned by the local processor core (when $CC = 0$) or it is cooperatively cached (when $CC = 1$). Another bit f is used in the *index-bit flipping scheme* and takes effect only when the CC bit is set. If the f bit is one, it means that the line is cooperatively cached with the last bit of its original set index flipped. There is also a G/T bit incorporated in

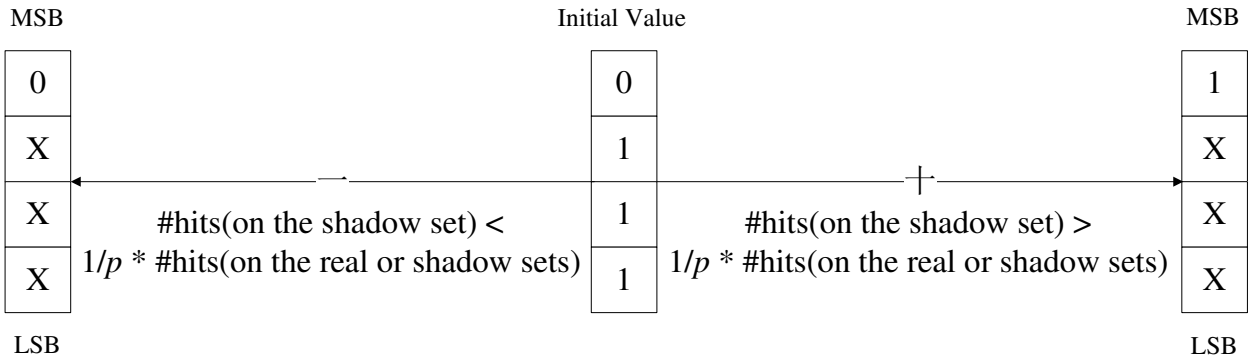


Figure 3.5: Illustration of the Operations on a 4-bit Saturating Counter

each L2 set, which is used to indicate whether the set is a giver (when $G/T = 0$) or taker (when $G/T = 1$) set. The G/T bits for all L2 sets form a G/T vector, each entry of which is addressable independent of addressing the L2 sets.

Each entry in a shadow set has a tag field, a valid bit and LRU bits. The shadow set retains the "shadows", namely the tag fields, of locally evicted lines from the corresponding L2 set: when an L2 set needs to replace a line, and the victim line is owned by the local processor core, the shadow set will retain the tag field of the victim line in one of its entries and set the entry valid. Additionally, the shadow L2 set maintains its own independent LRU ranking for all of its valid entries and applies the information to replacement. It is required that the shadow set entries be strictly exclusive with the local lines in the corresponding L2 set in terms of their tag fields. Therefore, if a formerly-evicted block with its tag present in the shadow set is revisited by the local core, two actions will be taken: (1) the shadow entry that has the target tag needs to be invalidated after the corresponding block enters the real set; (2) a hit on the shadow set is signaled to manipulate its saturating counter.

3.3.1.2 Monitoring Set-Level Capacity Demands

If an L2 set and its corresponding shadow set have the same associativity, the private and shadow sets implicitly form two buckets as defined in Section 3.2. Then, we can use the per-set saturating counter to monitor the set-level capacity demand, based on which set-level takers and givers are identified and grouped for cooperative caching.

Since an L2 set and its shadow set form two buckets, according to Equation 3.3, we can use the ratio σ (defined in Equation 3.6) to measure the potential hit rate increase if the capacity of the L2 set is doubled with respect to the number of cache blocks. If σ is greater than a predefined threshold $\frac{1}{p}$, where p is an integer, it is expected that doubling the capacity of the L2 set can lead to an increase in the hit rate by $\frac{1}{p}$. This is because $\sigma > \frac{1}{p}$ is equivalent to the relationship in Equation 3.7.

$$\sigma = \frac{\#hits(\text{on the shadow set})}{\#hits(\text{on the L2 set}) + \#hits(\text{on the shadow set})} \tag{3.6}$$

$$\#hits(\text{on the shadow set}) - \frac{1}{p} \times (\#hits(\text{on the L2 set}) + \#hits(\text{on the shadow set})) > 0 \tag{3.7}$$

To implement this idea, we define the operations on a saturating counter as follows (also shown in Figure 3.4): (1) every hit on the shadow set increments the saturating counter by 1; (2) after every p hits on the private or the shadow sets, the saturating counter is decremented by 1. Then, the outcome of the two operations can be reflected by the MSB (*most significant bit*) of the saturating counter. This is shown in an example in Figure 3.5: a one-valued MSB of the counter indicates that the L2 set has a higher capacity demand than that provided by its local L2 cache, and that doubling its capacity can potentially lead to an increase in the hit rate by at least $\frac{1}{p}$.

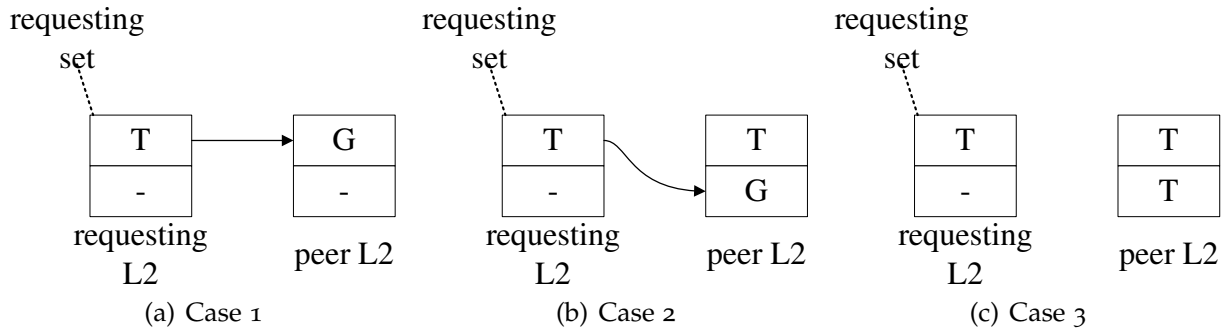


Figure 3.6: Index-Bit Flipping Scheme

3.3.1.3 G/T Sets Identification

As described above, we can differentiate taker and giver sets by just checking the MSB of the saturating counter of each set. A one-valued MSB indicates that extending the capacity of the set is beneficial; hence the set should be regarded as a taker and enabled to spill blocks in cooperative caching; otherwise, the set is defined as a giver and receives spilled blocks from its peer taker set. Thus, the MSB of the saturating counter can be directly used to update the corresponding entry of the G/T vector.

3.3.2 Grouping Sets for Spilling & Receiving

After the *G/T sets identifying stage*, the SNUG caches enter the *sets grouping stage* to group different cores' sets with complementary capacity demands to perform block spilling and receiving. The simplest grouping strategy is to couple different cores' sets with an identical index, as is done in CC or DSR. But this naïve approach only allows the sets with the same index to form a receiving & spilling group, which is too restrictive. Ideally, we would like to group taker and giver sets based on their capacity demands and supplies, totally independent of their index values (related designs will be discussed in the next chapter). Here, for the work presented in this chapter, we propose a simple *index-bit flipping* scheme that is able to flexibly group sets with complementary capacity

demands for spilling and receiving, at the low hardware complexity of just one extra f bit per cache line.

The *index-bit flipping* scheme works as follows: in an L2 cache, when a taker set needs to spill a local cache line, the L2 cache will put a CC spilling request together with the address of the spilled line on the interconnection bus. By snooping on the bus, other peer caches can detect the CC request as well as the address of the spilled block. Each peer cache will look up its own G/T vector to find the G/T information of the two adjacent entries that have the same index as the CC-spilling block but with the last index bit being don't-care. There can be three cases as shown in Figure 3.6. In Case 1, if the set with an identical index is a giver set in the peer L2 cache, the peer L2 cache will attempt to retain the spilled block in its set with exactly the same index. In Case 2, if the set with the same index in the peer L2 cache is a taker set while the other set with the last index bit different is a giver set, this giver set will attempt to retain the spilled block. In Case 3, if the corresponding two adjacent sets are both taker sets, this peer L2 cache will not respond to the CC request. Any peer cache that first responds to the CC request on the interconnection bus will get the spilled block. Based on whether the block is cooperatively cached in the set with an identical index or with the last index bit flipped, the f bit of the cooperatively-cached block will be set to zero (if the last index bit is not flipped) or set to one (otherwise).

Now suppose a block is missed in its local L2 cache, the cache will signal a retrieving request for the block with its address on the snoop bus. After a peer cache detects the request, it will first lookup its G/T vector to get the information of the two adjacent G/T bits that have the same index as the block address but with the ending bit being don't-care. If the G/T bit with exactly the same index, or otherwise with only the last index bit different, indicates a giver set, the L2 cache will try to find the block in the corresponding giver set; if both of the adjacent peer sets are indicated as taker sets, it means that the block

being retrieved can't be located in this L2 cache. This leads to at most one unambiguous search for the block in a peer L2 cache. Because the cooperatively-cached block can only be located in a giver set of at most one peer L2 cache, the peer cache that has the cooperatively-cached block will directly forward the block to the requesting L2 cache. At the same time, the peer cache will invalidate its cooperatively-cached copy of the block to free space for other blocks. If no peer caches respond to the retrieving request, the requested block is not on chip and will need to be fetched from the DRAM.

3.3.2.1 Maintaining Cache Coherence

In the SNUG cache design, we use two restrictions to maintain coherence between different L2 caches. First, only when a locally-evicted block is clean can it be cooperatively cached in a peer L2 cache. If the block is dirty, it will be directly put in the local L2 write buffer. Second, if a peer cache forwards a cooperatively-cached block to the original owner cache of the block, the copy of the block in the peer cache needs to be invalidated.

3.4 Experiments & Evaluation

In this section, we describe the configurations of our simulated system and workload combinations, and compare our SNUG design against other LLC management schemes available in the literature.

3.4.1 Simulation Configurations

In our experiment, we use the cycle-accurate PolyScalar [87], a multi-core simulator with detailed memory hierarchy models and SimpleScalar out-of-order cores [84]. We implement and evaluate five L2 cache organizations, L2P, L2S, CC (Best), DSR, and SNUG. According to [32], one of the spill-probabilities 0%, 25%, 50%, 75% and 100% that

Table 3.2: Configurations of the PolyScalar Simulator

Out-of-Order Cores	
Cores	4
Address Bits	32
Fetch/Issue/Commit	8/8/8
LSQ/RUU Entries	64/128
ALU/FPU/Mult/Div	4/4/1/1
Branch Predictor	2-Level, 1024 Entry, History Length 10
BTB Size	512 Sets, 4 Way
Branch Penalty	3 Cycles
RAS Entries	8
Memory Hierarchy	
L1I/D	1 Cycle, 4 Way, 32KB, 64B Lines
Each L2 Slice	10 Cycles, 16 Way, 1MB, 64B Lines, Write Back
L2 Write Buffer	FIFO, Mergeable, 16 Entries $\times$ 64B/Entry, Support Direct Read
DRAM Latency	300 Cycles
Snoopy Bus	16B-Wide Split Transactional Bus, 4:1 Speed Ratio, 1 Cycle for Arbitration

Table 3.3: Performance Metrics

Metric	Definition (n is the core count)
Throughput	$TP(Scheme) = \sum_{i=1}^{n} IPC_i(Scheme)$
Average Weighted Speedup	$AWS(Scheme) = \frac{1}{n} \times \sum_{i=1}^{n} \frac{IPC_i(Scheme)}{IPC_i(Baseline)}$
Fair Speedup	$FS(Scheme) = n / \sum_{i=1}^{n} \frac{IPC_i(Baseline)}{IPC_i(Scheme)}$

produces the best performance is selected as CC (Best) for a given workload. Table 3.2 lists the configurations shared by the five L2 schemes above. The difference between the L2 schemes is the remote L2 access latency: for L2P, CC and DSR, we assume that the remote L2 access latency is 30 cycles, while the remote latency for SNUG is assumed to be 40 cycles covering the additional delays of looking up the G/T vector of each L2 cache.

For the purpose of thorough comparisons, three standard metrics (listed in Table 3.3) are used to quantify performance. Specifically, *throughput* that is the sum of IPCs

Table 3.4: Application Classification

Type	Workload Class	Application-Level Capacity Demand	Applications
set-level non-uniformity of capacity demands	A	>1MB	*ammp*, *parser*, *vortex*
	B	<1MB	*apsi*, *gcc*
set-level uniformity of capacity demands	C	>1MB	*vpr*, *art*, *mcf*, *bzip2*
	D	<1MB	*gzip*, swim, *mesa*

Table 3.5: Workload Combinations & Characteristics

Class	Characteristics
C1	4 identical applications from class A without data sharing (stress test)
C2	4 identical applications from class C without data sharing (stress test)
C3	(2 different applications from class A) + (2 different applications from class C)
C4	(2 different applications from class A) + (1 application from class B) + (1 application from class C)
C5	(2 different applications from class A) + (2 different applications from class D)
C6	(2 different applications from class A) + (1 application from class B) + (1 application from class D)

Table 3.6: Workload Selection

C1	4 *ammp* 4 *parser* 4 *vortex*	C3	(*ammp*+*parser*)+(*bzip2*+*mcf*) (*parser*+*vortex*)+(*mcf*+*art*) (*vortex*+*ammp*)+(*art*+*vpr*)	C5	(*ammp*+*parser*)+(*swim*+*mesa*) (*parser*+*vortex*)+(*mesa*+*gzip*) (*vortex*+*ammp*)+(*swim*+*gzip*)
C2	4 *vpr* 4 *bzip2* 4 *mcf* 4 *art*	C4	(*ammp*+*parser*)+(*apsi*)+(*bzip2*) (*parser*+*vortex*)+(*gcc*)+(*mcf*) (*vortex*+*ammp*)+(*apsi*)+(*art*) (*ammp*+*parser*)+(*gcc*)+(*vpr*)	C6	(*vortex*+*ammp*)+(*apsi*)+(*gzip*) (*parser*+*vortex*)+(*gcc*)+(*mesa*) (*ammp*+*parser*)+(*apsi*)+(*swim*) (*vortex*+*ammp*)+(*gcc*)+(*mesa*)

(*instructions per cycle*) evaluates the utilization of a system; *average weighted speedup* indicates reduction in execution time; *fair speedup* balances both performance and fairness.

3.4.2 Workload Combinations

Table 3.4 classifies the 12 SPEC CPU2000 benchmarks used in our studies. Our evaluation takes into account 6 different classes of workload combinations described in Table 3.5. Specifically, workload combination class C1 and C2 are both stress tests, which means that the four co-scheduled applications from C1 or C2 are all identical, but with the

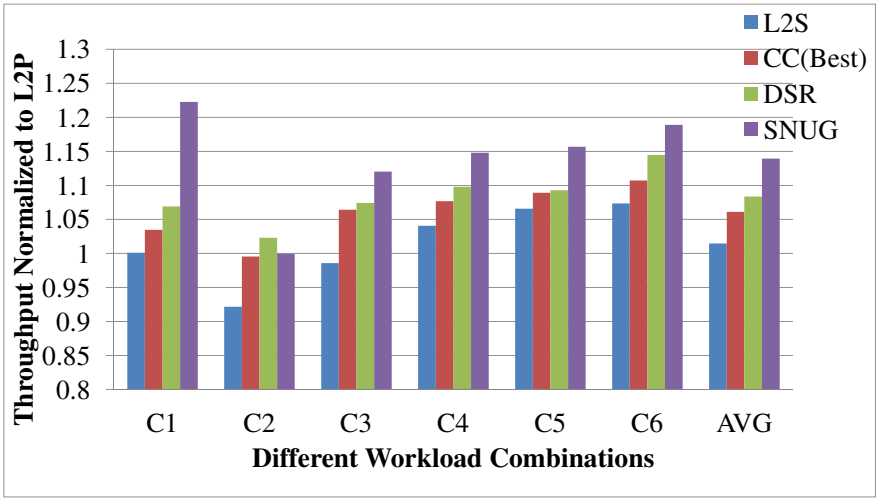


Figure 3.7: Throughput

assumption that there can be only capacity sharing among the co-scheduled applications, excluding any data sharing. The purpose of the stress tests is to see how different L2 cache designs can respond to applications' set-level capacity demands, since the identical co-scheduled applications have the same capacity demand at both application and set levels. Within a class in C3 - C6, all of the co-scheduled applications are different, and at least two applications showing set-level non-uniformity of capacity demands are chosen in each workload combination. Table 3.6 lists the 21 workload combinations that are categorized into the 6 different classes respectively.

3.4.3 Result Analysis

For each instance of simulation, we fast-forward the execution by 6 billion cycles to bypass the initialization section of the programs, and then execute each workload combination with the detailed out-of-order core model and different cache schemes for additional 3 billion cycles. In the results analysis, geometric means are calculated for all of the workload combinations within each given class.

Figure 3.7 shows the throughput of the L2S, CC(Best), DSR and SNUG schemes

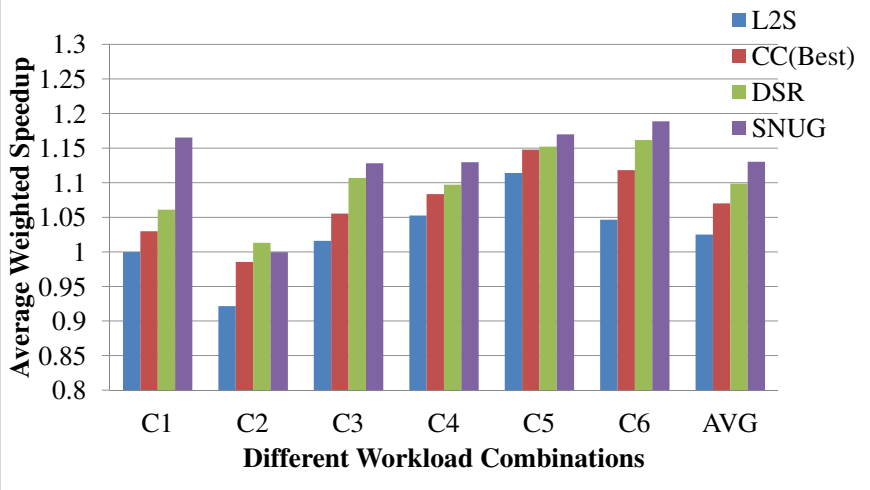


Figure 3.8: Average Weighted Speedup

normalized to L2P (1.0). In class C1 that is the stress test, because all of the applications have an application-level capacity demand of over 1MB and also exhibit the set-level non-uniformity of capacity demands, the SNUG cache organization can utilize the complementary capacity demands of interleaved taker and giver sets by the *index-bit flipping* scheme and then capture more opportunities for cooperative caching. Therefore, SNUG achieves a throughput improvement over the baseline L2P cache by 22.3% in class C1, better than the performance gain of CC(Best) by 3.5% and that of DSR by 6.9%. In C2, DSR achieves a throughput improvement over the baseline by 2.3%, and performs slightly better than CC(Best) (-0.5% performance degradation) and SNUG (-0.2% performance degradation), because DSR can assign some of the identical applications as taker applications while assigning others as giver applications to achieve biased performance improvement. In C3, C4, C5 and C6, SNUG outperforms any other cache scheme. Overall, on average, SNUG can improve the Quad-core CMP throughput by 13.9% for all of the 6 classes of workload combinations, in contrast to 8.4% of DSR.

Because the throughput metric is not fair to the application with a lower absolute IPC, we also use the metric of average weighted speedup to consider the change of relative

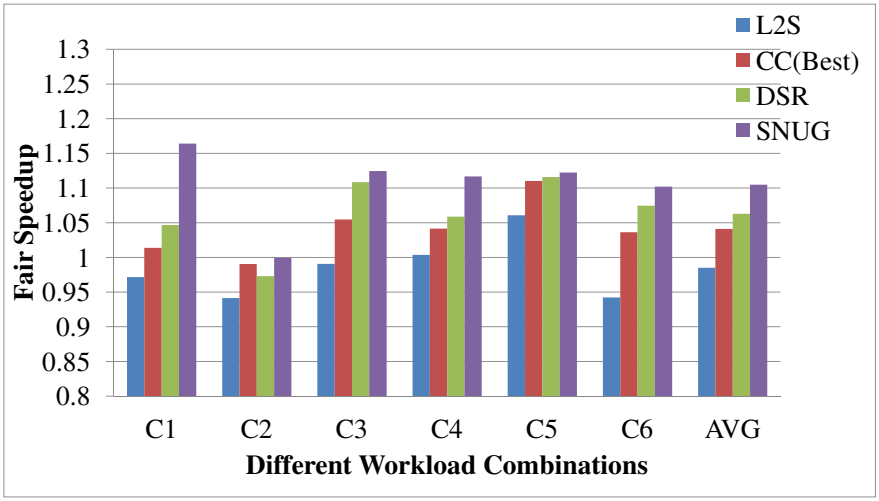


Figure 3.9: Fair Speedup

IPCs (the absolute IPC of a scheme over that of the baseline) of the applications. From Figure 3.8, it can be concluded that SNUG can also improve the average weighted speedup by 13.0%, while DSR, CC(Best) and L2S improve it by 9.9%, 7.0%, and 2.5%, respectively.

Figure 3.9 demonstrates the performance on the fair speedup metric (the harmonic mean of programs' relative IPCs) that balances both performance and fairness for different classes of workload combinations as well as different L2 cache schemes. On average, the SNUG scheme improves the performance by 10.4%, better than L2S (-1.5% degradation), CC(Best) (4.2%) and DSR (6.3%).

3.4.4 Space & Time Overhead Analysis

Since the SNUG caches require the per-set capacity demand monitor, the shadow sets and saturating counters account for the major hardware overhead in our design. Then, the storage overhead of the SNUG cache can be calculated by using Equation 3.8.

$$overhead = \frac{storage\_with\_SNUG - storage\_without\_SNUG}{storage\_without\_SNUG} \tag{3.8}$$

Table 3.7: Length of Each Field in the SNUG L2 Design

Field	Length
address length	32 bits
#(cache sets)	1024
set associativity	16
cache line size	64 byte
tag field length	16 bits
CC, f, v, d	1 bit each
RRPV field length	4 bits
$\log p$ (the length of the module p counter)	3 bits ($p = 8$)
k (the length of a saturating counter)	4 bits

Table 3.8: Overhead of Different Memory Address and Block Size Combinations

	32-bit address	64-bit address
64B/cache line	4.0%	5.6% (assuming that only 41 bits are used)
128B/cache line	2.1%	3.2%

Table 3.7 lists the length of each storage field in the SNUG design if we use the cache configurations in Table 3.8. Under such a cache configuration, the storage overhead of the SNUG cache design is only 4.0% by Equation 3.8, which is reasonably low when we consider the abundant silicon resources available as a result of technology scaling.

However, many processors, such as SUN's UltraSPARC-III [88], use 64-bit wide memory addresses. A longer memory address leads to a longer tag field in the shadow set that introduces more hardware overhead, although typically some leading bits of the memory address are unused (e.g., the leading 20 and 23 bits of the virtual address and physical addresses are unused in UltraSPARC-III respectively). We can offset the hardware overhead by adopting a larger cache block size while keeping the cache capacity fixed. Table 3.8 shows the hardware overhead of different memory address and cache line size combinations for a 1MB private L2 cache.

The tag field in the shadow set accounts for the largest portion of the storage overhead. However, because the tag field in the shadow set does not affect the semantics of running

threads at all, we plan to design a hash function for shadow caches to shorten their tag fields. For instance, for an UltraSPARC-like CPU with 64-byte L2 cache lines, if we can design and get a 16-bit hash value of a block's original 41-bit tag filed and use it as a new shadow tag, we can significantly reduce the storage overhead from 5.6% to 1.1%. We leave this part in the next chapter.

With respect to the time overhead, in our SNUG cache implementation, we experimentally observed that a combination of 5 million cycles for the *G/T sets identifying stage* and another 100 million cycles for the *sets grouping stage* produces a good performance outcome. Therefore, the parameters are adopted in our experiments. During the 5 million cycles of the *G/T sets identifying stage*, the cache can still accept retrieving requests but no spilling requests from others. At the end of this stage, each L2 cache maintains a new G/T vector, and continues to use the set-level G/T information to group sets for block spilling or spilling.

3.5 Summary

Although the *cooperative caching* allows CMP private L2 caches to share their capacity, its effectiveness is quite limited by its working principle of eviction-driven spilling and receiving. The *dynamic spill and receive* (DSR) technique improves cooperative caching by taking into account the differences in capacity demands that appear at the application level. However, DSR is less effective when such differences manifest themselves at the cache-set level but not at the application level. Our investigations reveal that this situation is common, motivating our proposal of the *Set-level Non-Uniformity identifier & Grouper* (SNUG) scheme that can exploit the fine-grained non-uniformity via cooperative caching to improve system performance. Experiments show that, for six classes of workload combinations, our SNUG cache can improve the Quad-Core CMP throughput by 22.3% at

best and by 13.9% on average over the baseline configuration, outperforming the prior-art DSR scheme that achieves an improvement by up to 14.5% and 8.4% on average.

Chapter 4

Spatiotemporal Capacity Management for Intra-Core Last Level Caches

4.1 Problem Definition

As a result of LLCs' vital importance to the overall system performance, since the uniprocessor era, LLC management schemes have been studied extensively [7, 8, 9, 10, 44, 46, 75]. In particular, previous work has shown that the traditional LRU replacement policy cannot be optimal when workloads exhibit poor temporal locality. Several alternative policies, such as DIP [7] and PeLIFO [8], have been proposed to improve LLCs' performance by employing sophisticated block insertion, aging, promotion, and victimization strategies. Moreover, researchers have observed that, independent of the replacement policy, LLCs can exhibit very distinct resource demands at the set level because of the non-uniform characteristics of working sets that are mapped to individual LLC sets. As a result, several recent proposals, such as V-Way [9] and SBC [10], by aiming to provide better cooperation between LLC sets in retaining working sets, could outperform the aforementioned replacement policies under certain circumstances.

It is our contention that the two kinds of cache management approaches, mentioned above, are inherently different in that one is rooted in the temporal management and the other in the spatial management of LLC capacity resources. Specifically, we define *temporal resource management* as replacement policies (such as DIP and PeLIFO) that determine how the capacity of an LLC set is temporally shared among the competing blocks of a working set mapped to the LLC set, when the LLC set cannot retain all of them. Furthermore, we define *spatial resource management* as schemes (such as V-Way and SBC) that dynamically decides how the overall capacity of an LLC is spatially partitioned among LLC sets that are hosting different working sets. Our analysis indicates that neither the temporal nor the spatial LLC management schemes, working independently, can consistently and robustly deliver the best performance in all situations. To better understand the differences between the two dimensions of management, we characterize the non-uniform distribution of working sets' spatial and temporal capacity demands and its performance impact, and conclude that the effectiveness of a specific LLC management strategy is determined by how an LLC's set-level capacity provision and utilization meet its non-uniform set-level capacity needs.

Motivated by the observations on the different working principles between existing spatial and temporal LLC management schemes as well as the significant performance impact, we propose the adaptive *SpatioTEmporal Management* (STEM) scheme to regulate the two dimensions of capacity demands concurrently and dynamically. In the proposed scheme, a set-level monitor based on shadow-tag hash signatures and saturating counters is utilized to capture and measure both temporal and spatial capacity needs of individual working sets. Based on these measurements, the cache controller then judiciously identifies and pairs off sets with complementary capacity demands. Then, the controller enables the underutilized set in each pair to cooperatively cache the other's victim blocks, while interactively deciding the best temporal sharing patterns for both of them in the

event of intra-core cooperative caching. In addition, if a set does not have another set with a complementary capacity demand to pair with, the controller can still decide the best set-level replacement policy for it.

Our execution-driven simulation using 15 benchmarks shows that the proposed scheme performs robustly and consistently well under various workloads and HW configurations studied. Specifically, our STEM LLC design can improve the metrics of *misses per 1k instruction* (MPKI), *average memory access time* (AMAT) and *cycles per instruction* (CPI) by 21.4%, 13.5% and 6.3% over LRU respectively, which is better than the prior-art DIP, PeLIFO, V-Way and SBC schemes, at a manageable HW storage cost of only 3.1%.

4.2 Research Motivations

In this section, we first provide an in-depth analysis of conventional intra-core LLC management proposals, which qualitatively shows the two different classes of working principles for these schemes. Then, we apply real-world workloads to quantitatively compare the schemes, revealing their distinct performance comfort zones as a result of their different working principles.

4.2.1 The Problems of Conventional LLC Management

First, as observed in a few recent studies [9, 10, 34], an interesting LLC property known as the *set-level non-uniformity of capacity demands* can result in the underutilization of those LLC sets whose working sets require less than the associativity, while leaving other sets over-utilized because their capacity (i.e., associativity) is insufficient for their working sets. Therefore, prior-art LLC spatial management schemes such as V-way and SBC attempt to perform dynamic capacity allocation to different LLC sets according to their spatial

metrics. The metric adopted by the V-way cache is implicitly the per-set "access count", while SBC's metric is the "saturation level" defined as the difference between miss and hit counts at the set level.

Second, when a working set cannot be entirely retained in a cache set, its member blocks will compete for the set's cache lines, giving rise to policies that decide which block needs be evicted from a set in the event of replacement. Existing HW-replacement policies all use certain criteria to adjust the lifetime values of cached and incoming blocks so as to approximate the ideal "Belady's optimal algorithm" [41]. Such criteria can make a significant difference in the LLC performance. For instance, the simple and commonly used LRU replacement policy favors access (both hit and miss) recency when adjusting blocks' lifetime in cache. Therefore, it performs quite well when a working set exhibits excellent temporal locality but can thrash an LLC set when the locality is poor. The more advanced DIP replacement policy always advocates hit recency but duels between either favoring or penalizing miss recency (namely, assigning the recently-missed/incoming block with either the longest or shortest lifetime). As a result, DIP is more flexible and adaptive than LRU in making replacement decisions.

4.2.2 Unconventional Thinking of the Problems

From the analysis above, we argue that the two different types of approaches actually have fundamentally distinct working principles, of which one is to spatially manage LLC capacity resources among different LLC sets (such as V-Way and SBC) and the other is to temporally optimize the sharing pattern of an LLC set's capacity among the member blocks of its working set. More specifically, on the one hand, if an over-utilized LLC set can get sufficient cooperative capacity from another underutilized set, no replacement needs to take place because both of their working sets are already retained. In this

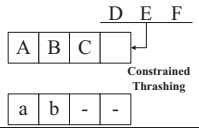
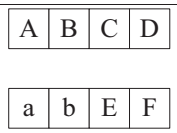
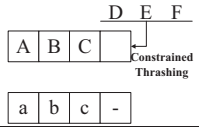
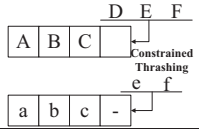
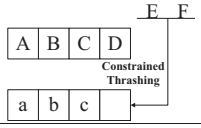
<table>
<tr><th>Ex. #</th><th>Synthetic Workloads</th><th>LRU</th><th>DIP</th><th>SBC</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">A→a→B→b→C →a→D→b→…</td><td>The entire working set 0 is thrashing, while working set 1 is well retained in LLC set 1.</td><td> 
</td><td> 
</td></tr>
<tr><td>Miss Rate = 1/2</td><td>Miss Rate = 1/4</td><td>Miss Rate = 0</td></tr>
<tr><td rowspan="2">2</td><td rowspan="2">A→a→B→b→C →c→D→a→E→ b→F→c→A→…</td><td>Working set 0 is thrashing, while working set 1 is well retained in LLC set 1.</td><td> 
</td><td>It is a dynamic process, and we write a simple trace simulator by following the procedure in the SBC proposal to get the miss rate.</td></tr>
<tr><td>Miss Rate = 1/2</td><td>Miss Rate = 1/4</td><td>Miss Rate = 1/3</td></tr>
<tr><td rowspan="2">3</td><td rowspan="2">A→a→B→b→C →c→D→d→E→e →F→a→A→b→…</td><td>Working set 0 is thrashing, and so is working set 1.</td><td> 
</td><td>It behaves exactly the same as LRU, due to the absence of underutilized LLC sets.</td></tr>
<tr><td>Miss Rate = 1</td><td>Miss Rate = 1/4+1/5</td><td>Miss Rate = 1</td></tr>
<tr><td colspan="2" rowspan="2">An Extensional Example for Ex. #2</td><td> 
</td><td colspan="2" rowspan="2">Still for Ex. #2, if SBC was able to combine its spatial management capability with a more advanced temporal scheme, e.g., retaining “D” in set 0’s local capacity and let E and F compete for the cooperative capacity in set 1, the overall miss rate would be reduced from 1/3 to no greater than 1/6.</td></tr>
<tr><td>Miss Rate ≤ 1/6</td></tr>
</table>

Figure 4.1: Conceptual Illustration with Synthetic Workloads

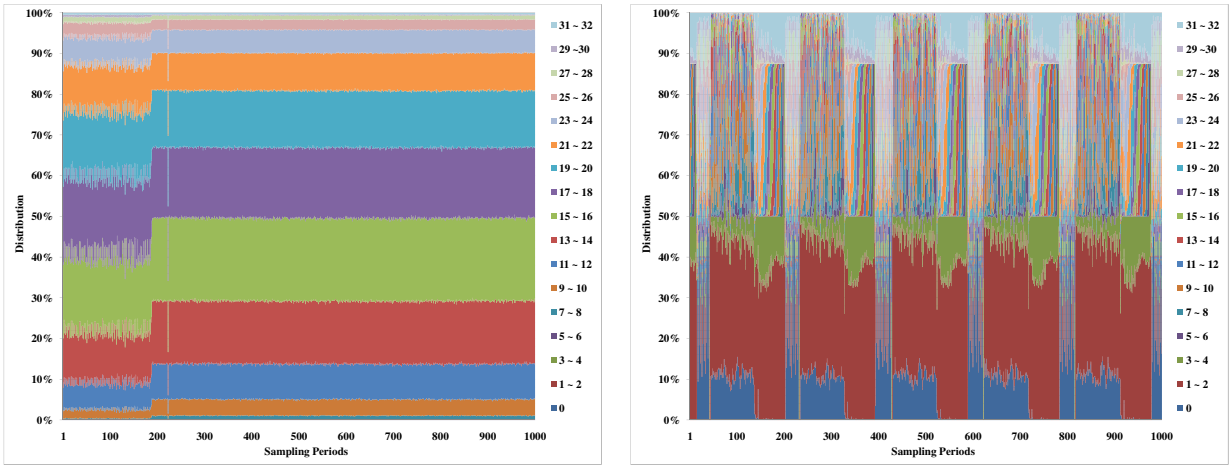
situation, the spatial management schemes will apparently be more effective than the temporal approaches that manage the two sets separately. On the other hand, if an LLC set does not have enough local space for its working set or cannot find external capacity for cooperation, adopting an advanced replacement policy such as DIP will be more sensible. But one of the challenges here is that existing adaptive temporal approaches such as DIP and PeLIFO all depend on application/LLC-level sampling, monitoring and decision-making, rendering them unable to work on an individual set basis. Yet, working at the set level, we believe, is essential in addressing the issue of set-level uniformity of capacity demands. More challenging is the fact that, if a set can only find some but insufficient cooperative capacity for the additional requests of its working set, both spatial and temporal management should simultaneously and interactively take effect on the

set and its cooperative set to make the best spatial and temporal use of their aggregate capacity.

For an intuitive illustration, we assume a simple 4-way associative LLC with just two sets. The LLC receives a sequence of repetitive requests from the upper level memory hierarchy components. After mapping the reference sequence to individual LLC sets, we can obtain two cyclic working sets, as shown in Figure 4.1. For the resulting performance, we measure the LLC's miss rate after its initialization. We consider SBC and DIP as representatives, respectively, as examples of spatial and temporal LLC management schemes. Furthermore, we assume that DIP has the knowledge of working sets' patterns without the need for dedicated sampling and dueling monitors [7].

In Example #1 of Figure 4.1, the cyclic working Set 0 is "A→B→· · ·→F→A→B→· · ·", while working Set 1 is "a→b→a→b→· · ·". We find that SBC performs better than DIP because SBC enables the over-utilized LLC Set 0 to place its blocks in Set 1, and this perfect match does not even trigger replacement in the long run. In Example #2, the only difference from Example #1 is that working Set 1 has an additional element "c". Although in SBC Set 0 is able to utilize the cooperative (albeit, insufficient) space of Set 1, their underlying LRU replacement policies cannot help the two sets produce the best performance. In Example #3, working Set 1 has two more elements "d" and "e" than that in Example #2. Therefore, both LLC sets are over-utilized, leaving SBC no choice for inter-set cooperation but to thrash both LLC sets. DIP can keep part of the working sets in both LLC sets, though Set 0 and Set 1 still contribute 1/4 and 1/5 overall misses respectively.

The intuitive illustration above enables us to better understand the different properties and comfort zones between temporal and spatial LLC management schemes. It also reveals that if a spatial management strategy like SBC could incorporate a more advanced temporal management mechanism, a better performance over both spatial and temporal



(a) *omnetpp* (b) *ammp*

Figure 4.2: Distribution of the Set-Level Capacity Demands for *omnetpp* and *ammp* During 1000 Sampling Periods

schemes would be achievable.

4.2.3 Quantitative Experiments

In this section, we use real-world applications to back our analysis in Section 4.2.2.

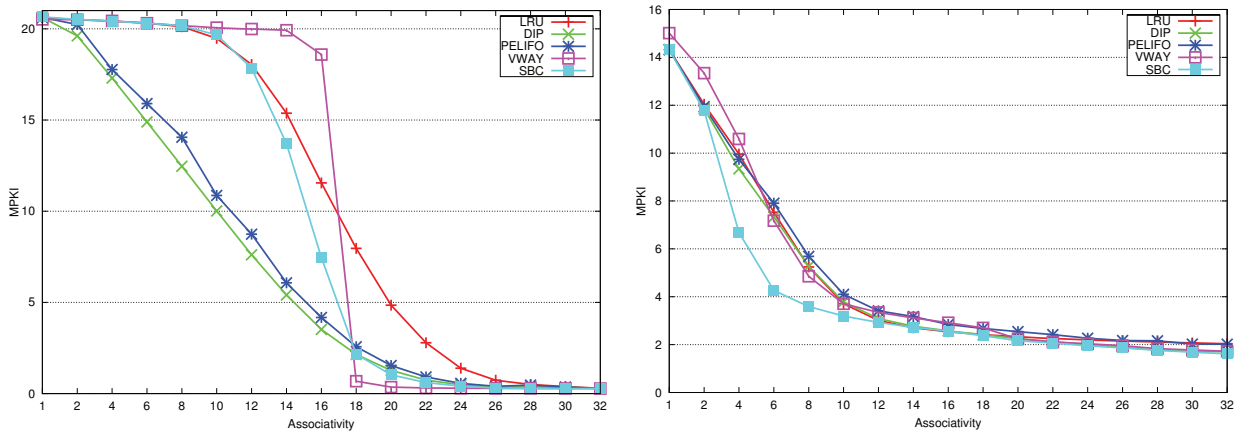
4.2.3.1 Non-Uniform Set-Level Capacity Demands

Although the non-uniformity of set-level accesses and saturation levels (defined as the difference between set-level hit and miss counts) has been noted, respectively, in V-Way [9] and SBC [10], we argue that neither the "access count" [9, 46] nor the "saturation level" [10] is an accurate or direct metric of set-level capacity demands. For example, if a set is experiencing a number of accesses, and further if these accesses only touch a small working set, it is highly likely that all accesses eventually turn out to be hits and the working set can be retained in the set without the need for extra capacity. Thus, a high access count is not always indicative of extra capacity demands. Moreover, an LLC set with misses dominating may not benefit from receiving extra capacity at all if its working

set is of streaming features, while an underutilized set with 80% accesses, e.g., as hits, may be able to further resolve its remaining 20% missed accesses by receiving a small amount of extra capacity.

In the previous chapter, we develop a more accurate and direct model that defines the set-level capacity demand as the minimum number of blocks required to resolve all conflict misses of the set during a time interval. With this definition, we experiment on two representative benchmarks *omnetpp* and *ammp* to characterize the features of their set-level capacity demands, as illustrated in Figure 4.2(a) and Figure 4.2(b) respectively. In Figure 4.2, each color represents 2 cache ways in the associativity range, according to the legend shown on the right. E.g., for *omnetpp*, the "light green" band (corresponding to legend "15-16") indicates that about 20% of the sets require 15-16 cache lines per set to meet their capacity needs; for *ammp*, the "blue" band (corresponding to legend "0") reveals that the corresponding sets are streaming-like and thus require little LLC capacity. The detailed description of the experimental setup appears in Section 4.4. Here, we only list the most important four parameters: 2048 LLC (L2) sets; 64-byte cache lines; 50000 accesses per time interval; and a total of 1000 time intervals during the workload characterization. With the settings, we first identify that the entire application/LLC-level capacity demands are no greater than 32 ways in both cases, which means that an associativity of 32 can help the LLC resolve all conflict misses for the workloads. Then, for an LLC set, we obtain the minimum number of ways/blocks required by it to resolve as many conflict misses as with an associaitivity of 32, and then define the value as the set's current capacity demand. As illustrated in Figure 4.2, the non-uniformity of set-level capacity demands is very evident for both benchmarks: for *omnetpp*, almost 50% sets require no more than 16 cache lines per set, while for ammp, about 50% sets require no more than 4 cache line per set.

Next, we show the impact of the non-uniformly distributed set-level capacity demands



(a) *omnetpp* (b) *ammp*

Figure 4.3: MPKIs of *omnetpp* and *ammp* for Different Associativity Configurations on LLC management schemes with real-world applications.

4.2.3.2 Demonstration with Real-World Workloads

We still use *ammp* and *omnetpp* as the representative workloads to evaluate LRU, DIP, PeLIFO, V-WAY and SBC in terms of their misses per 1k instructions (MPKI) under various associativity configurations, as shown in Figure 4.3.

In Figure 4.3(a), from associativity 2 to 16, both temporal schemes DIP and PeLIFO outperform the two spatial schemes V-Way and SBC, as well as the baseline LRU, for *omnetpp*. From associativity 12 on, the best spatial scheme SBC begins to outperform LRU. From associativity 18 to 24, both spatial schemes perform the best among all schemes. Beyond associativity 24, there is little difference among the five schemes. SBC's identical performance to LRU when the associativity is smaller than 12 is consistent with the conclusion drawn from Example #3 in Figure 4.1 because few sets that are less saturated [10] can be found for an associativity lower than 12. From associativity 12 to 16, SBC's performance is better than LRU but still worse than DIP/PeLIFO. This is because there are some but insufficient less-saturated sets for spatial cooperation in this range, SBC is not able to best utilize the limited cooperative capacity, which is consistent with the

conclusion drawn from Example #2 in Figure 4.1. When the associativity is greater than 18, SBC and V-Way perform the best, because there are an appropriate number of less-saturated sets for cooperation, which is consistent with the conclusion drawn from Example #1 in Figure 4.1. Beyond associativity 24, the performance curves of all schemes begin to converge as expected.

In Figure 4.3(b), from associativity 2 to 10, the best spatial scheme SBC outperforms any temporal schemes for *ammp*. It is because about 50% LLC sets require no more than 4 cache lines per set, as demonstrated in Figure 4.2(b). Therefore, in the integer range [4, 10], the spatial scheme is the most effective, which is consistent with the conclusion drawn from Example #1 in Figure 4.1. When the capacity is beyond 10, the effectiveness of both temporal and spatial LLC management diminishes, because the local/native capacity of each LLC set is sufficient for them to retain their working sets. This is why no other schemes significantly improve over LRU for *ammp* in the associativity range [12, 32].

4.3 The STEM Architecture

It has been clearly illustrated in the motivational experiments that LLCs can exhibit non-uniform capacity demands in both spatial and temporal dimensions. The spatial capacity demand refers to if a working set can fit most of its blocks into the current space of its LLC set, while the temporal capacity demand implies whether the working set is making the best use of the cache space it possesses. The two types of capacity demands have different impacts on the effectiveness of LLC management schemes. This is the principal reason why none of the existing cache management schemes working in either dimension alone can perform robustly and constantly well under all circumstances. Therefore, an adaptive LLC management is necessary to harness both dimensions of capacity demands concurrently and dynamically to optimize LLCs' performance.

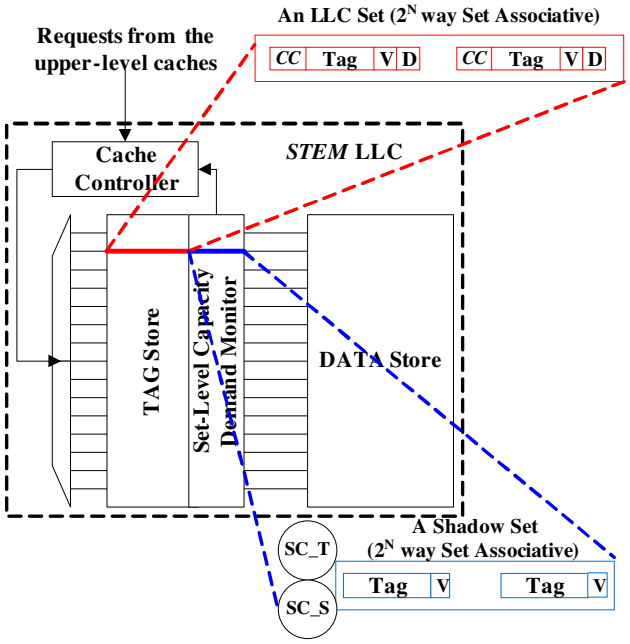


Figure 4.4: STEM Architecture

4.3.1 The STEM LLC Architecture

To accomplish this objective, we propose a novel LLC design named the *SpatioTEmporally Managed Last Level Caches* (STEM LLCs). STEM aims to achieve three specific design goals: (1) it identifies the spatial capacity demands of individual sets and couples two sets with complementary needs to perform inter-set cooperative caching; (2) in the event of inter-set space cooperation, it determines the best temporal capacity sharing patterns for both of the coupled sets so as to optimally utilize both local and cooperative capacity; and (3) for those uncoupled sets, it is still able to decide the best set-level replacement policies for them individually.

Figure 4.4 provides an architectural view of the STEM LLC. The STEM cache controller accepts access requests from upper-level caches. Then, the controller looks up the referenced block in the tag store, which is decoupled from the data store, to see if it is present in the LLC. There can be two scenarios if the requested block is on-chip: the block is either in its local set with the same index as indicated in the physical address of the

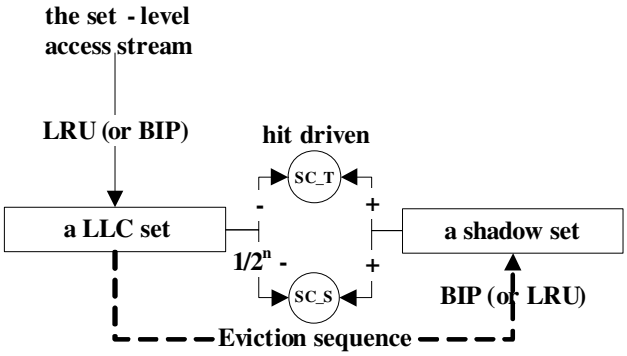


Figure 4.5: Set-Level Capacity Demand Monitor (SCDM)

block, or in a different set where the block is cooperatively cached. Therefore, each tag store entry needs an additional bit called the *CC* bit to indicate whether the block is local ($CC = 0$) or cooperatively cached ($CC = 1$), as shown in Figure 4.4. Then, the requested block is forwarded to the upper-level cache if it is found on-chip or otherwise fetched from DRAM. Meanwhile, the *set-level capacity demand monitor* (SCDM) is operated to capture and measure the dynamic information of individual sets' spatial and temporal capacity demands and feed it back to the cache controller. Based on the feedback information, the controller couples two sets with complementary spatial capacity needs and decides their best temporal capacity sharing behaviors for inter-set cooperative caching. For an uncoupled set, STEM will also adapt the set's replacement policy to either LRU or BIP (*bimodal insertion policy*) [7]. The design details and working principles of each critical component in STEM will be elaborated in the following subsections.

4.3.2 Set-Level Capacity Demand Monitors

The *set-level capacity demand monitor* (SCDM) is devised to capture and monitor both spatial and temporal capacity demands of individual sets. Associated with each LLC set, as illustrated in Figure 4.5, there is a shadow set [89] and two k-bit saturating counters "SC_S" and "SC_T" in the SCDM. Each shadow set has the same associativity as the

corresponding LLC set and stores an m-bit hash value taken from the tag field of a victim block that is evicted off the LLC set, where m is much shorter than the length of a tag field. Here, we still call this hashed tag value as a shadow tag. Thus, an LLC set appears to have "double" capacity with the additional "virtual" space provided by its shadow set. Then, the two saturating counters measure the spatial and the temporal capacity demands of each LLC set by using the information embodied in the shadow tags.

4.3.3 Operations on Shadow Sets

There are three essential operations on a shadow set: (1) if a local block is evicted from its original LLC set, the hash value of its tag field will be calculated by STEM's hashing module and inserted into the corresponding shadow set; (2) the shadow set maintains its own independent ranking for all of its valid entries and uses it for replacement; (3) if an access on a local block is missed in an LLC set, the corresponding shadow set will be looked up to check if the tag of the requested block is present in a valid shadow set entry. Additionally, it is required that the shadow set entries be strictly exclusive with the local blocks in the corresponding LLC set in terms of the complete/hash values of the tag fields. Therefore, if a previously-evicted block with its tag present in the shadow set is revisited by the owner set, two operations must be performed: (1) the shadow entry that has the hashed tag needs to be invalidated after the corresponding block is inserted into the LLC set; (2) a hit on the shadow set is signaled to operate its saturating counters.

The information of an LLC set's spatial capacity demands can be naturally captured by the shadow set because it contains the information of the set's victim blocks, as shown in Figure 4.5. If there are a considerable number of hits on the shadow set, it implies that the blocks previously evicted from the LLC set will soon be revisited and extending the LLC set's space will be beneficial. In the STEM LLC design, the shadow

set adopts a replacement policy opposite to that of the corresponding LLC set to capture the information of the LLC set's temporal capacity demands. Specifically, as illustrated in Figure 4.5, if the LLC set is currently adopting the LRU replacement policy to favor temporal locality, the shadow set will use the BIP policy [7] to keep the shadow tags of LLC victim blocks. The rationale behind the specific design choice is that if a large working set cannot be well retained in the LLC set due to its poor temporal locality, e.g., by way of thrashing, the same information of poor temporal locality will also be reflected by its eviction stream, which in turn can be captured by adopting BIP in the shadow set that contains the information of the set's victim blocks. In contrast, if an LLC set is adopting BIP for insertion but actually its large working set shows good temporal locality (e.g., if the average reuse distance is shorter than the set associativity), the temporal locality information can be captured in the eviction stream as long as the shadow set takes the LRU replacement policy.

4.3.4 Operations on Saturating Counters

The two k-bit saturating counters "SC_S" and "SC_T" are used to measure a set's temporal and spatial capacity demands respectively, by comparing the hit count of a shadow set against that of the LLC set. Whenever there is a hit on the shadow set, both saturating counters will be incremented by one. The temporal saturating counter is always decremented by one upon a hit on the LLC set, while the spatial saturating counter is decremented by one for every 2^n hits on the LLC set, as demonstrated in Figure 4.5. We implement counting 2^n hits on the LLC set in a probabilistic way that the spatial saturating counter is decremented by one only when an n-bit value produced by a *random number generator* is zero. The random number generator can be simply incorporated in the LLC controller.

We look at the values of the two k-bit saturating counters of an LLC set to measure its spatial and temporal capacity demands. Specifically, on the one hand, if a spatial saturating counter reaches a saturated value, it implies that providing the LLC set with double capacity can result in at least $\frac{1}{2^n}$ increase in the hit rate. The LLC set should be identified as a taker set that can benefit from inter-set capacity cooperation; otherwise, if the MSB (*most significant bit*) of the spatial saturating counter is 0, it suggests that the LLC set has a very high hit frequency with its local capacity and it could be regarded as a giver set that can potentially contribute part of its capacity in inter-set space sharing. The spatial saturating counter is reset only upon system initialization. On the other hand, , if a temporal saturating counter is saturated, indicating that the shadow set's replacement policy is estimated to outperform the LLC set's current policy, it will send a request to the cache controller to swap the replacement policies for the LLC and the shadow sets as well as resetting the temporal saturating counter.

4.3.5 Coupling Sets with Complementary Capacity Demands

As described above, a saturated spatial saturating counter indicates that extending the capacity of the corresponding set is beneficial; hence the set is regarded as a taker set that can significantly reduce its conflict misses if its capacity is extended. On the other hand, a 0-valued MSB denotes a giver set that may need fewer blocks than it currently possesses. Thus, the STEM LLC should couple a taker set and a giver set so that the taker set can utilize part of the giver set's capacity to reduce conflict misses.

The coupling process needs the assistance of a hardware heap (similar to the *destination set selector* in [10]) that keeps track of a small number of uncoupled giver sets which are less saturated than others, as well as an *association table* [10] that maintains the association information of paired sets. If a set is not paired with any other set, the value of its

association table entry is the set's own index. Both the HW heap and association table are embedded in the STEM LLC controller. When a set is identified as a giver set by its monitor, it tries to post its index and saturating level information to the heap. The heap checks if there are any available/invalid entries to keep the set's information. If there are no such invalid entries and the set is less saturated than one of the sets already in the heap, replacement will take place in the heap to make room for this less-saturated one.

In addition, when an uncoupled taker set needs to evict a block, it first sends a coupling request to the HW heap. The heap returns the index information of the least saturated giver set for coupling, and the association table records the two sets' indices in each other's association table entry. If there are no available giver sets in the heap, the taker just evicts the victim block off chip.

4.3.6 Spilling and Receiving Control

Unlike SBC that allows a taker set to continuously evict blocks to its coupled set, our STEM LLC design imposes some restrictions on spilling and receiving for any pair of coupled sets. This is because a giver set can be overwhelmed if spilling from the taker set is excessive. However, whether or not a giver set is overwhelmed can be easily detected by checking the MSB of the spatial saturating counter of its corresponding shadow set. If a previously 0-valued MSB of a spatial saturating counter turns 1, it suggests that either the set might have been overwhelmed by another set's excessive spilling or it has changed its role from a giver set to a taker set. The set-level capacity monitor returns such information to the cache controller to form a feedback loop as depicted in Figure 4.4. With the feedback loop, only when a set has a 0-valued MSB in the corresponding spatial saturating counter can it receive victim blocks from its coupled taker set.

While the spilling process is straightforward and similar to the SBC scheme, the

receiving process is significantly different. In the SBC proposal [10], it is clearly stated that receiving (using MRU insertion, namely the LRU replacement policy) is not dependent on the giver set's saturating level as long as the two sets are coupled. Such a receiving mechanism in SBC can severely pollute the giver set's space, because the taker set can excessively spill victim blocks to the giver set without looking at the actual utility of doing so. In the STEM LLC design, we set such a receiving constraint that the giver set cannot receive a foreign block unless its saturating value indicates that the set is still unsaturated even with receiving. In other words, whether or not a cooperative set is still able to contribute its capacity to the taker set can be detected by its spatial saturating counter. Furthermore, how a foreign block is inserted into the cooperative set is decided by what the cooperative set's temporal saturating counter indicates.

4.3.7 Decoupling Two Sets

The disassociation between two coupled sets is triggered by the event that the (former) giver set has evicted all cooperatively-cached blocks, followed by the action of resetting the two sets' association table entries to their own original indices respectively [10]. In contrast to the SBC scheme that does not put any constraints on the spilling and receiving processes, the decoupling process of STEM will be much faster because the taker (or giver) set will not spill (or receive) blocks after a role change is detected for either of them, which can greatly accelerate the decoupling process.

4.4 Experiments & Evaluation

To evaluate our STEM LLC design, in this section, we present the experimental setup, the results analysis, the sensitivity study and the cost analysis.

Table 4.1: Major Configuration Parameters

Core	Alpha ISA, 5-Stage Pipeline 8-Wide Dispatch/Retirement 256/256 Int/Fp Registers 64/64-Entry Inst/Data TLBs 6-Int ALU, 2-Int Mul/Div, 4-Fp ALU, 2-Fp Mul/Div 64-Entry IFQ, 64-Entry LSQ, 192-Entry ROB
L1I/D	2-Way, 32KB, 64B/Line, 1/2-Cycle I/D Lat 8/16 I/D MSHRs, 8-Entry Write Buffer physically tagged and indexed (M5's built-in setting)
L2	16-Way, 2MB, 64B/Line, 6/8-Cycle Tag/Data Store Lat 64 MSHRs, 32-Entry Write Buffer physically tagged and indexed (M5's built-in setting)
Bus	16B/Cycle, 2:1 Speed Ratio, 1-Cycle Arbitration
Mem	300-Cycle Lat

4.4.1 Experimental Setups

We use the cycle-accurate M5 simulator [90] as our architectural simulator with the configuration listed in Table 4.1. The simulated processor is an Alpha21264-like [91] out-of-order core with a 5-stage pipeline. For the memory hierarchy, we model two levels of on-chip caches. The L1 instruction and data caches adopt the conventional set-associative configuration and LRU replacement policy, and we assume a coupled tag-data store organization. For the L2 cache, we model decoupled tag and data stores, and adopt the same latency parameters as those presented in [10]. Specifically, if an access to an uncoupled or coupled giver set turns out to be a miss, the latency of a tag-store access is assumed to be 6 cycles; if an access to a set is a hit, the total latency of one tag-store access and one data-store access is assumed to be 14 cycles. For SBC and STEM, if an access to a coupled taker set is a miss and the requested block is not found in its cooperative set either, the total latency of two consecutive tag-store accesses is 12 cycles; otherwise a second hit will cost 20 cycles in all because it involves two tag-store accesses

Table 4.2: Workload Classes

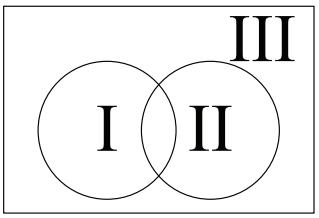


Table 4.3: MPKI Characteristics of Benchmarks

Class I	**M**PKI	**C**lass II	**M**PKI	**C**lass III	**M**PKI
*a*mmp	2.535	*a*rt	16.769	*g*obmk	2.236
*a*psi	5.453	*c*actusADM	3.459	*g*romacs	1.099
*a*star	2.622	*g*algel	1.426	*s*oplex	24.298
*o*mnetpp	11.553	*m*cf	59.993	*t*wolf	3.793
*x*alancbmk	14.789	*s*phinx3	10.969	*v*pr	3.306

as well as an additional data-store reference. We evaluate and compare LRU, DIP, PeLIFO, V-Way, SBC and our proposed STEM, among which both SBC and STEM may involve a second access to the cooperative set.

We select 15 benchmarks from the SPEC CPU 2000 & 2006 suites. In general, we assume that all applications can be categorized into three classes according to the features of their spatial and temporal capacity demands (at the LLC set level), as shown in Table 4.2. Class I includes the applications that exhibit set-level non-uniformity of capacity demands, whose performance is improvable by spatial schemes such as V-Way and SBC when the LLC capacity is in a certain range (e.g., *ammp*'s LLC performance can be improved over LRU by SBC in the associativity range [4,10], as shown in Figure 4.3(b)). Class II covers the programs that show poor temporal locality, so their performance can be promoted by an advanced temporal scheme like DIP or PeLIFO within a certain LLC capacity range (e.g., *art*'s LLC performance can be boosted by DIP when the LLC capacity is no greater than 1MB, as demonstrated in [7]). Class III consists of such applications that show uniform set-level capacity demands as well as good temporal locality, which can be well taken care of by the simple LRU scheme. Table 4.3 presents these 15 benchmarks in terms of their classification as well as their MPKI characteristics (under LRU).

The selected benchmarks are fast-forwarded and cache-warmed with 10 and 2 billion instructions respectively, followed by an execution of 3 billion instructions with the detailed architectural features listed in Table 4.1. In the evaluation, we use three

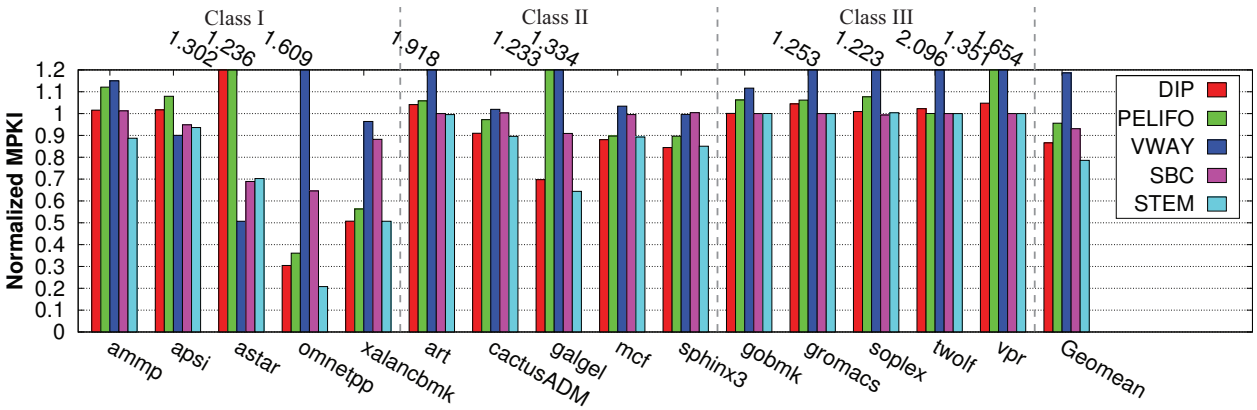


Figure 4.6: Normalized MPKI

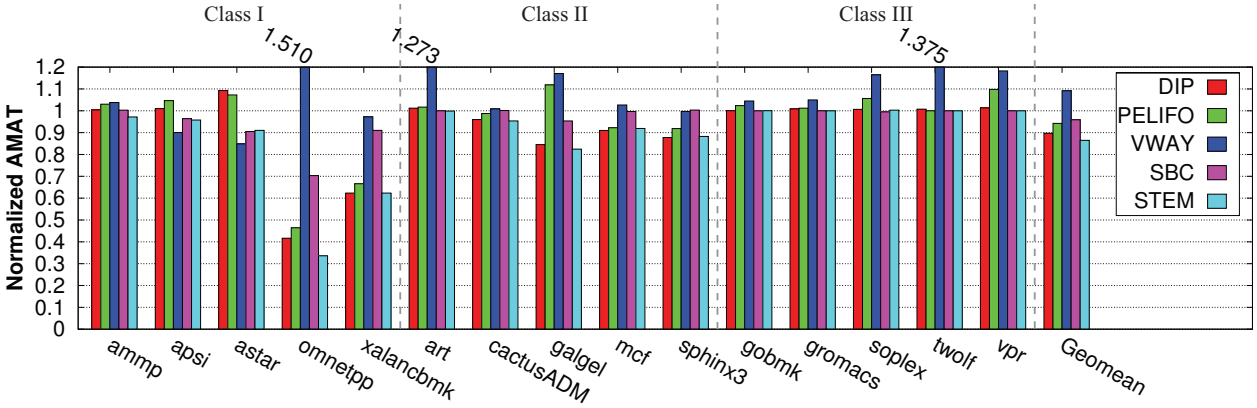


Figure 4.7: Normalized AMAT

performance metrics, namely, MPKI (*misses per 1K instructions*), AMAT (*average memory access time*) and CPI (*cycles per instruction*), to compare our STEM design against other prior-art schemes in various aspects. All results are normalized to those of LRU.

4.4.2 Performance Analysis

Figure 4.6 shows the performance comparison between STEM and the prior-art spatial and temporal LLC management schemes with respect to their MPKI results. For the benchmarks in Class I, as a result of the capability of spatial resource management, STEM is noticeably better than the existing temporal schemes DIP and PeLIFO. Specifically,

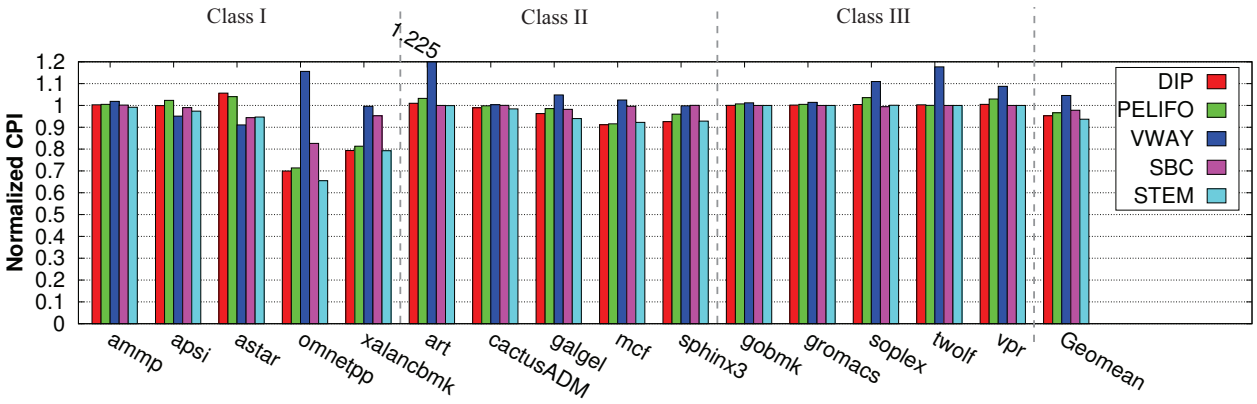


Figure 4.8: Normalized CPI

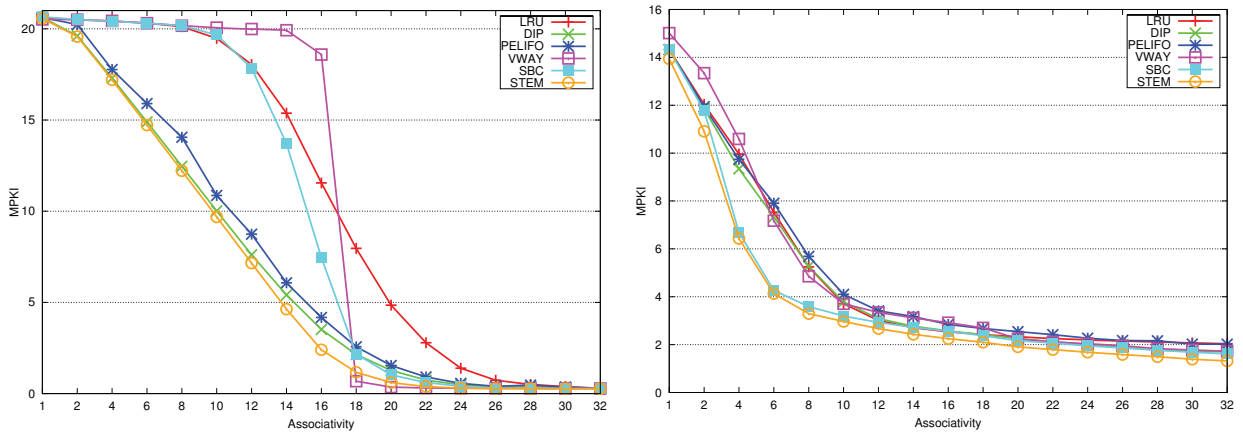
STEM outperforms the two temporal schemes by at least 12.9%, 8.1%, 53.4% and 9.7% for *ammp*, *apsi*, *astar* and *omnetpp* respectively. Interestingly, we find that temporal schemes can degrade the MPKI performance of *astar* significantly. That is because *astar* shows obvious set-level non-uniform capacity demands and, more importantly, good temporal locality for most LLC working sets. Since temporal schemes DIP and PeLIFO both dedicate several groups of sample sets to the policy comparison, e.g., BIP versus LRU in the DIP scheme, and the policy that incurs less (in DIP) or the least (in PeLIFO) misses will be imposed upon other non-sample sets. However, due to the set-level non-uniform features, *astar*'s LLC working sets are quite different from each other, and the winning policy of the sample sets is not necessarily suitable for the non-sample LLC sets most of which have good temporal locality. This is why DIP and PeLIFO make inappropriate application/LLC-level replacement decisions for *astar* (e.g., in DIP, BIP is selected as the winning policy and adopted for the non-sample LLC sets). Unlike DIP and PeLIFO, STEM is able to decide on better replacement policies for individual LLC sets based on their set-level temporal capacity demands for certain benchmarks like *astar*.

For the five schemes in Class II, we obtain the expected better performance of temporal LLC management schemes than that of the spatial ones, because the existing spatial

schemes are unable to handle the cases of poor temporal locality. Since STEM also has a temporal management module, it is capable of dueling between LRU and DIP under this circumstance, but at the LLC set-level rather than at the application/LLC level as in DIP and PeLIFO. STEM performs as well as DIP for the benchmarks of Class II. The reason why none of the schemes improves over LRU for *art* is because *art* is improvable by advanced temporal schemes only when its capacity is no greater than 1MB, as evaluated in [7], but the standard LLC capacity configured here is 2MB. With regard to the benchmarks in Class III, for which LRU is sufficient, we find that STEM performs as well as LRU and SBC both of which are among the best.

In Figure 4.6, we can also infer that the HW metric used by STEM to measure set-level capacity demands is better than those used by SBC and DIP. Among the 15 benchmarks, we see that V-Way underperforms LRU in 7 out of them, while STEM either outperforms or performs no worse than LRU. In addition, for the benchmarks in Class I, where spatial schemes have opportunities to significantly improve over LRU, STEM outperforms SBC just with the exception of astar for which it slightly underperforms by 0.3%. This comparison reveals that the HW metric in STEM, which utilizes the virtual capacity of shadow tags to directly measure the benefit of extending an LLC set's capacity, is more accurate than the (implicit) metric of "access count" of V-Way as well as the "saturation level" of SBC in estimating the capacity demands of individual LLC sets.

Because both SBC and STEM can involve a second access to a cooperative LLC set, MPKI is not a direct metric for comparing the throughput of different LLC management schemes, but it sheds light on the implication of MPKI reduction on throughput. Figure 4.7 and Figure 4.8 show the AMAT and CPI results of the schemes with the timing parameters of Section 4.4.1 and Table 4.1 incorporated into the simulation. We find that the comparison results of AMAT and CPI are consistent with that of MPKI in Figure 4.6. Specifically, the STEM LLC design can improve the AMAT performance of LRU by 13.5%



(a) *omnetpp* (b) *ammp*

Figure 4.9: Sensitivity Study

and the CPI performance by 6.3%, while DIP, PeLIFO, V-Way and SBC improve the two throughput metrics by (10.3%, 4.7%), (5.8%, 3.4%), (-9.2%, -4.6%) and (4.1%, 2.2%) respectively.

All in all, benchmarks in Class I and Class II together highlight the adaptive capabilities of STEM. More specifically, STEM has generally noticeable performance advantages over the existing temporal schemes for benchmark Class I, and significantly outperforms the prior-art spatial schemes for benchmark Class II. If a benchmark belongs to both Class I and Class II, STEM can outperform both temporal and spatial schemes simultaneously, which is consistent with the Extensional Example shown in Figure 4.1. In addition, STEM is capable of deciding different replacement polices for individual LLC sets and overcoming the pathological cases that expose the weaknesses of advanced application/LLC-level temporal schemes; and STEM's set-level spatial capacity demand monitors that take advantage of the virtual capacity of shadow tags are shown to be more accurate than those of V-Way and SBC.

4.4.3 Sensitivity Study

We use benchmarks *omnetpp* and *ammp* that are illustrated in Section 4.2 as examples for our sensitivity study. From Figure 4.9(a), we find that in the small associativity range of [1, 6] STEM performs as well as DIP that is the best out of all existing schemes under this condition, with noticeable performance improvement over the spatial schemes such as V-Way and SBC. In the moderate associativity range of [6, 16], STEM is able to outperform all existing LLC management schemes by leveraging the strengths of spatiotemporal capacity management. In the high associativity range of [18, 24], STEM is still be better than others except that it is slightly worse than V-WAY.

As illustrated in Figure 4.9, for *ammp* and throughout the entire experimented associativity range of [1, 32], STEM outperforms or performs no worse than the existing LLC management schemes, but with significant advantages over DIP, PeLIFO and V-Way in the associativity range of [2, 10].

From the two cases, we find that STEM is able to dynamically adapt its management strategy to both spatial and temporal capacity demands of workloads, which indicates that STEM may bridge the performance gap between existing spatial and temporal LLC management schemes.

4.4.4 Overhead Analysis

The *set-level capacity demand monitor* (SCDM) and the *association table* account for the vast majority of STEMs hardware overhead. Table 4.4 lists the length of each storage field in the STEM L2 cache. The overall storage overhead of both monitor store and association table of the LLC controller is 3.1% compared to LRU by estimation.

Table 4.4: Hardware Overhead Analysis with the Configurations in Table 4.1

address length	44-bit effective physical address in a Alpha21264 processor simulated by M5
# (LLC sets)	2048
association table	2048 entries with 11 bits each
set associativity	16
cache line size	64 bytes
tag field length	27 bits
m (the length of a shadow tag entry)	10 bits with the hash function defined in [92]
CC, V, D bits	1 bit each
replacement rank field	4 bits
k (the length of a saturating counter)	4 bits
n (2^n is the ratio that multiplies the number of hits on an LLC set in spatial measurement)	3 bits

4.5 Summary

This chapter proposes a novel LLC design, which is called the STEM (*SpatioTEmporally Managed*) LLC, to dynamically identify both spatial and temporal dimensions of capacity demands at the set level, couple two sets with complementary spatial resource needs for inter-set capacity sharing and decide on the best replacement policies for coupled and uncoupled LLC sets. Our executing-driven simulation shows that the STEM LLC design can improve the performance metrics of MPKI (*misses per 1k instruction*), AMAT (*average memory access time*) and CPI (*cycles per instruction*) over LRU by 21.4%, 13.5% and 6.3% respectively, better than the performance benefits obtained by the prior-art DIP, PeLIFO, V-Way and SBC LLC management schemes, at a manageable HW storage cost of only 3.1%.

Chapter 5

Co-optimizing Locality and Utility in Thread-Aware Capacity Management for Shared Last Level Caches

5.1 Problem Definition

The shared last level cache (SLLC) organization is commonly adopted in chip multiprocessor (CMP) products to simplify both cache capacity sharing and coherence support for processing cores. Most commodity CMPs nowadays, whether multi-core (e.g., AMD's Phenom^{TM} II X6 and Intel's Core i7) or many-core (e.g., Tilera's 100-core processors [2]), have large SLLCs to help retain a substantial amount of data on-chip. But a large aggregate capacity alone does not guarantee optimal performance without an effective SLLC management strategy. This is especially true when the cores are running a heterogeneous mix of applications/threads, as is increasingly common with the widespread deployment of CMPs in complex application environments such as virtual machines and cloud computing [93, 94].

Because of its vital importance to the system performance, SLLC capacity management has been extensively studied. We categorize these studies into two groups: those proposing alternatives to the LRU replacement policy [14, 52, 15, 16] and those proposing cache partitioning schemes [12, 13]. Since the commonly-used LRU replacement policy aims to favor cache access recency (or *temporal locality* [1]) only, it can result in thrashing when the working set size of a workload is larger than the cache capacity and the cache access pattern is locality-unfriendly (e.g., a large cyclic working set) [7]. Alternative replacement policies, such as TADIP [14] and NUCACHE [15], are proposed to overcome the thrashing problem by judiciously assigning and adjusting lifetimes for cached blocks. The *utility* of a thread represents its ability to reduce misses with a given amount of SLLC capacity [12]). Although threads may vary greatly in their utility, an LRU-managed SLLC is oblivious of such differences when threads are co-scheduled and their cache accesses are mixed. In response to this shortcoming, several recent studies, such as UCP [12] and PIPP [13], propose to partition the SLLC space among competing threads based on the utility information captured by per-thread LRU-stack profilers, notably improving the performance over the baseline LRU replacement policy. More details about these proposals can be found in Chapter 2.

In our view, the prior-art alternative replacement policies and cache partitioning schemes have fundamentally different working principles. Specifically, the alternative replacement policies (of TADIP and NUCACHE) determine how the competing cores should temporally share the SLLC capacity to accommodate workloads' locality, while the cache partitioning schemes (of UCP and PIPP) decide on how the SLLC resources should be spatially divided among the cores on a utility basis. Our analysis and evaluation show that alternative replacement policies and cache partitioning schemes represent essentially two independent dimensions of solving the overall shared cache management

[1] In this and next chapters, locality is specifically referred to as temporal locality.

problem, and that optimizing in just one dimension misses the benefits available from co-optimization in both. Specifically, the alternative replacement policies, lacking a utility monitor, cannot coordinate the best capacity provisioning for all of the co-scheduled threads, while the cache partitioning schemes, fail to realize opportunities for higher utility achievable by individual threads with an replacement policy other than LRU. In order to gain a deeper understanding of this issue, we characterize the locality and the utility features for a spectrum of workloads and construct different workload combinations to evaluate the existing solutions. Our observations confirm distinct performance comfort zones for the two categories of existing approaches, neither performing consistently and robustly well under all workloads.

Motivated by these observations, we propose a novel design, called CLU, to interactively co-optimize the locality and utility of workloads in thread-aware SLLC capacity management. The key design challenge is how to estimate the utility information with a replacement policy other than LRU. Based on the observation that the hit curve of the thrashing-prevention policy BIP (*bimodal insertion policy* [7]) is concave and can be approximated by using logarithmic samples, CLU employs two lightweight runtime monitors for each thread in a CMP workload: a classic LRU stack profiler and a novel *logarithmic-distance-curve-fitting* BIP utility profiler to capture the interleaved locality and utility of the thread. Leveraging the information about all co-scheduled threads, CLU spatially partitions the SLLC cache ways among the threads and temporally makes use of the allocated capacity for individual threads in an interactive way, so that the highest utility provided by the best replacement policies can be exploited. Our evaluation shows that CLU improves the throughput by 24.3%, 45.3% and 43.3% for our simulated dual-core, quad-core and eight-core systems (with 0.26%, 0.27% and 0.53% storage overhead) respectively, outperforming the existing alternative replacement policies and cache partitioning schemes under a wide-range of CMP workloads.

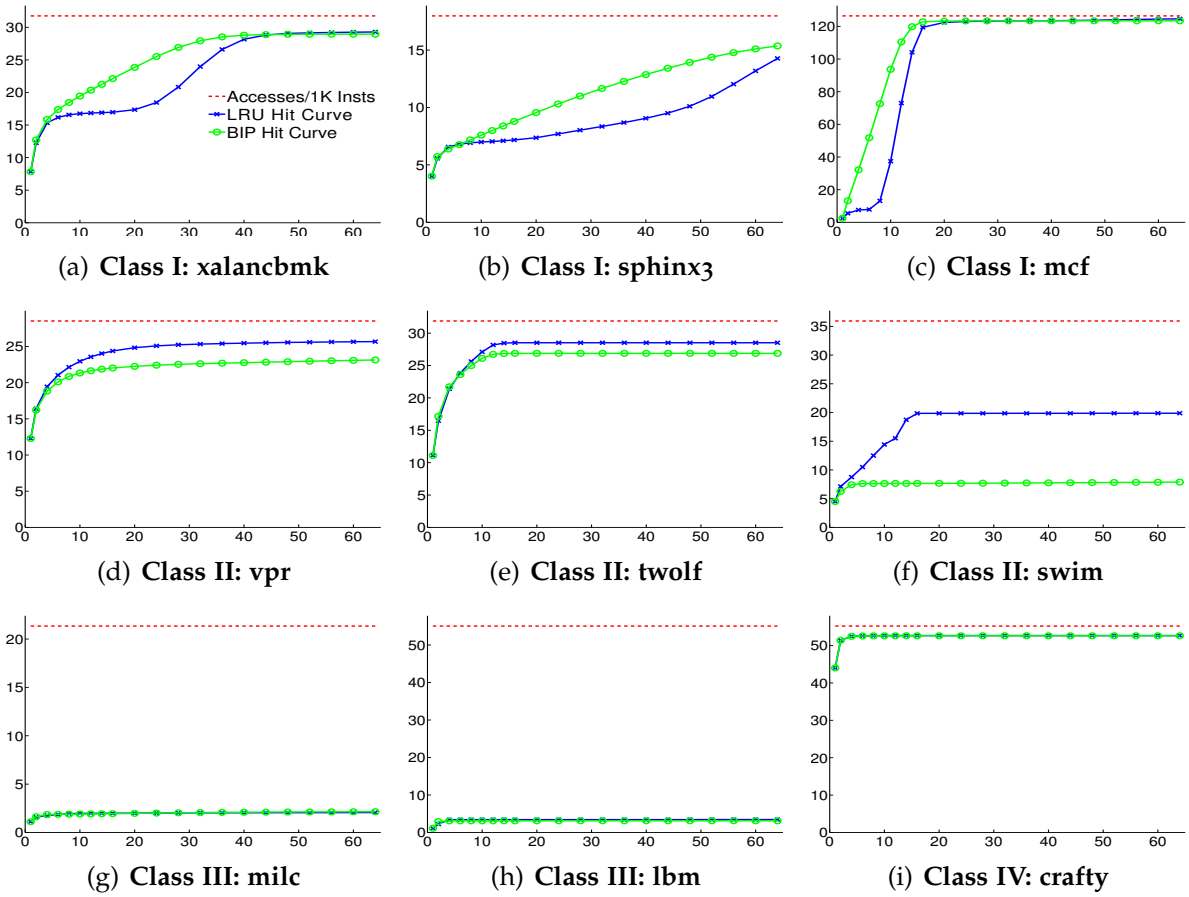
5.2 Research Motivations

Although the entire SLLC capacity can be accessed by all cores, allowing free accesses with the LRU replacement policy does not necessarily lead to an effective utilization of the SLLC resources. This is because regulating the contention for capacity among co-scheduled threads is beyond the capability of LRU. Therefore, various alternative replacement policies and cache partitioning schemes have been proposed for better utilization of the SLLC capacity. Here, we briefly describe their working principles and discuss their strengths and weaknesses revealed by our experiments, which motivates us to view the SLLC capacity management from a unique perspective.

5.2.1 Shared LLC Capacity Management

On the one hand, it has been noted that the LRU replacement policy performs quite well when a thread's block-reuse distance is no longer than its cache set associativity [7], or in other words, when the thread has excellent locality. However, LRU can cause a thread with poor locality to thrash its cache space [7] or severely interfere with other co-scheduled threads in capacity use [14]. In general, the thrashing problem can be solved by adaptively assigning the lifetime of a block according to the locality of the thread that brings it into the cache, and the solutions are termed as *locality-oriented alternative replacement policies*. The existing proposals like TADIP and NUACHE fall into this category.

On the other hand, the miss-driven nature of the LRU-based SLLC capacity management implicitly partitions the SLLC capacity among co-scheduled threads in a way that a thread incurring more misses will be allocated a greater amount of SLLC capacity by default. But the miss-driven capacity allocation is oblivious of a thread's efficiency of utilizing the SLLC resources for performance delivery, exemplified by the pathological



(a) **Class I: xalancbmk** (b) **Class I: sphinx3** (c) **Class I: mcf**

(d) **Class II: vpr** (e) **Class II: twolf** (f) **Class II: swim**

(g) **Class III: milc** (h) **Class III: lbm** (i) **Class IV: crafty**

Figure 5.1: HPKIs (Hits Per 1K Instructions) of LRU and BIP as a Function of the LLC Capacity for the SPEC Benchmarks

case where a streaming thread occupies a large amount of capacity with little performance contribution. Thus, UCP [12] and PIPP [13] are proposed to partition the SLLC space among co-scheduled threads according to their utility, which is defined as the ratio of the number of SLLC hits to the SLLC capacity that is required to maintain the hit count for a thread under LRU. We name these schemes as *utility-oriented capacity partitioning schemes*.

5.2.2 Our Perspective and Supporting Experimental Data

In our view, the aforementioned alternative replacement policies and cache partitioning schemes have fundamentally different working principles: the replacement alternatives

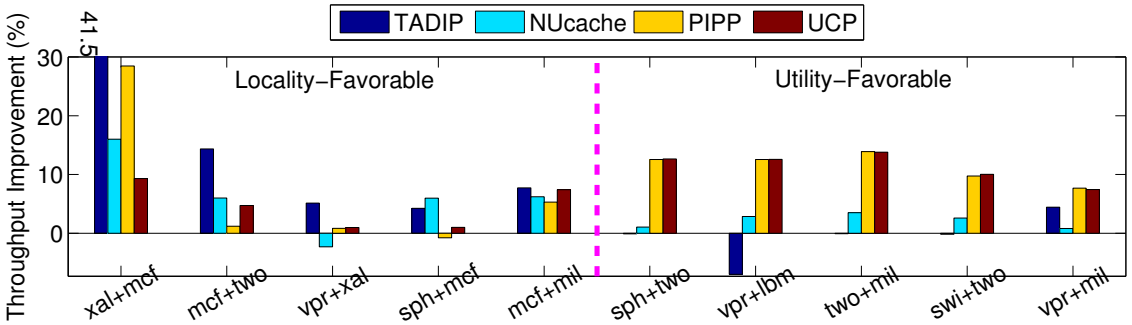


Figure 5.2: Performance Disparity between Locality-Oriented and Utility-Oriented Approaches

aim to temporally optimize the sharing of SLLC capacity for co-scheduled threads mainly by adapting to their locality features, while the cache partitioning schemes are targeted at spatially provisioning SLLC resources among competing threads according to their LRU-based utility characteristics. Unfortunately, the replacement policies are unable to coordinate the best capacity provisioning for all co-scheduled threads due to the lack of utility monitors, while the existing cache partitioning schemes cannot estimate or exploit the utility information for a replacement policy alternative to LRU. As a result, focusing on optimizing locality or utility alone in SLLC capacity management, the two categories of approaches miss delivering robust performance under a variety of workloads. In the following, we elaborate on why it is beneficial to treat locality and utility concurrently and interactively. Our argument is based on workload characterization as well as an evaluation of the two categories of approaches on the workloads that expose their performance comfort and discomfort zones.

Figure 5.1 illustrates the LLC performance for 9 of the benchmarks in our study as a function of assigned cache capacity, managed by LRU and BIP respectively (see Section 5.3 and Section 5.4 for more details). The x-axis shows the LLC capacity measured in the number of ways (with fixed 2048 sets and 64B lines assumed), while the y-axis represents *hits per 1K instructions*. The dotted roofline in each figure indicates the total number of

LLC accesses per 1k instructions (independent of the LLC capacity). The 9 benchmarks are divided into four classes according to their locality and utility characteristics. The 9 benchmarks can be divided into four classes depending on their locality and utility features. The first two classes represent the cases where the performance can be improved with allocation of extra capacity, but they differ in their LRU vs. BIP utility. The last two classes saturate in performance after a minimal allocation of capacity, but with very different hit rates. In the first class, as indicated in Figure 5.1 (a)-(c), benchmarks *xalancbmk*, *sphinx3* and *mcf* all have inferior locality because their LRU curves are significantly below the BIP curves within a certain capacity range (e.g., from associativity 2 to 20 for *mcf*). If any of them runs in a mix of co-scheduled threads on a CMP, an alternative replacement policy such as TADIP can potentially apply an alternative replacement decision better than LRU to improve the SLLC hit performance. In contrast, existing cache partitioning schemes like UCP are oblivious of locality due to their LRU-based utility monitors. For instance, if a cache partitioning scheme decides to allocate 8 cache ways to *mcf*, without the locality information, the scheme will never realize that *mcf*'s hit performance can still be improved by 4.5x ($\approx \frac{72.6-13.1}{13.1}$) with the same capacity allocation by simply altering the replacement policy from LRU to BIP.

In contrast, the workloads in the second class, represented by applications *vpr*, *twolf* and *swim* (illustrated in Figure 5.1 (d)-(f)), show good locality since their LRU curves are never below the BIP curves. However, they can still be set apart from each other with respect to their utility. For instance, when assigned 16 ways, *twolf* has a higher utility than *swim* in that it can yield 28.5 hits per 1K instructions (HPKI) (corresponding to a hit rate of 95.2%) while *swim* can deliver only 19.8 HPKI (with a hit ratio of 55.2%). Further, if *twolf* and *swim* are running concurrently and compete for the SLLC resources such as the 16-way SLLC, an alternative replacement policy like TADIP will detect LRU's better hit performance than BIP (especially for *swim*) and thus adopt the LRU module for both of

them. But since *swim* inherently has many more misses than *twolf* (e.g., the ratio between the MPKIs of *swim* and *twolf* are 5.0 and 11.5 at associativity 8 and 16 respectively), *swim* will occupy much greater capacity than *twolf* due to the underlying miss-driven capacity allocation by LRU. A cache partitioning scheme such as UCP or PIPP, being utility-aware, can do a better job of space partitioning in this case by favoring *twolf*.

Figure 5.1 (g)-(i) illustrate the third and the fourth classes whose applications require very few SLLC resources. In particular, *milc* and *lbm* are both streaming applications due to their high miss rates, while *crafty* is CPU-bound and can yield very high hit rates given a small amount of SLLC capacity.

To better understand the performance impact of the different working principles between alternative replacement policies and cache partitioning schemes, we construct 10 simple dual-core CMP workloads by pairing some of the benchmarks illustrated in Figure 5.1 to expose their performance gaps. We then use the workloads to evaluate the alternative replacement policies TADIP and NUCACHE, as well as the cache partitioning schemes PIPP and UCP, on a dual-core CMP with a 16-way 2MB SLLC (see the experimental setup details in Section 5.4). Figure 5.2 shows that, based on their throughput performance over the baseline LRU, the ten workloads can be divided into two categories, namely *locality-favorable* and *utility-favorable*. For a locality-favorable workload that consists of at least one of the benchmarks with inferior locality, e.g., *xalancbmk+mcf*, an alternative replacement policy like TADIP can greatly optimize the temporal capacity-sharing behavior for co-scheduled threads, which a cache partitioning scheme often fails to do. On the other hand, a utility-favorable workload consists of benchmarks with significantly diverse utility (e.g., *swim+twolf*) such that a cache partitioning scheme can make a better decision on space partition, yielding a better performance than an alternative replacement policy. We also note that an alternative replacement policy like TADIP performs much worse in certain utility-favorable workloads like *sphinx3+twolf*,

even though *sphinx3* presents opportunities for locality improvement. This is because *twolf* begins to interfere with *sphinx3* when they are managed by LRU and BIP respectively, due to the lack of a dedicated space partition for performance isolation in TADIP. In summary, we can infer from this motivational experiment that neither alternative replacement policies nor cache partitioning schemes can consistently perform well under a variety of workloads due to their different working principles.

5.3 The CLU Architecture

CLU is designed to achieve three specific goals: (i) to be thread-aware, which means that it should be able to differentiate between the diverse features of individual threads; (ii) to dynamically profile both utility and locality of co-scheduled threads and fully exploit the interactions between the two dimensions for co-optimization; and (iii) to decide on the optimal management policy by taking into account the locality and utility characteristics of all the threads.

5.3.1 The Overall Architecture

Figure 5.3 depicts an architectural view of CLU. On an N-core CMP, a *locality & utility monitor* is associated with each core and dynamically captures both the utility and locality information about the SLLC access sequence from its host core. In particular, the locality & utility monitor consists of an LRU profiler and a BIP profiler, both of which are based on the set sampling technique [7]. Therefore, only a small group of sampler sets out of all SLLC sets are monitored and the samplers' information is used to deduce the characteristics of the entire SLLC. On every time interval boundary, the profilers feed the information back to the *decision unit* that uses it to determine the space partitioning and replacement policy for all of the co-scheduled threads during the next time period.

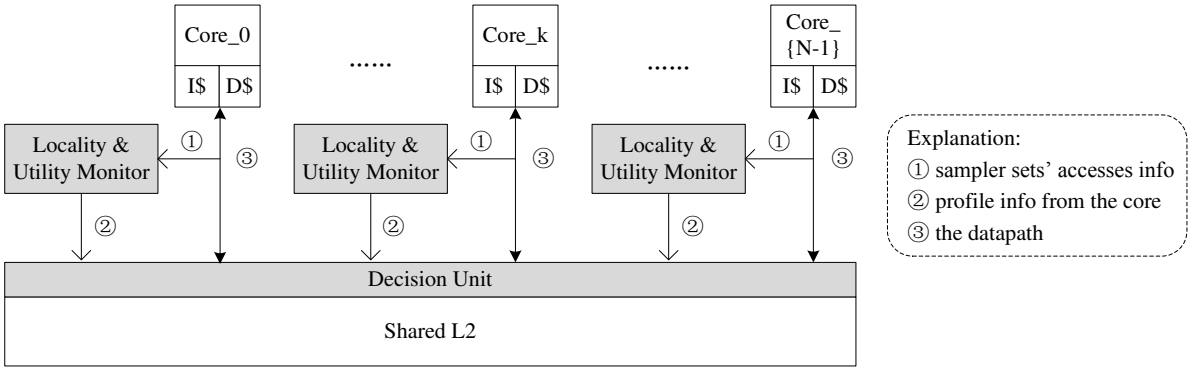


Figure 5.3: CLU Architecture

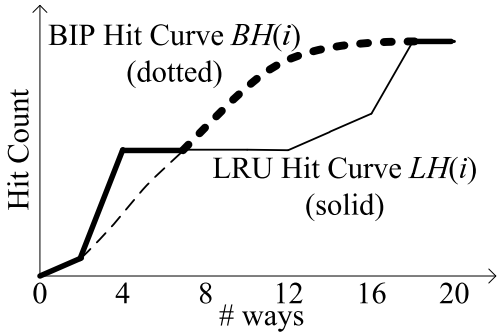


Figure 5.4: Deriving a Composite Hit Curve (Bold, Consisting of the Higher Segments of LRU/Solid and BIP/Dotted Hit Curves)

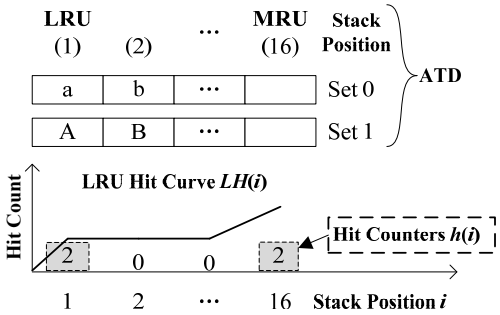


Figure 5.5: Profiling an LRU Hit Curve with the Mattson's LRU Stack Algorithm

5.3.2 The Locality & Utility Monitor

The *locality & utility monitor* counts the SLLC hits that a thread would contribute if it were running alone, while the amount of space it is assigned and the replacement policy (LRU vs. BIP) adopted to manage the allocated space are both varied. By so doing, the monitor attempts to capture the runtime interplay between the locality and the utility optimizations in SLLC management. Assuming that an SLLC has an associativity of 64, for example, the monitor counts the number of hits a thread would contribute if it were allocated 1-, 2-, . . . , or 64-way SLLC space, being managed by LRU and BIP respectively. Consequently, the monitor is able to deduce both the LRU and the BIP hit curves that

are a function of cache ways respectively, as illustrated in Figure 5.1 and generalized in Figure 5.4. The two curves can jointly convey two critical pieces of information:

- Which replacement policy should be adopted under a given capacity quota for the thread. As depicted in Figure 5.4, if the thread can get 4 cache ways, it should apply the LRU replacement policy to manage the given amount of space, since the LRU hit curve (solid) is above the BIP curve (dotted) when the way count equals 4; but if the assigned way-count is 12, the thread should alter the policy to BIP that can help it obtain far more hits. Therefore, with the two curves, CLU can implicitly derive a *composite hit curve* (bold) which consists of the higher segments of the LRU or BIP curves.
- What the preferred utility is under the best replacement policy. For instance, if the hit counts of the derived composite hit curve at the way counts of 10 and 12 are assumed to be 100 and 110 respectively, we know that the utility of 10 ways is better than that of 12 ways because $\frac{100}{10} > \frac{110}{12}$. In this way, CLU fully exploits the interactions between the locality and the utility dimensions.

To be detailed next, we apply two different profiling mechanisms to respectively deduce the LRU and the BIP hit curves of a thread, since LRU satisfies the stack property [42] while BIP does not. Specifically, the stack property stipulates that the blocks that would be in an A-way associative cache should be subsumed by those that would be in an $(A+1)$-way associative cache.

5.3.2.1 Profiling the LRU Hit Curve

To obtain the LRU hit curve, we leverage the well-established profiling technique [12] that leverages the *set sampling* strategy and the *Mattson's LRU stack algorithm* [42]. Specifically, an *auxiliary tag directory* (ATD) with an associativity of A and a size-A array of *stack-hit*

$$LH(i) = \sum_{1 \le k \le i} h(k), \text{ where } h(k) \text{ is the hit counter at LRU-stack position } k \text{ and } 1 \le k \le i \le A \tag{5.1}$$

$$BH(i) = \begin{cases} BH(2^k), & \text{where } i = 2^k \text{ and } BH(2^k) \text{ can be monitored by ATD}(2^k) \\ BH(i+1) - \Delta, & \text{where } \Delta = \frac{BH(2^{k+1}) - BH(2^k)}{2^k} \text{ and } 1 \le 2^k < i < 2^{k+1} \le 2^m = A \end{cases} \tag{5.2}$$

counters are adopted to implement the *Mattson's LRU stack algorithm*, where A is also the SLLC's set associativity, as shown in Figure 5.5. Here, an ATD structure, with each of its entries containing only the tag field, mimics the LRU stack of a small group of sampler SLLC sets, as if the monitored thread were exclusively occupying the whole space of these sampler sets. Upon every hit on the ATD, it reports the LRU-stack position where the hit takes place so that the corresponding stack-hit counter $h(i)$ can be incremented by one. As a result of the stack property of LRU, the value of the LRU hit curve at way count i, denoted $LH(i)$, can be expressed by Equation 5.1.

5.3.2.2 Profiling the BIP Hit Curve

The profiling of the BIP hit curve, on the other hand, is more challenging because BIP violates the stack property by placing incoming blocks at the LRU position of any cache set with a high probability or at the MRU position with the complementary (low) probability. Thus, the simple stack algorithm cannot be applied to deducing the BIP hit curve. To resolve this issue, we first propose an exact but complex approach and follow it with an approximate but practical solution.

The exact approach is also based on set sampling, and uses a number A of ATD structures representing the A different associativities from 1 to A. Therefore, in the exact approach, we use a group of A ATD structures, {ATD(1), ATD(2), ..., ATD($A-1$) and ATD(A)}, to mimic BIP's operations on the sampler SLLC sets with an associativity ranging from 1 to A respectively, where ATD(k) stands for an ATD structure with an

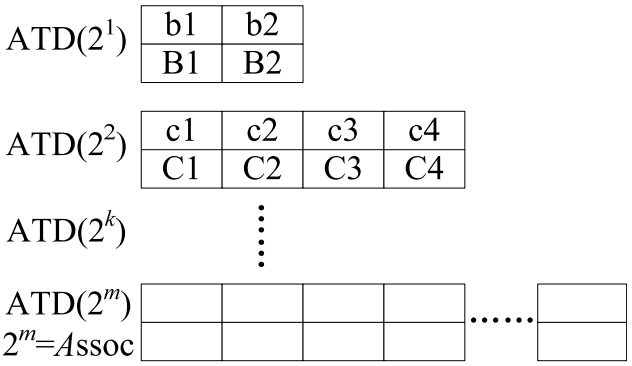


Figure 5.6: Practically but Approximately Profiling a BIP Hit Curve

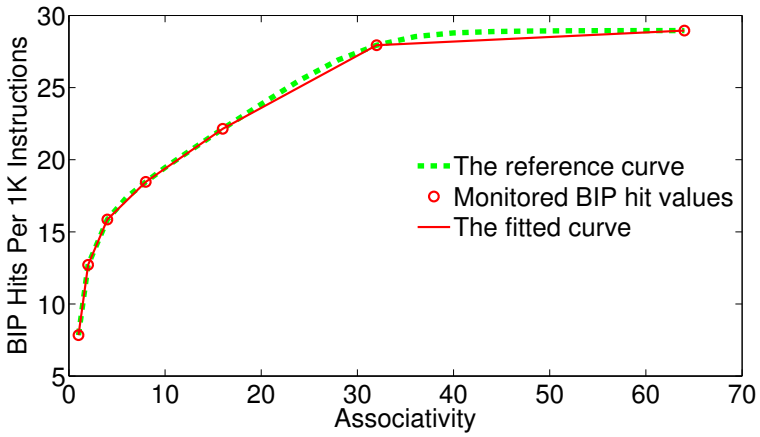


Figure 5.7: An Example of Applying the Logarithmic-Distance Monitoring & Curve-Fitting Approach to Profiling the BIP Hit Curve of Benchmark *xalancbmk* (by Approximating the Exact Reference Curve)

associativity of k. By monitoring any ATD(k) structure, the corresponding $BH(k)$, namely the value of the BIP hit curve at way count k, can be determined as the total hit count of ATD(k) under BIP. Although this approach provides an exact measure of the BIP curve, it requires a significant number A of ATD structures, which makes the implementation prohibitively expensive if A is large, even when a single ATD structure is lightweight [12, 13].

The practical solution is based on four key observations derived from an analysis of the BIP hit curves for the benchmarks in our study (exemplified in Figure 5.1): (i) the BIP

hit curve is monotonically non-decreasing with respect to the assigned way count; (ii) the BIP hit curve is a concave function, which means that the curve's gradient is always non-increasing as the way count increases. The intuition behind concave BIP curves is that, at non-LRU stack positions, the blocks hardly get evicted by incoming blocks (namely, stationary) and are also ranked from MRU to descending positions based on recency; (iii) the BIP hit curve has a long flat tail as the way count approaches a high value; and (iv) it is provable that the LRU and the BIP hit curves have the same value at way count 1 ($BH(1) = LH(1)$), since the LIP (LRU insertion policy) module in BIP does not let an incoming line bypass the cache [7]. Therefore, it is sufficient to monitor the BIP hit values at a small number of discrete logarithmic way-count points by using a dedicated ATD for each of these points, and then apply the *curve fitting* technique to deduce the entire BIP hit curve. Specifically, we employ m ATD structures $\{\text{ATD}(2^1), \text{ATD}(2^2), \ldots, \text{ATD}(2^m)\}$ to capture the BIP hit counts $\{BH(2^1), BH(2^2), \ldots, BH(2^m)\}$ in a small number of way-count cases $\{2^1, 2^2, \ldots, 2^m\}$, where $m = \log_2 A$. We carry out curve fitting based on the m discrete BIP hit values by linearly interpolating between the two monitored BIP curve counts $(2^k, BH(2^k))$ and $(2^{k+1}, BH(2^{k+1}))$. Then, the $BH(i)$ value can be calculated iteratively by Equation 5.2. Figure 5.7 shows an example of applying our logarithmically discrete monitoring and curve-fitting approach with up to 64 ways for the benchmark *xalancbmk*. The specific design choice of monitoring at logarithmic way-count points stems from our empirical observations mentioned above, suggesting a denser number of monitoring points to more accurately profile the high-gradient portion of the BIP hit curve when A is small, which is also a property of a logarithmic/geometric series.

As described above, the practical solution needs only $m = log_2 A$, instead of A, BIP-managed ATD structures at the associativities of 2, 4, $\ldots$, $\frac{A}{2}$ and A respectively, as well as m BIP-hit counters. It is worth remarking that the storage overhead (measured in the total

number of ATD ways) required by the practical BIP profiling is $2+4+8+\cdots+\frac{A}{2}+A = \frac{2\times(A-1)}{2-1} = 2\times(A-1) < 2\times A$, *which is less than twice the storage overhead required by a single A-way ATD structure for the LRU profiling and makes our solution very practical in hardware implementation.* It needs to be noted that, upon an access to one sampler SLLC set, the LRU-managed ATD and the m BIP-managed ATDs are operated concurrently for both the LRU- and BIP-curve profiling.

5.3.3 The Decision Unit

With the locality and utility characteristics of co-scheduled threads profiled during each time interval, the decision unit will periodically determine the optimal space partition and replacement policy for individual threads by leveraging on all their locality and utility information fed by the monitors. Since the space partitioning logic of CLU is also utility-based, aimed at maximizing the overall performance, we adopt the framework of the *lookahead utility-based cache partitioning* algorithm [12]. The original algorithm evaluates every potential partitioning decision and provisions cache ways to a thread that currently has the highest utility of these ways. We modify the algorithm to determine the best utility-based partitioning of the cache ways according to the composite hit curves, each of which is composed of the higher segments of the LRU and the BIP hit curves. Other studies only examine the utility of an LRU hit curve, which has been shown to be ineffective in the case of poor locality in Section 5.2.

On each time interval boundary, the SLLC's space-partitioning result for $Core_i$ is kept in an m-bit *partition quota counter*, denoted as Q_i, where $0 \leq i \leq N-1$ and $m = log_2 A$. Assuming that A is greater than N, CLU also guarantees that at least one way is provisioned to every core. With each core $Core_i$ the *decision unit* in CLU also associates a (*locality management*) bit, LM_i, to indicate either LRU (LM=0) or BIP (LM=1) to

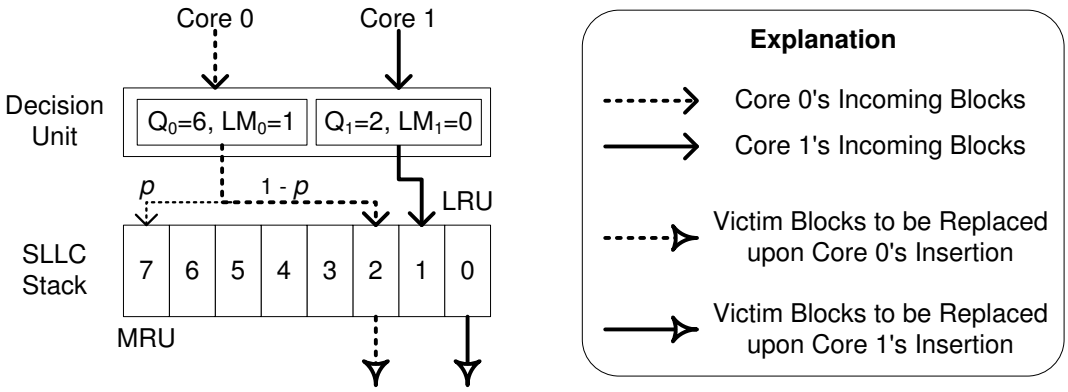


Figure 5.8: An Example of Enforcing the Management Decisions

be adopted for the core in its allocated SLLC space. LM_i can be determined by examining the difference between the LRU and the BIP curves at the value k of the partition quota counter Q_i: the bit is set 0 if $LH(k) \geq BH(k)$ or 1 otherwise.

CLU enforces its space partitioning and replacement policy with specific promotion, insertion and victimization strategies. The *single-step promotion* policy [13] is adopted as CLU's cache-block promotion mechanism. As illustrated in Figure 5.8, we assume that the SLLC's LRU stack is numbered $0, 1, \cdots, A-1$ from the LRU position to the MRU position. When a new block is brought in by $Core_i$, if its LM_i is 0, the LRU block of the target set is replaced and the incoming block is inserted at position $k-1$, where k is the value of Q_i. This part is similar to the promotion, insertion and victimization module of PIPP [13]. On the other hand, if $Core_i$'s LM bit is 1, the block at position $A-k$ will be victimized, and the block brought in by the core will be inserted at position $A-k$ with a high probability and at the MRU position with the complementary low probability. Therefore, if a core's incoming block stream shows a poor locality (i.e. its LM bit is 1), part of its working set can be preserved well in its allocated space with the BIP-like victimization and insertion. Figure 5.8 demonstrates a dual-core example with an 8-way SLLC managed by CLU: for $Core_1$, its incoming blocks are always placed at position 1

with the LRU blocks victimized, because it has good locality and gets a space quota of 2 SLLC ways; but for $Core_0$, since it exhibits inferior locality and is allocated with 6 ways, the blocks at position 2 ($= 8 - 6$) will be replaced upon insertion, and its incoming blocks will be inserted at the MRU position with a low probability (1/32 in our study and other BIP-related work [7, 14]) and at position 2 otherwise.

With respect to the time complexity, similar to existing cache partitioning approaches UCP and PIPP, the runtime performance overhead of CLU is negligible for the following reasons: (*i*) monitoring is in parallel and not intrusive with normal cache operations; (*ii*) every 5 million cycles, decision making is conducted in the background and not in the critical path of cache accesses; (*iii*) for A cache ways, curve fitting only involves addition, subtraction and shift operations (see Equation 5.2), while deriving a composite curve just needs to compare LRU and BIP hit counts at each of the A way-points, and both of them can be accomplished in linear time; (*iv*) only several partition quota counters and locality management bits will be modified to embody the new management decisions, of which the time complexity is trivial; (*v*) decision enforcement only changes the RRPVs of cache blocks in one set upon a cache hit, miss or fill, without moving or flushing a number of blocks.

5.4 Experiments & Evaluation

In this section, we first briefly describe our simulation-based experimental methodology and then present and analyze the evaluation results.

5.4.1 Evaluation Methodology

Simulation Setup: We use the cycle-accurate M5 full system simulator [90] with the configuration parameters listed in Table 5.1. For the memory hierarchy, we model two

Table 5.1: Major Configuration Parameters

Core (2/4/8)	Alpha ISA, in-order, IPC=1 except for memory accesses, 128/128 I/D TLBs
L1	2-way, 32KB, 64B/line, 1-cycle delay, 16 MSHRs, write back & 4 write buffer entries for L1D
L2	64B/line, 6/8-cycle tag/data store delay, totally 1024 MSHRs, write back & totally 256 write buffer entries, physically tagged and indexed (M5's built-in setting), 2MB & 16 ways/4MB & 16 ways/8MB & 32 ways for 2/4/8-core configurations
NoC	mesh topology $(1 \times 2, 2 \times 2, 2 \times 4)$ with 1 cycle delay per hop
Mem	300-cycle delay
Others	#(sample sets/core) = 32, BIP's low probability = 1/32, hit counter length = 16 bits, single-step promotion probability = 3/4, stream promotion probability = 1/128, NUCACHE L2 setups are the same as in [15]

Table 5.2: Selected Benchmarks & Classification

Class	Descriptor	Benchmarks
I	Poor Locality	galgel, mcf, libquantum, omnetpp, sphinx3, xalancbmk
II	Good Utility	ammp, swim, twolf, vpr, bzip2 calculix, gcc, GemsFDTD
III	Streaming	lucas, lbm, milc
IV	CPU-Bound	crafy, fma3d

levels of on-chip caches. The L1 instruction and data caches adopt the conventional set-associative configuration, the LRU replacement policy, and a coupled tag-data store organization. For the shared L2 cache, we model decoupled tag and data stores for each L2 slice, and take into account the NoC latency when calculating the L2 access time. Using representative and specially-constructed workloads, we evaluate and compare the performance of LRU (baseline), TADIP, NUCACHE, UCP, PIPP and the proposed CLU for the dual/quad/eight-core configurations. TADIP reevaluates its management decisions whenever any saturating counter of its monitor has its MSB altered, while NUCACHE, PIPP, UCP and CLU make management decisions every 5M cycles. The 16-bit profiler hit

counters in PIPP, UCP and CLU are reset upon each periodic decision boundary, and we have not found any overflow problems with the counters in our experiments. Particularly, the aforementioned schemes are all rooted in the true-LRU environment that the RRPV of each cache block has $\lceil \log_2 Associativity \rceil$ bits. In essence, for the true-LRU based schemes, the representative and specially-constructed workloads are generated to expose the performance gap between the locality-oriented and the utility-oriented approaches and demonstrate CLU's ability of bridging the gap. Then, we also use random workloads to evaluate the general overall performance for all of the true-LRU based schemes, as well as the pseudo-LRU based approach TA-DRRIP [16] that is also an alternative replacement policy but uses only 2 bits per cache block for the RRPV.

Performance Metrics: We adopt two standard metrics of *throughput* and *fair speedup* to quantify the CMP performance. Specifically, *throughput* measures the utilization of a system, while *fair speedup* balances both performance and fairness. Let IPC_i be the *instructions per cycle* performance of the ith thread when it is co-scheduled with other threads and $SingleIPC_i$ be the IPC of the same thread when it executes in isolation. Then, for a system where N threads execute concurrently, the formulas for the two metrics are shown in Equation 5.3 and Equation 5.4.

$$\text{throughput} = \sum_{i=1,2,\cdots,N} IPC_i \tag{5.3}$$

$$\text{fair speedup} = \frac{N}{\sum_{i=1,2,\cdots,N} SingleIPC_i / IPC_i} \tag{5.4}$$

Workload Construction: As listed in Table 5.2, we select 19 benchmarks from the SPEC CPU 2000 and 2006 benchmark suites and categorize them into four classes according to their locality and utility. Class I is a collection of benchmarks that exhibit poor locality and can be improved by judicious replacement policies. The benchmarks in Class II

Table 5.3: Workload Construction

MIX2	Apps	MIX4	Apps	MIX8	Apps
1	mil+omn	1	two+omn+lbm+mcf	1	2 sph + 2 omn + 2 mcf + 2 two
2	omn+xal	2	lbm+amm+mcf+omn	2	2 two + 2 vpr + 2 omn + 2 sph
3	xal+mcf	3	mcf+two+omn+sph	3	2 swi + 2 bzi + 2 luc + 2 omn
4	omn+fma	4	omn+two+mcf+gal	4	lib+sph+mcf+amm+swi+two+fma+mil
5	lbm+omn	5	amm+omn+two+mcf	5	2 swi + 2 mcf + 2 sph + 2 omn
6	mcf+two	6	xal+two+omn+mcf	6	2 mil + 2 vpr + 2 omn + 2 mcf
7	sph+mcf	7	two+swi+mcf+omn	7	2 bzi + 2 luc + 2 sph + 2 omn
8	fma+xal	8	fma+omn+lbm+swi	8	2 mcf + 2 swi + 2 bzi + 2 luc
9	xal+vpr	9	cra+swi+lbm+omn	9	2 omn + 2 mcf + 2 swi + 2 bzi
10	mcf+lbm	10	omn+lbm+gal+mcf	10	2 sph + 2 omn + 2 luc + 2 swi
11	swi+two	11	amm+omn+vpr+lib	11	gcc+sph+mcf+amm+vpr+omn+two+mil
12	lbm+xal	12	gal+amm+omn+vpr	12	two+gal+omn+mcf+gcc+lib+xal+vpr
13	vpr+lbm	13	mcf+omn+vpr+sph	13	omn+sph+mil+gcc+lib+two+swi+lbm
14	Gem+two	14	lbm+xal+fma+omn	14	vpr+swi+two+lib+fma+mcf+omn+xal
15	two+mil	15	mcf+lib+omn+amm	15	lib+sph+omn+gcc+two+xal+gal+lbm
16	lib+xal	16	omn+lib+xal+cra	16	2 omn + 2 swi + 2 two + 2 bzi
17	two+sph	17	two+xal+lib+omn	17	2 xal + 2 omn + 2 luc + 2 swi
18	xal+swi	18	mil+cra+omn+xal	18	2 sph + 2 xal + 2 omn + 2 mcf

have excellent utility and need dedicated SLLC space partitions. Class III is a group of streaming applications that require little SLLC capacity and need to be prevented from polluting the SLLC. Finally, Class IV benchmarks are CPU-bound with small working sets in the SLLC. From the four classes of benchmarks, we can construct dual/quad/eight-core workloads in Table 5.3, which can be further divided into locality-favorable and utility-favorable categories shown in the top and the bottom halves respectively. Every locality-favorable workload consists of at least one Class I benchmark and should enable either TADIP or NUCACHE to outperform both capacity-partitioning schemes. In contrast, for every utility-favorable workload constructed using benchmarks with diverse utility, PIPP and UCP are supposed to achieve a better performance than the alternative replacement policies.

Simulation Control: In the experiments, all threads in a workload are started from a checkpoint that has already had the first 20 billion instructions bypassed. They are cache-warmed with 1 billion instructions and then simulated in detail until all threads

finish another 1 billion instructions. Performance statistics are reported for a thread when it completes the latter 1 billion instructions. If one thread finishes the 1 billion instructions before others, it continues to run so as to still compete for the SLLC capacity, but its extra instructions are not taken into account in the final performance report.

5.4.2 Performance Comparison Using Representative and Specially-Constructed Workloads

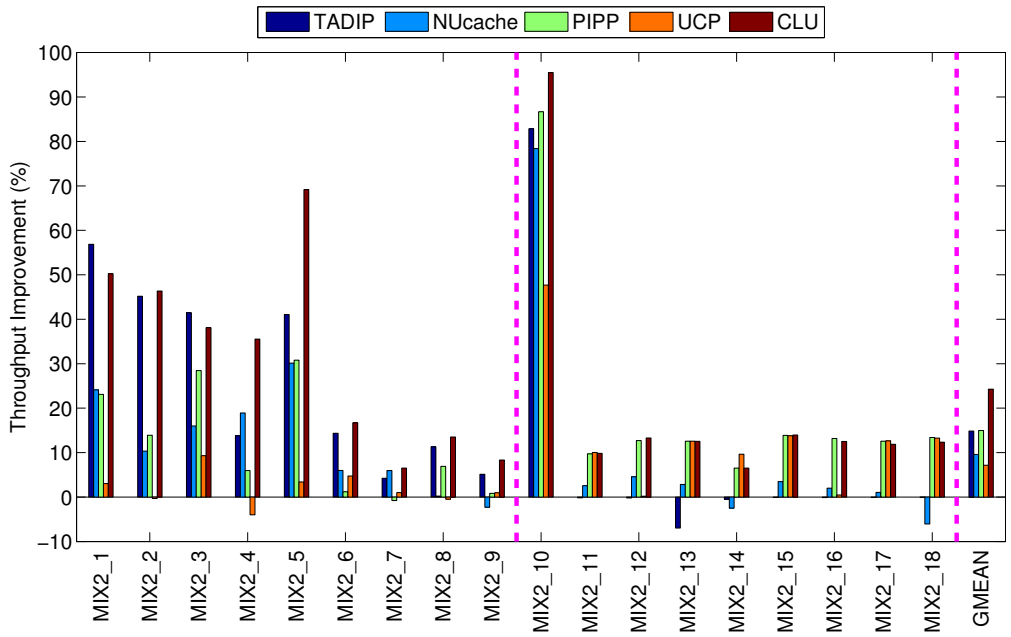


Figure 5.9: Throughput of Handpicked Dual-Core Workloads

Figure 5.9 shows the throughput performance of TADIP, NUCACHE, PIPP, UCP and CLU normalized to the baseline (LRU) on the simulated dual-core configuration. For 18 dual-core workloads, CLU provides a throughput improvement of 24.3% on average (and up to 95.5%), which is much higher than the improvements by the locality-oriented (TADIP: 14.9%, NUCACHE: 9.6%) and the utility-oriented (PIPP: 15.0%, UCP: 7.2%) approaches. If we look closer at the specific categories of workloads, we can

find that the higher improvements of CLU come from its capability of bridging the performance gap between alternative replacement policies and capacity partitioning schemes. Specifically, for the locality-favorable workloads MIX2_1-MIX2_9, the better alternative replacement policy TADIP, can improve their throughputs by 24.6% on average, while the better capacity partitioning scheme (PIPP here) can only yield 11.7% higher performance over the baseline, in contrast to CLU's 30.1%. In terms of the utility-favorable workloads MIX2_10-MIX2_18, however, PIPP and UCP can improve their performance by 18.3% and 12.7% respectively, while TADIP and NUCACHE only improve by 5.9% and 7.5%, compared to CLU's 18.8%. Therefore, while CLU outperforms all of the existing approaches throughout both locality-favorable and utility-favorable workloads, none of the locality-oriented or utility-oriented approaches can perform consistently well.

We can further explain the performance implications of existing approaches and CLU by means of case studies. On the one hand, for the locality-favorable workload MIX2_1 which is the combination *omnetpp+milc*, TADIP can provide a much better performance than both of the capacity partitioning schemes. Specifically, *milc* is a streaming application, and *omnetpp*, belonging to benchmark Class I, can be improved by smart replacement policies. In this scenario, TADIP will adopt its BIP module to manage both threads so as to prevent thrashing for *milc* and significantly promote the performance of *omnetpp* by preserving a large part of its working set in the SLLC. PIPP and UCP cannot do as well as TADIP through SLLC partitioning alone (e.g., giving at least 15 ways to *omnetpp* in a 16-way 2MB SLLC), because they cannot detect *omnetpp*'s being in Class I. So, PIPP and UCP will insert the incoming blocks of *omnetpp* at the high position of the SLLC stack (position 14 for PIPP and position 15/MRU for UCP), still thrashing its working set. With its locality & utility monitor and BIP-like insertion, however, CLU can match TADIP's performance. On the other hand, for the utility-favorable workload MIX2_11 *swim+twolf* that exhibits diverse utility, no opportunities are present for locality-oriented improvement. Thus,

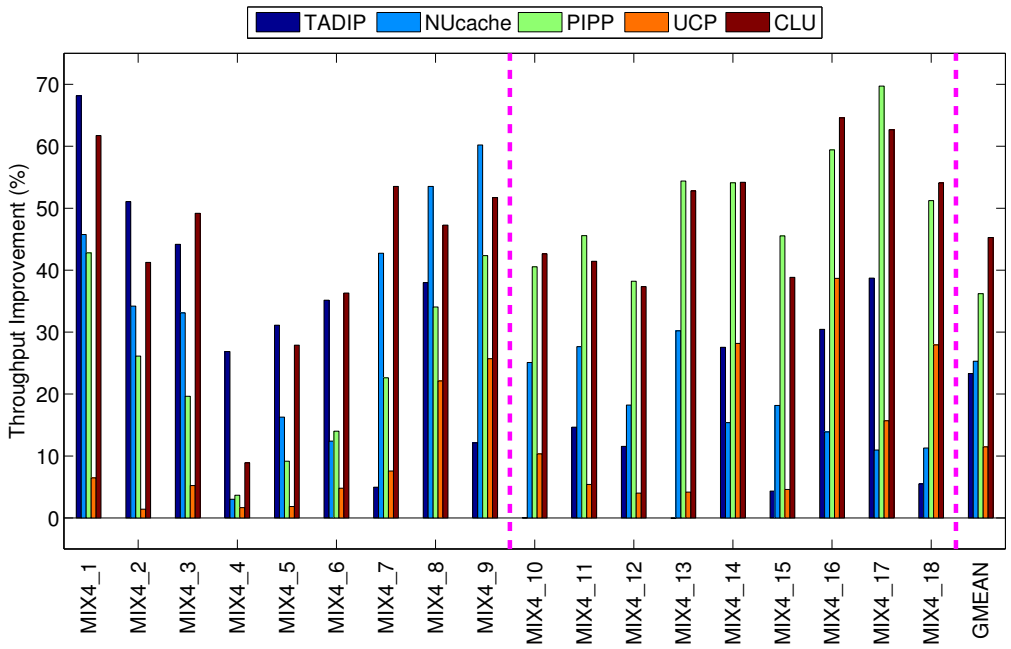


Figure 5.10: Throughput of Handpicked Quad-core Workloads

PIPP and UCP can improve the performance by 9.7% and 10.0% respectively, better than TADIP (-0.2%) and NUCACHE (2.6%). Since CLU also has a utility-management module, it can improve the performance of this benchmark combination by 9.8%.

Figure 5.10 and Figure 5.11 present the performance comparison among the schemes for the quad-core and eight-core configurations. For both configurations, the gap between alternative replacement policies and cache partitioning schemes is similarly manifested by a significant impact on the performance. Again, CLU exploits the opportunities for performance improvement in both locality and utility dimensions interactively and provides 45.3% and 43.3% higher throughputs over the baseline for quad-core and eight-core systems respectively, significantly outperforming other approaches. It must be noted that CLU slightly underperforms some of the locality-oriented and the utility-oriented approaches under a few workload combinations, such as the cases MIX4_1 and MIX8_17. This is because CLU is designed to strike a good balance/compromise between the locality

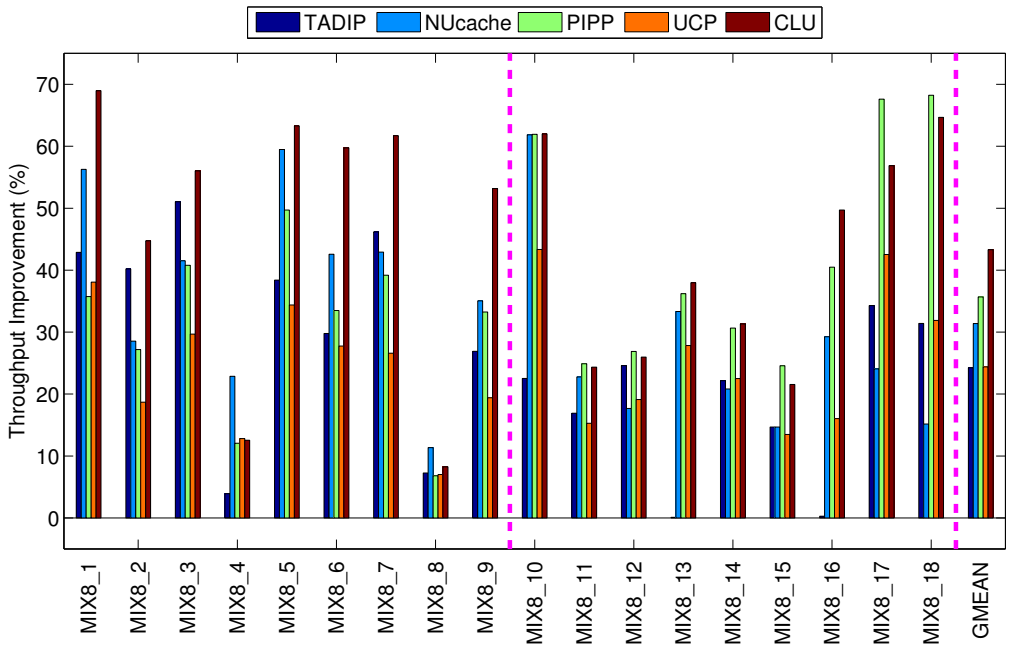


Figure 5.11: Throughput of Handpicked Eight-Core Workloads

and the utility optimizations. Therefore, it may not work as aggressively as a single-dimension management approach when the opportunities for performance optimization dominate in exactly one dimension for a workload. However, CLU can still win out robustly in a much broader range as a result of its adaptive management capabilities. More interestingly, when the opportunities of interactively exploiting both locality and utility are significant for a workload, CLU is able to co-optimize the management decisions, leading to its superior performance to the existing approaches (e.g., MIX2_10, MIX4_16 and MIX8_1).

Figure 5.12 illustrates the performance impact of different SLLC management schemes for the *fair speedup* metric. For the dual-, quad- and eight-core configurations, CLU outperforms the baseline by 23.1%, 43.0% and 36.5% on average (geometric mean) respectively, which are much better than any of the locality-oriented or the utility-oriented approaches (TADIP: 16.1% / 25.0% / 22.4%, NUCACHE: 11.4% / 30.5% /32.6%, PIPP: 16.1% / 34.7%

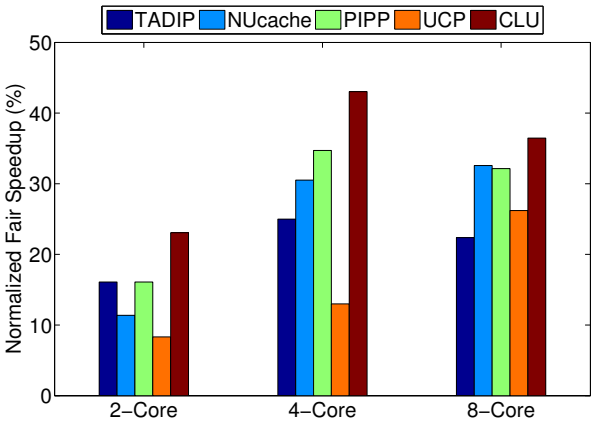


Figure 5.12: Fair Speedup Improvement

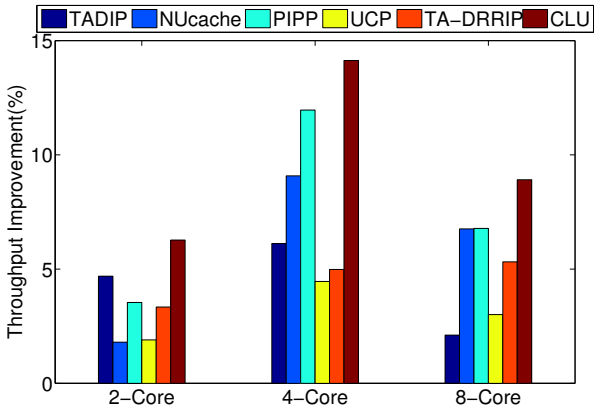


Figure 5.13: Throughput of 50 Random 2/4/8-Core Workloads

/ 32.1%, UCP: 8.3% / 13.0% / 26.2%). This set of results reveal that not only can CLU provide higher absolute throughput but also it is able to improve on the fairness over the existing schemes.

5.4.3 Performance Comparison Using Randomly-Selected Workloads

So far, we have demonstrated CLU's capabilities of locality and utility co-optimization with representative and specially-constructed workloads. Next, we show that CLU also performs well in randomly-selected workload combinations. In particular, the pseudo-LRU based scheme TA-DRRIP that uses 2 bits ($2 < \lceil \log_2 Associativity \rceil$) per block for the RRPV (see Chapter 2) is also included in the comparison. Specifically, for the dual-core, quad-core and eight-core configurations, we generate 50 random workload combinations from the pool of 19 SPEC CPU benchmarks in Table 5.2. Figure 5.13 illustrates the five schemes' average throughputs on the 50 randomly-selected dual-core, quad-core and eight-core workloads respectively, where CLU is shown to outperform all other schemes with a throughput improvement of dual cores by 6.3%, quad cores by 14.1% and eight cores by 8.9% over the baseline (true-)LRU scheme.

We find that the normalized performance of TADIP under random workloads in our study (6.1% / 2.1% for quad/eight-core systems respectively) is different from those reported in [14] (18% and 15% respectively). We speculate that several factors may have contributed to this discrepancy. For instance, [14] uses a CMP simulator based on the X86 ISA that has much more sophisticated memory addressing modes than a RISC ISA (e.g., the Alpha ISA in the M5 simulator), such as the register/immediate/direct/indirect/indexed addressing modes, which may make memory accesses more intensive in X86. In addition, our benchmark pool differs from the one adopted in [14], which can also contribute to the discrepancy. We notice, however, that the mutual/relative performance trend among TADIP, NUCACHE, PIPP and UCP remains consistent with the conclusions in their respective studies [14, 15, 13, 12].

Another interesting observation is that the performance gap between CLU and PIPP shrinks under the random workloads, compared to the gap under the specially-constructed ones. The underlying reason might be that, as analyzed in Section 5.2.1, the capacity partitioning scheme PIPP has an *ad hoc* ability for locality-oriented improvement via such mechanisms as the stream handler or the single-step promotion [13]. We speculate this ability helps PIPP perform better in the locality dimension under the random workloads than under the specially-constructed workloads. However, CLU's ability to co-optimize locality and utility is systematic and enables it to consistently outperform PIPP in both kinds of workloads.

Furthermore, TA-DRRIP underperforms TADIP by 1.4% and 1.1% for the dual-core and the quad-core random workloads respectively. This is mainly because the pseudo-LRU based TA-DRRIP uses only 2 bits for each block, while the true-LRU based TADIP uses $\lceil \log_2 Associativity \rceil$ bits (>2). Besides, it was experimentally shown in [16] that RRIP's design choice of using 2 bits per block achieves almost the same performance as using a higher number of bits (e.g., $\lceil \log_2 Associativity \rceil$). This is why the 2-bit TA-DRRIP is

Table 5.4: Hardware Overhead Details

address length	44-bit physical address
cache line size	64 bytes
associativity	16 ways for 2MB dual-core and 4MB quad-core, 32 ways for 8MB eight-core
#(sample sets)/core	32
tag bits	27 bits for 2-core, 26 bits for 4/8-core
monitor tag (hash)	10 bits for 2/4/8-core configurations
valid bit	1 bit
RRPV	4 bits per line for 2/4-core, 5 bits per line for 8-core
hit counter	16 bits each

chosen for evaluation and comparison in our experiments. We also notice that TA-DRRIP outperforms TADIP by 3.2% under the eight-core workloads. Our speculation is that, in this case, TA-DRRIP's capability of being both thrashing-resistant and scan-resistant [16] enables it to outperform TADIP that is only thrashing-resistant. Nevertheless, TA-DRRIP underperforms CLU under all of the dual-core, quad-core and eight-core random workloads, although CLU is not equipped with a scan-resistant module. We speculate that it is CLU's capability of locality and utility co-optimization that enables it to consistently outperform TA-DRRIP's locality-oriented management of being thrashing-resistant and scan-resistant.

5.4.4 Overhead Estimation

Since CLU requires an LRU profiler and a BIP profiler for per-core locality & utility monitoring, the shadow sets and hit counters will dominate the hardware overhead in its design. Specifically, each shadow set entry consists of a tag field, a valid bit and a RRPV field, as listed in Table 5.4. We also found in experiments that 16 bits are sufficient for a hit counter. Therefore, we can estimate that the storage overhead of CLU (with the practical BIP profiler) in dual-core, quad-core and eight-core systems are 5.79KB,

5.61KB and 11.46KB for the locality & utility monitor of each core, amounting to 0.56%, 0.55% and 1.12% of the overall SLLC capacity respectively. Moreover, we found that if we apply the 10-bit hash function in the study [92] to the tag field in the locality & utility monitoring logic, we can further reduce the overhead to 0.26%, 0.27% and 0.53% for dual-core, quad-core and eight-core configurations with negligible performance change compared to using full tag bits.

5.5 Summary

In this chapter, we demonstrate that the existing alternative replacement policies or cache partitioning schemes cannot adapt to a wide spectrum of workloads with diverse locality and utility since they are oriented towards either of the two optimization goals only. Therefore, we propose CLU, a novel SLLC capacity management scheme that is capable of interactive locality and utility co-optimization. By employing lightweight monitors that profile both LRU and BIP hit curves, CLU can exploit the co-optimized locality and utility of concurrent threads and effectively manage the SLLC capacity for CMP workloads. Our execution-driven simulation shows that CLU can improve the throughput by 24.3%, 45.3% and 43.3% for our simulated dual-core, quad-core and eight-core systems (with 0.26%, 0.27% and 0.53% storage overhead) respectively, outperforming the existing alternative replacement policies and cache partitioning schemes.

Chapter 6

Locality & Utility Co-optimization for Practical Capacity Management of Shared Last Level Caches

6.1 Problem Definition

For shared last level caches, it has been noticed that the *least-recently-used* (LRU) replacement policy becomes less effective due to the diminished *access locality* at the last cache level [14, 52, 57, 15] and the uncoordinated capacity allocation among heterogeneous threads [12, 13]. As detailed in Chapter 2, in response to LRU's limitations, two approaches have emerged in the literature. First, *alternative replacement policies*, such as TADIP [14], SDBP [52] and NUcache [15], have been proposed to manage locality by temporally assigning and adjusting lifetime for blocks. Second, working with a different principle, *cache partitioning schemes*, including UCP [12] and PIPP [13], try to optimize *utility* by spatially partitioning the SLLC capacity among concurrent threads to maximize performance.

Although the aforementioned proposals have demonstrated desirable performance improvement over LRU in simulations, they are not practically useful due to the high storage overhead entailed by them. Specifically, they are all based on the assumption that each cache line has $\log A$ bits for its *re-reference interval prediction value* (RRPV) [16] that is used to estimate how soon an accessed block will be reused, where A is the set associativity. But the $\log A$-bit overhead per line is considered to be prohibitive for SLLCs according to industry standards [57]. As a result, since the uniprocessor era, commodity processors have relied on lightweight LRU approximations for cache management, such as the *not-recently used* (NRU) replacement policy that requires just one-bit overhead in each cache line. It has been experimentally shown that the lightweight NRU is able to perform almost (99.52% [16]) as well as LRU but still cannot provide optimized performance for CMPs either.

Recent efforts have attempted to bridge the gap between the theoretical cache research and practical SLLC designs. Jaleel *et al.* [16] propose to use 2 bits in each line's RRPV field for a thread-aware dynamic SLLC replacement policy called TA-DRRIP. TA-DRRIP outperforms the baseline LRU and NRU policies by coordinating locality optimization for all of the co-scheduled threads. Rooted in the same two-bit RRPV substrate, the recent work SHiP [17] further improves over TA-DRRIP by considering the differences in locality at the finer-grained memory instruction level for re-reference interval prediction, but incurs more overhead than the thread-level TA-DRRIP approach.

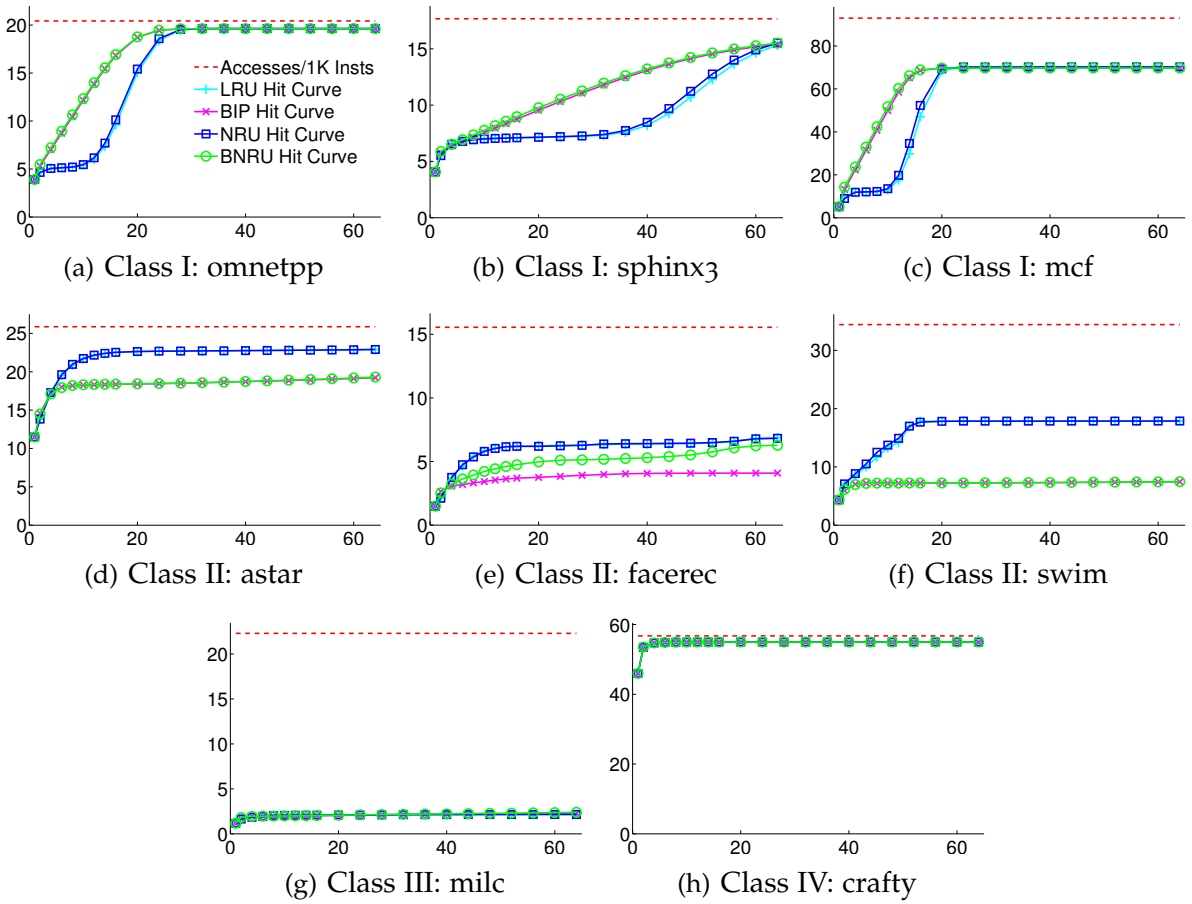
Through our analysis of and experimental study on practical SLLC capacity management solutions, we obtain two important insights that counter the previous research: (*i*) since the minimal-overhead NRU achieves almost the same practical performance as LRU but lacks such theoretical traits as the LRU stack property, it is possible to adopt the minimal-overhead 1-bit RRPV substrate in the entire SLLC and utilize monitors with good theoretical properties yet at a slightly more storage cost for just sample sets, so

that the goals of overhead reduction and performance improvement can be achieved at the same time; (*ii*) both locality and utility optimization opportunities are present in heterogeneous CMP workloads, but the practical schemes such as TA-DRRIP and SHiP are oriented only towards locality management, missing performance potentials provided by utility optimization.

Hence, we propose a novel practical SLLC management design, called COOP (an acronym for *locality and utility CO-OPtimization*), which achieves higher performance than both TA-DRRIP and SHiP but at comparable or lower overhead. COOP uses a single bit in each line's RRPV field, and employs a classic LRU-stack hit profiler and a novel *logarithmic-distance* BNRU (a.k.a., *bimodal* NRU, for thrashing prevention) hit profiler to monitor the interleaved locality and utility of each thread. Leveraging the information about all co-scheduled threads, COOP spatially allocates SLLC cache ways among the threads and temporally makes the best use of their partitions in an interactive way, so that the highest utility provided by either NRU or BNRU, whichever is better, can be exploited by locality and utility co-optimization for all of the threads. Our evaluation shows that COOP improves the throughput performance for 200 random workloads by 7.67% on a quad-core CMP with a 4MB SLLC, all at the cost of only 17.74KB storage overhead which is comparable to TA-DRRIP (16KB) but lower than SHiP (25.75KB), while outperforming both of them (compared to TA-DRRIP's 4.53% and SHiP's 6.00% throughput improvements).

6.2 Research Motivations

Although the entire SLLC capacity can be accessed by all cores, allowing free use, without constraints, does not lead to efficient utilization of SLLC resources. Therefore, various strategies have been proposed to make the best use of the SLLC capacity. But



(a) Class I: omnetpp (b) Class I: sphinx3 (c) Class I: mcf

(d) Class II: astar (e) Class II: facerec (f) Class II: swim

(g) Class III: milc (h) Class IV: crafty

Figure 6.1: HPKIs (Hits Per 1K Instructions) of LRU, BIP, NRU and BNRU as a Function of the SLLC Capacity for the SPEC Benchmarks

since practical SLLC management is our main focus in the chapter, we will analyze the strengths and weaknesses of the prior-art practical schemes and provide quantitative evidence in support of our conclusions.

6.2.1 Theoretical Shared LLC Management Proposals

We note that most previous SLLC management proposals [12, 14, 13, 52, 15] are based on the assumption of $\log A$-bit RRPVs. For instance, given the set associativity A, the TADIP replacement policy adopts $\log A$ bits for each line's RRPV field to indicate its

current position in the LRU stack; and the RRPVs of the *most-recently-used* and the *least-recently-used* blocks are 0 and $A-1$ respectively. But since this overhead is prohibitive for the SLLCs that have large set associativity from the industry's point of view [57], those proposals may arguably only be used for theoretical research. In general, the theoretical proposals can be categorized into either alternative replacement policies for locality management (e.g., TADIP [14], SDBP [52] and NUcache [15]) or capacity partitioning schemes for utility optimization (e.g., UCP [12] and PIPP [13]). The detailed background about these schemes is introduced in Chapter 2.

6.2.2 Practical Shared LLC Management Schemes

The LRU approximations, such as the *not-recently-used* (NRU) replacement policy, are practically adopted in commodity processors because its RRPV filed requires just a single bit. Chapter 2 provides detailed information of how NRU works. According to extensive experimental statistics [16], NRU achieves a desirable 99.52% performance approximation to LRU. But since LRU is not performance-effective for CMPs, the NRU replacement policy that closely approximates LRU is not performance-effective for the CMP SLLC management either.

Recently, Jaleel *et al.* [16] have proposed a high-performance practical replacement policy called RRIP (an acronym for *Re-Reference Interval Prediction*). With 2 bits in the RRPV field, a block can have any of the three different categories of re-reference intervals: *near* (RRPV=0 or 1), *long* (RRPV=2) and *distant* (RRPV=3). RRIP always predicts a long re-reference interval for incoming blocks in an effort to prevent cache pollution due to a subset of incoming blocks being dead-on-fill. Additionally, the bimodal variant of RRIP (called BRRIP) can prevent thrashing by predicting a distant (or a long) re-reference interval for an incoming block with a high (or a complementarily low) probability. TA-

DRRIP is a thread-aware extension of RRIP to CMPs with SLLCs by coordinating either RRIP or BRRIP for individual threads under set-dueling and feedback control. Rooted in the same 2-bit RRPV substrate, SHiP, proposed in the most recent work [17], assigns either a distant or a long re-reference interval to an incoming block, depending on whether or not it is predicted to be dead-on-fill. Specifically, SHiP leverages a history table and sample sets to dynamically learn which memory instructions (identified by their PC signatures) tend to insert dead-on-fill blocks, and predicts a distant (or a long) re-reference interval for new blocks if they are inserted by those PCs (or otherwise).

6.2.3 Our New Perspective and Its Supporting Experimental Evidence

If we apply the same categorization in Section 6.2.1 to TA-DRRIP and SHiP, they are both classified as alternative replacement policies but for practical use in that they aim to optimize locality for SLLCs. While the alternative replacement policies excel in locality management, they are likely unable to coordinate the best capacity provisioning among all co-scheduled threads for utility optimization. This is due to their lack of the *utility monitor* [12], a critical component in judicious capacity partitioning, which estimate how many SLLC hits each thread would deliver with various capacity allocated. Therefore, one of our research motivation lies in the question of whether or not locality optimization alone can provide high enough performance for practical SLLC capacity management. The answer to this question, to be shortly backed with workload characterization and performance comparison, is *no*, suggesting that locality and utility co-optimization is indispensable to the best utilization of SLLC resources. Further, if locality and utility co-optimization is out of necessity, another key question is whether or not the minimal-overhead 1-bit RRPV substrate is sufficient for such a purpose. The experiments detailed in the following provide an affirmative answer to this question.

6.2.3.1 Workload Characterization

In Chapter 2, we introduce that LRU and BIP are the two basic optional replacement policies used in the *thread-aware dynamic insertion policy* (TADIP) [14] which functions for locality optimization. Given that NRU is shown to closely approximate LRU, we are motivated to propose a new practical replacement policy, called the *bimodal NRU* (BNRU), which approximates BIP by filling 1 (or 0) into the RRPV of an incoming block with a high (or complementarily low) probability.

Figure 6.1 illustrates the SLLC performance in terms of *hits per thousand instructions* (HPKIs), for 8 of the benchmarks in our study, as a function of assigned cache capacity, managed by LRU, BIP, NRU and BNRU respectively (see Section 6.3 and Section 6.4 for more details of experimental setups). The x-axis shows the SLLC capacity measured in the number of ways (given that the number of sets and line size are fixed), while the y-axis represents *HPKIs*. The dotted roofline in each figure indicates the total number of SLLC *accesses per 1k instructions* (independent of the SLLC capacity). The 8 benchmarks are divided into four classes according to their locality and utility characteristics. Here, with fixed 2048 sets and 64B lines assumed, we can measure the capacity in terms of the associativity. Looking at the performance aspects of the 4 policies in Figure 6.1, we can make the following observations: (1) the NRU and the LRU hit curves overlap each other nearly completely for all of the 8 figures, indicating that NRU approximates LRU almost perfectly regardless of the SLLC capacity and associativity configurations; (2) the BNRU and the BIP hit curves also match each other very well, except for the benchmark *facerec*, in a variety of SLLC configurations; (3) BNRU is as capable as BIP for thrashing prevention, as evidenced in Figure 6.1 (a)-(c) where the BNRU and the BIP hit curves are higher than the NRU and LRU hit curves for the three benchmarks *omnetpp*, *sphinx3* and *mcf* within certain ranges of SLLC capacity configurations. In addition, *the three*

observations also hold for other benchmarks in our study (see Table 6.2).

From the perspective of benchmark characteristics, the 8 benchmarks can be divided into four classes depending on their locality and utility features. The first two classes represent the cases where the performance can be improved with extra capacity allocated, but they differ in their NRU vs. BNRU utility. The last two classes saturate in performance after a minimal allocation of capacity, but with very different hit rates. In the first class, as indicated in Figure 6.1 (a)-(c), benchmarks *omnetpp*, *sphinx3* and *mcf* all have inferior locality because their NRU curves are significantly below the BNRU curves within certain capacity ranges (e.g., from associativity 2 to 20 for *mcf*). If any of them runs on a CMP with a mix of co-scheduled threads, an alternative replacement policy such as BNRU can potentially improve the SLLC hit performance over NRU. These are the cases where locality optimization can come into more prominent play for SLLC management. For example, with the replacement policy simply altered from NRU to BNRU for the same allocated capacity of 8 cache ways, the SLLC hit performance of *mcf* can be improved by 2.5x ($\approx \frac{42.4-12.2}{12.2}$).

In contrast, the workloads in the second class, represented by applications *astar*, *facerec* and *swim* (illustrated in Figure 6.1 (d)-(f)), show good locality since their NRU curves are never below the BNRU curves. However, they can still be set apart from each other with respect to their utility. For instance, when assigned 16 ways, *astar* has a higher utility than *facerec* in that *astar* can yield 22.53 HPKI (corresponding to a hit rate of 87.3%) while *facerec* can deliver only 6.2 HPKI (with a hit ratio of 40.0%). If a CMP workload consists of applications all from this category, much less room is available for locality improvement that alternative replacement policies are good at, while utility optimization is still likely to make a difference in performance by favoring threads with higher utility in SLLC capacity partitioning (e.g., preferring *astar* to *facerec*).

Figure 6.1 (g) and (h) illustrate the third and the fourth classes whose applications

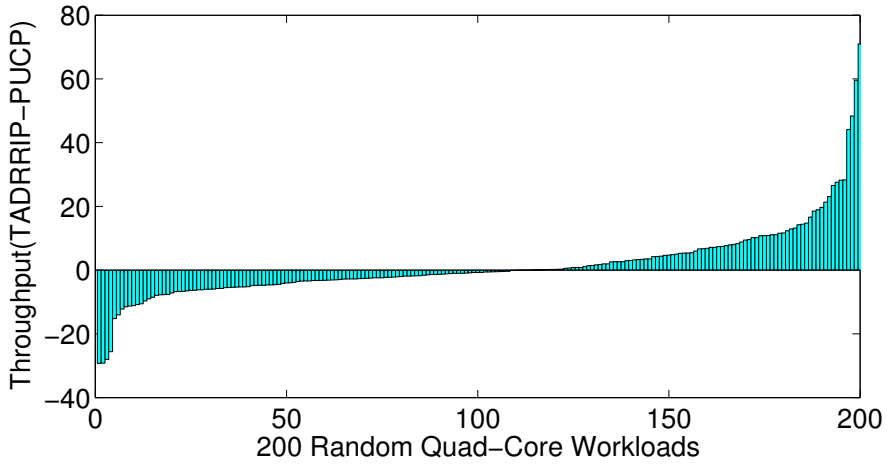


Figure 6.2: Difference in Normalized Throughput between TA-DRRIP and PUCP for Individual Quad-core Workloads

require very few SLLC resources. In particular, *milc* is a streaming application due to its high miss rate regardless of the amount of allocated SLLC capacity, while *crafty* is CPU-bound and can yield a very high hit rate given a small amount of allocated SLLC capacity.

6.2.3.2 Performance Comparison

The workload characterization experiments above indicate that (*i*) NRU and its bimodal variant BNRU are competent for favoring good locality and preventing thrashing respectively, and (*ii*) locality optimization alone cannot work consistently well for heterogeneous CMP workloads consisting of threads with various locality and utility features. To quantitatively demonstrate the limitation of practical alternative replacement policies that are oriented towards locality optimization only, we compose a simple *practical utility-based cache partitioning* (PUCP) scheme and compare it against the locality-oriented TA-DRRIP on 200 random quad-core workloads (see Section 6.4 for experimental details). In essence, PUCP makes cache-way partitioning decisions for co-scheduled threads by relying on the per-core *LRU utility monitors* (the same as in [12]) and then leverages the NRU

replacement policy (instead of LRU in [12]) to manage each thread's allocated ways. More related details will be presented in Section 6.3. The final performance comparison result is that TA-DRRIP and PUCP improve the throughput performance over LRU by 4.53% and 3.56% on average respectively, which indicates TA-DRRIP is better than PUCP overall. However, if we take a closer look at Figure 6.2, which illustrates the difference in throughput normalized to that of LRU between TA-DRRIP and PUCP for individual workloads (sorted in ascending order), we can clearly see that there are 112 out of the 200 workloads for which TA-DRRIP underperforms PUCP. Since PUCP does nothing beyond utility optimization with its LRU-based utility monitors and the 1-bit NRU substrate, the detailed view of performance difference in Figure 6.2 verifies our speculation on the limitation of practical alternative replacement policies with locality management alone. But utility optimization alone as is provided by PUCP is not sufficient either, since TA-DRRIP outperforms PUCP for the remaining 88 workloads, which also contributes to TA-DRRIP's better overall performance in spite of its performance disadvantage for the 112 workloads. In summary, we can infer from this motivational experiment that neither locality nor utility optimizations alone can consistently perform well under a variety of workloads for practical SLLC capacity management due to their different working principles and optimization objectives.

6.3 The COOP Architecture

Our practical scheme, called COOP (an acronym for *locality & utility CO-OPtimization*) is designed to achieve three specific goals: (*i*) to base the SLLC capacity management on the 1-bit RRPV substrate; (*ii*) to be aware of locality and utility features of individual threads by profiling their NRU and BNRU hit curves; and (*iii*) to decide on the SLLC optimal management policy by conducting interactive locality and utility co-optimization for all

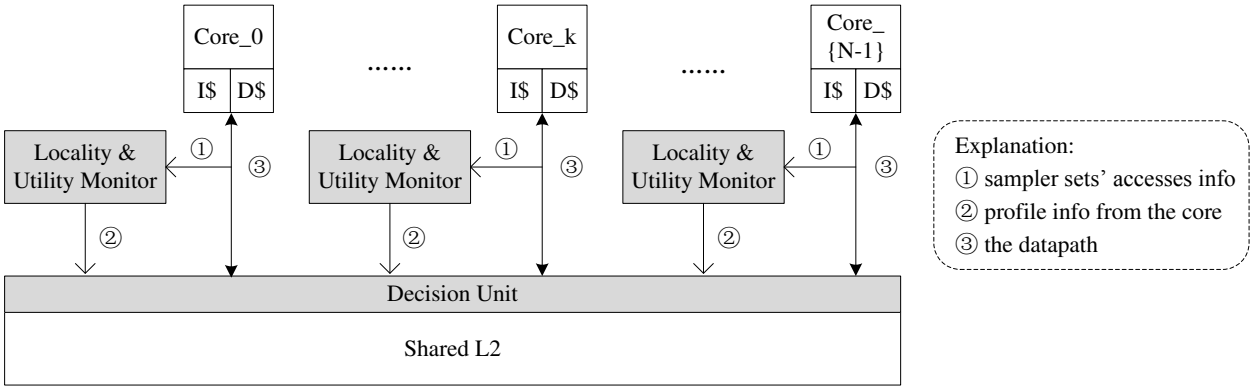


Figure 6.3: COOP Architecture

co-scheduled threads.

6.3.1 The Overall Architecture

Figure 6.3 depicts an architectural view of COOP, in which the grey boxes are the major extra hardware logic required by COOP on top of the SLLC with 1-bit RRPVs. In the SLLC, every cache line has a single bit in its RRPV field. Associated with each core, there is a *locality & utility monitor* that dynamically profiles both NRU and BNRU hit curves from the core's SLLC reference sequence. In particular, the NRU and the BNRU hit curves are captured by an LRU profiler and a BNRU profiler respectively, both of which are based on *auxiliary tag directories* (ATD) and the *set sampling* technique [7]. On every time interval boundary, the profilers feed the locality and utility information about sampler sets to the *decision unit*. Based on the information, the decision unit makes and enforces the capacity management decisions of cache-way partitions and replacement policies for all co-scheduled threads during the next time period.

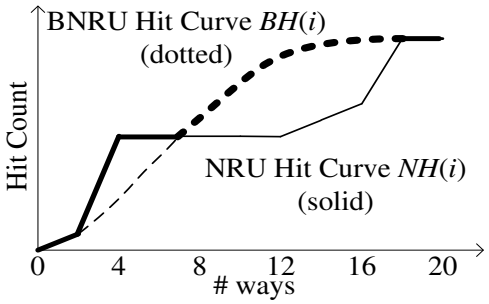


Figure 6.4: Deriving a Composite Hit Curve (Bold, Consisting of the Higher Segments of NRU/Solid and BNRU/Dotted Hit Curves)

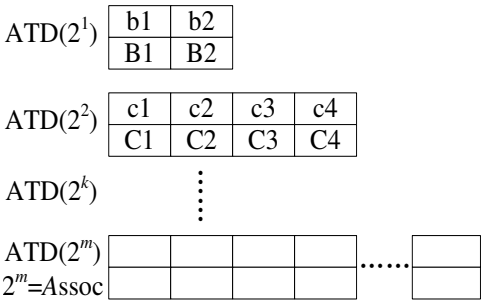


Figure 6.5: Approximately Profiling a BNRU Hit Curve

6.3.2 The Locality & Utility Monitor

The *locality & utility monitor* counts the SLLC hits that a thread would contribute if it were running alone, while the amount of space assigned to it and the replacement policy (NRU vs. BNRU) adopted to manage its allocated space are both varied. By doing so, the monitor attempts to capture the runtime interplay between locality and utility optimizations in the SLLC management. Assuming that an SLLC has an associativity of 64, for example, the monitor counts the number of hits a thread would contribute if it were allocated 1-, 2-, . . . , or 64-way SLLC space, being managed by NRU and BNRU respectively. Consequently, the monitor is able to deduce both NRU and BNRU hit curves as functions of cache ways, as illustrated in Figure 6.1 and generalized in Figure 6.4. The two curves can jointly convey two critical pieces of information:

- Which replacement policy should be adopted under a given capacity quota for the thread. As depicted in Figure 6.4, if the thread can get 4 cache ways, it should apply the NRU replacement policy to manage the given amount of space, since the NRU hit curve (solid) is above the BNRU curve (dotted) when the way count equals 4; but if the assigned way-count is 12, the thread should alter the policy to BRNU that can

help it obtain far more hits. Therefore, with the two curves, COOP can implicitly derive a *composite hit curve* (bold) which consists of the higher segments of the NRU or BNRU curves, as illustrated in Figure 6.4.

- What the preferred utility is under the best replacement policy. For instance, if the hit counts of the derived composite hit curve at the way counts of 10 and 12 are assumed to be 100 and 110 respectively, we know that the utility of 10 ways is better than that of 12 ways because $\frac{100}{10} > \frac{110}{12}$. In this way, COOP fully exploits the interactions between locality and utility dimensions.

As described below, we apply two different profiling mechanisms to deduce the NRU and the BNRU hit curves of a thread respectively because of their distinct features.

Profiling an NRU Hit Curve: Since NRU and LRU hit curves have been experimentally shown to be almost identical, as exemplified in Figure 6.1, we approximate an NRU hit curve with its corresponding LRU hit curve, which can be easily obtained by the well-established *LRU utility monitor* (LRU UMON) [12]. In an LRU UMON, an *auxiliary tag directory* (ATD) with an associativity of A and a size-A array of *stack-hit counters* are adopted to implement the *Mattson's LRU stack algorithm* [42], where A is also the SLLC's set associativity. Here, an ATD structure, with each of its entries containing only a hashed tag, a valid bit and a $\log A$-bit RRPV field mimics the LRU stack of a small group of sampler SLLC sets, as if the monitored thread were exclusively occupying the whole space of these sampler sets. Upon every hit on the ATD, it reports the LRU-stack position where the hit takes place so that the corresponding stack-hit counter $h(i)$ can be incremented by one. As a result of the LRU stack property, the values of the NRU and the LRU hit curves at way count i, denoted $NH(i)$ and $LH(i)$ respectively, can be expressed by Equation 6.1, where $h(k)$ is the hit counter at LRU-stack position k and $1 \leq k \leq i \leq A$.

$$NH(i) \approx LH(i) = \sum_{1 \leq k \leq i} h(k) \tag{6.1}$$

Profiling a BNRU Hit Curve: The profiling of a BNRU hit curve, on the other hand, is more challenging because BNRU violates the stack property with its non-deterministic 0-valued or 1-valued RRPV assignment for incoming blocks. Thus, the simple stack algorithm cannot be applied to deducing a BNRU hit curve. To resolve this issue, we first propose an exact but complex approach and follow it with an approximate but practical solution.

The *exact approach* is also based on set sampling, and uses a number A of ATD structures representing the A different associativities from 1 to A. Therefore, in the exact approach, we use a group of A ATD structures, {ATD(1), ATD(2), ATD(3) ..., ATD($A-1$) and ATD(A)}, to mimic BNRU's operations on the sampler SLLC sets with an associativity ranging from 1 to A respectively. Although this approach provides an exact measure of the BNRU curve, it requires a significant number A of ATD structures, rendering the implementation prohibitively expensive when A is large.

The *practical solution* is based on four key observations derived from an analysis of the BNRU hit curves for the benchmarks in our study (exemplified in Figure 6.1): (*i*) the BNRU hit curve is monotonically non-decreasing with respect to the assigned way count; (*ii*) the BNRU hit curve is a concave function, which means that the curve's gradient is always non-increasing as the way count increases. *The intuition behind concave BNRU curves is that, with the high-probability 1-valued RRPV assignment for an incoming block, the block will most likely be victimized due to its 1-valued RRPV upon a subsequent cache miss in the same set, thus preventing other blocks from getting evicted* (namely, *stationary*); (*iii*) the BNRU hit curve has a long flat tail as the way count approaches a high value; and (*iv*) it is provable that BNRU and NRU hit curves have the same value at way count 1 ($BH(1) = NH(1)$),

since neither of them lets an incoming line bypass the cache. Therefore, it is sufficient to monitor the BNRU hit values at a small number of discrete logarithmic way-count points by using a dedicated ATD for each of these points, and then apply the *curve fitting* technique to deducing the entire BNRU hit curve. Specifically, as illustrated in Figure 6.5, we employ $m = \log A$ ATD structures $\{\text{ATD}(2^1), \text{ATD}(2^2), \ldots, \text{ATD}(2^m)\}$ to capture the BNRU hit counts $\{BH(2^1), BH(2^2), \ldots, BH(2^m)\}$ in a small number of way-count cases $\{2^1, 2^2, \ldots, 2^m\}$. We carry out curve fitting based on the m discrete BNRU hit values by linearly interpolating between the two monitored BNRU curve counts $(2^k, BH(2^k))$ and $(2^{k+1}, BH(2^{k+1}))$ for $1 \leq k \leq m-1$. Then, the $BH(i)$ value can be calculated iteratively by Equation 6.2. Figure 6.6 shows an example of applying our *logarithmic-distance monitoring* & *curve fitting* approach with up to 64 ways for benchmark *mcf*.

$$BH(i) = \begin{cases} BH(2^k), & \text{where } i = 2^k \text{ and } BH(2^k) \text{ is monitored by } ATD(2^k) \\ BH(i+1) - \Delta, & \text{where } \Delta = \frac{BH(2^{k+1}) - BH(2^k)}{2^k} \text{ and } 1 \leq 2^k < i < 2^{k+1} \leq 2^m = A \end{cases} \tag{6.2}$$

The specific design choice of monitoring at logarithmic way-count points stems from our empirical observations mentioned above, suggesting a denser number of monitoring points to more accurately profile the high-gradient portion of the BNRU hit curve when A is small, which is also a property of a logarithmic/geometric series. As a result, the practical solution needs only $m = \log A$, instead of A, BNRU-managed ATD structures at the associativities of 2, 4, $\ldots$, $\frac{A}{2}$ and A respectively, as well as m BNRU-hit counters. It is worth remarking that the storage overhead (measured in the total number of ATD ways) required by the practical BNRU profiling is $2 + 4 + 8 + \cdots + \frac{A}{2} + A = \frac{2\times(A-1)}{2-1} = 2 \times (A-1) < 2 \times A$, *which is less than twice the storage overhead required by a single A-way ATD structure for the NRU/LRU hit curve profiling and makes our solution very practical in hardware implementation.* It must be noted that, (*i*) the RRPV field of each BNRU-managed

ATD entry has only a single bit, and (*ii*) upon an access to one sampler SLLC set, the LRU-managed ATD and the m BNRU-managed ATDs are operated concurrently to profile both NRU and BNRU hit curves.

6.3.3 The Decision Unit

With the locality and utility characteristics of co-scheduled threads profiled during each time interval, the decision unit will periodically determine the optimal space partitions and replacement policies for individual threads. Since the space partitioning logic of COOP is also utility-based and targeted at maximizing the overall performance, we adopt but with modification the framework of the *lookahead utility-based cache partitioning algorithm* [12] in the decision unit. The original algorithm evaluates every potential partitioning decision and provisions cache ways to a thread that currently has the highest utility of these ways based on its LRU hit curve. We modify the algorithm to determine the best utility-based partitioning of the cache ways according to the *composite hit curve*, which is composed of the higher segments of the NRU and the BNRU hit curves.

In the decision unit, there is an m-bit *partition quota counter*, an A-bit *global replacement mask* [95, 43] and a single *locality management* bit associated with each core, where $m = \log A$ and A is the associativity. On each time interval boundary, the SLLC's space partitioning result for $Core_i$ is kept in the *partition quota counter*, denoted as Q_i, where $0 \leq i \leq N-1$. Assuming that A is greater than N, COOP also guarantees that at least one way is provisioned to every core. The *global replacement mask* is used to specify which cache ways are currently allocated to the corresponding core. For example, if a core is allocated with two cache ways, say, way 0 and way 1, only the first and the second bits on its *global replacement mask* are set to one. A core can access any lines in an SLLC set but is only allowed to replace a line in its own allocated ways. The *locality management* bit, LM_i,

Table 6.1: Major Configuration Parameters

Core	four cores, Alpha ISA, in-order, IPC=1 except for memory accesses, 128/128 I/D TLBs, 44-bit physical addresses
L1	2-way, 32KB, 1-cycle delay, 16 MSHRs, write back & 4 write buffer entries for L1D, LRU-managed
L2	16-way, 4MB, 6/8-cycle tag/data store delay, totally 1024 MSHRs, write back & totally 256 write buffer entries, mesh NoC topology (2×2) with 1 cycle delay per hop
Mem	300-cycle off-chip memory access delay
Schemes	#sample_sets $= \frac{1}{32} \times$ #SLLC_sets for all sampling-based schemes; BNRU's low probability = $\frac{1}{32}$; PUCP/COOP's LRU-managed ATD: 16-bit hit counters, 10-bit hashed tags, 4-bit RRPVs, 1 valid bit; TA-DRRIP: 10-bit saturating counters; SHiP: 16K 3-bit saturating counters in the history table, 14-bit PC signatures + 1 reuse bit in each sampled cache line; COOP's BNRU-managed ATDs: 16-bit hit counters, 10-bit hashed tags, 1-bit RRPVs, 1-bit valid bits.

is utilized to indicate whether the NRU (LM_i=0) or BNRU (LM_i=1) policy is adopted for the core to manage its allocated SLLC cache ways. LM_i can be determined by examining the difference between the NRU and BNRU curves at the way count k that is also the value of Q_i: the bit is set 0 if $NH(k) \geq BH(k)$ or 1 otherwise.

6.4 Experiments & Evaluation

In this section, we first briefly describe our simulation-based experimental methodology and then present and analyze the evaluation results.

6.4.1 Evaluation Methodology

Simulation Setup: We simulate all schemes using the cycle-accurate M5 full system simulator [90] with the key configuration parameters listed in Table 6.1. We model a quad-core CMP with two levels of on-chip caches. The L1 instruction and data caches adopt a coupled tag & data store organization. For the shared L2 cache, we model decoupled tag

Table 6.2: Selected Benchmarks & Classification

Class	Descriptor	Benchmarks
I	Poor Locality	*galgel*, *libquantum*, *mcf*, *omnetpp*, *sphinx3*, *xalancbmk*
II	Good Utility	*astar*, *ammp*, *bzip2*, *calculix*, *facerec*, *GemsFDTD*, *swim*, *twolf*, *vpr*
III	Streaming	*lucas*, *milc*
IV	CPU-Bound	*crafty*, *eon*, *fma3d*

and data stores for each L2 slice and also account for the NoC latency when calculating the L2 access time. The SLLC capacity management schemes in comparison include LRU (baseline), NRU, PUCP, TA-DRRIP, SHiP and our proposed COOP scheme. PUCP and COOP make management decisions periodically every 5M cycles [1], and their profiler hit counters are reset upon each periodic decision boundary. We have not found any overflow problems with these 16-bit hit counters in our experiments.

Performance Metrics: We adopt two standard metrics, *throughput* and *fair speedup*, to quantify the CMP performance. Specifically, *throughput* measures the utilization of a system, while *fair speedup* balances both performance and fairness. Let IPC_i be the *instructions per cycle* performance of the ith thread when it is co-scheduled with other threads and $SingleIPC_i$ be the IPC of the same thread when it executes in isolation. Then, for a system where N threads execute concurrently, the formulas for the two metrics are shown in Equation 6.3 and Equation 6.4 respectively.

$$\text{throughput} = \sum_{i=1,2,\cdots,N} IPC_i \tag{6.3}$$

$$\text{fair speedup} = \frac{N}{\sum_{i=1,2,\cdots,N} SingleIPC_i / IPC_i} \tag{6.4}$$

Workload Construction: As listed in Table 6.2, we select 20 benchmarks from the SPEC CPU 2000 and 2006 benchmark suites and categorize them into four classes by investi-

[1] The period is experimentally tuned.

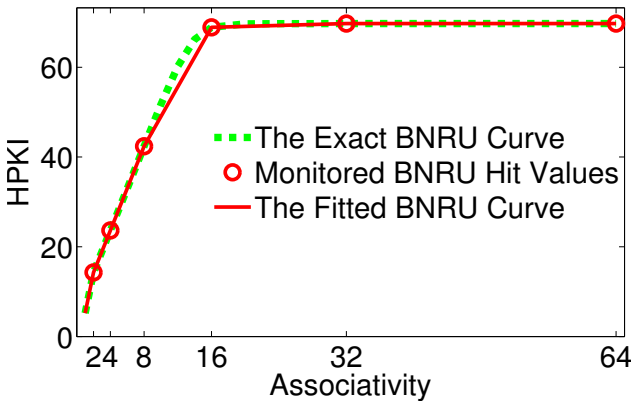


Figure 6.6: Applying the Logarithmic-Distance Monitoring & Curve Fitting Approach to Profiling the BNRU Hit Curve of Benchmark *mcf* (by Approximating the Exact Curve).

gating their locality and utility features via experiments similar to Section 6.2.3.1. Class I is a collection of benchmarks that exhibit poor locality and can benefit from judicious replacement policies. The benchmarks in Class II have excellent utility and need dedicated SLLC space partitions. Class III is a group of streaming applications that require little SLLC capacity and need to be prevented from polluting the SLLC. Finally, Class IV benchmarks are CPU-bound with small working sets in the SLLC. From the four classes of benchmarks, 200 random quad-core workloads are generated by randomly selecting 200 4-benchmark combinations out of the 20 individual benchmarks.

Simulation Control: In the experiments, all threads under a given workload are executed starting from a checkpoint that has already had the first 10 billion instructions bypassed. They are cache-warmed with 1 billion instructions and then simulated in detail until all threads finish another 1 billion instructions. Performance statistics are reported for a thread when it reaches the completion of the latter 1 billion instructions. If one thread finishes the 1 billion instructions before others, it continues to run so as to still compete for the SLLC capacity, but its extra instructions are not taken into account in the final performance report. This is in conformation with the standard practice in CMP cache research [14, 12, 13, 52, 15, 16, 17].

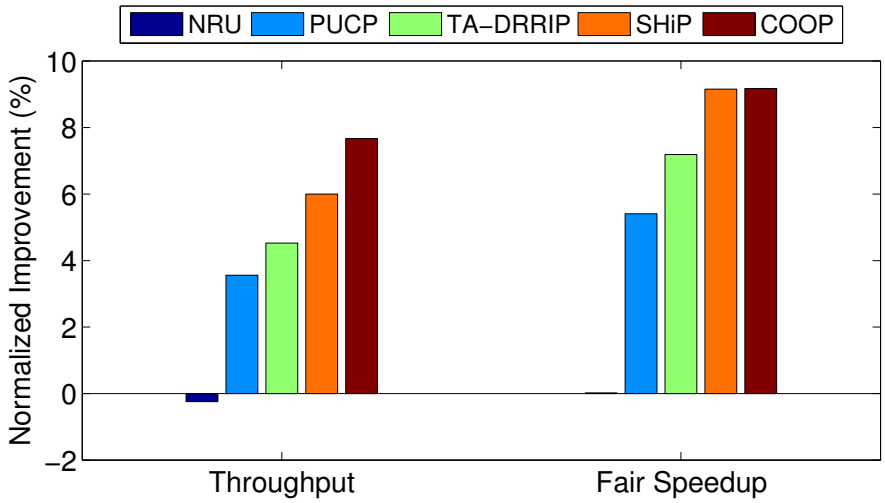
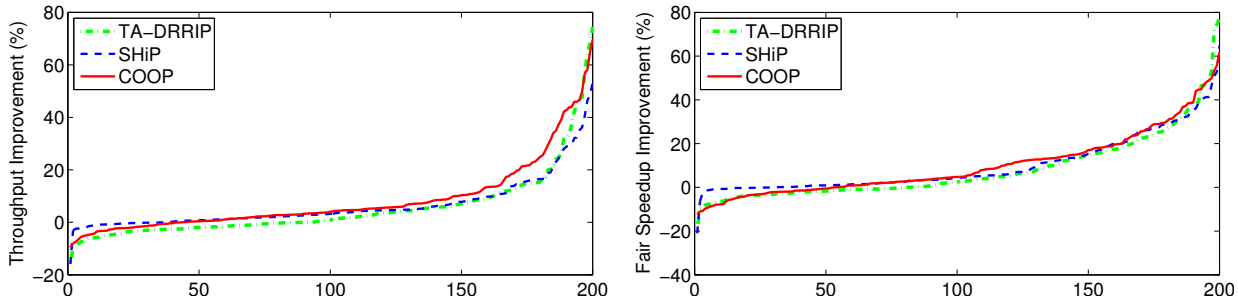


Figure 6.7: Throughput and Fair Speedup Improvement for the 4-Core Configuration

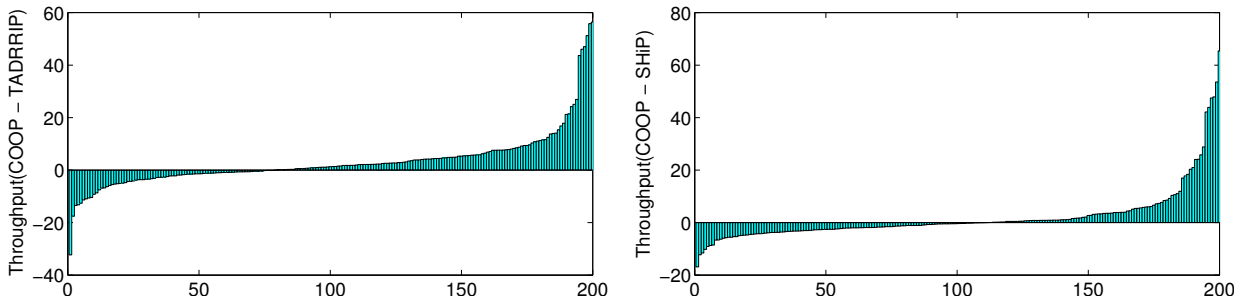
6.4.2 Performance Comparison

Figure 6.7 shows the geometric mean of throughput performance for NRU, PUCP, TA-DRRIP, SHiP and COOP, normalized to the baseline (LRU) on the simulated quad-core configuration. Averaged over all of the 200 random workloads, COOP provides a throughput improvement of 7.67%, which is noticeably higher than the improvements achieved by the practical replacement policies (TA-DRRIP: 4.53%, SHiP: 6.00%) and the simple utility-oriented scheme (PUCP: 3.56%). The NRU replacement policy degrades the throughput performance by 0.24%, which indicates that both NRU and LRU are clearly inadequate for CMPs with SLLCs. If we look closer at the details of the throughput performance in Figure 6.8(a), we can find that the worst case performance of COOP is -9.57% and its best improvement is up to 69.67%, while the (worst, best) performance margins for TA-DRRIP and SHiP are (-15.47%, 74.49%) and (-15.90%, 53.21%) respectively. Figure 6.8(a) also shows that the throughput performance curve of COOP is almost always above that of TA-DRRIP except for a very small fraction of the 200 workloads (i.e., a very small x-axis range at the end). COOP's curve is also above that of SHiP except for a small



(a) 4-Core Throughput (b) 4-Core Fair Speedup

Figure 6.8: Detailed Views of the Quad-Core Throughput and Fair Speedup Improvement for TA-DRRIP, SHiP and COOP (with Values Sorted in Ascending Order)



(a) Comparison between COOP and TA-DRRIP (b) Comparison between COOP and SHiP

Figure 6.9: Difference in Normalized Throughput between COOP and TA-DRRIP/SHiP for Individual Quad-Core Workloads

fraction of the 200 workloads (i.e., a small x-axis range at the beginning), which means that COOP offers consistent and robust throughput performance that is generally better than the two existing practical replacement policies.

Figure 6.9(a) and Figure 6.9(b) show the head-to-head comparison between COOP and the two prior-art schemes for individual workloads. COOP outperforms the two practical replacement policies for the majority of the workloads, and in many cases significantly. However, it does not do so in all cases. We speculate that it is because COOP must strike a balance between both locality and utility optimizations. COOP may not optimize locality fully when a workload heavily or exclusively favors locality optimization but does show stronger performance improvement over both of the prior-arts

(indicated by the portions of bars above the 0/equal line in Figure 6.9(a) and Figure 6.9(b)). If we specifically compare Figure 6.9(a) against Figure 6.2 in Section 6.2.3.2, we can make the following two observations: on the one hand, COOP significantly boosts the performance beyond capacity partitioning provided by PUCP by additionally adapting to the better replacement policy between NRU and BNRU; on the other hand, it relies on its partitioning module besides the alternative replacement mechanism, which is based on 1-bit RRPVs, to surpass a TA-DRRIP-like replacement policy which only conducts locality optimization even with 2-bit RRPVs. Therefore, COOP is arguably capable of bridging the performance gap between the locality and the utility optimization schemes via its co-optmization strategies.

In terms of the fair speed improvement, COOP improves over LRU by 9.17%, which matches the 9.16% performance improvement of SHiP. Other schemes, NRU, PUCP and TA-DRRIP, improve the fair speedup metric by 0.02%, 5.41%, 7.19% respectively. Figure 6.8(b) depicts the detailed result of fair speedups for individual workloads. While the curves of COOP and SHiP are both above that of TA-DRRIP in most cases, the difference between the curves of the two bests seems to be minor.

However, we notice in both Figure 6.8(a) and Figure 6.8(b) that SHiP's curves are barely below the zero horizontal line, while COOP's curves have a small portion below. This may suggest that SHiP is slightly more robust in the sense that it seldom underperforms LRU. But this robustness may come at the cost of its weakened ability to exploit the highest performance when available, which is evident in Figure 6.8(a) because the curves of both COOP and TA-DRRIP are above that of SHiP at the end of the x-axis range. TA-DRRIP seems to be able to exploit the highest performance, but it underperforms LRU in quite a few cases. In this sense, COOP also shows its ability to strike a reasonable balance between the exploitation of the highest possible performance and keeping performance robust in the worst cases.

Table 6.3: Overhead & Throughput

	LRU	NRU	TA-DRRIP	SHiP	COOP
RRPVs	32KB	8KB	16KB	16KB	8KB
Monitors	0	0	5B	9.75KB	9.74KB
Total Overhead	32KB	8KB	16KB	25.75KB	17.74KB
Throughput	1	0.9976	1.0453	1.0600	1.0767

6.4.3 Overhead Analysis

Table 6.3 gives a detailed comparison of hardware overhead for the various schemes in this chapter (see their specific configurations in Table 6.1). For all of the schemes, the total number of their RRPV bits as well as the extra monitoring logic is counted into their total hardware costs. In particular, for COOP, since it requires an LRU profiler and an BNRU profiler for per-core locality & utility monitoring, the ATD structures will dominate the hardware overhead in its design. But the hashing function that has shown a provably low collision rate in [92] and the *logarithmic-distance monitoring* technique in the BNRU profiler contribute to much less cost in the per-core locality & utility monitor. Most importantly, COOP is based on the minimal-overhead 1-bit RRPV substrate, which can greatly help reduce the overhead compared to the costs of 2-bit RRPVs in both TA-DRRIP and SHiP in spite of COOP's extra monitoring logic. As a result, COOP's 3.88KB LRU profiler, 5.86KB BNRU profiler and 8KB RRPVs contribute to its 17.74KB total overhead. In contrast, the recently-proposed practical scheme SHiP, requires more hardware resources due to the large prediction history table (6KB) and also additional PC signature stores (3.75KB), in addition to the 16KB 2-bit RRPVs. Overall, COOP can provide a better performance improvement than both TA-DRRIP and SHiP but at a comparable cost of storage overhead to TA-DRRIP and a lower cost than SHiP.

6.5 Summary

The research community has introduced a substantial volume of theoretical proposals to optimize either locality or utility in the SLLC capacity management. But their high storage overhead for re-reference interval prediction discourages the industry from adopting them in practical CMP designs. Although there are already two practical replacement policies TA-DRRIP and SHiP that significantly reduce overhead by relying on a 2-bit RRPV substrate, their performance is suboptimal due to their single-pronged approach of locality optimization. Different from the existing studies, our proposed COOP design (*i*) combines the strengths of both the minimal-overhead 1-bit RRPV substrate and the profilers with good theoretical traits and, importantly, (*ii*) carries out locality & utility co-optimization in capacity management. By employing lightweight monitors that profile both NRU (approximated by LRU) and BNRU hit curves (curve-fitted with *logarithmic-distance monitoring*), our proposed design can exploit the co-optimized locality and utility of concurrent threads and thus effectively manage the SLLC capacity for CMP workloads with heterogeneous resource requirements. Our execution-driven simulation shows that the proposed scheme improves the throughput performance over the baseline LRU for 200 random workloads by 7.67% on a quad-core CMP with a 4MB SLLC, outperforming both TA-DRRIP (4.53%) and SHiP (6.00%), all at the cost of only 17.74KB storage overhead that is slightly higher than TA-DRRIP (16KB) but lower than SHiP (25.75KB).

Chapter 7

Directions for Future Research

So far, our solutions to exploiting the spatiotemporal interplay in cache management have been discussed based on the following assumptions: (*i*) in cache partitioning, there are always more cache ways than processing cores; (*ii*) the CMP workloads consist of a mix of single-threaded applications; (*iii*) there is only one application/thread running on a core; (*iv*) each thread is always pinned to its assigned core. These assumptions do simplify our effort to address the LLC management problems and develop corresponding solutions at the architecture level. But they are subject to limitations in light of the CMP core-count scaling, the increasing popularity of thread-level parallelism as well as the interactions among architecture, runtime systems and operating systems. In what follows, we will discuss how the scope of our dissertation research can be further broadened from both architecture's and systems' perspectives.

7.1 From Architecture's Perspectives

At the architecture level, since the CMP core count is expected to double every 18 months [1], it is an important issue how to make our proposed spatiotemporal solutions

scalable with the number of cores. In addition, the aggressive core count growth is now propelling the exploration of massive thread-level parallelism through new programming frameworks [96] or automatic parallelization techniques [97]. Therefore, the implications of data sharing in parallel applications [98] will also need to be taken into account in cache management.

7.1.1 Scaling with the Core Count

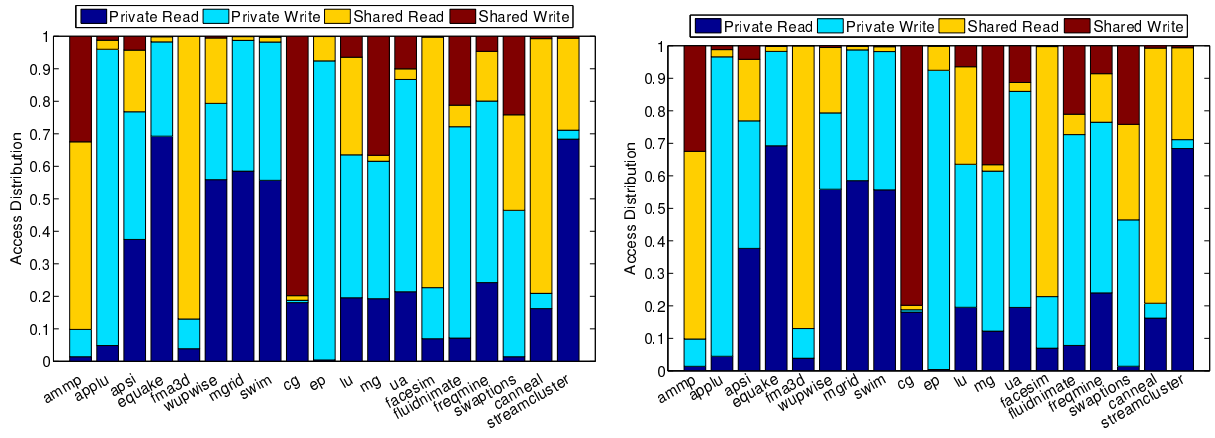
We note that our implicit assumption of more SLLC ways than CMP cores in way partitioning might soon be rendered invalid. This is because, while the core count keeps growing exponentially, the number of SLLC ways may just increase moderately because of the high energy consumption and latency associated with a large number of cache ways in the SLLC [99]. As a result, there will eventually be more cores than SLLC ways on a CMP, inevitably invalidating the aforementioned assumption. In the following, we describe two potential solutions to the problem.

First, we can decouple the logical cache associativity from physical cache ways, which allows for much higher logical associativity than the number of physical ways (e.g., a 1024-associative cache with just 64 physical ways) based on *cuckoo hashing*, as proposed in the recent research on ZCache [11]. Thus, it may be possible to partition the high logical associativity rather than a limited number of physical ways among many cores in our future exploration of more design space. We also note that a partitioning scheme for ZCache has been demonstrated recently in [100]. Therefore, it will be quite straightforward to leverage the new logical-associavitity partitioning technique to make our spatiotemporal solutions scalable with the ever-increasing core count.

Second, as advocated by some researchers, future many-core CMPs may adopt a hybrid LLC organization, combining the benefits of large aggregate capacity of the

shared LLC and low latency of the private LLC. For instance, as proposed in [101], a small number (e.g., 16) of physically adjacent cores can be clustered (either statically or dynamically) with the capacity of their LLC slices exclusively shared among each other, while globally the CMP can be partitioned into a group of such private clusters (e.g., a 1024-core may be configured into 64 16-core clusters). In this context, the spatiotemporal solutions can directly be applied to the shared LLC within each cluster because the assumption behind the way-partitioning approach is still valid. We plan to evaluate our spatiotemporal solutions with this type of hybrid LLC organization in our future work.

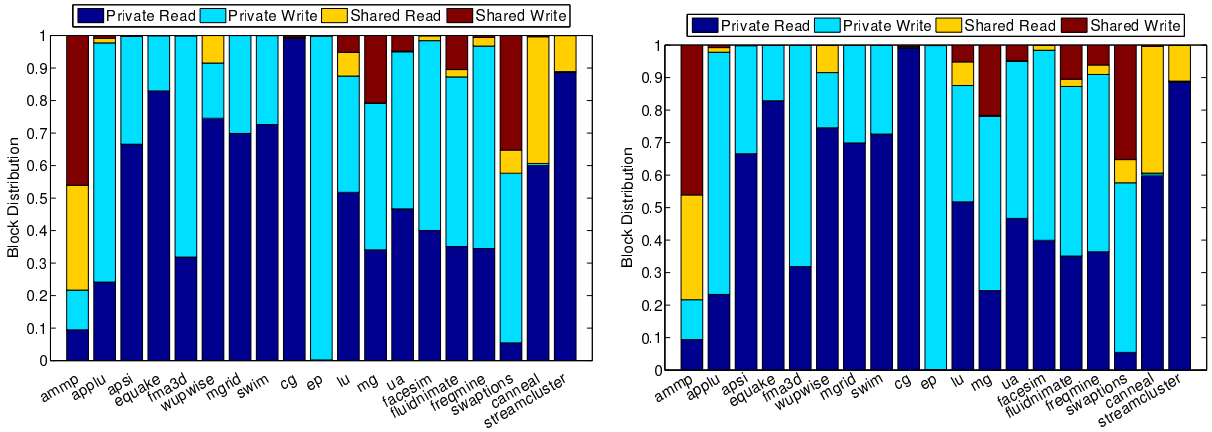
7.1.2 Implications of Multithreaded Workloads



(a) **16MB** (b) **32MB**

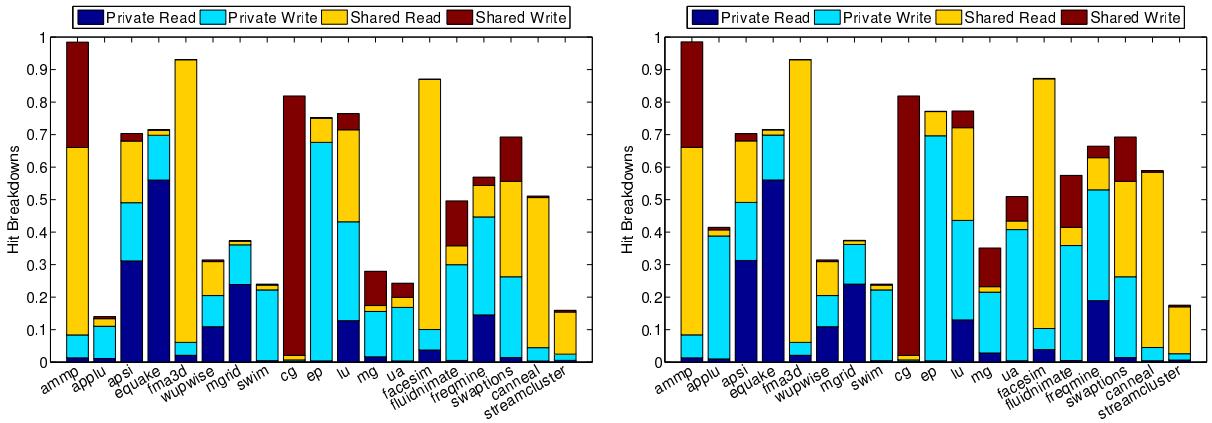
Figure 7.1: Distributions of Accesses to the SLLCs

Our current proposed solutions are evaluated using multiprogrammed workloads, for which the heterogeneous capacity requirements of co-scheduled applications are the most important concern. However, with the rising importance of thread-level parallelism, strategies of cache management may need to be adjusted accordingly. In the following, we illustrate the potential influence of data sharing in parallel applications on cache



(a) **16MB** (b) **32MB**

Figure 7.2: Distributions of Blocks in the SLLCs



(a) **16MB** (b) **32MB**

Figure 7.3: Hit Ratio Breakdowns in the SLLCs

management and also provide our suggestions on how to adapt cache management to multithreaded workloads.

To study the implications of parallel applications on cache management, we investigate the performance differences between shared and private data under read and write operations on SLLCs. Specifically, we characterize 19 representative parallel applications (from SPEC OMP 2001, NPB 3.3 and Parsec 2.1 benchmark suites) with respect to their distributions of accesses and blocks as well as the breakdowns of hit ratios in SLLCs. We

simulate a 16-core CMP with 16-way 16MB and 32-way 32MB SLLCs managed by LRU in two different experiment settings respectively. The L1D/I caches are set to 2-way 32KB. 16 threads are spawned in each application without being pinned to cores. After the bypass period, 200M instructions are executed for each benchmark to collect the SLLC cache performance statistics.

From Figure 7.1, we find that the distributions of accesses to the 16MB and the 32MB SLLCs are almost identical for each benchmark. But the applications can still be categorized into two groups, of which one has dominant accesses to shard data (e.g., *ammp*, *fma3d* and *cg*) while the other features much more references to private data (e.g., *applu*, *apsi* and *equake*). However, if we look at the distributions of shared and private cache blocks in the SLLCs, as indicated in Figure 7.2, we find that all of the applications have far higher volumes of private blocks except *ammp*. We speculate from both Figure 7.1 and Figure 7.2 that, for certain benchmarks, their shared data can contribute more to their SLLC hit rates than private data because a large number of accesses are targeted at a small volume of (shared) blocks. Figure 7.3 does confirm the correctness of this speculation, as evidenced by the high hit rates of shared data in such benchmarks as *ammp*, *fma3d* and *cg*.

Since shared data can provide much higher hit ratios, it may need special care so as to stay longer in the SLLC. However, awareness of shared data is absent in our current spatiotemporal solutions. Especially for the applications belonging to the data parallel programming model, their internal threads tend to be homogeneous. So, it may be difficult to utilize the capacity management solutions oriented towards heterogeneous threads to boost their performance. In our future exploration of more design space of our spatiotemporal solutions, we may need to enhance the current designs to recognize private and shared data and assign higher priority to the shared one in temporal and spatial capacity management. Moreover, as private data has a much larger volume, we

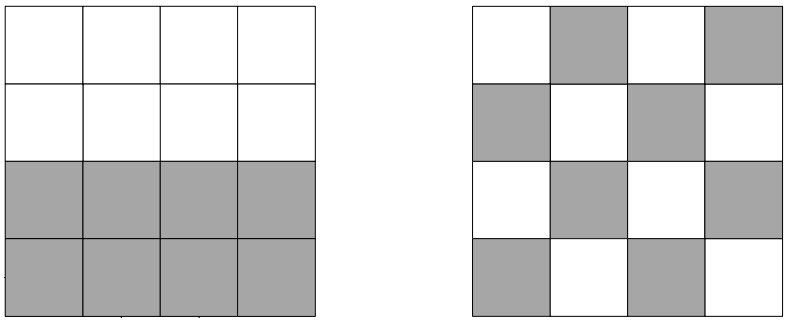
can essentially promote its proximity to the requesting core in the SLLC by such means as page coloring [78].

7.2 From Systems' Perspectives

At the system level, there are at least two implications accompanied with the new trend of aggressive thread-level parallelism. On the one hand, emerging many-core applications typically have very large working sets [102], pressing the slowly-scaled hardware resources such as the on-chip LLC capacity. On the other hand, a number of concurrent threads need to be well synchronized and scheduled to make good use of the computation power of processing cores. As a result, more challenging issues will pop up in a broader context that covers not only LLC capacity management but also scheduling and synchronization.

7.2.1 Interactions Between LLC Capacity Management and Scheduling

The scheduler is an indispensable OS component that supports multiprogramming in computer systems. Conventionally, an OS scheduler is designed to enable all processes/threads to have fair access to processor resources, enforce priority ordering for the co-scheduled processes/threads, guarantee load balance and minimize the idleness of processing cores. For CMPs, there are two major types of thread scheduling, *space-scheduling* and *time-scheduling* [103]. Specifically, space scheduling is about determining which core a running thread is mapped to, while time scheduling is referred to as time-multiplexing multiple threads on a core. By enabling a scheduler to be aware of applications' LLC capacity requirements, the scheduler can play an active role in promoting the efficacy of



(a) A Space-Scheduling Plan with the Threads from the Same Group Clustered Together

(b) A Space-Scheduling Plan with the Threads from Different Groups Interleaved with Each Other

Figure 7.4: A Comparison between Different Space-Scheduling Plans

LLC capacity management by spatially or temporally scheduling concurrent threads close together (sharing many LLC resources) or far apart (sharing minimal LLC resources).

7.2.1.1 The Impact of Space Scheduling on LLC Management

Here, we are formulating a space-scheduling problem that is related to enhancing our proposed spatiotemporal LLC capacity management at the system level. For an $N \times N$ mesh-based CMP architecture that has been integrated with our spatiotemporal capacity management framework, assume that there are N^2 threads that need to be space-scheduled on the CMP. Hence, there can be $N^2!$ possible thread-to-core mapping permutations in all. We claim that different space-scheduling plans can have distinct impacts on facilitating inter-core cooperative caching, thus leading to very different overall performance. In this regard, we illustrate an intuitive example as follows.

If $N = 4$, as is demonstrated in Figure 7.4, suppose that two groups of 8 threads are co-scheduled on the 4×4 CMP tile and need to be mapped to the 16 cores. Group 1 consists of streaming-like threads (represented in white in Figure 7.4) with sufficient underutilized LLC capacity, while Group 2 is formed by the threads that can benefit

from more LLC capacity (shown in gray). A simple case is that the two groups can be two parallel applications respectively, each with 8 threads spawned. Figure 7.4 shows two distinct space-scheduling plans. Plan (a) clusters the threads from the same group with similar capacity requirements, while plan (b) fully interleaves the threads from the two groups. It is obvious that plan (b) will produce better performance of inter-core cooperative caching than plan (a), because a thread in plan (b) can always find its complementary peer in one of its immediate neighbors for inter-core cooperative caching, whereas at least half of the threads in plan (a) cannot.

The significance of the problem lies in that an intelligent space-scheduling plan can enable our proposed spatiotemporal capacity management framework to perform inter-core cooperative caching more effectively, which will in turn lead to better overall performance of many-core CMPs. But because there are up to $N^2!$ potential space-scheduling plans, one research challenge is how to come up with a heuristic method to efficiently choose a space-scheduling plan with satisfactory performance out of the $N^2!$ possibilities. Then, based on the heuristic method (if it were ready for use), we plan to develop a dynamic space-scheduling algorithm to promote the effectiveness of our proposed spatiotemporal LLC management solutions by leveraging the hints about all cores' LLC capacity requirements and utilization. Furthermore, to make the scheduling algorithm more efficient, we will need to take into account data-sharing patterns of multithreaded workloads as well.

7.2.1.2 The Impact of Time Scheduling on LLC Management

When there are more concurrent threads than cores, the time-scheduling functionality of an OS scheduler will come into play. Since the uniprocessor era, time scheduling has been playing a vital role in promoting the utilization of processing cores. For example, if a running thread gets blocked upon a long-latency I/O event, the OS scheduler will

replace the thread with another one from the processing core's ready queue by means of context switching, so that the core does not sit idle waiting for the previous thread's I/O response. Here, we are demonstrating that time scheduling may also be helpful to improving the efficacy of our proposed spatiotemporal LLC management framework, which seems like a promising research problem for further investigation.

Let us consider such an scenario: given a 16-core CMP, suppose that there are two streaming parallel applications as well as another two multithreaded applications with good utility of LLC capacity. We further assume that each of the four applications spawns 8 threads with quite few internal synchronization contentions. When all of the applications are simultaneously executed on the CMP, the OS scheduler needs to time schedule them so that while 16 threads are running the other 16 are waiting in the processing cores' ready queues. But among all possible time schedules, some may not help make good use of the LLC capacity at all: e.g., (*i*) if the two streaming applications are co-scheduled together, they will have a large portion of the LLC capacity underutilized; (*ii*) if the two applications with good LLC utility become active at the same time, they may severely contend with each other for the LLC capacity. A much better time-scheduling strategy is to mingle two applications with complementary capacity needs at a time, so that the streaming one can always contribute its LLC capacity to its peer application by taking advantage of our spatiotemporal solutions. From the comparison above, it is obvious that time scheduling can have a significant impact on the LLC capacity management.

Although the examples above illustrate the promise of applying time scheduling to LLC management, it is still a question whether or how this potential functionality can be made compatible with other existing purposes of time scheduling such as to guarantee fairness among co-scheduled threads. We plan to leave the issue for future research. Even more challengingly, time scheduling and space scheduling should be made to intelligently interact with each other so as to maximize their supports for our spatiotemporal LLC

capacity management framework.

7.2.2 Taking into Account the Interplay Among LLC Capacity Management, Thread Synchronization and OS Scheduling

There are two fundamental synchronization mechanisms in operating systems, *spinning* and *blocking* [104]. In the spinning-based synchronization, only the thread that is holding a lock can make forward progress in its critical section, while any other contender thread needs to wait by spinning until it acquires the lock. The spinning-based synchronization allows waiting threads to respond to lock hand-offs very quickly, but it typically wastes a significant amount of processor time in making a number of threads spin idly. In contrast, in the blocking-based synchronization, waiting threads are evacuated from cores via context switching. The blocking-based synchronization frees processing cores from sitting idle for long, but it can bring high overhead to the critical path of computation due to frequent context switching.

Both spinning and blocking have close interactions with OS scheduling because a contemporary OS scheduler typically needs to enforce time slicing, priority ordering and load balancing for co-scheduled threads. For instance, in the spinning-based synchronization, *priority inversion*, which indicates that a high-priority task is forced to wait an indefinite period for the completion of a low-priority task, usually occurs when the thread count is large. This is because an ordinary OS scheduler cannot distinguish between the thread with a lock and those that are spin-waiting when adjusting threads' priorities. Then, the lock-holder thread can be given low priority and thus preempted from running, forcing other high-priority but spinning threads to wait even longer. Even if all threads are assigned with the same priority, a *lock convoy* may also occur when the lock-holder thread gets preempted due to timeout. In this case, a large number of contender threads

have to relinquish their time quanta due to failures of acquiring the lock, leading to repeated preemption overhead, poor utilization of scheduling quanta and thus dramatic deterioration in overall performance.

To alleviate the aforementioned problems with spinning and blocking, several hybrid schemes have been introduced in an OS, such as OpenSolaris' adaptive mutex and Linux's futex. Both of the schemes use the *spin-then-block* contention management policy. In the spin-then-block policy, based on the assumption that the time spent in spinning is less than that of being taken to block, threads spin-wait if the lock-holder thread is now running on another processing core or get blocked if none of the current running threads has the lock. But the spin-then-block policy is also quite limited for the systems with high loads [104, 105]. This is because the OS adjusts threads' priorities fast in this condition, and context switching can still unconsciously preempt a lock-holder thread and force other contender threads to wait for a large amount of time.

There are two root causes of the priority inversion and the lock convoy problems. On the one hand, a lock-holder thread is not visible to the operating system, and thus by no means can the OS scheduler avoid preempting it prematurely. On the other hand, before the lock-holder thread finishes its critical section and releases the lock, there is no reason to assign time quanta to other threads that contend for the same lock. Feng *et al.* [106] devise a subtle mechanism, called the *contention-aware scheduler* (CAScheduler), to address the two underlying causes by taking advantage of the runtime information about lock usages in Java Virtual Machines. Two data structures, the *contention vector* and the *lock count*, play a key role in exposing each thread's lock usage information to the OS scheduler. The *contention vector* is essentially a sliding window that keeps track of how many times different locks have recently been acquired by a thread. The *lock count* tells how many locks are currently held by a thread. The CAScheduler clusters the threads with similar lock usages according to their *contention vectors*, time schedules those from

the same cluster (with many lock contentions) and space-schedules the threads from different clusters (with few contentions). CAScheduler also favors the threads that have larger *lock counts* by assigning them with longer time quanta or higher priority to work around the regular priority adjustment and the time slicing routines.

Although CAScheduler has been demonstrated to outperform the regular Linux scheduling routines, we speculate that higher performance improvements could have been obtained if smart LLC capacity management had been incorporated. For instance, since the critical thread that holds a number of locks is made visible in CAScheduler, we can also favor its LLC capacity requests besides giving it longer time quanta. This is because, for certain memory-intensive threads, it is the long memory latency instead of the processor quanta that acts as a performance bottleneck. So, better LLC capacity allocation strategies are likely to help this kind of parallel applications. In addition, CAScheduler excludes the consideration of LLC capacity contentions in making scheduling decisions. But LLC capacity management, thread synchronization and thread scheduling in fact closely interact with each other on the fly, as analyzed above and also in Section 7.2.1. Therefore, it is necessary for us to take into account their interplay to optimize the performance holistically for future many-core CMPs.

Chapter 8

Conclusion

Efficient utilization of the last-level cache capacity is vital to mitigating the memory wall problem facing chip multiprocessors. During our study of CMPs' LLC capacity management, we realize the need for a holistic perspective on the different natures of contemporary LLC management mechanisms. This thesis makes the key point that replacement policies and balancing/partitioning schemes fundamentally fall into temporal and spatial dimensions respectively. It also highlights our experimental observation that the interactions between the two dimensions present unique but unexplored performance improvement opportunities for CMPs with private or shared LLCs. Based on this fundamental idea, the dissertation research identifies four specific LLC capacity management problems related to the spatiotemporal interplay and proposes cost-effective solutions accordingly, which is recapitulated as follows.

- In Chapter 3, we derive an accurate metric by mathematical modeling to measure the capacity demands of individual LLC sets. By applying the measurement to workload characterization, we observe that much underutilized capacity of private LLCs cannot be exploited unless the inter-core cooperative caching works at the fine-grained cache set level. Therefore, we propose the "SNUG" LLC design that

leverages the non-uniformity of set-level capacity demands and thus improves over the prior-art approaches which are functional at the coarse-grained application level.

- In Chapter 4, we reveal that set-balancing schemes and replacement policies represent two different goal orientations, spatial versus temporal, towards optimizing the utilization of intra-core private LLC capacity. However, neither of them can excel in a wide spectrum of workloads exhibiting diverse spatial or temporal capacity requirements. We devise the "STEM" LLC management scheme that interactively redistributes space among LLC sets and adjusts the lifetime of a set's blocks. The spatiotemporal management strategy enables "STEM" to deliver more robust performance than other approaches.

- In Chapter 5, we observe that prior research on the shared LLC management has focused on exploiting either locality or utility of co-scheduled applications and thus provides distinct performance advantages. To cohesively get the combined benefits, we propose the "CLU" solution to co-optimize locality and utility for shared LLCs. The key idea behind CLU is to derive and join the hit curves of both LRU and BIP replacement policies for all possible number of ways allocated to each application. Based on the combined hit curves, CLU interactively decides both partition quotas and replacement algorithms for all of the applications. With the co-optimization capability, CLU outperforms the prior-art proposals that are oriented towards improving either locality or utility alone.

- In Chapter 6, we study reducing the hardware cost involved in the locality & utility co-optimization for shared LLCs. We demonstrate that the two recent replacement policies rooted in 2-bit RRPVs are oriented towards improving locality only and thus miss the performance opportunities uniquely available to utility optimization. To practically co-optimize both locality and utility, we find that NRU and bimodal NRU,

which entail single-bit RRPVs to re-reference prediction, can provide performance closely approximating those of LRU and BIP respectively regardless of the number of cache ways. Therefore, we devise the "COOP" SLLC management scheme that bases the co-optimization on single-bit RRPVs with additional lightweight locality & utility monitors. COOP outperforms the two practical replacement polices at almost the smallest storage cost.

Besides presenting the aforementioned problems and solutions, the dissertation also opens up opportunities for more in-depth research in the future. At the architecture level, we realize that our implicit assumption of more ways than cores in cache partitioning may soon be invalidated by the rapid core-count growth. Thus, the spatial modules in our proposed schemes will need an overhaul in response to the new trend. Furthermore, the ever-increasing core count will propel more and more applications to become multi-threaded to exploit thread-level parallelism. In parallel applications, data sharing among the threads within the same address space can have a significant impact on the LLC capacity management, which presents both opportunities and challenges to improving our spatiotemporal solutions that are currently targeted at independent threads. At the system level, both time and space scheduling are shown to be capable of taking a more active role in boosting the efficacy of CMP LLC capacity management. Especially for multihreaded applications, we will need to make scheduling, synchronization and LLC capacity management cooperate more efficiently and intelligently with each other in effort to optimize overall performance. These topics are recommended for further research investigation.

Bibliography

[1] International Technology Roadmap for Semiconductors 2011. http://www.itrs.net/Links/2011ITRS/Home2011.htm. 1, 7.1

[2] The World's First 100-Core Processor. http://www.tilera.com/about_tilera/press-releases/tilera-announces-worlds-first-100-core-processor. 1, 5.1

[3] William A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995. 1

[4] Bruce L. Jacob. *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It*. Morgan & Claypool Publishers, 2009. 1

[5] Sally A. McKee. Reflections on the Memory Wall. In *Proceedings of the 1st Conference on Computing Frontiers*, pages 162–167, 2004. 1

[6] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 371–382, 2009. 1

[7] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel Emer. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of*

the 34th International Symposium on Computer Architecture, pages 381–391, 2007. 1, 1.1, 2.4.1, 2.4.2, 4.1, 4.2.2, 4.3.1, 4.3.3, 4.4.1, 4.4.2, 5.1, 5.2.1, 5.3.1, 5.3.2.2, 5.3.3, 6.3.1

[8] Mainak Chaudhuri. Pseudo-LIFO: the Foundation of a New Family of Replacement Policies for Last-Level Caches. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 401–412, 2009. 1, 1.1, 2.4.2, 4.1

[9] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. The V-Way Cache: Demand Based Associativity via Global Replacement. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 544–555, 2005. 1, 1.1, 2.4.2, 4.1, 4.2.1, 4.2.3.1

[10] Dyer Rolán, Basilio B. Fraguela, and Ramón Doallo. Adaptive Line Placement with the Set Balancing Cache. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 529–540, 2009. 1, 1.1, 2.4.2, 4.1, 4.2.1, 4.2.3.1, 4.2.3.2, 4.3.5, 4.3.6, 4.3.7, 4.4.1

[11] Daniel Sanchez and Christos Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 187–198, 2010. 1, 2.4.2, 7.1.1

[12] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, 2006. 1, 1.1, 3.2.1.1, 5.1, 5.2.1, 5.3.2.1, 5.3.2.2, 5.3.3, 5.4.3, 6.1, 6.2.1, 6.2.3, 6.2.3.2, 6.3.2, 6.3.3, 6.4.1

[13] Yuejian Xie and Gabriel H. Loh. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In *Proceedings of the 36th International Symposium on*

Computer Architecture, pages 174–183, 2009. 1, 1.1, 2.4.4, 5.1, 5.2.1, 5.3.2.2, 5.3.3, 5.4.3, 6.1, 6.2.1, 6.4.1

[14] Aamer Jaleel, William Hasenplaugh, Moinuddin K. Qureshi, Julien Sebot, Simon C. Steely Jr., and Joel S. Emer. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 208–219, 2008. 1, 1.1, 2.4.4, 3.2, 5.1, 5.2.1, 5.3.3, 5.4.3, 6.1, 6.2.1, 6.2.3.1, 6.4.1

[15] R Manikantan, Kaushik Rajan, and R Govindarajan. NUcache: An Efficient Multicore Cache Organization Based on Next-Use Distance. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, pages 243–254, 2011. 1, 1.1, 2.4.4, 5.1, 5.1, 5.4.3, 6.1, 6.2.1, 6.4.1

[16] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proceedings of the 37th International Symposium on Computer Architecture*, pages 60–71, 2010. 1, 1.1, 2.2, 2.4.1, 2.4.4, 5.1, 5.4.1, 5.4.3, 6.1, 6.2.2, 6.4.1

[17] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely Jr., and Joel Emer. SHiP: Signature-Based Hit Predictor for High Performance Caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–12, 2011. 1, 1.1, 2.4.4, 6.1, 6.2.2, 6.4.1

[18] Stefan G. Berg. Cache Prefetching. Technical Report UW-CSE 02-02-04, Department of Computer Science & Engineering, University of Washington, 2002. 1, 2.6

[19] Wi-fen Lin, Steven K. Reinhardt, and Doug Burger. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 301–312, 2001. 1, 2.6

[20] Alaa R. Alameldeen and David A. Wood. Interactions between Compression and Prefetching in Chip Multiprocessors. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, pages 228–239, 2007. 1

[21] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback-Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, pages 63–74, 2007. 1, 2.6

[22] Xiaotong Zhuang and Hsien-Hsin S. Lee. Reducing Cache Pollution via Dynamic Data Prefetch Filtering. *IEEE Transactions on Computers*, 56(1):18–31, 2007. 1, 2.6

[23] Eiman Ebrahimi, Onur Mutlu, Chang J. Lee, and Yale N. Patt. Coordinated Control of Multiple Prefetchers in Multi-Core Systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 316–326, 2009. 1, 2.6

[24] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely Jr., and Joel Emer. PACMan: Prefetch-Aware Cache Management for High Performance Caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 442–453, 2011. 1, 2.6

[25] James Tuck, Luis Ceze, and Josep Torrellas. Scalable Cache Miss Handling for High Memory-Level Parallelism. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 409–422, 2006. 1

[26] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A Case for MLP-Aware Cache Replacement. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 167–178, 2006. 1

[27] Feihui Li, Chrysostomos Nicopoulos, Thomas D. Richardson, Yuan Xie, Narayanan Vijaykrishnan, and Mahmut T. Kandemir. Design and Management of 3D Chip Multiprocessors Using Network-in-Memory. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 130–141, 2006. 1

[28] Gabriel H. Loh. 3D-Stacked Memory Architectures for Multi-Core Processors. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 453–464, 2008. 1

[29] Gabriel H. Loh. Extending the Effectiveness of 3D-Stacked DRAM Caches with an Adaptive Multi-Queue Policy. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 201–212, 2009. 1

[30] Lan Young, Edris Mohammed, Jason Liao, Alexandra Kern, Samuel Palermo, Bruce Block, Miriam Reshotko, and Peter Chang. Optical I/O Technology for Tera-Scale Computing. In *Proceedings of the 2009 International Solid-State Circuits Conference*, pages 468–470, 2009. 1

[31] The Road to Exascale: Hardware and Software Challenges. http://sc09.supercomputing.org/?pg=panels.html. 1

[32] Jichuan Chang and Gurindar S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 264–276, 2006. 1.1, 2.4.3, 3.1, 3.4.1

[33] Moinuddin K. Qureshi. Adaptive Spill-Receive for Robust High-Performance Caching in CMPs. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, pages 45–54, 2009. 1.1, 2.4.3, 3.1, 3.2

[34] Dongyuan Zhan, Hong Jiang, and Sharad C. Seth. Exploiting Set-Level Non-Uniformity of Capacity Demand to Enhance CMP Cooperative Caching. In *Proceedings of the 24th Annual IEEE International Parallel and Distributed Processing Symposium*, pages 222–233, 2010. 1.2, 4.2.1

[35] Dongyuan Zhan, Hong Jiang, and Sharad C. Seth. STEM: Spatiotemporal Management of Capacity for Intra-Core Last Level Caches. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 163–174, 2010. 1.2

[36] Dongyuan Zhan, Hong Jiang, and Sharad C. Seth. CLU: Co-optimizing Locality and Utility in Thread-Aware Capacity Management for Shared Last Level Caches. *IEEE Transactions on Computers*, in Press. 1.2

[37] Dongyuan Zhan, Hong Jiang, and Sharad C. Seth. Locality & Utility Co-optimization for Practical Capacity Management of Shared Last Level Caches. In *Proceedings of the 26th International Conference on Supercomputing*, pages 279–290, 2012. 1.2

[38] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Higher Education, 2005. 2.1

[39] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5th Edition)*. Morgan Kaufmann, 2012. 2.1, 2.2, 2.2

[40] AMD A6-Series Microprocessor Family. `http://www.cpu-world.com/CPUs/K10/TYPE-A6-Series.html`. 2.1

[41] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966. 2.4.1, 4.2.1

[42] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM System Journal*, 9(2):78–117, 1970. 2.4.1, 3.2.1.1, 5.3.2, 5.3.2.1, 6.3.2

[43] Kamil Kedzierski, Miquel Moreto, Francisco J. Cazorla, and Mateo Valero. Adapting Cache Partitioning Algorithms to Pseudo-LRU Replacement Policies. In *Proceedings of the 24th International Parallel and Distributed Processing Symposium*, pages 1–12, 2010. 2.4.1, 2.4.4, 6.3.3

[44] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache Bursts: a New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 222–233, 2008. 2.4.2, 4.1

[45] André Seznec. A Case for Two-Way Skewed-Associative Caches. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 169–178, 1993. 2.4.2

[46] Mazen Kharbutli, Keith Irwin, Yan Solihin, and Jaejin Lee. Using Prime Numbers for Cache Indexing to Eliminate Conflict Misses. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pages 288–299, 2004. 2.4.2, 4.1, 4.2.3.1

[47] Michael Ferdman, Pejman Lotfi-Kamran, Ken Balet, and Babak Falsafi. Cuckoo Directory: A Scalable Directory for Many-Core Systems. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, pages 169–180, 2011. 2.4.2

[48] Enric Herrero, José González, and Ramon Canal. Distributed Cooperative Caching. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 134–143, 2008. 2.4.3, 3.1

[49] Noel Eisley, Li-Shiuan Peh, and Li Shang. Leveraging On-Chip Networks for Data Cache Migration in Chip Multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 197–207, 2008. 2.4.3, 3.1

[50] Enric Herrero, José González, and Ramon Canal. Elastic Cooperative Caching: an Autonomous Dynamically Adaptive Memory Hierarchy for Chip Multiprocessors. In *Proceedings of the 37th International Symposium on Computer Architecture*, pages 419–428, 2010. 2.4.3

[51] Hyunjin Lee, Sangyeun Cho, and Bruce R. Childers. CloudCache: Expanding and Shrinking Private Caches. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, pages 219–230, 2011. 2.4.3

[52] Samira M. Khan, Yingying Tian, and Daniel A. Jimenez. Sampling Dead Block Prediction for Last-Level Caches. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 175–186, 2010. 2.4.4, 5.1, 6.1, 6.2.1, 6.4.1

[53] 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. 2.4.4

[54] Kyle J. Nesbit, James Laudon, and James E. Smith. Virtual Private Caches. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 57–68, 2007. 2.4.4

[55] Xing Zhou, Wenguang Chen, and Weimin Zheng. Cache Sharing Management for Performance Fairness in Chip Multiprocessors. In *Proceedings of the 18th International*

Conference on Parallel Architectures and Compilation Techniques, pages 384–393, 2009. 2.4.4

[56] Jean-Loup Baer and Wen-Harm Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 73–80, 1988. 2.5

[57] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 151–162, 2010. 2.5, 6.1, 6.2.1

[58] Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. Bypass and Insertion Algorithms for Exclusive Last-Level Caches. In *Proceedings of the 38th International Symposium on Computer Architecture*, pages 81–92, 2011. 2.5

[59] Jaewoong Sim, Jaekyu Lee, Moinuddin K. Qureshi, and Hyesoon Kim. FLEXclusion: Balancing Cache Capacity and On-Chip Traffic via Flexible Exclusion. In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 1–12, 2012. 2.5

[60] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, 2002. 2.7

[61] James Balfour and William J. Dally. Design Tradeoffs for Tiled CMP On-Chip Networks. In *Proceedings of the 20th International Conference on Supercomputing*, pages 187–198, 2006. 2.7

[62] Michael Zhang and Krste Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 336–345, 2005. 2.7

[63] Bradford M. Beckmann, Michael R. Marty, and David A. Wood. ASR: Adaptive Selective Replication for CMP Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 443–454, 2006. 2.7

[64] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2011. 2.8

[65] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999. 2.8

[66] Nick Barrow-Williams, Christian Fensch, and Simon Moore. Proximity Coherence for Chip Multiprocessors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 123–134, 2010. 2.8

[67] Hongzhou Zhao, Arrvindh Shriraman, and Sandhya Dwarkadas. SPACE: Sharing Pattern-Based Directory Coherence for Multi-Core Scalability. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 135–146, 2010. 2.8

[68] George Kurian, Jason E. Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, Lionel C. Kimerling, and Anant Agarwal. ATAC: a 1000-Core Cache-Coherent Processor with On-Chip Optical Network. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 477–488, 2010. 2.8

[69] John H. Kelm, Matthew R. Johnson, Steven S. Lumettta, and Sanjay J. Patel. WAYPOINT: Scaling Coherence to Thousand-Core Architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 99–110, 2010. 2.8

[70] Blas A. Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José F. Duato. Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks. In *Proceedings of the 38th International Symposium on Computer Architecture*, pages 93–104, 2011. 2.8

[71] Daniel Sanchez and Christos Kozyrakis. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture*, pages 129–140, 2012. 2.8

[72] R. E. Kessler and Mark D. Hill. Page Placement Algorithms for Large Real-Indexed Caches. *ACM Transactions on Computer Systems*, 10(4):338–359, 1992. 2.9

[73] Sangyeun Cho and Lei Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *Proceedings of the 39th Annual IEEE/ACM Internationl Symposium on Microarchitecture*, pages 455–468, 2006. 2.9

[74] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture*, pages 367–378, 2008. 2.9

[75] Livio Soares, David Tam, and Michael Stumm. Reducing the Harmful Effects of Last-Level Cache Polluters with an OS-Level Software-Only Pollute Buffer. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–269, 2008. 2.9, 4.1

[76] Manu Awasthi, Kshitij Sudan, Rajeev Balasubramonian, and John Carter. Dynamic Hardware-Assisted Software-Controlled Page Placement to Manage Capacity Allocation and Sharing within Large Caches. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, pages 250 –261, 2009. 2.9

[77] Mainak Chaudhuri. PageNUCA: Selected Policies for Page-Grain Locality Management in Large Shared Chip-Multiprocessor Caches. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, pages 227 –238, 2009. 2.9

[78] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 184–195, 2009. 2.9, 7.1.2

[79] Xiaoning Ding, Kaibo Wang, and Xiaodong Zhang. SRM-Buffer: an OS Buffer Management Technique to Prevent Last Level Cache from Thrashing in Multicores. In *Proceedings of EuroSys*, pages 243–256, 2011. 2.9

[80] Gabriel H. Loh. 3D-Stacked Memory Architectures for Multi-Core Processors. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 453–464, 2008. 3.1

[81] Chang J. Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. Prefetch-Aware DRAM Controllers. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 200–209, 2008. 3.1

[82] Ian A. Young, Edris Mohammed, Jason T. S. Liao, Alexandra M. Kern, Samuel Palermo, Bruce A. Block, Miriam R. Reshotko, and Peter L. D. Chang. Optical

I/O Technology for Tera-Scale Computing. In *Proceedings of the 2009 International Solid-State Circuits Conference*, pages 468–470, 2009. 3.1

[83] SPEC CPU 2000 Benchmark Suite. http://www.spec.org/cpu2000/. 3.2.1.3

[84] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002. 3.2.1.3, 3.4.1

[85] SPEC CPU 2000 Memory Characterization. http://www.jaleels.org/ajaleel/workload/SPEC_CPU2000/. 3.2.1.4

[86] Kevin Skadron and Douglas W. Clark. Design Issues and Tradeoffs for Write Buffers. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, pages 144–155, 1997. 3.3

[87] PolyScalar Digital Signal Processor Simulator. http://www.cs.ucsb.edu/~franklin/PolyScalar/Home.htm. 3.4.1

[88] Tim Horel and Gary Lauterbach. UltraSPARC-III: Designing Third-Generation 64-Bit Performance. *IEEE Micro*, 19(3):73–85, 1999. 3.4.4

[89] Thomas Roberts Puzak. *Analysis of Cache Replacement Algorithms*. Dissertation, University of Massachusetts - Amherst, 1985. 4.3.2

[90] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006. 4.4.1, 5.4.1, 6.4.1

[91] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, 1999. 4.4.1

[92] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient Hardware Hashing Functions for High Performance Computers. *IEEE Transactions on Computers*, 46(12):1378–1381, 1997. 4.4, 5.4.4, 6.4.3

[93] The Single-Chip Cloud Computer Project in Intel. http://techresearch.intel.com/ProjectDetails.aspx?Id=1. 5.1

[94] Iñigo Goiri, Josep Oriol Fitó, Ferran Julià, Ramon Nou, Josep Lluis Berral, Jordi Guitart, and Jordi Torres. Multifaceted Resource Management for Dealing with Heterogeneous Workloads in Virtualized Data Centers. In *Proceedings of the 11th International Conference on Grid Computing*, pages 25–32, 2010. 5.1

[95] Derek Chiou, Srinivas Devadas, Larry Rudolph, Boon S. Ang, Derek Chiouy, Derek Chiouy, Larry Rudolphy, Larry Rudolphy, Srinivas Devadasy, Srinivas Devadasy, Boon S. Angz, and Boon S. Angz. Dynamic Cache Partitioning via Columnization. In *MIT CSAIL Computation Structures Group Memo 430*, pages 1–22, 1999. 6.3.3

[96] Apache Hadoop. http://hadoop.apache.org/. 7.1

[97] J. Gregory Steffan and Todd C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture*, pages 2–13, 1998. 7.1

[98] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Does Cache Sharing on Modern CMP Matter to the Performance of Contemporary Multithreaded Programs? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 203–212, 2010. 7.1

[99] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. Way-Predicting Set-Associative Cache for High Performance and Low Energy Consumption. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, pages 273–275, 1999. 7.1.1

[100] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proceedings of the 38th International Symposium on Computer Architecture*, pages 57–68, 2011. 7.1.1

[101] Shekhar Srikantaiah, Emre Kultursay, Tao Zhang, Mahmut Kandemir, Mary Jane Irwin, and Yuan Xie. Morphcache: A reconfigurable adaptive multi-level cache hierarchy. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, pages 231–242, 2011. 7.1.1

[102] Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin, and Ling Shao. Allocation Wall: a Limiting Factor of Java Applications on Emerging Multi-core Platforms. In *Proceeding of the 24th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 361–376, 2009. 7.2

[103] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors. *ACM Computing Surveys*, 45(1):1–31, 2013. 7.2.1

[104] F. Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C. Mowry. Decoupling Contention Management from Scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, 2010. 7.2.2

[105] Kishore K. Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. No More Backstabbing... A Faithful Scheduling Policy for Multithreaded Programs. In *Proceedings of the 20th*

International Conference on Parallel Architectures and Compilation Techniques, pages 1–12, 2011. 7.2.2

[106] Feng Xian, Witawas Srisa-an, and Hong Jiang. Contention-Aware Scheduler: Unlocking Execution Parallelism in Multithreaded Java Programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 163–180, 2008. 7.2.2