Cooperative Caching for Chip Multiprocessors

by

Jichuan Chang

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
University of Wisconsin-Madison
2007

Abstract

Chip multiprocessor (CMP) systems have made the on-chip caches a critical resource shared among co-scheduled threads. Limited off-chip bandwidth, increasing on-chip wire delay, destructive inter-thread interference, and diverse workload characteristics pose key design challenges. To address these challenge, we propose CMP cooperative caching (CC), a unified framework to efficiently organize and manage on-chip cache resources. By forming a globally managed, shared cache using cooperative private caches. CC can effectively support two important caching applications: (1) reduction of average memory access latency and (2) isolation of destructive inter-thread interference.

CC reduces the average memory access latency by balancing between cache latency and capacity optimizations. Based private caches, CC naturally exploits their access latency benefits. To improve the effective cache capacity, CC forms a "shared" cache using replication control and LRU-based global replacement policies. Via cooperation throttling, CC provides a spectrum of caching behaviors between the two extremes of private and shared caches, thus enabling dynamic adaptation to suit workload requirements. We show that CC can achieve a robust performance advantage over private and shared cache schemes across different processor, cache and memory configurations, and a wide selection of multithreaded and multiprogrammed workloads.

To isolate inter-thread caching interference, we add a time-sharing aspect on top of spatial cache partitioning. Our approach uses Multiple Time-sharing Partitions (MTP) to simultaneously improve throughput and fairness while maintaining QoS over the longer term. Each MTP partition unfairly improves at least one thread's throughput, and partitions favoring different threads are scheduled in a cooperative, time-sharing manner to either maintain fairness and QoS, or implement priority. We also integrate MTP with CC's LRU-based capacity sharing policy to combine their benefits. The integrated scheme—Cooperative Caching Partitioning (CCP)—divides the total execution epochs into those controlled by either MTP or the

baseline CC policy, respectively, according to the fraction of threads that can benefit from each of them. Our simulation results show that for a wide range of multiprogrammed workloads, CCP can improve throughput, fairness and QoS for workloads suffering from destructive interference, while achieving the performance benefit of the baseline CC policy for other workloads.

Acknowledgments

First of all, I thank my Savior and Lord Jesus Christ for His love and guidance. I could have never started this dissertation without God's inspiration, and never finished it without all the support and help He has arranged in my life.

I am deeply indebted to my advisor Professor Guri Sohi. Guri has been a great mentor in teaching me the ways to look at problems, conduct research, parse results and present ideas. He guided me through the difficult process of finding dissertation topic, and provided me with much freedom in seeking out challenging problems. His patience, calmness and compassion have been a great source of inspiration to me. My professional and personal life has greatly benefited from Guri's encouragement and support.

I'd like to thank the rest of my thesis committee — Professors Mark Hill, David Wood, Jim Smith and Karu Sankaralingam for their feedbacks. Besides their daily influence and suggestions to improve this dissertation, Mark and David kindled my interests in parallel computer architecture and introduced me to select Guri as my advisor. They also lead a great team of students who have provided numerous help to me throughout my graduate career. Jim challenged me to always look at the big picture and approach the problem from a different angle. I especially thank Karu for his constructive comments and his dedication to students.

Throughout my years in Madison, I had the privilege to interact with and learn from many professors in other research groups. Professor Mikko Lipasti has provided valuable insights to my research. Professors Miron Livny and Stephen Wright had been great supervisors when I worked on the big Condor project. Professor Mary Vernon taught me about analytical models. Professors Somesh Jha, Remzi Arpaci-Dusseau, Charles Fischer, and Jeff Naughton had been wonderful teachers to me both in and out of the classroom.

My career in computer architecture is especially enriched by the architecture group at Wisconsin, its alumni and industrial affiliates. I thank Professor Emeritus Jim Goodman and current architecture faulty

for their tireless effort in building an environment that is both intellectually critical and personally inviting. I especially thank our alumnus and Professor Doug Burger at UT-Austin, who collaborated with us on coherence decoupling and wrote reference letters for me. I also thank Wisconsin alumni T.N. Vijaykumar, Andreas Moshovos, Amir Roth, Ravi Rajwar, Kevin Lepak, Milo Martin, Trey Cain, Ashutosh Dhodapkar, Alaa Alameldeen, Shiliang Hu, Min Xu and Brad Beckmann for their technical and professional advices.

I thank the current and previous Multiscalar group members I have met in Madison: Craig Zilles, Adam Butts, Param Oberoi, Sai Balakrishnan, Allison Holloway, Philip Wells, Matthew Allen, Koushik Chakraborty and Vikas Garg. Before I joined the group, Craig, Adam and Param generously shared their experiences with Guri. Adam especially provided me with valuable feedbacks on my first architecture paper. Sai has been my "runahead" fellow throughout the years, helping me prepare for my qualification exam, build processor simulators, crystallize my ideas, and search for jobs. I totally enjoyed the many discussions with him on computers, technologies and cultures. Allison was my officemate for almost 5 years: she not only put up with my English but also kept helping me to better it. I am grateful for her advices on cultural and language issues, and have always loved to hear her speaking Chinese. I thank Philip, Matt, Koushik and Vikas for always willing to chat with me on both technical and personal issues, and for reading many of my write-ups. I also thank Philip and Koushik for educating discussions on processor power and reliability issues. I also thank Philip that he first suggested the idea of time-sharing based cache partitioning.

My research could have been much harder without using Multifacet group's Ruby cache simulator and their commercial workloads. I appreciate them for their willingness to share the research infrastructure with me, their help when I first played with Ruby and the valuable comments/discussions from them. I especially thank Carl Mauer and Alaa Alameldeen for simulator helps, Min Xu for being a great friend and sounding board, Brad Beckmann and Mike Marty for their inspirations on CMP caching and cache coherence research and Luke Yen for interesting discussions.

My research has been funded by by NSF grants EIA-0071924, CCR-0311572 and CNS-0551401, and

donations from Intel Corporation. Many thanks also go to the Computer Systems Lab and the Condor project for their infrastructure support.

Lastly I thank my family. My deepest thanks to my own life and wife Haining for her enduring and refreshing love: this dissertation could have never been finished without her support! I have discovered many of her virtues during our marriage, especially when we were both in the dissertation mode. I thank my two wonderful children John and Suzanne for their patience with me and their ways of teaching me to be humble. I thank my parents and parents-in-law for their love, support and always encouraging me to pursue my plan of life. I also thank my brother Hongchuan who stay in my hometown Qingdao to be with my parents, so I could come to the US for PhD study.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

Chip Multiprocessors (CMPs) have been widely adopted and commercially available [4,54,72,87,148] as the building blocks for future computer systems. Instead of building highly complex, power-hungry, single-threaded processors, CMP designers integrate multiple, potentially simpler, processor cores on a single chip to improve the overall throughput while reducing power consumption and design complexity. As the number of processor cores increases [73], a key aspect of CMP design is to provide fast data accesses for on-chip computation resources. Although caching has been one of the first and most widely used techniques to improve memory access speed in single-core chips, it faces several challenges when used in multi-core environments. To address these challenges, this dissertation studies the organization and management of CMP on-chip cache resources and proposes a unified caching framework to satisfy both performance and non-performance (e.g., fairness and Quality-of-Service (QoS)) requirements of future CMP systems.

1.1 CMP Caching Challenges

Unlike conventional designs with caches dedicated to a single processor core, CMP caches serve multiple threads running concurrently on physically distributed processor cores. This change of execution paradigm both aggravates caching demands and introduces new challenges that can not be sufficiently addressed by prior caching proposals.

1.1.1 Limited Off-chip Bandwidth

The main purpose of on-chip cache memory is to streamline processor operation by reducing the number of long-latency off-chip accesses. Based on a von Neumann architecture, processor computation involves frequent accesses of the memory system to fetch/store instructions and data. Historically, the performance gap between processor and DRAM has been increasing exponentially for more than two decades [118], which makes memory operations very expensive (costing hundreds of processor cycles). Even with large on-chip caches, high-performance processors often spend more than 50% of the time waiting for memory operations to complete [111].

Recently, improvement of single processor performance has slowed down as frequency scaling approaches its limit, but the "memory wall" problem is likely to persist for CMPs due to limited off-chip bandwidth. Technology trends [51] indicate that off-chip pin bandwidth will grow at a much lower rate than the number of processor cores (and thus their aggregate memory bandwidth requirement) on a CMP chip. Without disruptive technology (e.g., proximity communication [39]), the increasing bandwidth gap has to be bridged by efficient organization and use of available CMP cache resources.

Emerging software such as "Recognition, Mining and Synthesis" (RMS) workloads [71] can also stress the memory bandwidth requirement. Many such programs have poor data locality, either due to inherent streaming/scanning behaviors in the workload, or because better locality is only possible when their large working sets can be simultaneously satisfied by the last-level cache [75]. Therefore, both technology and software trends demand the CMP cache resources to be well utilized to reduce off-chip accesses.

1.1.2 Growing On-chip Wire Delay

In future technology, on-chip wire delay [63] will increase to a point that cross-chip cache accesses are far more expensive than local cache accesses. Without careful data placement, such non-uniform latencies reduce the benefit of on-chip caches because on average only a small fraction of blocks are located in cache

banks that are close to their consuming processors.

To reduce on-chip cache access latency, single-core designs exploit non-uniform cache architecture (NUCA) [26, 83] by migrating frequently used data into closer and faster cache banks. However, with multiple distributed cores accessing shared data, migration may be ineffective because the competition between different cores often leaves shared data in banks that are farther to all requesters [15]. In contrast, private cache organization reduces on-chip access latency by locally replicating frequently accessed data, but such replication can waste capacity and incur more expensive off-chip misses. CMP caching thus faces the conflicting requirements of saving off-chip bandwidth and reducing on-chip latency, and has to trade off between techniques that reduce off-chip vs. cross-chip misses [13].

1.1.3 Destructive Inter-thread Interference

The ineffectiveness of shared data migration among CMP cores demonstrates that competition of shared resources can lead to destructive inter-thread interference. For multiprogrammed workloads, threads also compete for cache capacity and associativity which can cause lowered performance (e.g., due to thrashing [35]), unexpected performance (e.g., due to unfair resource allocation [84]), lack of performance QoS [74] (i.e., no guarantee in providing certain baseline performance [151]) and lack of control over the per-thread and overall performance (e.g., no priority support). Without hardware solutions, their remedy can complicate the task of operating systems (e.g., to improve fairness and avoid priority inversion) and server administration (e.g., to maintain QoS for consolidated server workloads).

Because fairness, QoS and priority support are important requirements for CMP users, and are often assumed by software running on CMPs, CMP caching schemes have to attack the problem of destructive interference and answer the challenge of simultaneously satisfying multiple, potentially conflicting, requirements such as throughput and fairness improvement.

1.1.4 Diverse Workload Characteristics

With multiple execution contexts available, CMPs can support both single-threaded and multithreaded workloads as well as their multiprogramming combinations. These workloads demonstrate different caching characteristics, therefore prefer different cache organizations or caching policies. For example, multi-threaded workloads with large working sets prefer a shared cache for better effective capacity [75] while smaller workloads prefer a private cache organization for better on-chip latency [13]. Workloads with little sharing can benefit from dynamic migration [15], but aggressive sharing requires careful tradeoff between replication and migration. Furthermore, LRU-based cache replacement performs well for workloads with good temporal locality, while frequency-based policies (e.g., LFU) are more suitable for workloads with poor locality [142].

Diverse workload preferences suggest that using a single caching scheme with fixed policies is unlikely to provide robust performance for a wide range of workloads. An ideal CMP caching scheme should be able to combine the strengths of different cache organizations and policies, and dynamically adapt to suitable behaviors to accomodate individual workload's caching requirements.

1.2 Prior Caching Proposals

The importance of CMP caching has spurred many research proposals to answer some of the aforementioned challenges. Below we provide a brief overview of them.

1.2.1 Private and Shared Caches

Proposals for CMP on-chip memory hierarchy have borrowed heavily from the memory hierarchies of traditional multiprocessors. Here, each core have private L1 data and instruction caches tightly coupled with the processor pipeline. L2 caches are usually private because building a shared L2 for processors

on multiple chips requires significant complexity and pin bandwidth (for L1 to L2 cache communication). Proposals for CMPs also use the private L1 cache structures of traditional multiprocessors, although the L1 caches may not use inclusion [9, 12]. The interesting question has to do with the organization of large, last-level cache resources (in this dissertation we focus on L2 cache design).

Most current CMP designs adopt a logically shared L2 cache organization [4,72,87,148] to make efficient use of the overall cache capacity and reduce off-chip accesses. However, the latency of a shared L2 cache is heavily influenced by the increasing wire delay, and its unconstrained capacity sharing can cause destructive inter-thread interference. Private L2 caches are more tolerant of on-chip wire delay, and naturally avoid inter-thread interference by statically partitioning the aggregate capacity among processor cores. However, this organization incurs many more off-chip accesses due to inefficient use of the aggregate on-chip capacity (e.g., replication of shared data and static capacity allocation).

Because private and shared cache organizations have their unique advantages and disadvantages, two separate lines of research have been launched to combine their strengths to: (1) reduce average memory access latency and (2) improve throughput, fairness, and QoS provisioning via cache partitioning.

1.2.2 Latency Reduction Proposals

As a practical step towards the ideal zero-latency infinite-capacity cache, these proposals attempt to reach the off-chip miss rate of a shared cache with only the latency of a private cache. Private cache based optimizations improve capacity utilization by controlling replication [13, 27, 138] and allowing capacity sharing between caches [27, 156]. Shared cache based optimizations reduce on-chip latency by cache block migration [15, 158], replication [159] or limiting the scope of sharing within a small cluster [68]. However the proposed solutions are either limited in only improving a certain class of workloads [13, 15, 138, 156] or relying on specific cache or coherence protocol implementations [15, 27, 68, 138, 156, 159]. Furthermore, most of these proposals are unable to deal with workloads having poor locality and destructive interference

(with the exception of PDAS [156], which provides QoS support for multiprogrammed embedded workloads).

1.2.3 Cache Partitioning Schemes

CMP cache partitioning schemes [68, 74, 84, 96, 121, 123, 144, 156] extend a private cache organization's static capacity partitioning to isolate inter-thread interference. Specifically, they orchestrate cache allocation with dynamically adjustable, often heterogeneous, cache partitions to match the perceived capacity requirements of co-scheduled threads. Despite their differences in metrics, mechanisms and policies, prior cache partitioning schemes have two common limitations. (1) Limited functionality. None of the current proposals addresses all CMP caching requirements, including thrashing avoidance, fairness improvement, QoS guarantee and priority support, partially due to the difficulty of satisfying multiple, often conflicting, goals with a single cache partition. (2) Limited scope of application. Cache partitioning can only outperform LRU-based latency-reduction schemes for some multiprogrammed workloads. An attempt to use cache partitioning for a wide range of workloads either causes sub-optimal performance [141], or requires more complex partitioning scheme to close this performance gap [123].

1.3 Overview of Our Approach

Although prior CMP caching proposals can meet subsets of CMP caching requirements, an integrated scheme is needed not only to answer performance, fairness and QoS related challenges, but also support other optimizations such as power efficiency and reliability. The goal of this dissertation is to: (1) develop a unified framework to facilitate and integrate CMP caching optimizations and (2) demonstrate its effectiveness in improving both performance and non-performance objectives (such as fairness and QoS).

The basic ideas of our CMP Cooperative Caching (CC) are embodied in the CC framework, which consists of the following three layers.

- **Mechanisms.** This layer lies in the middle of the framework, using three key components to enable and control cooperative cache resource sharing: (1) private cache organization that reduces latency, bandwidth and design complexity while improving fairness and power-efficiency; (2) cooperation mechanisms that control data placement and replacement in the aggregate on-chip cache to support resource sharing among private caches; and (3) cooperation throttling mechanisms that control the amount and flow of resource sharing to achieve certain caching behaviors. The combination of these primitives can render many possible caching behaviors for CMP designers to explore.

- **Policies.** On top of the mechanism layer, this layer includes cooperative caching policies to achieve various optimization goals using the basic mechanisms. The cooperation philosophy is applied at this layer to resolve conflicts among competing peer caches as well as different caching objectives.

- **Implementations.** The underlying implementation layer realizes cooperative caching mechanisms by either augmenting existing cache designs and coherence protocols or introducing new designs. The separation of mechanisms, policies and implementations allows different layers to be extended independently, so that the framework can be used for different caching optimizations, and incorporated by various CMP implementations.

CC answers CMP caching challenges in the following ways.

- **Reducing off-chip accesses.** Via cooperation among private caches, CC can form an aggregate cache having an effective capacity comparable to a shared cache, to reduce costly off-chip misses. Specifically, three capacity sharing policies are proposed: (1) The first policy facilitates cache-to-cache transfers of on-chip "clean" blocks to eliminate unnecessary off-chip accesses to data that already reside elsewhere on the chip. (2) The second policy reduces replication to make room for unique on-chip copies (called **singlets**), thereby making better use of the on-chip cache resources. (3) The third policy places locally evicted blocks into peer caches (called **spill**). It lets the private caches

cooperatively identify singlet but inactive blocks, and keep globally active data on-chip. By combining local LRU with global spill/reuse history, this policy approximates global LRU replacement for efficient capacity sharing.

- **Improving on-chip latencies.** By using private caches as the baseline organization, CC attracts data locally to reduce remote on-chip accesses, thus lowering the average on-chip cache access latency. When combined with capacity improving policies, CC can achieve an off-chip miss rate similar to that of a shared cache, and a local cache hit rate similar to that of using private caches. Our evaluation using full-system simulation shows that CC performs robustly over a range of system/cache sizes and memory latencies. For an 8-core CMP with 1MB L2 cache per core, CC can improve the performance of multithreaded commercial workloads by 5-11% compared with a shared cache and 4-38% compared with private caches. For a 4-core CMP running multiprogrammed SPEC2000 workloads, CC is 5-23% faster than a shared cache, and at worst 1.5% slower than using private caches. CC also outperforms the victim replication scheme [159] by 9% on average over a mixture of multithreaded, single-threaded and multiprogrammed workloads, while the performance advantage increases for workloads with large working sets.

- **Dealing with inter-thread interference.** CC extends previous cache partitioning schemes with Multiple Time-sharing Partitions (MTP), to improve throughput and fairness while maintaining QoS. Specifically, each MTP partition improves at least one thrashing application's throughput by temporarily shrinking the capacity of other applications to make room for it. By time sharing cache resources among multiple unfair partitions that favor different applications, the problems of fairness improvement and priority support are translated into well-studied time-sharing resource management problems. Fairness can thus be improved by giving different applications equal opportunity to speed up, while priority can be supported by allocating different numbers of time slices to different unfair partitions. The MTP partitioning algorithm further guarantees QoS by using partitions that, on

average, can bound each application's slowdown against the uniform partitioning baseline. Comparing with the best single spatial partition based scheme without QoS constraint, MTP can achieve up to 60%, and on average 12%, better performance.

- **Adapting to workload characteristics.** CC can adapt to diverse workload characteristics by providing a spectrum of caching behaviors and selecting the best policy to meet workload preference. In terms of latency reduction, CC can throttle the amount of block replication and capacity sharing to find the best tradeoff between the two extremes of private and shared caches. CC further integrates latency reduction policies with MTP to combine the benefits of LRU-based fine-grained sharing with cache partitioning schemes. The complementary advantages of MTP and LRU-based optimizations are achieved by dividing the total execution epochs into those controlled by MTP and the baseline CC policies, respectively, according to the fraction of applications that can benefit from each of them. The integration not only provides the best performance for a wide range of workloads, but also can simplify MTP partitioning by focusing only on applications with large speedup potentials, leading to a heuristic-based, practical implementation. Our evaluation shows that the resulted scheme— Cooperative Cache Partitioning (CCP)—achieves comparable performance as an (impractical) exhaustive search of MTP partitions for workloads that need cache partitioning, which is up to 80%, and on average 7%, better than the baseline CC policies. For workloads where cache partitioning hurts, CCP usually defaults to the baseline CC and consistently outperforms all other cache partitioning schemes.

1.4 Thesis Outline

We have motivated CMP cooperative caching with technology and software trends in this chapter. Chapter 2 discusses related work in CMP caching. Chapter 3 then presents the cooperative caching framework, detailing its mechanism and implementation layers. Chapter 4 and Chapter 5 use latency reduction and throughput/fairness/QoS improvement as two example applications to demonstrate the benefits of our ap-

proach. We conclude in Chapter 6 by summarizing thesis contributions and pointing out future research directions.

The gist of Chapter 3 and Chapter 4 appeared in our ISCA 2006 paper [23], while we extend the published work in several aspects: (1) the ideas of CMP cooperative caching are now presented in a layered framework, (2) the mechanisms and implementations are introduced to support general cache cooperation instead of only reducing memory latency, (3) a discussion of scalable CC and a possible implementation are included, and (4) the evaluation is extended with adaptive throttling results.

A modified version of Chapter 5 was published in ICS 2007 [24], which is augmented in this dissertation with motivating examples, detailed discussion of implementation options, as well as an extensive presentation of the evaluation results.

CHAPTER 2

BACKGROUND AND PREVIOUS WORK

In this chapter, we discuss the background and previous work on CMP caching. Section 2.1 overviews the background on caching, with emphasis on the aspects where our proposed scheme (CC) differs from conventional uniprocessor caching: (1) cache placement and replacement, (2) support for non-uniform cache latencies, and (3) extensions to manage the sharing of cache resources among multiple competing consumers. Section 2.2 and Section 2.3 survey recent CMP caching proposals to improve average memory access latency and mitigate destructive inter-thread interference, respectively, and compare them with CC. Section 2.4 summarizes prior work with a taxonomy of hardware CMP caching schemes.

2.1 Caching

Although CMP caching presents a set of new challenges to processor cache designers, these challenges are not new in the history of general caching research and have been individually addressed in other caching systems such as virtual memory paging, web caching and conventional shared-memory multiprocessor memory designs.

The idea of caching was first documented in the IBM System/360 implementation [95] which used a high-speed buffer to bridge the processor-memory speed gap by exploiting the locality of references principle [37]. For a given cache size (which is determined by engineering tradeoffs), the cache's efficacy is largely determined by its data placement and replacement policies. Theoretically, Belady's MIN replacement algorithm is optimal by providing a provable upper limit for hit ratio [16]. However, because this algorithm evicts data with farthest reuse distance, it is limited in: (1) being an offline algorithm that requires both future knowledge and unbounded lookahead to make replacement decisions, (2) assuming

uniform cache hit latencies and miss latencies and (3) not considering conflicts between multiple cache resource consumers. Below we discuss extensions in these aspects separately.

2.1.1 Cache Replacement and Placement

To attack the first limitation of the MIN algorithm, many practical, online replacement policies have been proposed, among which the least recently used (LRU) and least frequently used (LFU) policies are two typical examples. The two policies are near optimal, respectively, for programs with strong and weak temporal locality [43][1]. LRU is arguably the most widely used policy because its implementation is simpler than LFU and it can quickly adapt to working set changes. To provide good caching performance for a wide range of workloads, many software policies (e.g., [77, 106]) and hardware designs (e.g., [42, 142]) are proposed to combine the benefits of both LRU and LFU. CC achieves the same goal, but instead by using cache partitioning to isolate workloads with weak locality from those with good locality and by integrating LRU replacement with cache partitioning.

In the context of CMP caching, flexible data placement is also needed to exploit the benefits of advanced cache replacement policies [57], reduce inter-thread conflict misses [158], provide QoS support via cache partitioning [81], and keep frequently accessed data close to the processor [26]. Highly associative caches (used by most CMP caching proposals) are thus needed to allow flexible data placement at the cost of extra area, latency, power and complexity overhead. Page coloring and remapping [130] can reduce conflict misses with low associativity, but require profile-based software optimization. Across cache banks, distance-aware placement attempts to keep frequently used data in the closest (and thus fastest) banks. For example, D-NUCA [83] dynamically migrates "hot" data towards the processor core, and NuRapid [26] further decouples data placement from tag placement using forward pointers.

[1]The notions of strong and weak temporal locality were formally treated by Coffman and Denning [43] and were later explained by Megiddo and Modha in the context of paging [106]: LRU is the optimal policy if the request stream is drawn from an LRU Stack Depth Distribution (SDD) which is "useful for treating the clustering effect of locality but not the nonuniform of page referencing." LFU is optimal if the workload can be characterized by the Independent Reference Model (IRM), which assumes that each reference is drawn in an independent fashion from a fixed distribution over the set of all pages in the auxiliary storage.

CC improves cache associativity through cooperation among private caches, each with lower associativity, without extra hardware and software overhead. The aggregate cache is managed by an approximation of global LRU via a combination of local LRU and global placement history, therefore can support fine-grained sharing for both multithreaded and multiprogrammed workloads. To achieve distance-aware data placement, CC relaxes the inclusion requirement between L1 and L2 caches, and improves data locality using private caches that keep frequently used data close to the requesting processors.

2.1.2 Non-uniform Latencies

In web caches that buffer files from multiple servers, cache misses have non-uniform latencies. Because Belady's MIN algorithm can only support uniform cache latency, new cache replacement policies are needed to prioritize data with variable costs and provide the best average access latency [157]. Similarly, processor caches can also exploit the cost difference between misses having non-uniform memory-level parallelism [52], as shown by proposals for cost-sensitive and MLP-aware uniprocessor caches [76, 122].

Distributed web and file caches can also cooperate to form a logically shared, global cache with non-uniform hit latencies [32, 46, 50]. By exploiting a high-speed network that makes inter-cache communication much faster than accessing remote servers, physically separated caches work closely together to improve the effective cache capacity without significantly hurting local hit rates. The situation is analogous to CMP caching, where private caches can exchange information and data over a high-bandwidth, low-latency, on-chip interconnect to reduce expensive off-chip misses. Our research is directly inspired by the above observation, and borrows from existing cooperative file caching policies.

The most related research to CMP caching, in terms of handling non-uniform access latencies, focuses on the memory organization of shared memory multiprocessors. To improve scalability of small-scale Uniform Memory Access (UMA) machines (SMP systems [25]), Non-uniform Memory Access (NUMA) systems statically partition the memory space among processor/memory nodes [91, 93]. Because local memory

DSM Memory	CMP Cache	Common Features
UMA [25]	Shared cache (UCA)	High capacity, uniformly long latency
NUMA [91, 93]	Shared, banked, cache (S-NUCA) [83]	High capacity, non-uniform latencies
COMA [56, 129]	Private caches [67]	Lowered capacity, low latency
RC-NUMA [160]	Adaptive private/shared NUCA [40]	Partition between private/shared space
VC-NUMA [108]	VC-CMP [115], Victim replication [159]	Victim caching
R-NUMA [45]	CMP-NuRapid [27]	Counter-based hints for relocation
AS-COMA [88]	CMP cooperative caching [23]	Biased replacement
OS support [150]	ASR [14]	Selective replication

Table 2.1: Corresponding Proposals for Organizing DSM Memory and CMP Caches

accesses can be several times faster than remote accesses in a NUMA system, a significant amount of research was done to improve data locality via a Cache-Only Memory Architecture (COMA) that uses local memory to attract frequently used data [56, 129], NUMA augmented with remote data caches or victim caches [108, 160], adaptation between COMA and CC-NUMA [45, 88] according to perceived memory pressure, and software techniques for page migration and replication [137, 150].

Due to similar latency/capacity tradeoffs, techniques to improve the average memory latency of distributed shared memory (DSM) systems can also be used to improve CMP caching. To illustrate this, Table 2.1 lists some of the corresponding proposals in DSM memory organization and CMP caching and their common features. CMP caching faces similar issues as their counterparts in DSM page caching such as cache coherence, replacement policies, control of replication/migration and scalability, although different tradeoffs and implementations are needed for DSM vs. CMP architectures.

2.1.3 Shared Resource Management

The third limitation of the MIN algorithm, its inability to deal with competition among multiple cache consumers, has been addressed by paging techniques (i.e., caching at the memory level). Sharing physical memory among competing threads, and specifically avoiding destructive interference such as thrashing [35], is a classic problem for virtual memory management [36]. This problem is solved in software by proactively co-scheduling programs whose aggregate working set can be contained by available resources (e.g.,

balanced-set schedulers [48]), or reactively reducing the level of multiprogramming (i.e., the number of in-memory programs) to fit resource constraints. Similarly, Fedorova studied CMP-aware operating system schedulers [47] to avoid cache thrashing [49] and maintain fair cache usage by adjusting the CPU time quantum allocated to different threads.

Memory partitioning is another mechanism to avoid interference. As an example, local memory management policies (such as the WS policy used in Windows operation systems) ensure that each thread has enough pages to hold its working set [34] and only replace pages from a thread's local page pool. The concept of working set and local replacement is also exploited by CMP caching partitioning schemes (e.g., [40, 123, 144, 156]). On the other hand, global replacement policies allow different threads to share space in a fine-grained manner, but are prone to inter-thread pollution. WSClock [22] achieves the benefits of both schemes by integrating working set based partitioning with global LRU replacement. In a similar vein, we integrate cache partitioning and LRU-based latency reduction policies to combine their strengths.

Verghese et al. [151] recognized the need for performance isolation in a central server environment, and proposed mechanisms to provide isolation under heavy load while allowing sharing under light load. Later, the notion of performance isolation was extended to provide flexible QoS [145], including QoS guarantee, fairness and differentiated services. Waldspurger [152] introduced several novel policies for virtual machine servers, which: (1) identify and reclaim least valuable pages, (2) eliminate redundancy overhead and (3) support performance isolation. This dissertation attack the same problems for processor caching via: (1) global replacement of inactive blocks, (2) replication-aware replacement and (3) cache partitioning support. To simultaneously improve throughput, fairness and QoS, we also extend spatial partitioning with time-sharing, which has been well studied and implemented by operating systems [62, 153].

2.2 CMP Caching Optimizations for Latency Reduction

Besides studies in CMP cache organizations [67, 113, 146], many hybrid caching schemes have been proposed to reduce the average memory access latency for CMPs [14, 15, 27, 59, 68, 94, 115, 138, 158, 159].

2.2.1 Shared Cache Based Proposals

Oi and Ranganathan [115] made the analogy between CMP caching and CC-NUMA memory organization, and evaluated the benefit of using part of the shared L2 cache as fix-sized victim caches. They discovered that aggressive replication (using large victim caches) can hurt performance, and for SPLASH2 workloads [155], suggested using 1/8 of total L2 capacity for victim caching. Zhang and Asanovic [159] proposed a dynamically adaptable form of victim caching, called Victim Replication (VR), in a tile-based CMP. Their scheme allows an L1 victim to be cached in the local L2 bank, potentially evicting data without replicas. A random replacement policy and a directory-based coherence protocol are required by their implementation to simplify replication control and replica identification. Because VR keeps replicas in both the home node cache and all consumer caches, it can waste capacity when little data sharing exists (e.g., in multiprogrammed workloads). This issue was resolved by victim migration [158] via home block migration, implemented with extra shadow tags to keep track of the migrated data.

Beckmann and Wood [15] studied CMP-NUCA schemes to mitigate the impacts of increasing on-chip wire delay. Their results showed that, different from single-core caching, dynamic migration (D-NUCA) in a CMP is ineffective for widely shared data. They also identified the issues with CMP-NUCA implementation and power consumption, and proposed using LC transmission lines to reduce wire delay. Li et al. [94] extended their work with 3D die-stacking and network-in-memory, and demonstrated better performance. The techniques proposed in this dissertation are orthogonal to these new technologies and can potentially exploit fast wires and 3D caches to improve latency and capacity.

Chishti et al. proposed CMP-NuRapid [27] to optimize replication and capacity in CMP caches. Their design, like CMP-NUCA, uses individual algorithms to optimize special sharing patterns (i.e., controlled replication for read-only sharing, *in-situ* communication for read-write sharing, and capacity stealing for non-sharing). In contrast, CC aims to achieve these optimizations through a unified technique: cooperative cache placement/replacement, which can be implemented in either a centralized or distributed manner.

CMP-NuRapid implements a distributed directory/routing service by maintaining forward and reverse pointers between the private tag arrays and the shared data arrays. This implementation requires extra tag entries that may limit its scalability, and increases the complexity of the coherence protocol (e.g., the protocol has to avoid creating dangling pointers). CC tries to avoid such issues by using a simple, centralized directory engine with less space overhead.

Based on NUCA, Huh et al. [68] introduced a cache organization to support a spectrum of sharing degrees, which denote the number of processors sharing a pool of their local L2 banks. The average access latency can be optimized by partitioning the aggregate on-chip cache into disjoint pools, to fit the running application's capacity requirement and sharing patterns. Their study showed that static mappings with a sharing degree of 2 or 4 can provide the best latency, and dynamic mapping can improve performance at the cost of complexity and power consumption. CC is similar in trying to support a spectrum of sharing points, but achieves it through cooperation among private caches and adaptive cooperation throttling.

Cho and Jin [28] recently proposed an OS-level page allocation approach to address CMP caching's locality, capacity and isolation issues. Based on shared cache organization, their proposal maps physical pages into cache slices to exploit locality and uses "virtual multi-cores" to provide isolation between multiprogrammed threads. However, this approach requires both hardware page mapping support [78], and significant modifications in the operating systems (e.g., location-aware page allocation, locality and capacity aware OS scheduling). Comparatively, CC answers CMP caching challenges with simple hardware extensions, while being amenable to software-based solutions by providing capacity sharing and isolation mechanisms.

2.2.2 Private Cache Based Proposals

Harris proposed synergistic caching [59] for large scale CMPs, in which neighboring cores and their private L1 caches are grouped into clusters to allow fast access of shared data. Synergistic caching offers three

duplication modes (i.e., beg, borrow and steal), corresponding to replication, use without replication and migration. Because no single duplication mode performs the best across all benchmarks, reconfiguration was suggested, although not evaluated, to statically or dynamically choose the best mode.

Speight et al. [138] studied adaptive mechanisms in private cache based designs to reduce off-chip traffic. They used an L2 snarf table to identify locally evicted blocks that might be reused soon. Upon local eviction, such blocks are kept on-chip via write-backs to peer caches. The host cache will replace either invalidated or shared clean blocks to make room for them, potentially reducing expensive off-chip misses when they are reused later. This scheme is limited by only supporting multithreaded workloads, and differs from CC in its write-back and global replacement policies.

Motivated by the need for dynamic adaptation, Beckmann et al. [14] proposed Adaptive Selective Replication (ASR) for multithreaded workloads. ASR uses private caches and dynamically seeks an optimal degree of replication for each individual thread. The cost and benefit of current and future replication levels are estimated using on-chip counters, which are considered when determining whether a thread can benefit from more aggressive replication. Their cost/benefit estimation mechanisms can be used to throttle CC's replacement-aware cache replacement. Although ASR was proposed to control replication in multithreaded workloads, its per-thread threshold and counters do support heterogeneous workloads and can potentially be extended and used in a multiprogramming environment.

2.3 CMP Cache Partitioning

Below we survey CMP cache partitioning proposals that prevent destructive inter-thread interference via resource isolation, according to their different optimization purposes.

Miss reduction. Stone et al. [141] studied the problem of partitioning cache capacity between different reference streams, and identified LRU as the near-optimal policy for their workloads. They also showed empirically that LRU can swiftly adapt to working set changes, without explicit repartitioning support. Liu

et al. [96] proposed Shared Processor-based Split L2 cache that partitions the shared L2 space in units of "split." Their scheme can be configured to suit for both inter-application and intra-application non-uniform caching requirements, but involves both profiling support and operating system modification to determine and enforce cache partitioning. Suh et al. [143, 144] applied way partitioning to shared CMP caches. Using in-cache monitoring mechanism, their partitioning algorithm assumes convex miss rate curves and allocates extra capacity to threads having the best marginal miss rate reduction. Qureshi and Patt [123] proposed UMON sampling mechanism to provide more precise measurement, and lookahead partitioning algorithm to handle workloads with non-convex miss rate curves. Dybdahl et al. [41] extended way partitioning [144] by overbooking cache capacity to account for non-uniform per-set requirements, and evaluated its effectiveness using private L1/L2 caches with a shared L3 cache. Dybdahl and Stenstrom [40] further extended CC with an adaptive shared/private partitioning scheme to avoid inter-thread interference. Their partitioning algorithm is essentially the same as Suh's proposal, but instead of in-cache monitors, "shadow tags" are used to measure the benefit of having one extra cache way.

Fairness improvement. Kim et al. [84] emphasized the importance of fair CMP caching, discussed the implication of unfairness (such as priority inversion) and proposed a set of fairness metrics as their goal of optimization. They evaluated both static and dynamic partitioning (both requiring profiling information), and discovered that, under heavy caching pressure, fair caching often improves overall throughput. Yeh and Reinman [156] proposed fast and fair partitioning, based on a NUCA cache consisted of ring-connected distributed banks. Their scheme ensures the "baseline fairness" by guaranteeing QoS for all co-scheduled threads. To enforce partitioning decision, each NUCA bank is divided between portions used by the local thread and remote threads. Such partitioning is dynamically adjustable based on program requirements and phase changes. Instead of cache partitioning, Kondo et al. [86] applied dynamic voltage and frequency scaling (DVFS) to maintain CMP fair caching. Using the same metrics and policies as Kim et al., they also observed throughput improvement over shared cache for many cases and energy saving due to decreased

voltage and frequency.

QoS provisioning. Iyer [74] motivated the importance of QoS guarantee and prioritization, not only between different users but also different types of access streams (e.g., demand vs. prefetch requests) generated by the same thread. He also proposed the CQoS framework that defines and implements QoS via priority classification, assignment and enforcement. Yeh and Reinman [156] focused on throughput improvement on top of data QoS guarantee. Kannan et al. [81] studied CMP resource management to support flexible QoS. Their work demonstrated the feasibility of QoS-aware hardware and software using prototypes and showed predictable performance in multiprogramming and virtualization environments. Vardarajan et al. [149] proposed molecular caches to accomodate the diverse requirements of multiprogrammed workloads. By varying the number of allocated tiles (or molecules) and the per-tile management policies, molecular caches can provide varied cache line sizes, associativities and cache sizes for different co-scheduled threads. Molecular caches achieve power efficiency via private tile based organization, and provides software-defined QoS (specified as target miss rates) by dynamically adjusting per-application capacity.

Generic support. Rafique et al. [125] and Petoumenos et al. [121] proposed spatially fine-grained partitioning support, which can be used by various partitioning policies (such as miss rate reduction, fair caching and QoS provision). Hsu et al. [66] studied various partitioning metrics and policies. Their study focused on three caching paradigms (communist caching for fairness, utilitarian caching for overall throughput and uncontrolled capitalist caching), and recognized the difficulties to improve both overall throughput and fairness using a single partitioning scheme.

Goals		Target Workloads	Baseline	Schemes
Latency Reduction		Multithreaded	Private	Adaptive L2 snarfing [138] ASR [14] Synergistic cache [59]
		Ineffective for shared data Ineffective for multiprogrammed	Shared	CMP-DNUCA [15] Victim replication [159]
		Multithreaded and multiprogrammed	Shared	Victim migration [158] CMP-NuRapid [27] NUCA substrate [68] OS-level page mapping [28]
Interference Isolation	Miss Reduction	Multithreaded and multiprogrammed Multiprogrammed Multiprogrammed Multiprogrammed Multiprogrammed	Shared Shared Shared Shared Private	Profile-based partitioning [96] Dynamic partitioning [144] Utility-based partitioning [123] Overbooked partitioning [41] Share/private partitioning [40]
	Fairness / QoS	Multiprogrammed	Shared Shared Shared Shared Private Private	Fair caching [84] Fair caching via DVFS [86] CQoS [74] QoS prototypes [81] Fast and fair (QoS) [156] Molecular caches [149]
	Mechanism	Multiprogrammed	Shared	STATSHARE [121] OS-driven partitioning [125] Hsu et al. [66]
All		Multithreaded and multiprogrammed	Private	CC/CCP (this dissertation)

Table 2.2: A Taxonomy of Hardware CMP Caching Schemes

2.4 A Taxonomy of CMP Caching Techniques

Table 2.2 provides a taxonomy of related hardware CMP caching proposals. These schemes are classified along three dimensions: (1) goals (latency reduction, interference isolation, or their combinations), (2) target workloads (multithreaded or multiprogrammed) and (3) baseline cache organizations (shared or private). A desirable CMP caching scheme should be able to simultaneously achieve multiple optimization goals, perform robustly for a wide range of workloads, while being amenable to simple and modular hardware implementations.

The first group of proposals in Table 2.2 aim at memory latency reduction, but 5 out of 9 of them are limited in only supporting multithreaded workloads or being ineffective for workloads with significant data sharing. The schemes that do support both multithreaded and multiprogrammed workloads are all based on a shared cache design, and they either rely on specific hardware implementations (e.g., victim migration [158] requires a directory protocol and CMP-NuRapid [27] depends on a snooping bus) or have to cooperate with software for cache management (e.g., NUCA substrate [68] and OS-level page mapping [28]).

The schemes in the second group use cache partitioning to achieve some of the interference isolation goals. Among them, the first 5 schemes only minimize off-chip miss rate to improve overall throughput. Except for profile-based partitioning [96], these schemes are only applicable to multiprogrammed workloads. Share/private partitioning [40] is the only proposal that combines cache partitioning with latency reduction techniques (it uses CC as the baseline design). Other cache partitioning schemes either focus only on fairness or QoS improvement, or only provide generic partitioning mechanisms. Except for fast and fair [156] which improves throughput while maintaining QoS, none of these proposals support multiple optimization goals.

Finally, CC is the only hardware CMP caching scheme that addresses all aspects of latency reduction and interference isolation optimizations, supports both multithreaded and multiprogrammed workloads and exploits the latency, power, and design modularity benefits of private caches.

CHAPTER 3

CMP COOPERATIVE CACHING FRAMEWORK

Consider the task of building a CMP cache hierarchy that not only has high capacity and low latency, but also is fair, reliable, power-efficient, and easy to design and verify. Each of these requirement may prefer a different way to organize the available resources and a different policy to strike the balance between resource sharing and isolation, while all optimizations have to fit in the same design. As shown in previous chapters, existing proposals do address some of the key challenges of CMP caching. However, without a unified solution, these schemes can not satisfy all of these important, yet potentially conflicting, requirements for workloads with diverse caching characteristics.

We need a holistic approach to address the challenges of CMP caching, which can integrate and trade off among available optimizations, preferably based on a unified supporting framework. The questions are whether such a framework exists and, if so, how to use it to accomodate conflicting requirements. This chapter answers the first question with the cooperative caching framework for efficient organization and management of CMP cache resources, while the following two chapters will demonstrate its uses in latency reduction and interference isolation, respectively. Below, we introduce the three-layer model of CC framework in Section 3.1. The mechanism and implementation layers are covered by Section 3.2 and Section 3.3, while example policies will be detailed in Chapter 4 and 5. Section 3.4 discusses the extension of CC for large-scale CMPs.

3.1 CMP Cooperative Caching Framework

The basic idea of CC is to form a globally-managed, aggregate on-chip cache via cooperative resource sharing among private caches. CC is inspired by software cooperative caching algorithms [32], which have

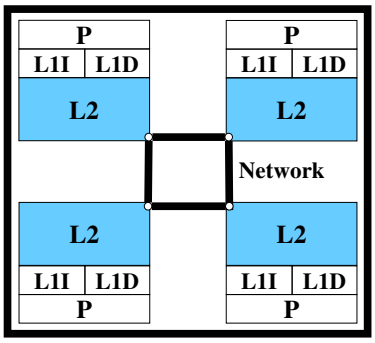


Figure 3.1: Cooperative Caching (The shaded area represents the aggregate cache formed via cooperative private caches.)

been proposed and shown to be effective in the context of file and web caching [32, 46]. The key principle of CC is to support a wide spectrum of sharing behaviors via the combination of private cache's resource isolation with various forms of cooperative capacity sharing/throttling. Such capabilities will be exploited by high-level cache cooperation policies to satisfy different sets of optimization goals.

Figure 3.1 shows a picture of the CC concept for a CMP with four cores. To simplify discussion in this dissertation, we assume a CMP memory hierarchy with private L1 instruction and data caches for each processing core. We focus on using L2 cache as the last level on-chip cache, although the ideas of CC are equally applicable if there are more levels of caches on the chip (e.g., Dybdahl and Stenstrom [40] evaluated CC with L3 caches). Each processor core's local L2 cache banks are physically close to it, and privately owned by it such that only the processor itself can directly access them. Local L2 cache misses can possibly be served by remote on-chip caches via cache-to-cache transfers as in a traditional cache coherence protocol. The key difference between CC and conventional private caches is that here the private caches are not isolated from each other, instead they act as parts of a logically shared cache by sharing information and resources. Such sharing activities are enabled by CC's cooperation and throttling mechanisms, and orchestrated by cooperation policies to achieve specific optimization goals.

Before elaborating on why we choose a private cache based organization and how to support inter-cache sharing in the CC framework, we now introduce the layered structure of CC framework. As depicted

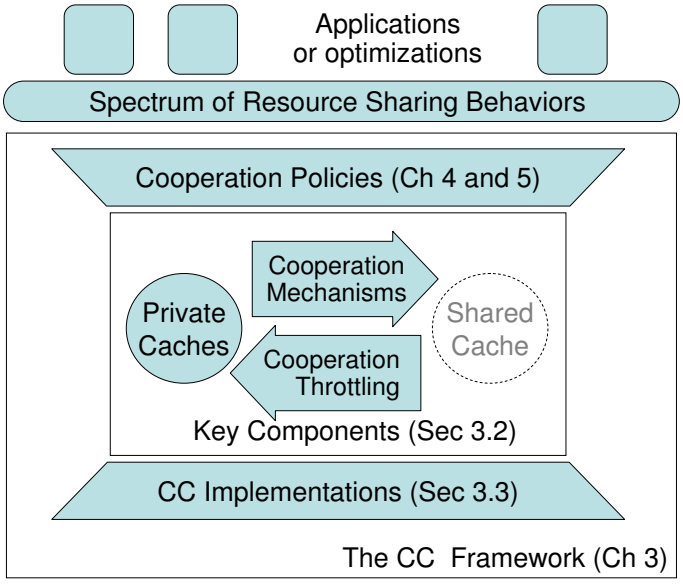


Figure 3.2: The CMP Cooperative Caching (CC) Framework

in Figure 3.2, the idea of cooperative caching is supported by three layers focusing on implementation, mechanism and policy, respectively. This separation of layers not only allows their independent extension, but also simplifies the design and analysis of CMP cache hierarchy by isolating and addressing system properties at appropriate levels of abstraction.

Implementation layer. At the lowest level, the layer contains various implementations of cooperative cache structures and resource sharing/isolation mechanisms. This layer encapsulates the implementation details such as cache sub-banking, chip layout, interconnection networks and coherence protocols, so that the key mechanisms can be supported by different designs. Correctness and scalability are the main system-level properties maintained at this layer.

Mechanism layer. In the center of the framework, the mechanisms layer provides abstractions of cache organization and resource management to support cooperative caching policies. This layer consists of three key components: **(1) private cache based organization** that provides on-chip locality and resource isolation; **(2) cooperation mechanisms** used by co-scheduled threads to explicitly share the aggregate cache

resources; and **(3) cooperation throttling mechanisms** used to control and orchestrate cooperative resource sharing. These components can each be populated with concrete mechanisms, and the key property of this layer is composability. When these mechanisms are combined by caching policies, a wide spectrum of sharing behaviors should be available to suit the needs of specific workloads and cache optimizations.

Policy layer. This layer optimizes the cache hierarchy's high-level properties such as performance, fairness, QoS, and reliability, to fit for its intended use. A specific policy can optimize for one property such as fairness, and potentially impact the other properties in positive or negative ways. CMP caching optimizations (e.g., power or reliability optimization) or applications (e.g., QoS provision) can further select from available policies according to their strengths and shortcomings.

3.2 CC Mechanisms

Below we describe the key components in CC's mechanism layer. This layer plays a similar role as the IP layer in the Internet protocol suite [29, 30] because they both aim to support diverse higher level applications with a small set of abstractions that are open to many possible implementations.

3.2.1 Private Cache Organization

Compared with many shared cache based CMP caching schemes, CC uses a private cache based organization because it has the following advantages that are likely to be of increasing importance for future CMPs.

1. **Latency.** Private caches reduce on-chip access latency by keeping frequently referenced data locally for fast later reuse.

2. **Bandwidth.** Locally cached data can filter out accesses to remote caches, significantly lowering the bandwidth requirement on the cross-chip interconnection network. This can translate into a simpler and potentially faster network, while the saved area and power budget can be used for other purposes.

3. **Associativity.** Instead of building a highly-associative shared cache to avoid inter-thread conflict misses, the same set-associativity is available for an aggregate cache formed by private caches each with much lower associativity, thus reducing power, complexity and latency overhead.

4. **Modularity.** Compared with a shared cache whose directory information is stored with cache tags and distributed across all banks, a private cache is more self-contained and thus serves as a natural unit for resource management (e.g., power off to save energy).

5. **Encapsulation.** Because private caches interact only through exchanges of cache coherence messages, the internal organization and operation of individual caches are encapsulated and thus hidden from other caches and processors. Encapsulation enables CMP caches to have different sizes, associativities, caching policies, voltage/frequency configurations and reliability characteristics to either support heterogeneous processors, or accomodate workloads with diverse performance, power, and reliability requirements.

6. **Explicit sharing.** Because encapsulation forces inter-cache resource sharing to be explicit, cooperation throttling is thus easy to implement. Because a cluster of cooperative private caches can be viewed as one larger private cache, explicit sharing based cooperation mechanisms and policies are reusable at the granularity of cache-clusters, thus simplifying the task of composing a large-scale CMP from small-scale clusters of cores and caches.

7. **Adoption of existing hardware optimizations.** Because many microarchitectural optimizations assume a uniprocessor with an exclusively owned cache hierarchy, private cache based designs allow a smooth adoption of such techniques into CMP systems.

8. **Software portability.** Existing parallel software (in particular the operating system) is written for processors with private caches, and thus directly portable for private cache based CMP systems. On the other hand, breaking the assumption of an exclusively owned cache could cause unexpected

problems (e.g., fairness and security issues [120]).

But partaking of these potential advantages first requires a solution to the major, and predominant, drawback of private cache designs: the larger number of off-chip cache misses compared to shared cache designs. CC attempts to make the ensemble of private L2 caches (the shaded area in Figure 3.1) appear like a collective shared cache via cooperative resource sharing. For example, cooperative caching will use remote L2 caches to hold (and thus serve) data that would generally not fit in the local L2 cache, if there is spare space available in a remote L2 cache. To support such capacity sharing, CC extends conventional private caches in two ways. It relaxes multi-level inclusion between L1 and L2 caches to enable flexible data placement, and it supports sharing of on-chip clean data to save unnecessary off-chip misses.

Non-inclusive Caches

Capacity sharing among private caches requires decoupled data placement in the L1 caches and their companion L2 caches. Conventionally, multi-level cache hierarchies often employ inclusion, where the lower level caches (which are closer to the processors and often smaller) can only maintain copies of data in their companion higher level caches. Cache inclusion implies that a block will have to be invalidated in an L1 cache when it is evicted from or invalidated in the companion L2 cache. This can greatly simplify the implementation of the coherence protocol because invalidation requests can be filtered by the higher level on-chip caches. However with CC, since the objective is to create a global L2 cache from the individual private L2 caches, maintaining inclusion with only a single L2 cache bank unnecessarily restricts the ability of an L1 cache to buffer a variety of data that reside in on-chip L2 caches. An arbitrary L1 cache needs to be able to cache any block from the aggregate L2 cache, and not only a block from the companion L2 cache bank. Thus, for CC to be effective, the L1 caches have to be either **non-inclusive** (where a block may be present in either the L1 or companion L2 cache, or both) or **exclusive** (where a block can be present in either the L1 or companion L2 cache, but not both), i.e., be able to cache a block that may not reside in the

associated L2 cache bank.

Sharing Clean Data Among Private Caches

With private L2 caches, memory access can be avoided on an L2 cache miss if the data can be obtained from another on-chip cache. Such inter-cache data sharing (via cache-to-cache transfers) can be viewed as a simple form of cooperation, usually implemented by the cache coherence protocol to ensure correct operation. Except for a few protocols that have the notion of a "clean owner,"[1] modern multiprocessors employ variants of invalidate-based coherence protocols that only allow cache-to-cache transfers of "dirty" data (meaning that the data was written by a processor and has not been written back to the higher level storage). If there is a miss on a block that only has clean copies in other caches, the higher-level storage (usually the main memory) has to respond with the block, even though it is unnecessary for correct operation and can be more expensive than a cache-to-cache transfer within a CMP.

There are two main reasons why coherence protocols employed by traditional multiprocessors do not support such sharing of clean copies. First, cache-to-cache transfer requires one and only one cache to respond to the miss request to ensure correctness, and maintaining a unique clean-owner is not as straightforward as a dirty-owner (i.e., the last writer). Second, in traditional multiprocessors, off-chip communication is required whether to source the clean copy from another cache or from the main memory, and the latency savings of the first option is often not big enough to justify the complexity it adds to the coherence protocol. In fact, in many cases obtaining the data from memory can be faster than obtaining it from another cache.

However, CMPs have made off-chip accesses significantly more costly than on-chip cache-to-cache transfers; currently there is an order of magnitude difference in the latencies of the two operations. Moreover, unlike traditional multiprocessors, (on-chip) cache-to-cache transfers do not need off-chip communication. Furthermore, a high percentage of misses in commercial workloads can be satisfied by sharing clean data,

[1] For example, the Illinois protocol [116], EDWP protocol [5] and Token Coherence [101].

due to frequent misses to (mostly) read-only data (especially instructions) [11, 13, 97]. These factors make it more appealing to let caches share clean data[2].

3.2.2 Cooperation Mechanisms

By sharing on-chip clean/dirty data and relaxing multi-level cache inclusion, an aggregate cache is formed by the collection of on-chip private caches. This aggregate cache differs from the baseline private cache organization in that, through cache cooperation, its content can be controlled to offer different capacities and latencies for different processor cores.

Cache cooperation, for example to share the aggregate cache capacity, is a new hardware caching opportunity afforded by CMPs. As we shall see, cooperation between caches will require the exchange of information and data between different caches, under the control of cooperation and throttling policies. Such an exchange of information and data, over and above the support of a basic cache coherence protocol, was not considered practical, or fruitful, in multiprocessors built with multiple chips. For example, for a variety of reasons, when a data block is evicted from the L2 cache of one processor, it would not be placed in the L2 cache of another processor. The complexity of determining which L2 cache to place the evicted block into, and transferring the block to its new L2 cache home, would be significant. Moreover, this additional complexity would provide little benefit, since the latency of getting a block from memory would typically be lower than getting it from another L2 cache. But, the situation for CMP on-chip caches is very different: the transfer of information and data between on-chip caches can be done relatively easily and efficiently, while the benefit of cooperation (e.g., avoiding costly off-chip misses) can be significant.

The CC framework provides cache placement and replacement based cooperation mechanisms to exploit the aforementioned opportunities. These mechanisms add local extensions to each cache in two aspects:

[2] IBM Power4 [148] and Power5 systems add two states (SL and T) to their MESI-based coherence protocol to select and transfer clean ownership. The first module that fetches a clean data naturally becomes its owner, and the ownership transfers to the next requester when the data is forwarded to a different cache. If the data is replaced before requested, the ownership is lost. But with the help of highly associative (8-way) L2 caches, this should happen only infrequently.

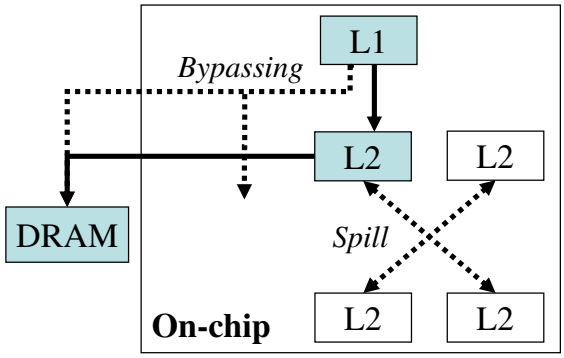


Figure 3.3: Examples of Cache Placement Based Cooperations (Dotted lines represent data paths introduced by placement based cooperations, while solid lines indicate the original data paths.)

(1) what blocks are placed into it and (2) what blocks are replaced (or displaced) from it. CC policies can combine these mechanisms to determine what data are kept in the aggregate on-chip cache, and specifically, in which individual caches.

Cache Placement Based Cooperation

Cache placement based cooperation treats each L2 cache as a black-box: without changing the internal caching operation, it affects cache content by modifying what data can be placed into the cache. This is achieved by modifying an L2 cache's local and remote request streams, as illustrated in Figure 3.3.

An L2 cache's local request stream consists of demand misses and prefetches generated by its associated L1 caches. Placement based cooperation can filter such requests to avoid L2 cache pollution. One such example is cache bypassing [80]: by using compiler-generated hints or runtime statistics to predict data locality in large chunks, it only insert data with good temporal locality in the cache. Similarly, CMP-NuRapid [27] reduces unnecessary data replication by only placing data in a local L2 when it has been recently reused (thus is likely to be reused in the future). Both examples selectively bypass L2 placement to improve the effective cache capacity.

CC also introduces a remote request stream by placing (or "spilling") locally evicted blocks into remote on-chip caches (labeled as **spill** streams in Figure 3.3). Spill allows capacity sharing among peer caches,

making all on-chip storage resources available for one processor core to use. Through spill, each cache can observe the reference and reuse streams from both local and remote computation threads. Such information can be used to devise approximate global cache management policies (such as global LRU replacement) without paying the overhead of global coordination and synchronization.

The spill mechanism can be tailored in different ways. For example, cooperation policies can vary in deciding (1) what data can be spilled, (2) which cache to host the spilled data, (3) what data should be replaced to make room for the spilled data, (4) whether spilled data can trigger further spills, and (5) how to balance the competition between local and remote data, etc. We leave policy decisions for later discussion, but make two mechanism-level decisions here. First, by default, we choose to randomly pick a host cache while giving higher probabilities to close neighbors. The random algorithm requires no global coordination, and allows one cache's victims to reach all other on-chip caches without requiring "rippled spilling". Keeping spilled blocks close to their previous caches can reduce both spilling time and access latency for later reuse. Second, because one of the purposes of spill is to extend the on-chip life cycle of local victims, the host cache should handle a newly-arrived spilled block in the same way as a demand miss. This implies that for LRU-based replacement policies, the spilled block is initially set as the most recently used (MRU) entry in the host cache.

Cache Replacement Based Cooperation

Through placement based cooperation, each on-chip cache can be potentially shared by many computation threads generating heterogeneous memory access streams and having different locality, communication and sharing properties. Consequently, conventional cache replacement policies (e.g., LRU, random or pseudo-LRU) that treat all references equally can cause poor QoS and sub-optimal performance. To recognize and exploit data heterogeneity, cache replacement based cooperation combines the default cache replacement with data priority.

This mechanism augments each cache block with a few bits for data classification, which can represent compiler generated locality hints [154] or dynamic sharing and communication properties [23, 159]. The cache replacement logic is extended to use such information and always evict blocks with the lowest priority. For example, giving lower priority to data with on-chip replicas can quickly yield space for unique data copies, thus increasing the aggregate cache's effective capacity. If multiple candidates exist in the lowest priority, the default cache replacement policy is used to break even. Because replacement based cooperation only changes how victim blocks are selected, it has no correctness implication and allows more flexible implementations (e.g., trading accuracy or even correctness for simplicity).

Replacement based cooperation basically allows the customization of individual caches to prefer different types of data. This mechanism can be combined with spill in various ways to manage the content of the aggregate on-chip cache as well as individual on-chip caches. Below we discuss three examples. First, caches with the same priority setting can be composed via spilling to collectively replace undesirable data. Consider a 2-core CMP where one core's L2 cache contains only high priority data while the other L2 cache only has low priority data. When isolated, each individual cache has only one class of data, so the prioritized replacement policy falls back to the default replacement policy. However, with the help of spill, the two caches can be glued together as an aggregate cache with two data classes. High priority data spilled from its original cache will cause global eviction of low priority data, leading to better utilization of the aggregate cache. Second, caches with opposite priority settings can be connected via spilling as producer-consumer pairs, where the cache blocks updated by a producer cache are evicted and spilled directly into its consumer cache. Such capability can be exploited by computation with task-level pipeline parallelism for streamlined data communication [53]. Lastly, caches with different performance, reliability and power characteristics can potentially be configured with different priority settings, so that heterogeneous cache resources can be matched to data classes with different properties [69].

On the other hand, priority-based replacement should be used wisely to avoid cache resources being

entirely occupied by high priority data, potentially leading to a Denial-of-Service (DoS) attack for low priority data. In this dissertation, DoS is not an issue because we use priority-based replacement only in L2 caches, and only to reduce the amount of data replication (Section 4.2.1). Other uses of priority-based replacement can avoid the DoS problem with a probabilistic implementation, where high priority data have a small chance of being evicted while low priority blocks exist.

3.2.3 Cooperation Throttling Mechanisms

Because CC's cooperation is based on modifications of cache replacement and placement logic, it simplifies cooperation throttling to a small set of control points. Below we consider two classes of throttling mechanisms.

Probability Based Throttling

To adjust the amount of cooperation, cooperation probabilities can be specified at the following three control points: (1) local L1 to L2 data path, (2) L2 replacement logic, and (3) L2 spill logic. These probabilities are used to decide how often to apply cooperation instead of taking the default action. Probability based throttling allows CC to provide a wide spectrum of sharing behaviors. If all three probabilities are set to 0, CC defaults to private caches (albeit with support for cache-to-cache transfer of clean data). CC's behavior moves towards more aggressive resource sharing as these probabilities increase.

Probability based throttling can be used for different workloads. For homogeneous workloads where all threads have similar caching behavior, only one set of system-wide probabilities is needed. However, heterogeneous workloads demand thread-specific probabilities to suit the caching requirements of individual threads. For example, programs with larger working sets can have higher probability for spill and cooperative replacement, while smaller programs can have higher probabilities on L2 bypassing. This way the smaller programs can save space in their local caches for larger programs, and the aggregate cache resource

are more efficiently shared.

Quota Based Throttling

CC also supports quota based throttling: each thread's maximum resource consumption can be specified and CC will make sure these quotas are honored. Resource quota can be either coarse-grained or fine-grained. Many cache partitioning schemes allocate capacity in large chunks, based on way partitioning [144]. Fine-grained quota, on the other hand, can be an arbitrary number of cache blocks.

CC maintains quota by tracking each thread's resource consumption and replaces data from over-quota threads with data from other threads. CC throttles a thread's data placement decisions based on whether the thread has used up its capacity quota. Specifically, CC disallows a thread to spill locally evicted data to other caches if the thread's current capacity exceeds its quota. As data spilled by another thread replaces data stored in its local cache, an over-quota thread's capacity usage will be decreased. On the other hand, CC avoids selecting a under-quota thread's private cache as a recipient of spilled data, so that this thread's capacity usage will only gradually increase.

3.3 CC Implementations

In this section, we present the implementation of the CC framework. Section 3.3.1 enumerates CC's functional requirements, and Section 3.3.2 proposes a possible implementation that exploits a CMP's high-bandwidth, low-latency, on-chip communication network and flexible topology to reduce space, latency and complexity overhead. Other CC implementations are possible by extending various existing implementations, which are discussed in Section 3.3.3.

3.3.1 General Requirements

The functional requirements for CC are described before and summarized below.

- **Cache coherence extensions.** Beyond a conventional cache coherence protocol for non-inclusive caches (e.g., implemented by Piranha [12]), CC also requires support for cache-to-cache transfer of clean blocks and block spill. As part of the coherence protocol, this support has to be implemented correctly.

- **Cache replacement extensions.** Data classification information needs to be created, exchanged and maintained, while L2 cache replacement logic uses such information to support priority-based replacement. Because these modifications only affect the selection of eviction candidates, an incorrect or slow implementation should only cause performance degradation rather than correctness problems.

- **Extensions to support throttling.** To allow L2 cache bypassing, L1 caches need to directly write to the L2 write-back buffer (assuming write-back caches), which is straightforward to implement. CC also adds (1) extra states to track the amount of capacity used by each core (which can be imprecise) and (2) extra logic to decide whether to use cooperation and whether/where to spill. Such extensions do not affect correctness, thus can be imprecise or slow.

3.3.2 Cluster-based CMP Organization

In this section we detail our proposed implementation of CC using a specialized, on-chip, centralized directory, which will be used to evaluate the performance of both private cache organization and CC in Chapters 4 and 5. We focus on the implementation of cache coherence extensions because they are critical for correctness. It should be noted that this design can be used for other CMP systems while CC can also be implemented in various other ways.

Centralized On-chip Directory

Our design is based on a directory protocol, which has two advantages over a snooping protocol. (1) Latency. In a snooping protocol, every L2 miss incurs long-latency arbitration overhead to gather responses from all

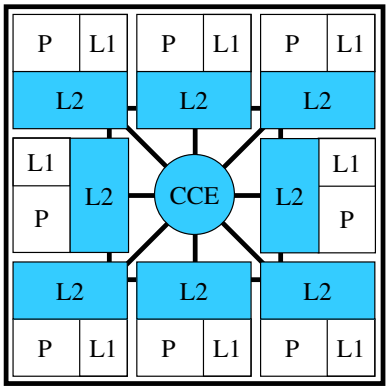


Figure 3.4: Private Caches with a Centralized Directory

on-chip caches. A directory protocol can reduce such overhead into a request transfer and a directory lookup. The reduced network switching activities can also save active power. (2) Bandwidth. Compared with snooping, a directory protocol can significantly reduce the number of broadcast requests and network bandwidth requirement. However, implementing a directory protocol has to solve two challenges: directory storage overhead and protocol design complexity.

The proposed implementation is based on a MOSI directory protocol to maintain cache coherence, but improves over a traditional directory-based system in several ways: (1) To reduce storage requirements, the directory memory for private caches is implemented by duplicating the tag structures of all private caches, requiring only 3% extra cache space (Table 3.1); (2) The directory is centralized to serve as the only serializing point for cache coherence, which can greatly simplify the implementation of the directory protocol; (3) Located at the center of the chip, the directory can provide fast access to all caches; (4) The directory is connected to individual cores using a special point-to-point ordered request network, separate from the network connecting peer caches for data transfers.

Figure 3.4 illustrates the major on-chip structures for an 8-core CMP. The Central Coherence Engine (CCE) embodies the directory memory and coherence engine, whose internal structure and directory memory organization is shown in Figure 3.5. The CCE consists of spilling buffers and the directory memory connected with router queues for incoming and outgoing messages. The spilling buffer is organized as a

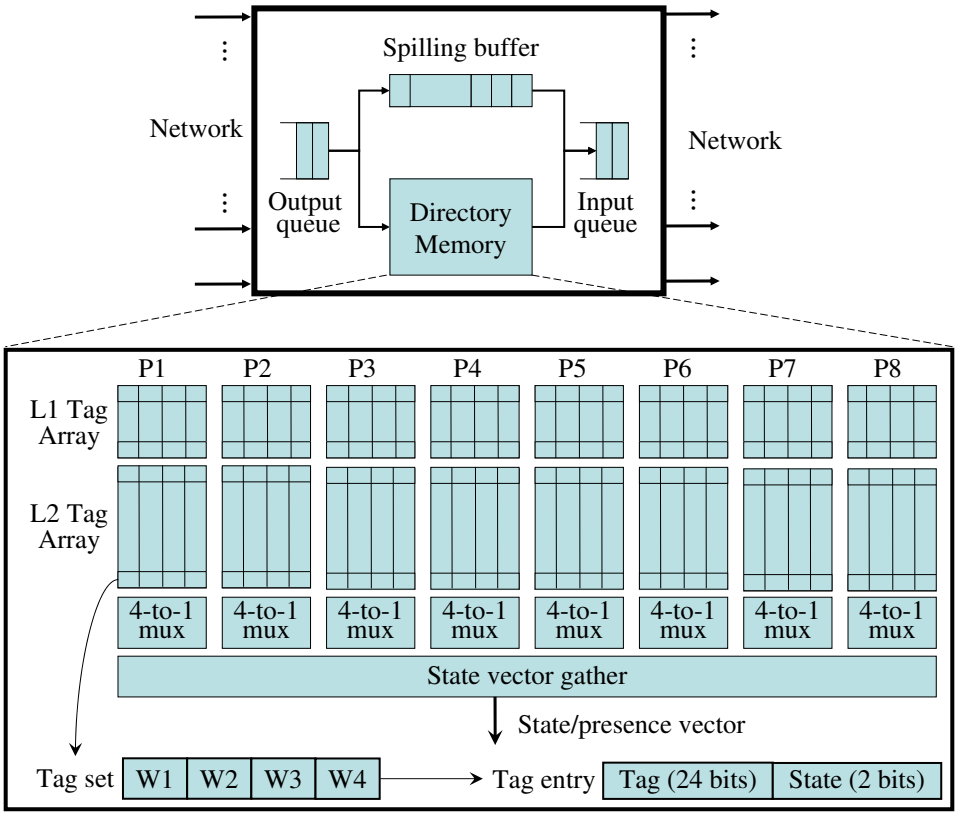


Figure 3.5: CCE and Directory Memory Structure (8-core CMP with 4-way associative L2 caches)

circular buffer, where each valid entry stores an in-flight spilling block (data and state) and its host cache ID. The lookup, insertion and deletion of spilling buffer entries will be discussed later (in Section 3.3.2). The CCE's directory memory is organized as a duplication of all private caches' tag arrays, similar to [112]'s remote cache shadow directory. Because CC requires the private caches to be non-inclusive, the CCE has to duplicate the tags for all cache levels. The tags are indexed in exactly the same way as in an individual core. A tag entry consists of both the tag bits and state bits.

In our implementation, the directory memory is multi-banked using low-order tag bits to provide high throughput. Incoming coherence requests trigger lookups in both the spilling buffer and the directory memory. A directory lookup will be directed to the corresponding bank, and to search the related cache sets in all cores' tag arrays in parallel. The results from all tag arrays are gathered to form a state/presence

vector as in a conventional directory implementation. An individual block's state may be updated according to the request type and current coherence state. The coherence engine will finish processing the request by generating requests for invalidation, data forwarding, or replies with data or acknowledgment messages. The latency of a directory lookup is expected to be almost the same as a private L2 cache tag lookup, since the extra gather step should only marginally increase the latency.

The coherence engine maintains each processor's duplicate tag arrays to reflect what blocks are stored in its corresponding local caches and what their states are. For non-inclusive caches, a cache block can remain in a processor core's private cache hierarchy (L1 and L2 caches) while moving frequently between local caches. CCE only keeps track of block installation and eviction from one processor's entire private cache hierarchy, because its correct operation only requires knowledge on whether a processor's private cache hierarchy has a block, but not its precise location within the hierarchy. However, lack of such information requires the CCE to carefully manage its tag arrays to avoid conflicts. Specifically, when allocating a new block in the directory, the CCE first attempts to allocate in the L2 tag array before filling the L1 tags. This is because for large L2 caches, blocks from multiple L2 sets can be mapped to the same L1 cache set and potentially cause an overflow in the L1 cache set. Filling the L2 tags first will guarantee the directory can find a free L1 tag when it is needed. Conversely, when CCE evicts a block from an L2 tag array, it also checks whether any L1 tags can be mapped and immediately moved to the corresponding L2 set. By keeping the L2 tag arrays as full as possible, the CCE ensures that no overflow occurs in its L1 tag arrays.

Table 3.1 lists the storage overhead for an 8-core CMP with a 4-way associative 1MB per-core L2 cache, 2-way 32K split L1 instruction/data caches, and 8-entry per-core spilling buffers. The tag bits storage overhead is estimated assuming a system having 4 Terabytes of physical memory, and a 128-byte block size. The total storage needed for extra tag bits (recording information used for cache cooperation), processor ID (used for quota-based cooperation throttling), duplicate tag arrays and spilling buffers is 271.5KB, increasing the on-chip cache capacity by 3.12% (or 6.10% for a 64-byte block size). This ratio is similar to

Component	Location	Size (KB)
Tag extension (2-bit)	Caches	17.0
Processor ID (3-bit)	Caches	25.5
Tag duplication	Directory	221.0
Spilling buffers	CCE	8.0
Total (3.12%)		271.5

Table 3.1: CCE Storage Overhead for an 8-core CMP with 1MB 4-way Per-core L2 Cache

Piranha [12] and lower than CMP-NuRapid [27]. Table 3.2 shows CCE's relative space overhead for several different CMP configurations. Although the absolute storage size increases with the number of cores and per-core cache size, the relative overhead remains stable and actually slightly decreases. We do not model the area of the separate point-to-point network as it requires the consideration of many physical constraints, which is not the focus of this dissertation. However, we believe it should be comparable to that of existing CMP's on-chip networks.

Configuration	Variable Parameter Value (Overhead)		
8-core, 4-way (varied L2 size)	512KB (3.30%)	1MB (3.12%)	2MB (2.98%)
1MB, 4-way (varied CMP size)	4-core (3.20%)	8-core (3.12%)	16-core (3.07%)

Table 3.2: CCE Storage Overhead under Different CMP Configurations

Cache Coherence Extensions

Besides maintaining cache coherence, the CCE also needs to support cooperation-related on-chip data transfers — (1) cache-to-cache transfers of clean data and (2) spills. The implementation of these functions is discussed below.

Sharing of clean Data. To support cache-to-cache transfers of clean data, the CCE needs to select a clean owner for a miss request. By searching the CCE directory memory, the CCE can simply choose any cache with a clean copy as the owner and forward the request to it, which will in turn forward its clean block to the requester. This implementation requires no extra coherence state or arbitration among the private caches. On the other hand, the CCE has to be notified when private caches replace clean blocks, in order

to keep the directory state updated. This requirement is met by extending the baseline cache coherence protocol with a "PUTS" (or PUT-Shared) transaction, which notifies the CCE about the eviction of a clean block. On receiving such a request, the CCE will invalidate the block in the corresponding core's tag arrays.

Spill. Figure 3.6 illustrates two implementations of spill using coherence messages communicated among the spilling cache, CCE and host cache. In a pull-based implementation (Figure 3.6 (A)), a local victim is locally buffered while the evicting cache notifies a randomly chosen host cache to fetch the block. The host cache then issues a special prefetch request to pull the block. In addition to serving the prefetch request, the spilling cache also transfers the state bits and removes its local copy (thus migrating the block to the host cache). The implementation of pull-based spill is straightforward as most modern processors support prefetching. As shown in Figure 3.6 (B), push-based spill consists of two pairs of data transfer and acknowledge messages. The first transfer is the same as a normal write-back initiated by the private cache. Upon receiving a spilled block, the CCE temporarily buffers the data in its spilling buffer and acknowledges the sender. The second transfer ships the block from the CCE to the chosen host cache. The host cache treats the incoming data similarly as a demand request, allocates space for it by possibly replacing another block, then acknowledges the directory to release the spill buffer. Race conditions can occur when the host cache issues a request for the spilled block during the second data transfer, in which case the CCE will receive the request message instead of the acknowledgment. The CCE handles it by searching the spilling buffer and releasing the entry with both matching block address and host cache ID. Similar as update-based coherence protocols, push-based spill can have deadlock issues, which are often solved by using different virtual channels for different types of communications. We have implemented push-based spill in our simulator to prove its feasibility, and avoided deadlock by using a dedicated virtual channel for block spilling.

Data Classification. Because the coherence engine can observe all on-chip transactions, it has been used extensively to detect sharing, communication and synchronization patterns [31, 70, 82, 89, 90, 92, 93,

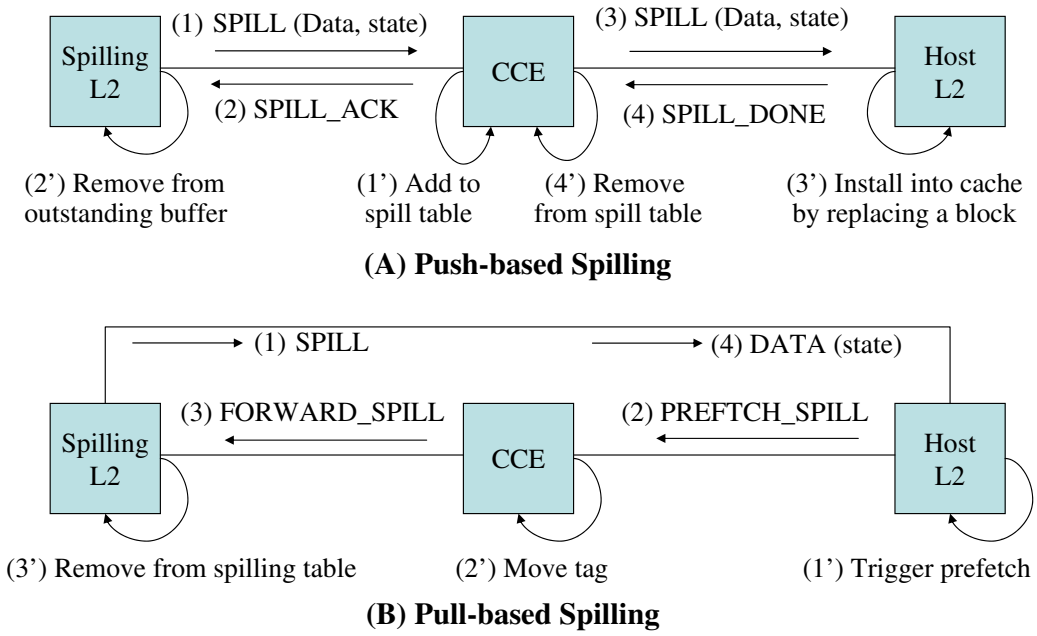


Figure 3.6: Push- and Pull-based Spill

110, 126, 140]. The CCE can classify data according to their coherence states or behaviors, and use such information for CC's prioritized cache replacement. For example, Chapter 4 presents replication-aware cache replacement that tries to keep unique data on chip. CCE detects unique on-chip copies when a write-back leaves only one cache holding the data, and communicates this information to that cache with a notification message.

3.3.3 Other Implementation Options

This section discusses other possible implementations of the three key components in CC's mechanism layer: the private cache organization, cooperation mechanisms and throttling mechanisms.

Cache Coherent On-chip Private Caches

Cache coherence among on-chip private caches can be maintained by snooping or directory based protocols, as well as token coherence [101]. Multiple private caches can be connected via snooping buses as in

traditional SMPs, and CMPs such as IBM Power4 and Power5 systems [148]. Non-inclusive L1/L2 caches are also supported by previous CMP designs. For example, Piranha [12] uses shadow tags to encode L1 cache states on the shared L2 cache side. Similarly, each private L2 cache can include duplicated L1 tags (with cache states) to simplify the implementation of cache coherence. Snooping requests can be filtered by simultaneously looking up both L2 and duplicate L1 tags, and forwarded to an L1 cache only if the data is actually stored in it. A similar implementation can be used for token coherence, as demonstrated in [13], while further optimizations can use soft-state directory information to reduce the number of broadcast requests.

Implementing private caches with a directory protocol is less straightforward, because the naive implementation of on-chip directory memory can incur prohibitive storage overhead. A concise on-chip directory can be implemented by duplicating private L1 and L2 cache tags (as shown in Section 3.3.2), or using a sparse directory cache [55]. The latter approach works well for workloads with good directory reference locality, but misses in the directory cache either incur expensive off-chip misses or require eviction of valid on-chip blocks.

Many options exist to support on-chip sharing of clean data. The clean owner can either be encoded in the protocol as a new state, or selected via arbitration of peer caches in a snooping protocol [148], or chosen by the directory if it keeps track of clean data write-backs (Section 3.3.2).

Supporting Cooperation

Spill-based cooperation can be implemented in either a “push” or “pull” strategy, which have been discussed in Section 3.3.2.

Both placement and replacement based cooperation need to associate policy-specific information with individual cache blocks, which can be recorded in the cache tag with a few bits. These bits are initialized upon cache allocation, possibly using information associated with the newly arrived data block. Updates of

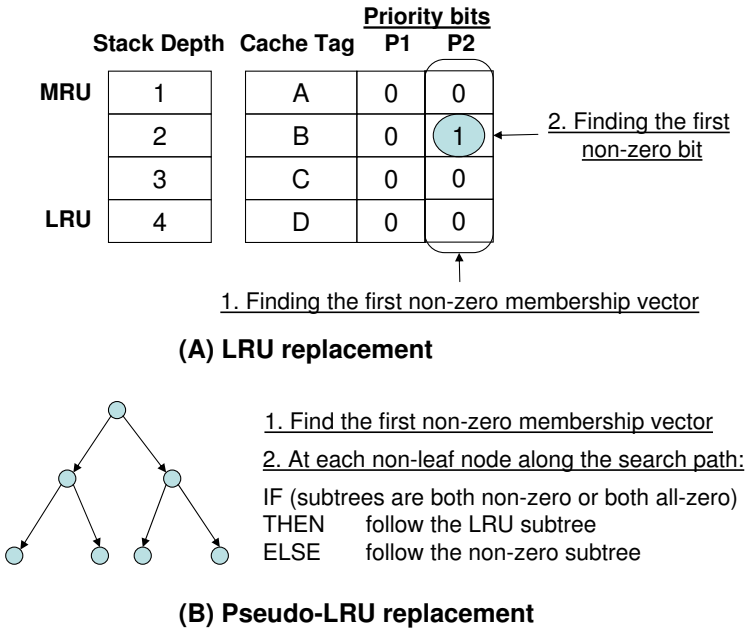


Figure 3.7: Implementing Priority-based Replacement (N=3, M=4)

these bits are triggered by external events, which can either be actively observed by caches in a snooping protocol or generated by the directory when it detects state changes. Because these bits are only used to make caching decisions, and not involved in cache coherence and computation efforts, they are allowed to be imprecise or out-dated without causing correctness problems.

Prioritized replacement can be implemented with extra circuitry in the cache replacement logic. Figure 3.7 illustrates its integration with two representative cache replacement policies, assuming `N` classes of data are prioritized in an `M`-way associative cache.

- **LRU replacement.** As shown in Figure 3.7 (A), each cache block is associated with `N-1` priority bits, each bit indicating whether it belongs to a certain data class between priority `1` and `N-1`. A block belongs to class `N` (the highest priority level) if all priority bits are `0`. We can also view the array of priority bits in a cache set as `N-1` class membership vectors (each vector has `M` bits corresponding to the `M` blocks). In a stack-based implementation [105], a block's priority bits move along with it to reflect changes of the block's position in the LRU stack. The replacement candidate is selected from

the lowest-priority non-zero membership vector, and the victim should have the lowest stack position among the blocks belonging to the selected priority class. This implementation can also be used for other stack-based replacement algorithm (such as random replacement [105]).

- **Pseudo-LRU replacement.** Figure 3.7 (B) shows a tree-based pseudo-LRU implementation [136]. To integrate with priority-based replacement, we first select the lowest-priority non-zero membership vector. At each non-leaf node along the binary search path, the LRU-based search logic is augmented to also consider priority information. Specifically, the augmented logic selects the LRU subtree if the membership vectors for both subtrees are simultaneously non-zero or all-zero; otherwise, it selects the subtree with a non-zero membership vector.

The size and complexity of these extra circuits grow with both levels of priority `N` and cache associativity `M`. We set (`N, M`) to be (2, 4) in this dissertation, while expecting `N` to be less than 4 and `M` no more than 8, so these changes can only add minimal storage and latency overhead. More aggressive assumptions are made by other CMP caching proposals because write-backs are not on the critical path and they have negligible performance impact. For example, victim replication [159] uses a 4-level prioritized replacement in a 16-way set-associative shared cache.

Supporting Throttling

To support cooperation throttling, each private cache should include two sets of registers: the "knob" registers to store specified probabilities or quotas, while the "measurement" registers to save performance measurements which are fed back to adjust the degree of cooperation throttling. For quota-based throttling, every cache block includes a processor-ID field to indicate for which core’s data it stores. Each private cache maintains a set of counters to reflect the number of cache blocks used by each thread, as well as the number of invalidated/unused blocks. By periodically sharing such information among different cores, CC can monitor the capacity usage of different cores.

Cooperation throttling can be either static or dynamic. Setting the throttling knobs statically is straightforward and requires no special support. Dynamic throttling consists of a feedback loop where current throttling performance and program behavior changes are used to adjust the degree of future throttling. In this dissertation, we assume that throttling decisions are made by the hardware, but CC itself has the flexibility to support software controlled adaptation. The software can periodically read the "measurement" registers, make adaptation decisions and update the "knob" registers, while CC is responsible for enforcing the specified throttling decisions.

3.4 CC for Large Scale CMPs

The advent of CMPs has changed the scaling trend from boosting frequency into increasing the number of on-chip cores [119]. With Intel announcing its 5-year 80-core CMP plan [73] and Rapport Inc.'s shipping of single-chip with 256 mini-cores [128], computer architects are now starting to consider how to build and use CMPs with 1000 cores in a few technology generations [7]. In this section, we discuss several directions in improving the scalability of CC and outline a possible implementation of CC for large scale CMPs. This proposal is by no means the best approach, but only serves as our first step towards the design of many-core systems.

3.4.1 Directions to Improve CC's Scalability

We believe that CC's private cache organization is essential for highly scalable CMPs due to its modularity and locality benefits. It will be difficult for a shared cache to support hundreds of threads, because the L1 miss traffic can saturate the on-chip network, and the needed cache associativity to avoid inter-thread conflict misses may incur prohibitive overhead. CC addresses the off-chip bandwidth bottleneck of private caches through inter-cache cooperation, which may limit the scalability in two aspects. The first bottleneck for current CC design is its central-directory based implementation. As the number of cores managed by the CCE increases, contention within CCE will delay cache coherence operations and cooperation activities.

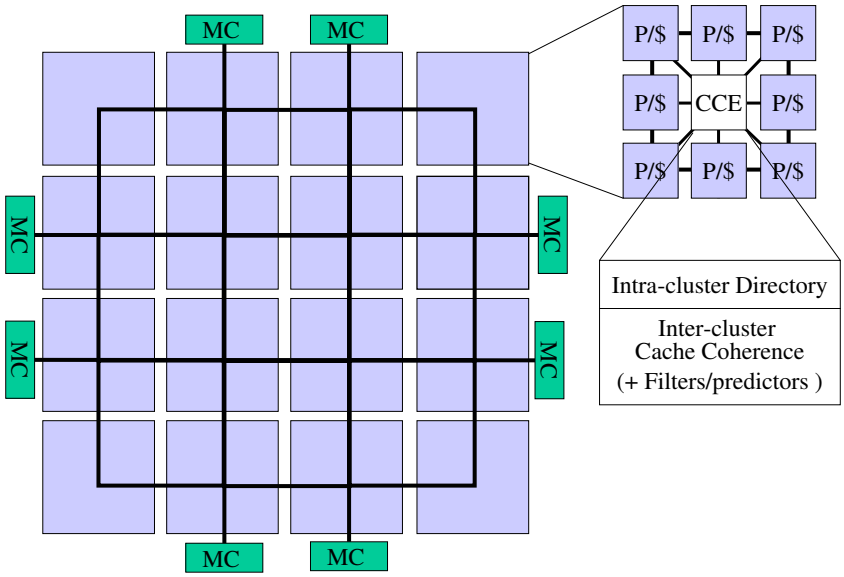


Figure 3.8: 128-core CMP with 16 8-core Clusters

Scalability barriers also exist at the policy level even if we have a scalable cache coherence protocol. As the number of cores increases, the average latency for cooperative capacity sharing among all cores (e.g., via spilling/reusing) will also grow, eventually hitting a point where the capacity benefit provided by global sharing is offset by the growing on-chip communication overhead. Cooperative capacity sharing policies should consider such tradeoffs and limit cooperation within a group of closely located caches. Other cache optimizations may also prefer such a scoping policy if their algorithms cannot scale to hundreds of cores.

A natural way to accomodate these requirements is to build large-scale CMPs via composition of small-scale clusters (e.g., 4-8 cores) and reuse the current CC design within each cluster. Comparing to a flat directory protocol possibly embedded in a mesh-base on-chip network, the hierarchical design can significantly improve latency and reduce bandwidth by exploiting intra-cluster data/communication locality. This approach provides a smooth transition path for small-scale workloads because it requires no extra modifications of cooperation policies and incurs little extra performance overhead. Figure 3.8 illustrates such a hierarchical design for a 128-core CMP with 16 8-core clusters. Within each cluster, the CCE is augmented to maintain cache coherence at two levels. Intra-cluster coherence is provided by the central

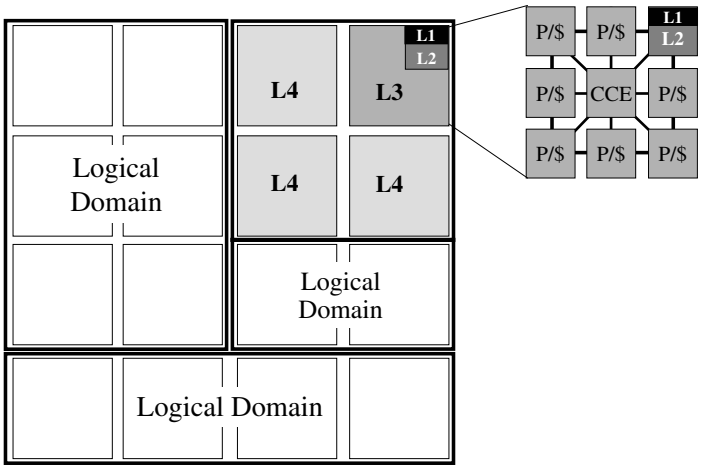


Figure 3.9: Logical Cooperation Domains (L3 = the aggregate cache within a cluster; L4 = the aggregate cache within a logical domain.)

directory (CCE) as discussed in Section 3.3.2, and inter-cluster coherence is achieved with a directory organization where, through address space partitioning, each CCE serves as the home node for a fraction of physical addresses. Multiple memory controllers (MC) are used, each responsible for servicing DRAM accesses generated by one or more neighboring inter-cluster directories.

This design mitigates the implementation bottleneck by limiting the number of cores within each clusters, while encapsulating each cluster as a single core with private caches to build larger systems. Such encapsulation not only reduces inter-cluster traffic for data forwarding, block invalidation, and write-back, but also decouples intra- and inter-cluster coherence, allowing flexible combination of coherence protocols at different levels.

Treating a cluster as a single core (by aggregating reference streams and cache resources) also allows the reuse of previously proposed cooperation mechanisms and policies across multiple clusters. Instead of using a fixed scope of cooperation defined by the cluster boundary, inter-cluster cooperation will take place in “logical domains” (as shown in Figure 3.9), which can be statically or dynamically formed according to management domains, communication patterns, or data locality. With cooperative caching at different

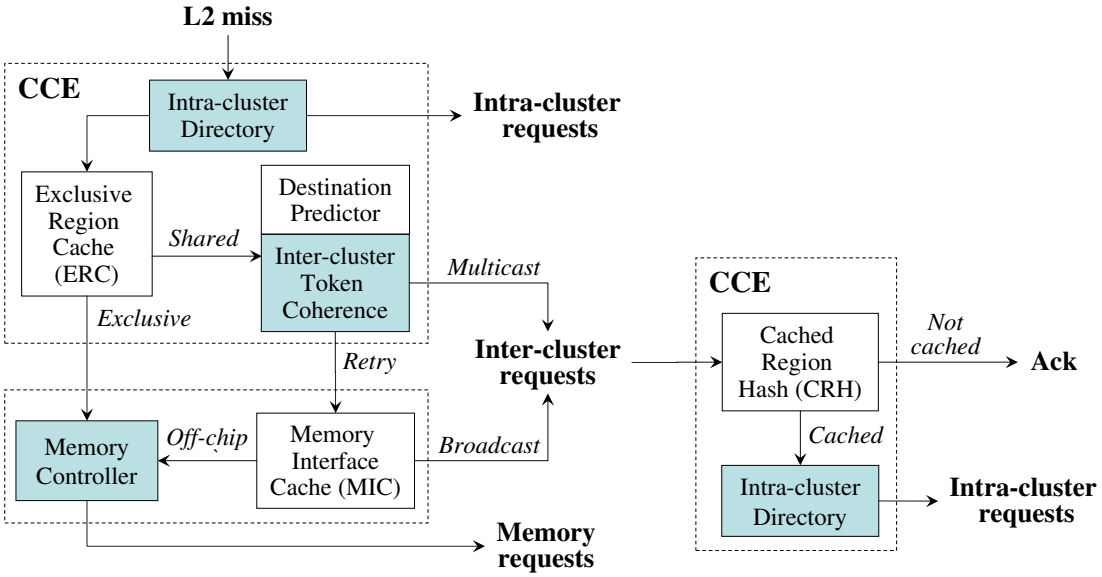
levels, a single core can have a deep on-chip cache hierarchy consisting of its private L1 and L2 caches, as well as the L3 and L4 caches formed through intra- and inter-cluster cooperation.

3.4.2 An Implementation of Large-scale CMP

Below we describe a cluster-based implementation of large-scale CMP in details. Most the techniques used here have been evaluated by other researchers for small-scale multiprocessors (e.g., no more than 16 nodes) [21, 101, 103, 104, 109]. We leave their evaluation under large-scale CMPs for future work because evaluating large-scale CMPs is still an open research question [7, 65].

Figure 3.10 shows the key components of this implementation, as well as their interaction when servicing (A) L2 cache misses and (B) inter-cluster coherence requests. The required components (shaded in the figure) are (1) intra-cluster directory and (2) inter-cluster token coherence engine which are collocated in the CCE, and (3) the memory controller for off-chip DRAM accesses. The other components are optional filters and predictors to improve performance and scalability.

At a high level, an L2 miss is handled in the following steps. First the CCE receives the miss request, looks up in the intra-cluster directory and generate intra-cluster forward or invalidation requests if the miss can be satisfied by other caches in the local cluster. If the request misses in its local cluster (the L3 cache in Figure 3.9), it will check the Exclusive Region Cache (ERC) to see whether the coarse-grain region (that the block belongs to) is exclusively cached by the local cluster. If the region is exclusively cached, implying that the miss cannot be serviced on-chip, the miss request is directly sent to the memory controller for off-chip memory access. Otherwise, the region is potentially shared by other on-chip clusters, the token coherence engine will try to service the miss via inter-cluster coherence requests. The token coherence engine predicts a set of destination clusters (the L4 cache in Figure 3.9), and multi-casts the request. The multicast request can either be satisfied, or needs to be retried by the Memory Interface Cache (MIC) if none of the destination clusters can serve it. Depending on whether the block has on-chip copies, the MIC will either generate an



(A) L2 Miss Handling (B) Inter-cluster Request Handling

Figure 3.10: A Possible Implementation for Large-Scale CMPs

on-chip broadcast request, or an off-chip memory request which will eventually satisfy the miss request.

Below we describe the operation and implementation of different components. As we have already discussed the internals of intra-cluster directory and memory controller, we only cover the remaining components used for large-scale CMPs.

- **Exclusive Region Cache (ERC).** ERC records frequently accessed coarse-grained regions (continuous, aligned, power of 2 sized memory areas) [109] that are exclusively cached within the local cluster. When an L2 miss enters the CCE (Figure 3.10 (A)), it first triggers a lookup in the intra-cluster directory. If the block hits in the local cluster, the directory will issue intra-cluster request messages; otherwise, the ERC is checked. The ERC directly sends out an off-chip request to the memory controller if the block belongs to a region exclusively stored in the local cluster, therefore filtering unnecessary inter-cluster traffic via the early detection of on-chip misses. Previous research has observed significant coarse-grained exclusive caching for both multiprogrammed and multithreaded workloads [21, 109] and shown that small ERCs (64-entry per 1MB cache for 16K regions) are

sufficient for effective filtering.

- **Inter-cluster token coherence.** We employ token coherence [101,103] to keep inter-cluster cache coherent (Figure 3.10 (A)) because it allows direct communication between the miss-causing cache and neighboring sharers without expensive intervention of global ordering points (e.g., bus or directory). A destination-set predictor [100] is used to predict the destination cluster sets for misses in the local cluster, which are used to generate multicast requests and avoid broadcast on the global network. By sourcing data directly from close-by sharers to the requesting cache, this design reduces average cache latency for workloads having significant sharing among neighboring clusters or between predictable producer-consumer pairs. For workloads that cannot exploit token coherence, we use the memory controller as the global ordering points to broadcast requests. We make two modifications to the original token coherence [101] for better scalability: (1) Each cluster is treated as one cache to reduce the token size and predictor storage; (2) For read requests (GETS), the multicast request is implemented as one cruise-control message (as used by Piranha [12] for on-chip invalidation) which sequentially visits each cluster in the destination set until it finds the first sharer. This optimization can further reduce latency by fetching data from the closest cluster.

- **Memory Interface Cache (MIC).** Requests are routed to the memory controller if initial searches in their local cluster and predicted destination clusters fail (Figure 3.10 (A)). For correctness and performance reasons, off-chip DRAM accesses should be avoided if on-chip copies exist. We adopt the memory interface cache proposed by Marty and Hill [104] to maintain such information, thus avoiding the complexity and space overhead of using an extra chip-level directory. Here, each memory block is augmented with an owner bit to indicate whether the memory should respond to on-chip requests. This bit is cleared (as 0) when data is first fetched on-chip, and set to 1 upon off-chip write-back. The MIC records on-chip regions with their block-level owner-bit vectors. When a request misses in the MIC, the memory controller will read the owner-bit from memory and fetch the data if

the memory should respond. Otherwise, a broadcast is issued to find the on-chip copies. To maintain the owner-bit, replacing a block with partial tokens needs to merge its tokens with other on-chip copies, as did in [104].

- **Cached Region Hash (CRH).** Figure 3.10 (B) shows the processing of inter-cluster request within a CCE. We use a Bloom filter [18] called Cached Region Hash (CRH) [109] to filters unnecessary directory lookups to save latency, bandwidth and power. CRH records a superset of regions cached in the local cluster using a small bit vector (1KB per 1MB cache) and can filter most unnecessary region lookups. The intra-cluster directory is only accessed if the request hits in the CRH, which generates either intra-cluster forwarding requests, or an acknowledgment if the data is not cache in the cluster.

From the viewpoint of a processor core, the proposed large-scale CC design can effectively support all levels of on-chip cache hierarchy (L1 to L4 caches as shown in Figure 3.9) because it avoids higher-level cache accesses and global communication whenever a miss request can be serviced by a lower-level cache. We believe that this design can improve scalability in several important scenarios: (1) locality is exploited at the following three cache levels to reduce global communication: private L2 cache, caches in a cluster managed by the intra-cluster directory, and neighboring clusters glued with token coherence (e.g., logical domains and stable sharers); (2) with ERC, no coherence overhead is incurred for exclusively owned data; (3) global ordering points (here the memory controller) are accessed only when global communication is needed; (4) off-chip accesses are reduced through on-chip cooperation and by the MIC structure.

The drawback of this design is its inefficiency for global communication, which can at worst involve both a multicast message generated by the token coherence protocol and a broadcast message generated by the global ordering point. To support workloads with less data and communication locality for CMPs with kilo-cores, future research is needed in understanding large-scale parallel workloads, and providing scalable communication infrastructure (e.g, high-dimensional interconnection networks), programming models (e.g., hybrid between shared memory in small scopes and message passing across clusters) and cache management

policies.

3.5 Summary

To meet both performance and non-performance related, potentially conflicting cache requirements, a wide spectrum of application/optimization specific cache resource sharing behaviors are needed. Because neither private nor shared cache organization can answer these challenges, we advocate a unified framework to manage the aggregate CMP on-chip cache resources.

The proposed CC framework includes three key mechanisms. (1) Private cache based organization provides both latency/bandwidth benefits and resource isolation. CC also removes the inclusion restriction within a processor core's multi-level private cache hierarchy for flexibility and supports cache-to-cache transfers of clean data to avoid unnecessary off-chip misses. (2) Cache placement and replacement based cooperation mechanisms enable inter-cache resource sharing. (3) Probability and quota based throttling mechanisms can orchestrate and control cooperative resource sharing.

Cooperative policies can thus combine these core mechanisms in various ways to suit the resource sharing needs for specific workloads and optimization goals. CC can also be implemented in different ways. As an example, we propose a cluster-based CMP organization using a on-chip central directory, and discuss possible extensions to support large-scale CMPs. Chapter 4 and 5 will elaborate on how to combine CC mechanisms with innovative cooperation policies for memory latency reduction and inter-thread interference isolation.

CHAPTER 4

LATENCY REDUCTION VIA COOPERATIVE CACHING

In this chapter we extend the CC framework with cooperation policies to reduce the average memory access latency. In Section 4.1 we motivate the problem and outline our proposed solution. Section 4.2 describes cooperation policies to reduce the number of long-latency off-chip misses. An evaluation of our approach is presented in Section 4.3, and we conclude with Section 4.4.

4.1 Motivation and Proposed Solution

4.1.1 Motivation

An important goal of a cache hierarchy is to improve performance by reducing the average memory access latency. For CMPs, the average memory access latency can be broken into cycles spent at different levels of the memory hierarchy: (1) local L1 caches; (2) local L2 cache, which can either be a processor core's private L2 cache or local L2 banks as part of a shared cache; (3) remote caches, which are on-chip caches that are not local to the requesting processors; and (4) off-chip storage (any external caches and DRAM). Correspondingly, Equation 4.1 calculates the average memory latency, where P_c denotes the probability that a memory access hits in cache level *c*, and L_c denotes the (round-trip) hit latency to cache level *c*. For a given workload and processor configuration, memory latency reduction usually correlates to performance improvement because the number of memory accesses to complete a certain amount of work (measured as the number of committed instructions or user-defined transactions) is mostly determined[1].

[1]Workload variability [2] can change the number of memory accesses per unit of work, usually due to spin locks and idle loops. However, such effects are largely filtered by private L1 caches commonly used in CMP designs, and we use the same methodology as suggested in [2] to compensate the effects that workload variability has on simulation results.

$$Latency = (P_{L1} * L_{L1}) + (\boldsymbol{P}_{LocalL2} * L_{LocalL2}) + (\boldsymbol{P}_{RemoteL2} * L_{RemoteL2}) + (\boldsymbol{P}_{Mem} * L_{Mem}) \quad (4.1)$$

Several terms in Equation 4.1 can be viewed as constants across different CMP caching schemes. First, the access latencies of various cache/memory levels are fixed by their tag/data array lookup time and network latencies[2]. Second, the L1 cache hit ratio is constant because L1 caches are similar across various CMP caching schemes and usually independent of the organization and management of L2 cache resources. These factors have left the probability distribution of L1 cache misses (or the relative hit ratios to L2 caches and memory) as the determining factor of CMP caching performance. These terms are marked bold in Equation 4.1.

This aspect is exactly where various CMP caching proposals differ. Pure private caches make efficient use of their portion of on-chip cache resources, leading to more local L2 hits (higher $\boldsymbol{P}_{LocalL2}$) as well as potentially more off-chip accesses (higher $\boldsymbol{P}_{Mem}$); a shared cache can reduce off-chip misses by storing data across the chip, but a local bank is only able to satisfy a fraction of total L1 misses (lower $\boldsymbol{P}_{LocalL2}$). Shared cache based hybrid schemes try to exploit data replication and migration to increase local L2 hit ratio (higher $\boldsymbol{P}_{LocalL2}$ and lower $\boldsymbol{P}_{RemoteL2}$), without significantly increasing off-chip accesses [15,27,94,115,158,159]. Private cache based hybrid schemes attempt to make use of remote on-chip caches (increasing $\boldsymbol{P}_{RemoteL2}$ and decreasing $\boldsymbol{P}_{Mem}$), while retaining higher local L2 hit ratio [14,59,68,138]. Depending on the aggregate caching requirement of the workloads, these schemes can deliver different results: small workloads that can be cached by a local L2 cache do not need the larger capacity of shared cache; larger workloads prefer shared cache based or hybrid designs to reduce off-chip misses.

In order to suit the diverse requirements of different workloads, an ideal CMP caching scheme must

[2]This is only a first-order approximation because congestion in the network and memory controller can cause extra delays.

be able to: (1) provide a spectrum of options between the two extremes of private and shared cache designs, and (2) dynamically adapt to the best sharing point for a given combination of workload and system configuration.

4.1.2 Proposed Solution

We try to optimize the average latency of memory requests with CC by combining the strengths of private and shared caches adaptively. This is achieved in three ways: (1) by using private caches as the baseline organization, CC attracts data locally to reduce remote on-chip accesses, thus lowering the average on-chip memory latency; (2) via cooperation among private caches, it can form an aggregate cache having similar effective capacity as a shared cache, to reduce costly off-chip misses; (3) by cooperation throttling, it can provide a spectrum of choices between the two extremes of private and shared caches, to better suit the dynamic workload behavior.

Our approach attempts to manage the aggregate on-chip caches with a set of unified heuristics. By mimicking a shared cache, CC does not distinguish between different sharing types (e.g., private, read-only, read-write) or treat individual threads separately (according to their different working set sizes and locality characteristics). The proposed cooperation policies are conceptually simple, only requiring modifications to the default cache placement and replacement policies, and are easily supported by the CC framework.

4.2 Policies to Reduce Off-chip Accesses

Because CC's baseline organization already uses private caches to reduce cache latency, we now consider cooperation policies to efficiently use the aggregate on-chip cache resources and thereby reduce the number of off-chip accesses. We choose to mimic the caching behavior of a shared cache with a group of cooperative private caches. Compared with private caches, a shared cache makes more efficient use of available capacity in three ways, corresponding to the three cooperation polices we discuss in this section.

First, a shared cache uses all on-chip data (both dirty and clean) to satisfy processor requests. On the contrary, traditional private caches only support the sharing of on-chip dirty data. As discussed in Section 3.2.1, CC matches a shared cache by supporting cache-to-cache transfer of clean data. Because unnecessary off-chip accesses are removed when there are clean copies residing elsewhere on the chip, we show in Section 4.3 that this baseline CC design can significantly outperform conventional private caches.

Second, a shared cache eliminates replication by storing only one copy of each unique data, while private caches may keep multiple copies of the same data on-chip. We introduce a cooperation policy that replaces replicated data blocks to make room for unique on-chip copies (called **singlets**), thereby making better use of the available cache resources.

Third, a shared cache observes references from all processor cores and chooses replacement victims globally. Consequently, different cache capacities are allocated to different threads according to their requests[3]. On the other hand, cache capacities are statically allocated to different threads in a private cache organization, and each private cache can only observe requests and select replacement victims locally. Using CC's spill mechanism, the last cooperation policy combines local replacement policies with global spill/reuse history to approximate a global replacement policy, thereby keeping potentially more useful data on the chip.

Because Chapter 3 has described the details of cache-to-cache transfer of clean data, below we will only discuss the two new cooperation policies and their throttling.

4.2.1 Replication-aware Cache Replacement

The baseline private L2 caches employed by CC allow replication of the same data block in multiple on-chip caches. When cache capacity can sufficiently satisfy the program's working set requirement, replication reduces cache access latency because more requests can be satisfied by locally replicated copies. However,

[3]The model here is that a thread runs on a processor core, whose requests are filtered by L1 private caches and triggers capacity allocation in the shared cache.

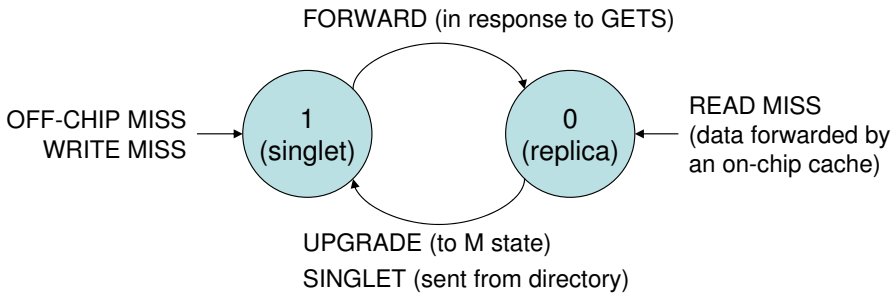


Figure 4.1: State Diagram for the Singlet Bit

when cache size is dwarfed by the working set size, replicated blocks will compete for the limited capacity with unique copies. CC uses replication-aware data replacement to optimize capacity, which discriminates against replicated blocks in the replacement process. This policy aims to increase the number of unique on-chip blocks, thus improving the probability of finding a given block in the aggregate on-chip cache.

We define a cache block in a valid coherence state as a **singlet** if it is the only on-chip copy, otherwise it is a **replica** because multiple on-chip copies exist. We employ a simple policy to trade off between access latency and capacity: evict singlets only when no replicas are available as victims. This can be implemented by CC using prioritized cache replacement. All on-chip data are classified as either singlets or replicas, and replicas are first selected when choosing victims.

With CC, a singlet block evicted from a cache can be further spilled into another on-chip cache. Using the aforementioned replacement policy, both invalidated and replica blocks in the receiving cache are replaced first, again reducing the amount of replication. By giving priority to singlets, all private caches cooperate to replace replicas with unique data that may be used later by other caches, further reducing the number of off-chip accesses.

To indicate whether a block is a singlet, each cache tag is augmented with a singlet bit. This bit is advisory and not needed for correctness. Figure 4.1 describes the state diagram for the singlet bit, which can be initialized in two ways: (1) it is set to 0 if the block is first fetched from off-chip memory or as a result of write miss (assuming an invalidation based coherence protocol), or (2) it is set as 1 if the block is forwarded

from other caches. In the second case, the forwarding cache also resets its singlet bit to 0, indicating the data now has replicas. The singlet information is also communicated from the directory to on-chip caches: when the directory receives a write back message, or a PUTS message which indicates the eviction of a clean block, it checks the presence vector to see if this action leaves only one copy of the data on-chip. If so, an advisory notification message (SINGLET) is sent to the cache holding the last copy of the block, which can set the block's singlet bit to 1.

4.2.2 Global Replacement of Inactive Data

Spilling a victim into a peer cache both allows and requires global management of cooperative private caches. The aggregate cache's effective associativity now equals the aggregate associativity of all caches. For example, 8 private L2 caches each with a 4-way associativity effectively offers a 32-way set associativity for CC to exploit.

Similar to replication-aware data replacement, we want to cooperatively identify singlet but inactive blocks, and keep globally active data on-chip. This is especially important for multiprogrammed workloads with heterogeneous access patterns. Because these applications do not share data and have little operating system activity, almost all cached blocks are singlets after the initial warmup stage. However, one program with poor temporal locality may touch many blocks which soon become inactive (or dead), while another program with good temporal locality but large working set will have to make do with its fixed, private cache space, frequently evicting active data and incurring misses.

Implementing a global-LRU policy for CC would be beneficial but is also difficult because all the private caches' local LRU information has to be synchronized and communicated globally. Practical global replacement policies have been proposed to approximate global age information by maintaining reuse counters [124] or via epoch-based software probabilistic algorithms [50]. We modify N-Chance Forwarding [32], a simple and fast algorithm from cooperative file caching research, to achieve global replacement.

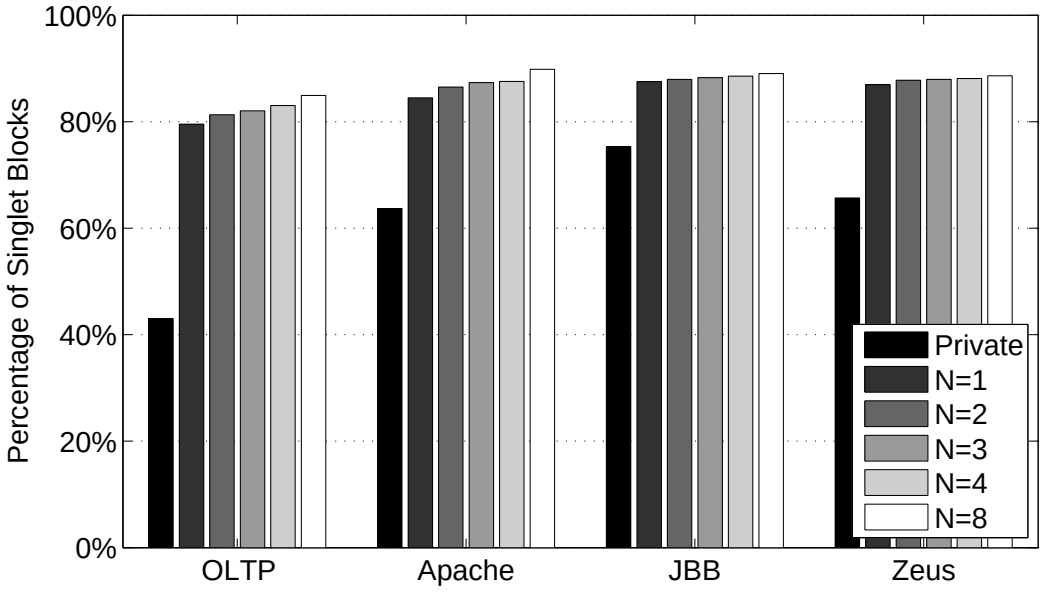


Figure 4.2: Percentages of Unique Cache Blocks in Different Schemes

N-Chance Forwarding was originally designed with two goals: it tries to avoid discarding singlets, and it tries to dynamically adjust each program's share of aggregate cache capacity depending on its activity level. Specifically, each block has a recirculation count. When a private cache selects a singlet victim, it sets the block's recirculation count to N, and forwards it to a random peer cache to host the block. The host cache receives the data, set it as the most recently used (MRU) entry in the chosen cache set and evicts the least active block in its local cache. The life cycle of the spilled block is thus extended, giving it new chances to compete with other cache blocks for on-chip space. If a recirculating block is later evicted, its count is decremented and it is forwarded again unless the count becomes zero. If the block is reused, its recirculation count is reset to 0. To avoid a ripple effect where a spilled block causes a second spill and so on, a cache that receives a spilled block is not allowed to trigger a subsequent spill.

The parameter N was set to 2 in the original proposal [32]. A larger N gives singlet blocks more opportunities to recirculate through different private caches, hence makes it more likely to reduce the amount of replication and improve the aggregate cache's effective capacity. We have studied CC schemes with different N values, and found that increasing N beyond 1 has little additional benefit for CMP caching. To

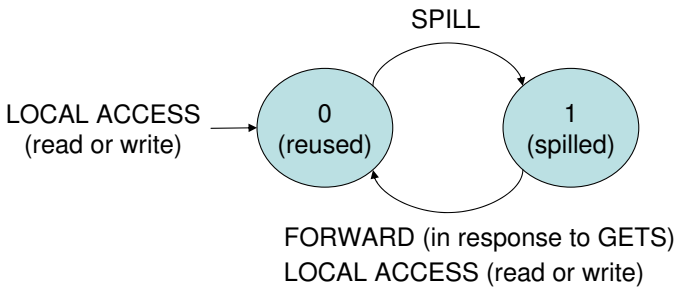


Figure 4.3: State Diagram for the Spilled Bit

explain this, we compare the percentages of singlet blocks under various N values in Figure 4.2. It shows that, for 4 commercial workloads with significant data sharing, CC with N=1 can achieve almost the same level of replication control as CC with N=8, therefore providing the same level of performance[4]. Figure 4.2 also shows that, even with N=8, N-Chance Forwarding cannot remove all replicas in the aggregate cache. This is because CC can only reduce replication in cache sets where singlets compete space with replicas, which may not cover all cache sets.

We therefore sets N to 1 in this dissertation, and call the modified policy **1-Fwd**. 1-Fwd dynamically balances each private cache's allocation between local data accessed by its own processor and global data stored to improve the aggregate cache usage. The active processors' local references will quickly force global data out of their caches, while inactive processors will accumulate global data in their caches for the benefit of other processors. This way, both capacity sharing and global age-based replacement are achieved.

Implementing 1-Fwd in CC is also simple. Each cache tag needs to be extended with one bit to indicate whether the block was once spilled but has not been reused. Figure 4.3 illustrates the operation of the spilled bit with a state diagram. This bit is initialized to 0 for blocks installed due to local accesses. It is set to 1 when a cache receives the spilled block, and reset to 0 if the block is reused by either local or remote processors. Similar to the singlet bit, the spilled bit is advisory and not needed for correctness.

[4]For multiprogrammed workloads with little replication, CC with larger N values perform essentially the same as CC with N=1.

4.2.3 Cooperation Throttling

At this point, CC can operate in one of two extreme modes: (1) shared cache mode by always using its cooperative capacity improvement policies, and (2) private cache mode that never uses cooperation policies. Now we use probability based cooperation throttling to provide a wide spectrum of caching behaviors between the two extreme modes, and discuss how to dynamically choose the best cooperation probabilities.

As discussed in Section 3.2.3, probability based throttling controls how often to apply the cooperation policies. In the context of memory latency reduction, CC behaves more like as a shared cache with higher cooperation probabilities and more like a group of isolated private caches with lower probabilities. Although cooperation throttling is needed for both multithreaded and multiprogrammed workloads, we focus on multithreaded workloads in this chapter, and apply quota based throttling in Chapter 5 to enforce isolation among multiprogrammed applications.

Several options exist in choosing the best cooperation probabilities for a given workload. Static tuning sets the optimal probability based on profile information, and dynamic tuning adapts by predicting the costs/benefits of various throttling degrees. Beckmann et al. [13, 14] examined the tradeoffs in balancing latency and capacity, and proposed adaptive selective replication (ASR) mechanisms and policies to reach the best replication level. ASR can be integrated with CC to optimize both homogeneous and heterogeneous workloads.

Alternatively, we can use dynamic set sampling (DSS) [122] to predict the memory latencies experienced under different cooperation probabilities simultaneously. The key intuition behind DSS is that a caching scheme's impact on on the whole cache can be accurately predicted by sampling its impact on a small fraction of cache sets. As shown in Figure 4.4, we divide each L2 cache into 5 disjoint cache set groups: 4 small sampling groups (each having 3% of the total cache sets) and one large group consisting of all the remaining cache sets. Each sampling group uses a different cooperation probability (0%, 30%, 70% and 100%, respectively), and periodically a global selector will choose the best performing sampling group

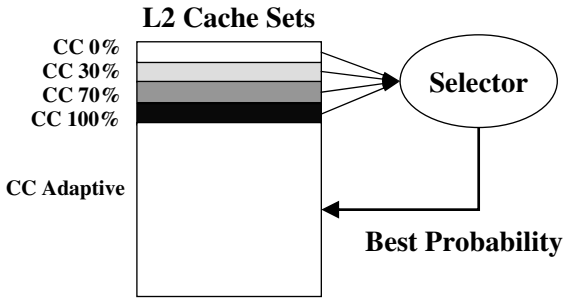


Figure 4.4: DSS-based Adaptive Throttling

(performance measured in average memory latency) and use its cooperation probability in the remaining cache sets.

In this dissertation we compare the average memory latencies of various throttling options by assuming strong correlation between memory latency and performance, similar to previous proposals [13, 122]. Techniques such as out-of-order processors, prefetching and memory-level parallelism optimizations can break this assumption by partially or totally overlapping cache misses with useful computation or concurrent memory accesses. More accurate prediction can be made by sampling the direct performance measurement such as IPC or user-specified throughput metrics (as suggested in [3]), which is left as future work.

4.3 Performance Evaluation

In this section we evaluate the effectiveness of CC using full-system simulations. We first describe our simulation and workload setup in Section 4.3.1, then present the performance, latency and bandwidth results for multithreaded commercial workloads and multiprogrammed SPEC2000 workloads in Sections 4.3.2 and 4.3.3, respectively. In Section 4.3.4, we evaluate CC's performance sensitivity with different system sizes, memory latencies and directory overheads. CC is then compared against the recently proposed Victim Replication (VR) scheme [159] in Section 4.3.5, using the same simulation parameters as in the original VR proposal. The benefits of adaptive cooperation throttling is evaluated in Section 4.3.6.

Table 4.1 summarizes the different setups we use in our evaluation. We use multiple setups to demonstrate

Variable	Values
Workloads	Multithreaded commercial workloads (OLTP, Apache, JBB, and Zeus) Multiprogrammed SPEC2000 workloads (heterogeneous and homogeneous) Multithreaded SPECOMP workloads (for VR comparison) Single-threaded MinneSPEC benchmarks (for VR comparison)
System sizes	4-core and 8-core CMPs
Processor models	4-way out-of-order, 12 FO4 cycle time [60, 64] (default) In-order blocking, 12 FO4 cycle time [60, 64] (for sensitivity study and throttling) In-order blocking, 24 FO4 cycle time [61, 139] (for VR comparison)
Cache sizes	1MB per-core L2 (default) 512KB and 2MB per-core L2 (for sensitivity study)
Memory latency	300 cycles (default) 600 cycles (for sensitivity study)

Table 4.1: Evaluation Scenarios

Component	Parameters
Out-of-order processor pipeline	4-wide issue, 10-stages
Instruction window / scheduler	128 / 64 entries
Branch predictors	12K YAGS + 64-entry RAS
Block size	128 bytes
L1 I/D caches	32KB, 2-way, 2-cycle hit latency
L2 caches/banks	Sequential tag/data access, 15-cycle hit latency
On-chip network	Point-to-point mesh network, 5-cycle per-hop latency
Main Memory	300 cycles total, 16 outstanding requests per core

Table 4.2: Processor and Cache/Memory Parameters

that CC achieves a robust performance advantage across many different processor, cache and memory configurations and a wide selection of workloads.

4.3.1 Simulator and Workload Setup

We use a Simics-based [99] full-system execution-driven simulator. The cache simulator is a modified version of Ruby from the GEMS toolset [102]. The processor module **ms2sim** is a timing-directed functional simulator that models modern out-of-order superscalar processors using Simics Microarchitecture Interface (MAI). Table 4.2 lists the relevant configuration parameters used in our default simulation setting.

For benchmarks, we use a mixture of multithreaded commercial workloads and multiprogrammed SPEC

Multiprogrammed (4-core)		
Name		**Benchmarks**
Mix1		apsi, art, equake, mesa
Mix2		ammp, mesa, swim, vortex
Mix3		apsi, gzip, mcf, mesa
Mix4		ammp, gzip, vortex, wupwise
Rate1		4 copies of twolf, small working set (< 1MB)
Rate2		4 copies of art, large working set (> 1MB)
Multithreaded (8-core)		
Name	**Transactions**	**Setup**
OLTP	400	IBM DB2 v7.2 EEE, 25000 warehouses, 128 users
Apache	2500	20000 files (500MB data), 3200 clients, 25ms think time
JBB	12000	Sun HotSpot 1.4.0, 1.5 warehouses per-core, 44MB data
Zeus	2500	Event-driven, other configurations similar to Apache

Table 4.3: Workloads

workloads. Table 4.3 provides more information on the workload selection and configuration. The commercial multithreaded benchmarks include OLTP (TPC-C), Apache (static web content serving using the open source Apache server), JBB (a Java server benchmark) and Zeus (another static web benchmark running the commercial Zeus server) [1]. To compensate for workload variability, we measure the performance of multithreaded workloads using a work-related throughput metric [1, 3] and run multiple simulations with random perturbation to achieve statistically valid conclusions. The number of transactions simulated for each benchmark is listed in Table 4.3.

Multiprogrammed workloads are combinations of heterogeneous and homogeneous SPEC CPU2000 benchmarks. We use the same set of heterogeneous workloads as [27] for their representative behaviors, and include two homogeneous workloads with different working set sizes to explore extreme cases. The commercial workloads are simulated with an 8-core CMP, while the multiprogrammed workloads use a 4-core CMP, as we believe the scale of CMP systems may be different for servers vs. desktops.

Our default configuration associates each core with a 1MB 4-way associative unified L2 cache. Inclusion is not maintained between local L1 and L2 caches for CC (thus the baseline private caches) for the reasons described in section 3.2.1. For the shared cache scheme, private L1 caches are inclusive with the shared

4-core, 4MB total L2 capacity			
	Network	L2 assoc.	Worst-case latency
Private	2X2 mesh	4-way	50-cycle
Shared	2X2 mesh	16-way	40-cycle
8-core, 8MB total L2 capacity			
	Network	L2 assoc.	Worst-case latency
Private	3X3 mesh	4-way	70-cycle
Shared	4X2 mesh	32-way	60-cycle

Table 4.4: Network and Cache Configurations

L2 cache to simplify the protocol design. Because the L1 cache capacity is only 6.4% of the L2 cache, the performance impact of multi-level inclusion/exclusion is negligible. Throughout our evaluation, we compare a group of private L2 caches with a shared L2 cache having the same aggregate associativity and total capacity. We classify L2 hits for a shared cache into local and remote L2 hits, meaning hits into a processor's local and remote L2 banks, respectively. Unless noted otherwise, all caches use LRU for replacement. We view prefetching as a orthogonal approach to improve performance via latency hiding, so none of the caching schemes incorporates prefetching. Independently, Beckmann [13] has evaluated state-of-the-art CMP caching schemes with token coherence and a IBM Power4 like prefetcher, and shown CC is able to perform competitively for both commercial and scientific workloads.

Table 4.4 reports the cache and network latencies for our default simulation setup. These latencies are modeled similar to those in previous proposals [27, 104, 159], and consistent with CACTI [132] results. The default setup assumes a 12-FO4 pipeline delay to model a high-performance design [60,64], and we scale the cache and network latencies in Section 4.3.5 to model a 24-FO4 pipeline delay based, performance/power balanced processor design. We use mesh networks for intra-chip data transfers, modeling the non-uniform hit latencies for the shared cache. CC and private caches communicate with the on-chip directory (detailed in Section 3.3.2) using one-hop point-to-point links, therefore adding extra latencies for local L2 misses (including the one-hop network latency and the directory access latency). The performance impact of directory overhead will be evaluated in the sensitivity study (Section 4.3.4).

	Thousand misses per transaction Off-chip (Private / Shared / CC)	L1 Misses breakdown (Private / Shared / CC)		
		Local L2	Remote L2	Off-chip
OLTP	9.75 / 3.10 / 3.80	90% / 15% / 86%	7% / 84% / 13%	3% / 1% / 1%
Apache	1.60 / 0.90 / 0.94	65% / 9% / 51%	15% / 77% / 36%	20% / 14% / 13%
JBB	0.13 / 0.08 / 0.10	72% / 10% / 57%	14% / 80% / 32%	14% / 10% / 11%
Zeus	0.71 / 0.46 / 0.49	67% / 9% / 45%	15% / 78% / 41%	19% / 12% / 13%

Table 4.5: Multithreaded Workload Miss Rate and L1 Miss Breakdown

4.3.2 Multithreaded Workloads

In this section, we compare the performance of CC against private and shared cache schemes, as well as an "ideal" caching scheme that models a shared cache (for its capacity advantage) but with only the latency of a local cache bank (15-cycle).

Table 4.5 shows the off-chip miss rates and L1 miss breakdowns for various workloads and caching schemes. For each benchmark, we report the off-chip miss rate in terms of thousand misses per transaction (column 2), and break down L1 misses into local and remote L2 hits as well as off-chip accesses (columns 3-5). The ideal caching scheme (not reported here) should have an off-chip miss rate as low as the shared cache, and local L2 hit ratio as high as the private scheme. CC has much lower off-chip miss rates (within 4-25% of a shared cache) than the baseline private caches, and 5-6 times higher local L2 hit ratios than the shared cache. According to Equation 4.1, these characteristics suggest CC will likely perform better than both private and shared cache schemes unless off-chip miss rates are very low (favoring private caches) or memory latencies are extremely long (favoring a shared cache).

Performance

Figure 4.5 compares the performance of private, shared, CC and the "ideal" caching schemes, as transaction throughput normalized to the shared cache. Four CC schemes are included for each benchmark: from left to right they use system-wide cooperation probabilities of 0%, 30%, 70% and 100% respectively. As discussed in Section 3.2.3, the cooperation probability is used by L2 caches to decide how often to apply replication-aware data replacement and spilling of singlet victims. The baseline CC design (without

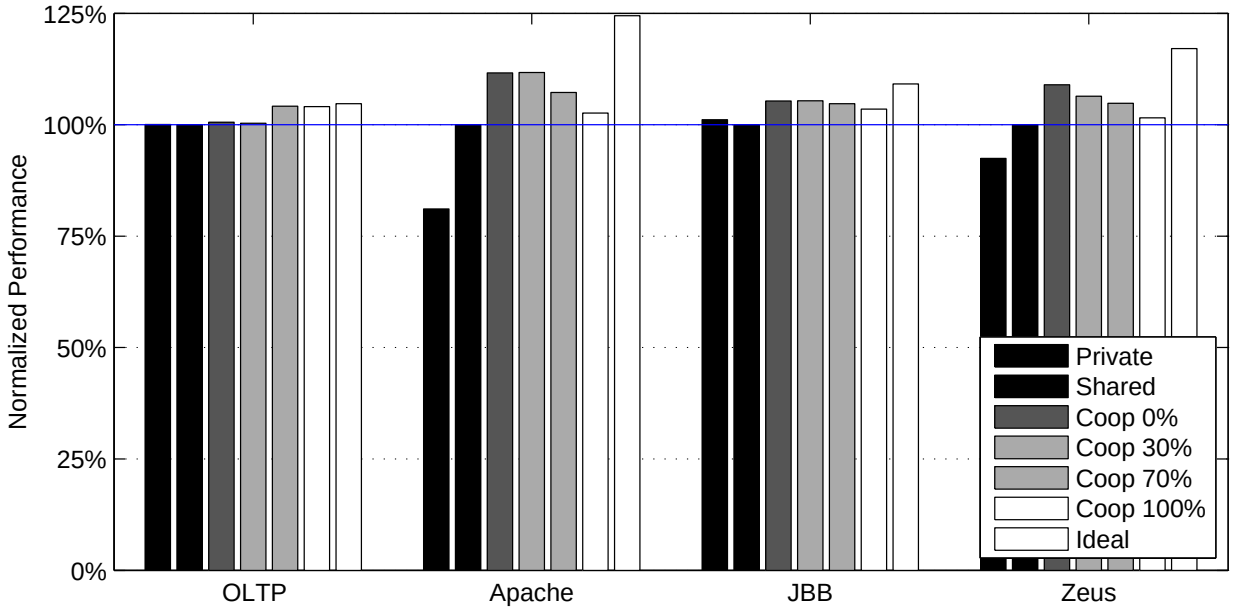


Figure 4.5: Multithreaded Workload Performance (The “ideal” scheme models a shared cache with only the latency of a local bank.)

capacity improvement policies) uses 0% cooperation probabilties, while the default CC scheme uses 100% cooperation probabilities to optimize capacity. We choose only four different probabilities as representative points along the sharing spectrum, although CC can support finer-grained throttling by simply varying the cooperation probabilities.

For our commercial workloads, the default CC scheme (“CC 100%”) always performs better than the private and shared caches. The best performing cooperation probability varies with different benchmarks, which boosts the throughput to be 5-11% better than a shared cache and 4-38% better than with private caches. CC achieves over half of the performance benefit of the ideal scheme for all benchmarks.

Memory Latency

The average memory access latencies (normalized to the shared cache) for different schemes are shown in Figure 4.6. In each case we break down the access latency into cycles spent in L1 hits, local and remote L2 hits and off-chip accesses. We calculate the average latency by assuming no overlap between memory

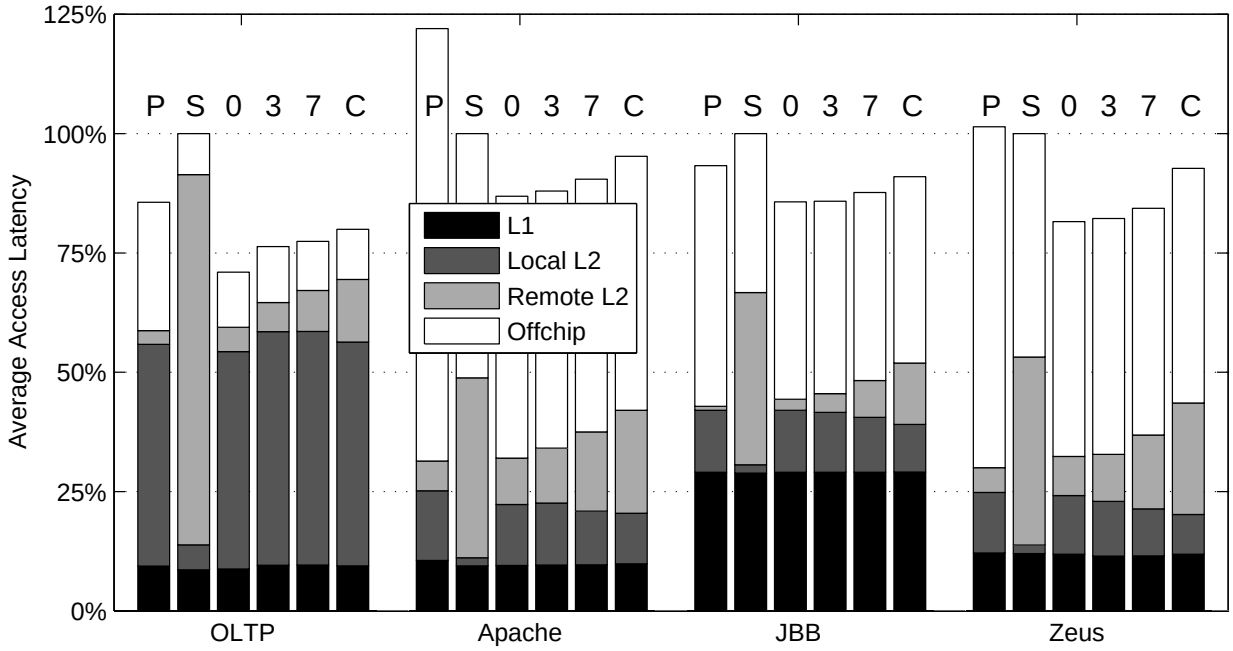


Figure 4.6: Multithreaded Workload Average Memory Access Latency (from left to right in each group: Private (P), Shared (S), CC 0% (0), CC 30% (3), CC 70% (7) and CC 100% (C))

accesses. Comparing Figures 4.5 and 4.6, we see that, for out-of-order processors, lower access latency does not necessarily result in better performance. For example, in the case of OLTP, private caches have a relatively lower access latency but effectively the same performance as a shared cache. This is because off-chip accesses have a much smaller contribution to the average latency for shared caches than they do for private caches, whereas the contribution of on-chip accesses is much larger. An out-of-order processor can tolerate some of the on-chip latencies, even to remote L2 cache banks, but can do little for off-chip latencies. Because CC's capacity improving policies can effectively reduce the impact of long off-chip latencies for private caches, it achieves better performance than both private caches and the CC scheme with no cooperation (CC 0%).

On the other hand, Apache and Zeus spend over 50% of the total memory cycles on off-chip accesses, suggesting these workloads either have large working sets or poor locality. The off-chip latency gap between private and shared cache is essentially removed by CC 0% with cache-to-cache transfer of clean data.

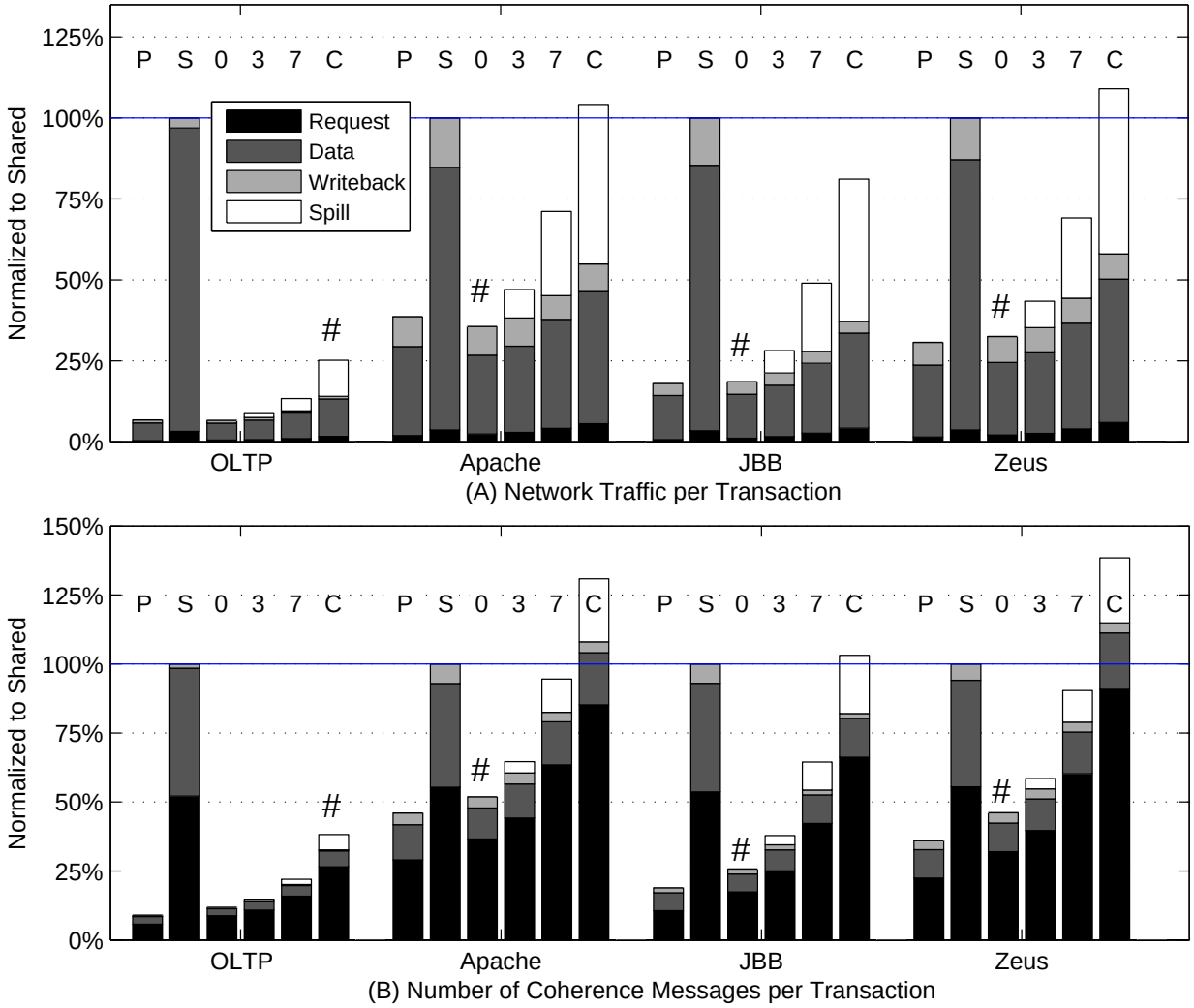


Figure 4.7: Multithreaded Workload Bandwidth ("#" indicates the best performing CC scheme.)

Increasing CC's cooperation probability has little benefit on reducing the off-chip latency, but consequently increases the on-chip latency. For this class of workloads, CC with 0% cooperation probability achieves the best performance.

Bandwidth

Figure 4.7 compares (A) the amount of on-chip network traffic and (B) the number of coherence messages generated to accomplish the same amount of work (e.g., an OLTP transaction) for different caching schemes.

	Misses per thousand instructions Off-chip (Private / Shared / CC)	L1 Misses breakdown (Private / Shared / CC)		
		Local L2	Remote L2	Off-chip
Mix1	3.1 / 2.0 / 2.4	78% / 19% / 67%	3% / 73% / 22%	19% / 9% / 11%
Mix2	3.0 / 1.6 / 1.8	64% / 35% / 75%	4% / 55% / 14%	32% / 9% / 11%
Mix3	1.2 / 0.7 / 0.8	91% / 20% / 87%	1% / 77% / 9%	7% / 3% / 4%
Mix4	0.6 / 0.3 / 0.3	95% / 12% / 90%	0% / 86% / 8%	4% / 2% / 2%
Rate1	0.8 / 0.6 / 0.8	90% / 20% / 80%	3% / 76% / 13%	7% / 4% / 6%
Rate2	53 / 51 / 41	31% / 7% / 24%	11% / 47% / 34%	58% / 46% / 42%

Table 4.6: Multiprogrammed Workload Miss Rate and L1 Miss Breakdown

These numbers quantify the requirements for (A) the on-chip network bandwidth and (B) cache coherence engine's execution bandwidth. The total bandwidth requirements are broken down into 4 categories: (1) control messages, (2) data forwarding messages, (3) block write-backs, and (4) block spills.

The network and coherence bandwidth requirements of CC without cooperation (CC 0%) are both comparable to those of the private caches, which are often several times lower than a shared cache. As the cooperation probability increases from 0% to 100%, CC consumes extra bandwidth to exchange control information (e.g., communicated via PUTS and SINGLET messages) and data (via block spills) between the on-chip directory and caches. Cache cooperation also increases the amount of data forwarding between peer caches because it causes (1) more misses in local L2 caches and (2) more local misses to be satisfied by a peer L2 cache. However, CC often requires less bandwidth than the shared cache because its use of private caches can filter most L1-L2 communications. Overall, although CC 100% can sometimes require more bandwidth than a shared cache, the best performing CC schemes' bandwidth requirements are usually less than 50% of a shared cache. The saved bandwidth not only leads to reduced network power consumption, but also allows a potentially simpler and faster network design.

4.3.3 Multiprogrammed Workloads

In this section, we analyze CC's performance for multiprogrammed SPEC2000 workloads. We compare performance using the aggregate IPCs from 1 billion cycles of simulation. No cooperation throttling is used because a single system-wide cooperation probability is not sufficient to accomodate the heterogeneity

across benchmarks, and we leave the adaptive throttling for heterogeneous workloads as future work.

Multiprogrammed SPEC2000 workloads differ from commercial multithreaded workloads in several ways: (1) no replication control is needed for private caches as little sharing exists among threads; (2) consequently most L1 cache misses in the private cache scheme are satisfied by the local L2 cache, which often leads to reduced average on-chip cache latency and better performance than a shared cache; (3) the aggregate on-chip cache resources still need to be managed globally to allow dynamic capacity sharing among programs with different working set sizes. Because CC supports global management of distributed private caches, we expect CC to retain the advantages of private caches while reducing the number of off-chip misses via cooperation.

Table 4.6 lists the off-chip miss rates and L1 miss breakdowns for private, shared and CC schemes. As with multithreaded workloads, CC can effectively reduce the amount of both off-chip (shown by the low miss rates in column 2) and cross-chip references (demonstrated by the high local L2 hit ratios in column 3). The off-chip miss rates are only 0-33% higher than a shared cache, and the local L2 hit ratios are close to those using private caches.[5]

Notice the high off-chip miss rates of Rate2 (over 40 misses per thousand instructions) are caused by running four copies of `art`, whose aggregate working set size significantly exceeds the total cache capacity. Thrashing occurs as the result of overcommiting cache resource usage, causing the shared cache to have an off-chip miss rate similar to private caches. CC has 20% fewer misses because its spill based global capacity allocation is less intrusive than the shared cache's request driven capacity allocation, therefore can mitigate the negative effect of thrashing. This case is an example of destructive inter-thread interference, which will be addressed in Chapter 5.

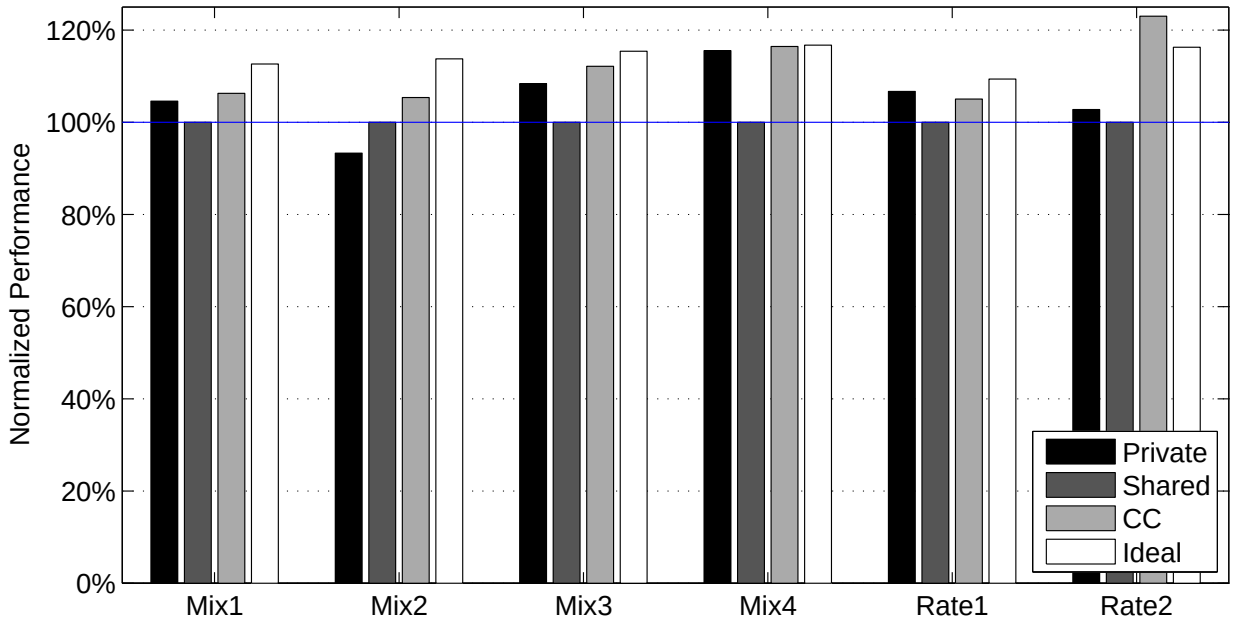


Figure 4.8: Multiprogrammed Workload Performance

Performance and Memory Latency

The aggregate IPCs for the different schemes, normalized to the shared cache scheme, are shown in Figure 4.8. CC outperforms the private scheme by an average of 6% and the shared caches by 10%. For Rate2, CC performs better than the ideal shared scheme because it has lower miss rate than a shared cache. Figure 4.9 shows the average memory access latency, assuming no overlap between accesses. It illustrates that private caches run faster than a shared cache for Mix1, Mix3 and Mix4 by reducing cross-chip references (also shown in Table 4.6), while a shared cache improves Mix2's performance by having many fewer off-chip misses. CC combines their strengths and outperforms both for all heterogeneous workloads. For homogeneous workloads, Rate1 consists of four copies of `twolf`, whose working set fits in the 1MB L2 cache, so private caches are the best choice while CC performs slightly worse. CC reduces the off-chip miss rate for Rate2, and consequently improves its performance by over 20%.

[5]For Mix2, CC's local L2 hit ratio is higher than the private scheme because one of the benchmarks (`ammp`) experiences much fewer off-chip misses thus progresses faster and reaches a different computation phase during the simulation.

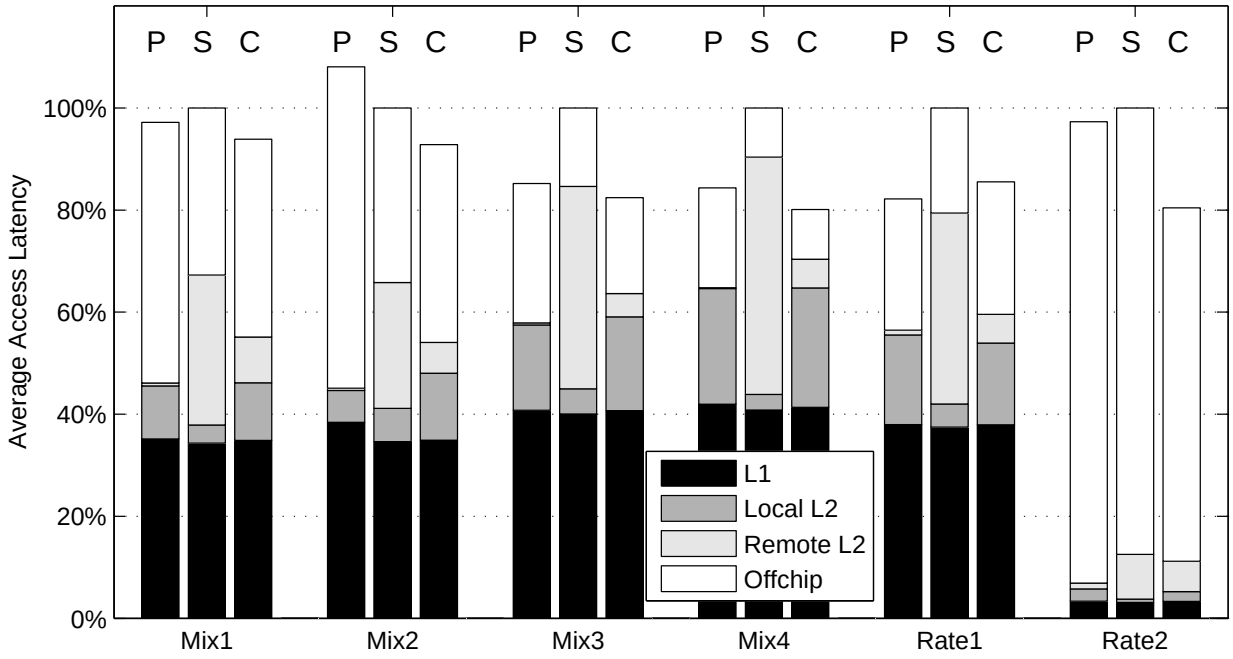


Figure 4.9: Multiprogrammed Workload Average Memory Access Latency (from left to right in each group: Private, Shared, and CC)

Bandwidth

Figure 4.10 shows the amount of network traffic and coherence messages generated per committed instruction, normalized to the shared scheme. Comparing with private caches, CC incurs extra network traffic mainly due to block spills and more frequent inter-cache data forwarding. On the other hand, the coherence messages generated by CC are mainly due to cooperation related information communication (e.g., PUTS and SINGLET) which can be easily combined with other coherence messages. Compared with the shared cache, CC often filter many more L1 to L2 messages than the added cooperation messages. Except for the pathological case of Rate2, where frequent local L2 evictions cause CC to generate many spills and more cross-chip traffic than the shared cache, CC only generates 25-60% of the network traffic and 28-82% of the coherence messages of a shared cache.

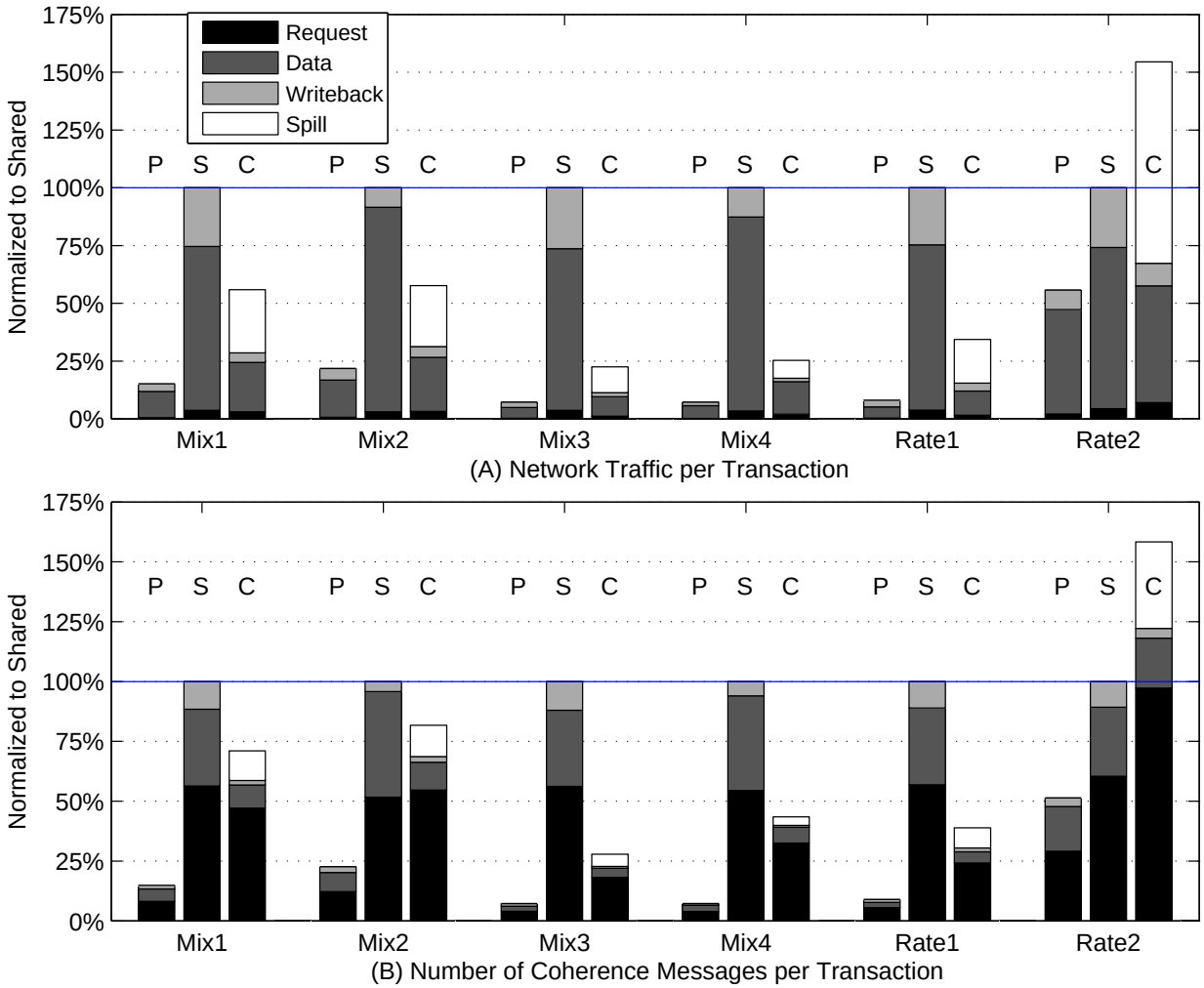


Figure 4.10: Multiprogrammed Workload Bandwidth

4.3.4 Sensitivity Study

After showing CC's benefits with the default simulation parameters, we now evaluate the performance robustness of CC with a sensitivity study of commercial workload performance using in-order, blocking processors. We choose to use in-order processors mainly to reduce simulation time, but they also represent a relevant design choice (e.g., [87]). The main idea here is to assess the benefit of CC across a spectrum of memory hierarchy parameters.

Figure 4.11 compares the relative performance (transaction throughput normalized to the shared cache

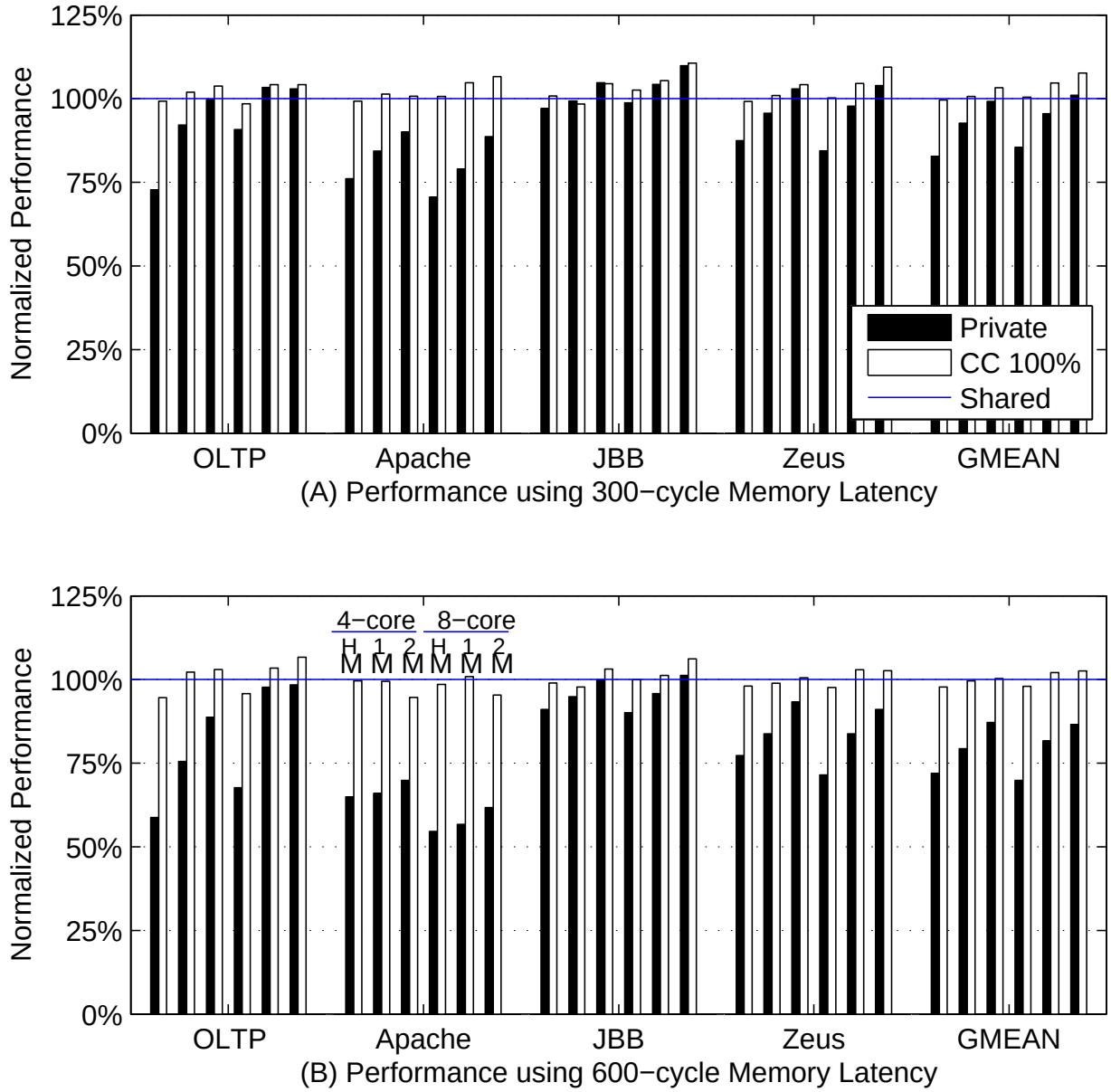


Figure 4.11: Performance Sensitivity (300 and 600 cycles memory latencies; from left to right in each group: 4-core and 8-core CMPs with 512KB, 1MB and 2MB per-core caches)

scheme) of private, shared and CC for different system sizes (4-core vs. 8-core), per-core cache capacities (512KB, 1MB, and 2MB) and memory latencies (300 cycles vs. 600 cycles). For a given CMP size (e.g., 4-core), the normalized throughput of both CC and private schemes increases with the per-core cache capacity, while the performance gap between CC and private schemes gradually decreases. Overall, CC achieves the

Extra cycles	0	+5	+10	+15
Multithreaded	7.5%	4.7%	3.2%	0.1%
Multiprogrammed	11%	8.0%	7.0%	5.9%

Table 4.7: Speedups with Varied CCE Latencies

best performance for most configurations with a 300-cycle memory latency. When memory latency doubles, the latency of off-chip accesses dominate, and CC becomes similar to a shared cache, having -2.2% to 2.5% average speedups.

Besides cache and memory parameters, the overhead of CC's implementation can also impact its performance. As discussed in Section 3.3.2, our CC implementation uses an on-chip, centralized directory called CCE. Because every local L2 miss has to be serviced through and potentially delayed by the CCE, we study CC's performance sensitivity to the CCE overhead. Table 4.7 shows the speedups of CC over the baseline shared cache design when the CCE latency (originally 5 cycles) is doubled, tripled and quadrupled. Speedups over private caches are not included because both CC and private caches are implemented using the CCE, and increasing CCE latency has a similar effect on them. We observe that CC can tolerate directory overhead, and perform better than a shared cache, even with quadrupled CCE latency.

4.3.5 Comparison with Victim Replication (VR)

In this section, we compare CC with victim replication [159], an example of recently proposed CMP caching optimizations. We choose to study VR but not other CMP caching schemes (e.g., [15, 26, 83]) for several reasons: (1) both VR and CC use cache replacement as the underlying technique to control replication; (2) both schemes are based on a traditional cache organization, while other proposals require significant changes in the cache organization; (3) they both use a directory-based protocol implementation, rather than requiring different styles of coherence protocols (e.g., snooping protocol or token coherence).

The comparison is conducted using in-order, blocking processors (12 FO4 cycle time) with the same L1/L2 cache sizes and on-chip latency parameters as [159] to specifically match its evaluation. We also

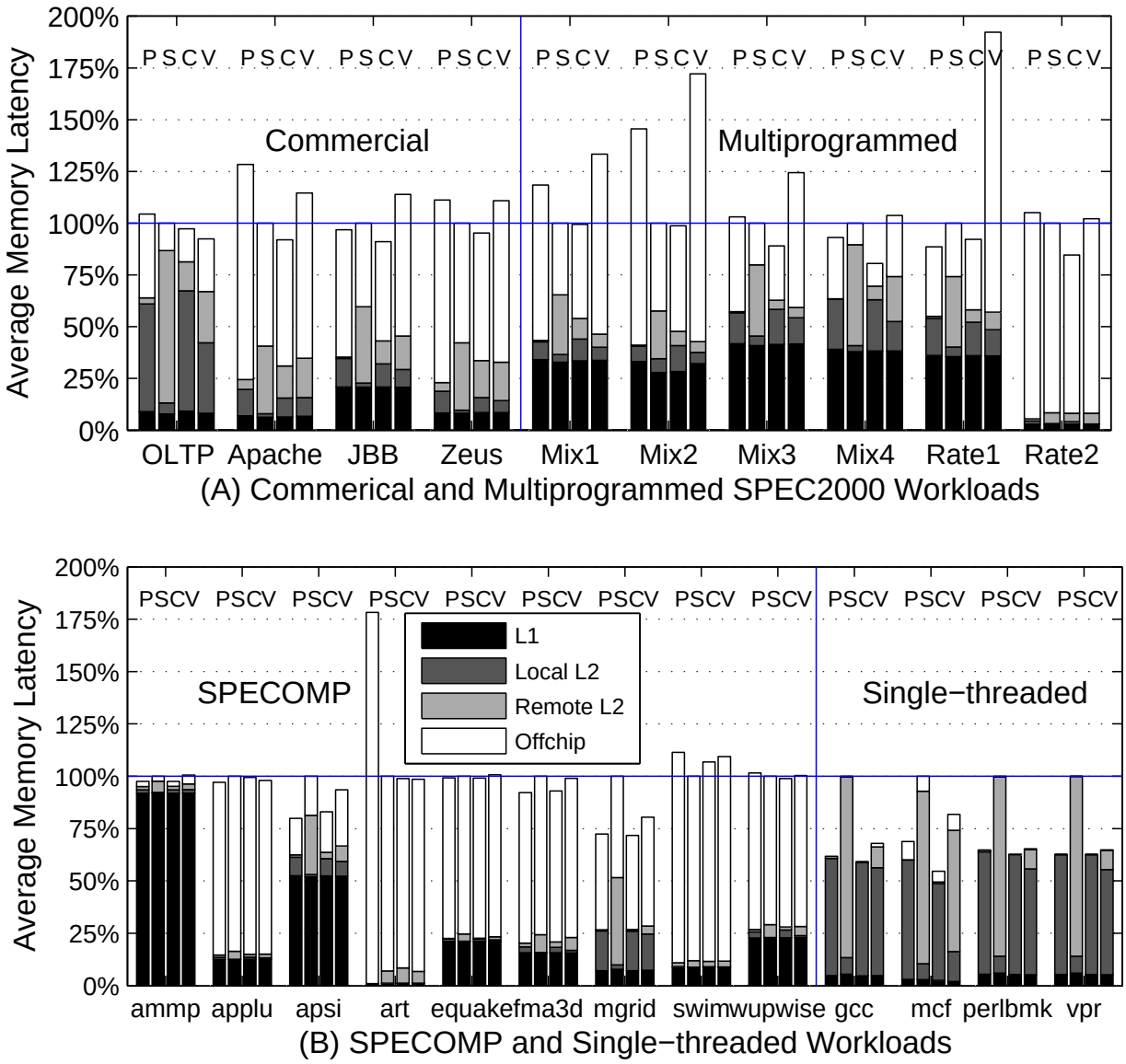


Figure 4.12: Latency Comparison with Victim Replication (from left to right in each group: Private (P), Shared (S), CC (C), and VR (V))

tried using the set of parameters as in previous experiments, however, VR performs worse than both shared and private schemes for 3 out of 4 commercial benchmarks. As in [159], we use random replacement for victim replication, because it's not straightforward to set the LRU information for the replica. To create as realistic a match to the previous paper as possible, we also include results for 9 SPECOMP benchmarks [8] and 4 single-threaded SPEC2000 benchmarks with the MinneSPEC reduced input set [85], all running on

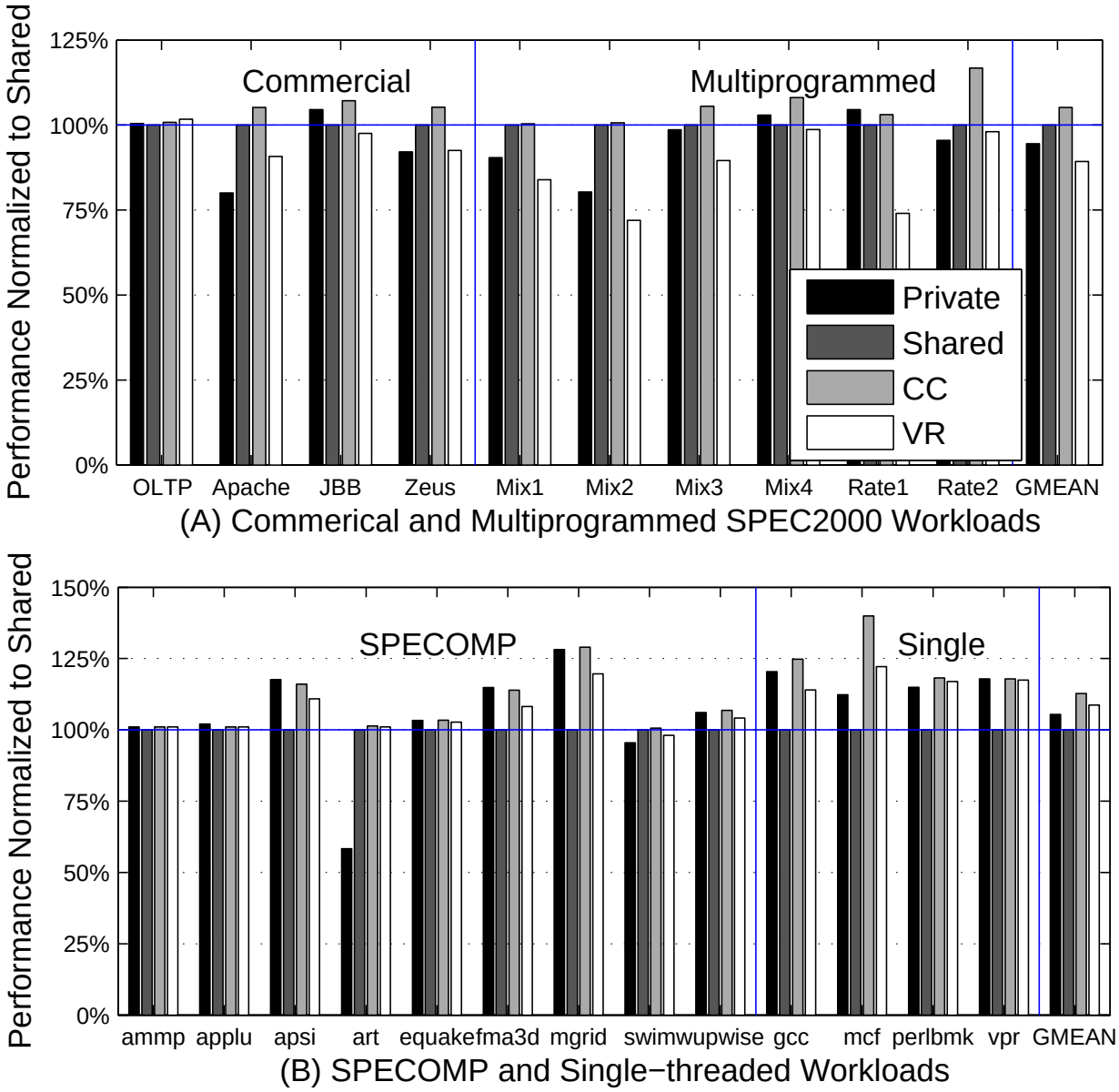


Figure 4.13: Performance Comparison with Victim Replication

8-core CMPs. The single-threaded benchmarks are in common with [159], and the SPECOMP benchmarks have characteristics similar to the multithreaded workloads it used.

Same as reported by [159], Figure 4.12 compares the average memory latencies of private, shared, CC and victim replication schemes. We also report the performance measurement (either transaction throughput

or IPC, normalized to the shared scheme) in Figure 4.13. By comparing the two figures, we can see that, with in-order processors, the relative ordering of different schemes' latency reduction capabilities correlates well with their relative performance ordering, however, latency reduction results cannot directly predict performance improvement. Therefore we focus on Figure 4.13 in the following discussion.

Figure 4.13 (A) includes (1) commercial multithreaded workloads with large working sets and substantial inter-thread sharing, and (2) SPEC2000 multiprogrammed workloads, while (3) SPECOMP multithreaded workloads with little sharing and (4) single-threaded workloads are covered by Figure 4.13 (B). Victim replication outperforms both private and shared schemes for SPECOMP and single-threaded workloads, on average by 5% and 18%. However, victim replication performs poorly for multithreaded commercial workloads and multiprogrammed workloads, being on average 6% slower than private caches and 11% slower than a shared cache.

CC consistently performs better than victim replication (except for OLTP). CC provides the best performance for 3 out of 4 commercial workloads, 5 out of 6 multiprogrammed workloads and all single-threaded workloads; it is less than 1.4% slower than the best schemes for all SPECOMP benchmarks. Across all of these benchmarks, CC is on average 9% better than private and shared schemes, and 10% better than victim replication.

Victim replication is especially ineffective for multiprogrammed workloads because it blindly replicate blocks in both the referencing processor's local bank and its home node bank. This can cause significant waste of on-chip capacity (as indicated by VR's higher "off-chip" bars shown in Figure 4.12), when the home node does not reference the data but has to keep the master copy of the data.

4.3.6 Adaptive Throttling

Previously we have evaluated CC schemes with different cooperation probabilities for multithreaded workloads, now we study the performance of adaptive cooperation throttling which dynamically selects the best

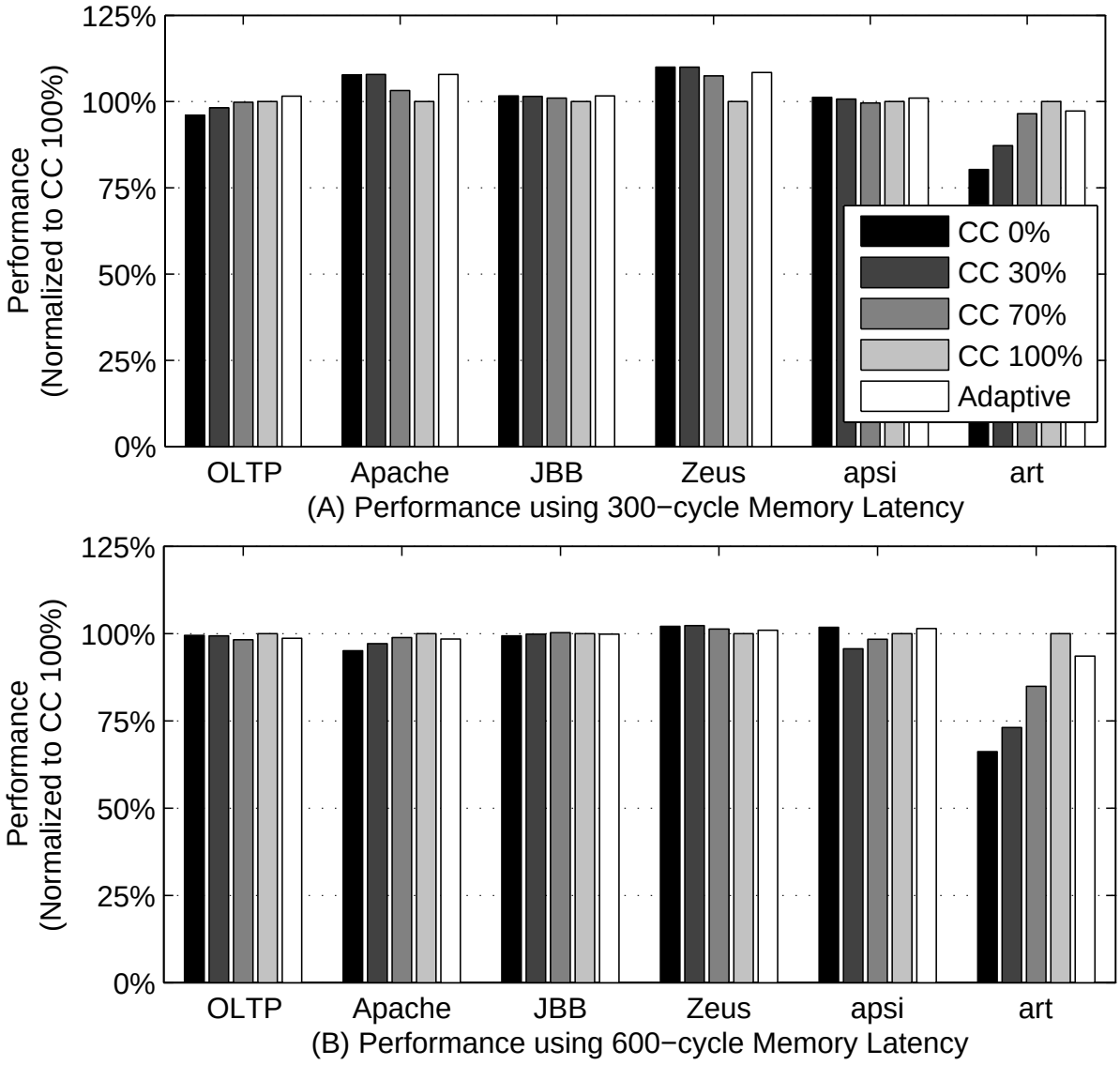


Figure 4.14: Adaptive Throttling Results

performing cooperation probability. Our experiments are based on Dynamic Set Sampling (DSS), which was discussed in Section 4.2.3. Figure 4.14 compares the performance of CC schemes with various cooperation probabilities and CC with adaptive throttling (Adaptive). Besides the commercial workloads, we also include two SPECOMP benchmarks `apsi` and `art` to represent multithreaded scientific workloads[6].

Figure 4.14 (A) shows that, except for `art`, adaptive throttling achieves the same performance as the

[6]Other SPECOMP benchmarks are excluded here because varying the cooperation probability has little impact on them.

best static throttling scheme (at worst 1% lower). For OLTP, adaptive throttling even outperforms the best static scheme because it can dynamically change the cooperation probabilities to accomodate program phase changes. For SPECOMP benchmark `art`, DSS-based dynamic adaptation performs worse then the best static scheme because it mispredicts future program behavior. Although this benchmark prefers the capacity optimizations of CC 100%, it repetitively goes through phases that prefer more replication. Without phase prediction support, our current adaptation scheme prematurely increases the amount of replication and evicts data that will be reused in later phases. This causes extra off-chip misses, leading to a 3% performance gap between the adaptive scheme and CC 100%. As a sensitivity test, Figure 4.14 (B) reports similar results using longer memory latency (600-cycle). Due to increases off-chip miss penalty, the performance gap between CC 100% and CC adaptive for `art` is increased to 7%. For other workloads, adaptive throttling continues to reduce memory latency and performs within 1.5% as the best static throttling scheme.

4.4 Summary

In this chapter we proposed cooperation policies for CC to reduce processor stalling cycles due to memory access latency. Our proposed solution is based on the CC framework for the latency benefit of private caches. Expensive off-chip misses are reduced by mimicking the behavior of a shared cache: (1) replication is controlled to keep unique blocks on-chip, and (2) local cache replacement is combined with global spill/reuse history to approximate a global cache replacement policy. Probability based throttling is applied to trade off between cycles spent on on-chip wire delays and off-chip memory accesses, and adaptively select the best cooperation option.

Our simulation shows that CC achieves the best performance for different CMP configurations and workloads. CC can reduce the runtime of simulated workloads by 4-38%, and performs at worst 2.2% slower than the best of private and shared caches in extreme cases. CC also outperforms the victim replication scheme [159] by 9% on average over a mixture of multithreaded, single-threaded and multiprogrammed workloads, while the performance advantage increases for workloads with larger working sets.

CHAPTER 5

COOPERATIVE CACHE PARTITIONING

With multiple execution contexts simultaneously sharing on-chip cache resources, CMPs must accomodate multiprogrammed, concurrently running threads with different localities and working set sizes. In the previous chapter, CC is extended with global replacement policies to dynamically adjust the cache allocation in a fine-grained manner. This policy is simple by assuming benign interaction among different threads and being oblivious to their caching characteristics. However, under high caching pressure, destructive inter-thread interference can happen, causing sub-optimal performance, unfair progress and poor Quality-of-Service (QoS). In this chapter, we further extend the CC framework with policies to mitigate the impacts of such destructive interference.

In Section 5.1, we illustrate the problems caused by cache resource contention, discuss previous cache partitioning proposals for interference isolation and pinpoint their limitations. We outline our proposed solution by addressing the limitations of prior cache partitioning schemes. Section 5.2 provides background information on metrics, workloads characteristics and evaluation methods. In Section 5.3 and 5.4, we detail the two aspects of our approach—time-sharing based cache partitioning and its integration with CC's LRU-based capacity sharing policy. The evaluation results are presented in Section 5.5, and we conclude in Section 5.6.

5.1 Motivation and Proposed Solution

5.1.1 Motivation

To make efficient use of the aggregate on-chip cache capacity and off-chip bandwidth, most CMP caching designs support dynamic capacity sharing either via a logically shared cache [12, 15, 27, 58, 113, 148, 159] or

by adding inter-core capacity sharing policies onto private cache based designs [23, 59, 138, 156]. However, capacity sharing in a conventional, unconstrained manner can cause destructive interference among co-scheduled threads, leading to sub-optimal overall performance and unfair impact on individual threads. In contrast, a private cache design avoids inter-thread interference by statically partitioning the aggregate capacity between processor cores. This design is simple, fair and guarantees QoS, but often incurs many more expensive off-chip misses for thread mixes with non-uniform caching requirements.

A Thrashing Example

Figure 5.1 shows an example of destructive inter-thread interference, by considering the task of executing many copies of SPEC2000 benchmark `art` on a 4-core CMP with a 4MB 16-way shared L2 cache. Figure 5.1 (A) shows the capacity allocation among co-scheduled threads, while Figure 5.1 (B) plots the corresponding throughput (measured as IPCs). In each bar graph, the 4 bars on the left represent an LRU-based shared cache with the number of co-scheduled threads ranging from 1 to 4, and the rightmost bar represents a cache partitioning scheme with 4 threads concurrently running.

For LRU-based shared cache, the total capacity is evenly divided among co-scheduled threads. The overall throughput doubles when the number of co-scheduled threads increases from 1 to 2, but starts to decrease when more (i.e., 3 and 4) threads share the aggregate cache resources. This is a typical example of thrashing [35], where the overall throughput drops after the system is overloaded beyond a certain threshold (2 copies of `art` being co-scheduled in this example). The performance degradation is actually caused by cache resource contention among threads with large working sets. Here `art`'s working set size is 1.75MB, which can only be satisfied when less than 3 threads share the aggregate 4MB L2 cache. Adding more threads will cause significantly more off-chip misses, leading to lowered system throughput.

To optimize the aggregate throughput, the operating system can be modified to consider cache resource contention and only schedule 2 copies of `art` at a time [49]. On the other hand, hardware cache partitioning

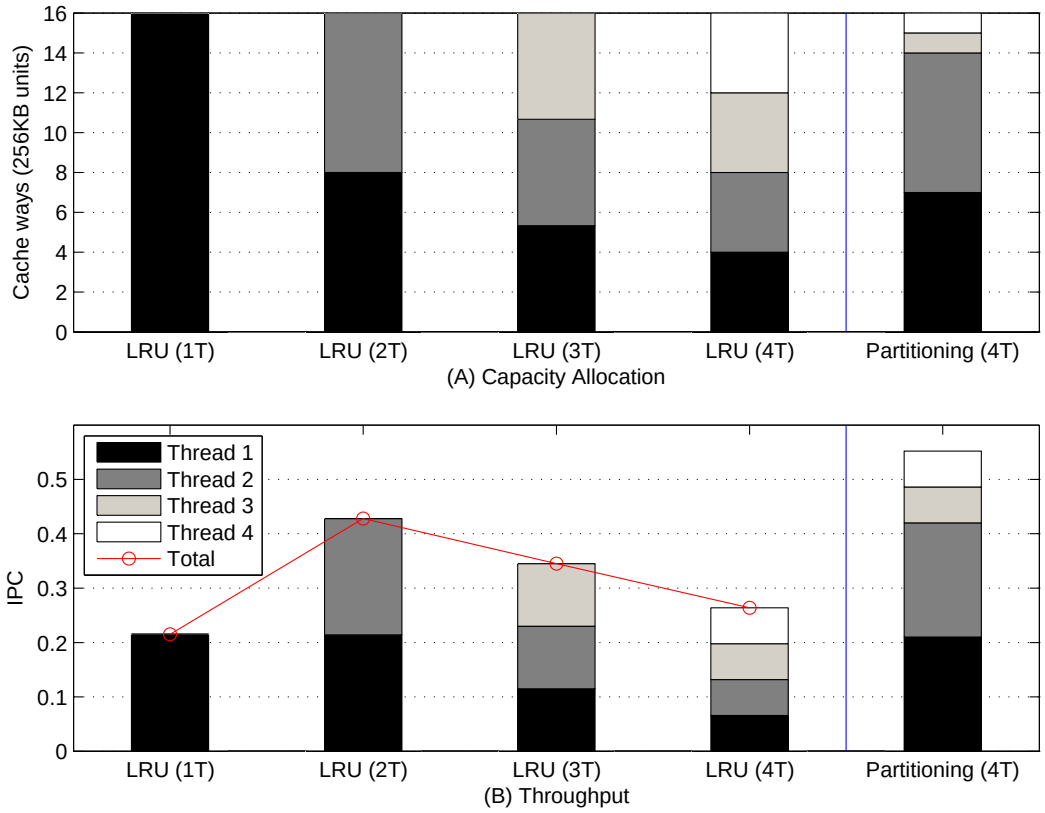


Figure 5.1: Different Ways to Run Many Copies of `art` on a 4-core CMP and Their Throughput

schemes (for example, [123, 144]) can achieve even higher throughput without adding software complexity. As shown by the rightmost bars in Figure 5.1, this is achieved by partitioning the total L2 capacity among 4-copies of `art` to: (1) provide the minimum capacity needed by threads 1 and 2 to satisfy their working set requirements and (2) allocate the remaining capacity between threads 3 and 4.

Cache Partitioning Background

Cache partitioning manages the aggregate cache resources via explicit allocation to different reference streams (e.g., generated by different threads), as opposed to caching policies that treat all requests as from a single source [144]. By dividing the total capacity among co-scheduled threads, cache partitioning can isolate the destructive interference among co-scheduled threads and potentially improve the system

throughput (as shown in Figure 5.1 and prior proposals [96, 123, 144]), fairness [84] or QoS [156].

We can view private caches as an example of static cache partitioning, where the aggregate cache resources are divided among processor cores at design time. However, cache capacity requirements are non-uniform across threads and across different program phases of a single thread. To accomodate such dynamic and often heterogeneous capacity requirements, recent cache partitioning proposals [68, 74, 84, 96, 121, 123, 144, 156] match the perceived requirements of different threads by orchestrating cache resource allocation with more flexible, usually heterogeneous partitions.

CMP cache partitioning schemes generally work in repetitive epochs, each consisting of three steps: (1) measurement, (2) partitioning, and (3) enforcement. The first step is to measure and estimate each thread's performance (in terms of miss rate or IPC) for candidate cache partitions. This information is then used to determine the next cache partition to reach a given optimization goal. The new partition will be enforced in the next execution epoch, while new measurement will be gathered and used in later epochs.

Cache partitioning proposals can differ in measurement and enforcement mechanisms, optimization goals and metrics as well as their partitioning policies. Measurement information can be gathered via profiling [66, 84], LRU stack hit position counting [156], monitoring [143], or dynamic set sampling [123]. Table 5.1 compares the optimization goals and policies used by prior schemes. The partitioning algorithm has to be simple by avoiding exhaustive search, thus often uses heuristics to prioritize capacity allocation according to the miss rate and speedup characteristics of co-scheduled threads. Both the measurement and partitioning steps can incur space or execution time overheads, while inaccurate information/decisions can lead to sub-optimal results.

Despite their differences in metrics, mechanisms and policies, prior cache partitioning schemes have two common characteristics and consequently two limitations.

	Optimization goals	Policies (threads with allocation priority)
Liu et al. [96]	Max throughput	Static partitioning
Suh et al. [144]	Min miss rate	Greedy (app. with best marginal miss reduction)
CQoS [74]	QoS	Generic framework, open for various policies
Fair Sharing [84]	Min slowdown difference	Greedy (app. with most extra misses)
Fast and Fair [156]	Max $\sum speedup$ under QoS	Greedy (app. with best speedup)
OS-managed [74]	Open	Generic mechanism, open policies
STATSHARE [74]	Open	Generic model/mechanism, open policies
Utility-based [123]	Max WS	Lookahead (app. with best marginal utility)
MTP (this chapter)	Max FS under QoS	Iterative (threads with high speedups)

Table 5.1: Comparing CMP Cache Partitioning Schemes.

Coarse-grained vs. Fine-grained Capacity Allocation

Most current cache partitioning schemes are coarse-grained, allocating large capacity units (e.g., in 64KB chunks) for long epochs (e.g., 5-million cycles [123]). A fine-grained partitioning scheme needs to (1) estimate the costs/benefits of capacity allocations in smaller units (e.g., 128B cache blocks) and (2) using such information to frequently trigger the partitioning step, both incurring significant overhead [121]. To reduce such overhead, most prior proposals use **way partitioning** [144] as the basic mechanism to enforce a cache partition[1]. Assuming a set-associative cache, way partitioning allocates cache resources in units of cache ways (each way having the same number of cache sets). This mechanism can be implemented with a modified cache replacement policy to ensure that the number of blocks used by a thread at a cache set level does not exceed its way quota.

On the other hand, the commonly used LRU-based capacity sharing is fine-grained and can outperform coarse-grained cache partitioning schemes when little inter-thread interference exists. To illustrate this, Figure 5.2 plots the amount of cache allocated by (A) an LRU-based scheme and (B) a way partitioning scheme that optimizes overall throughput, for workload `art-art-apsi-apsi` (2 copies of `art` and `apsi` sharing a 4MB L2 cache). The cache partitioning scheme is coarse-grained because it allocates

[1]STATSHARE [121] and cache-level-quota enforcement [125] do provide spatially, but not temporally, fine-grained partitioning mechanisms, however without specific policies to exploit such mechanisms. STATSHARE [121] also provides an analytical model to calculate the cost/benefit of block-based capacity allocation, but evaluating this model for every L2 request is expensive.

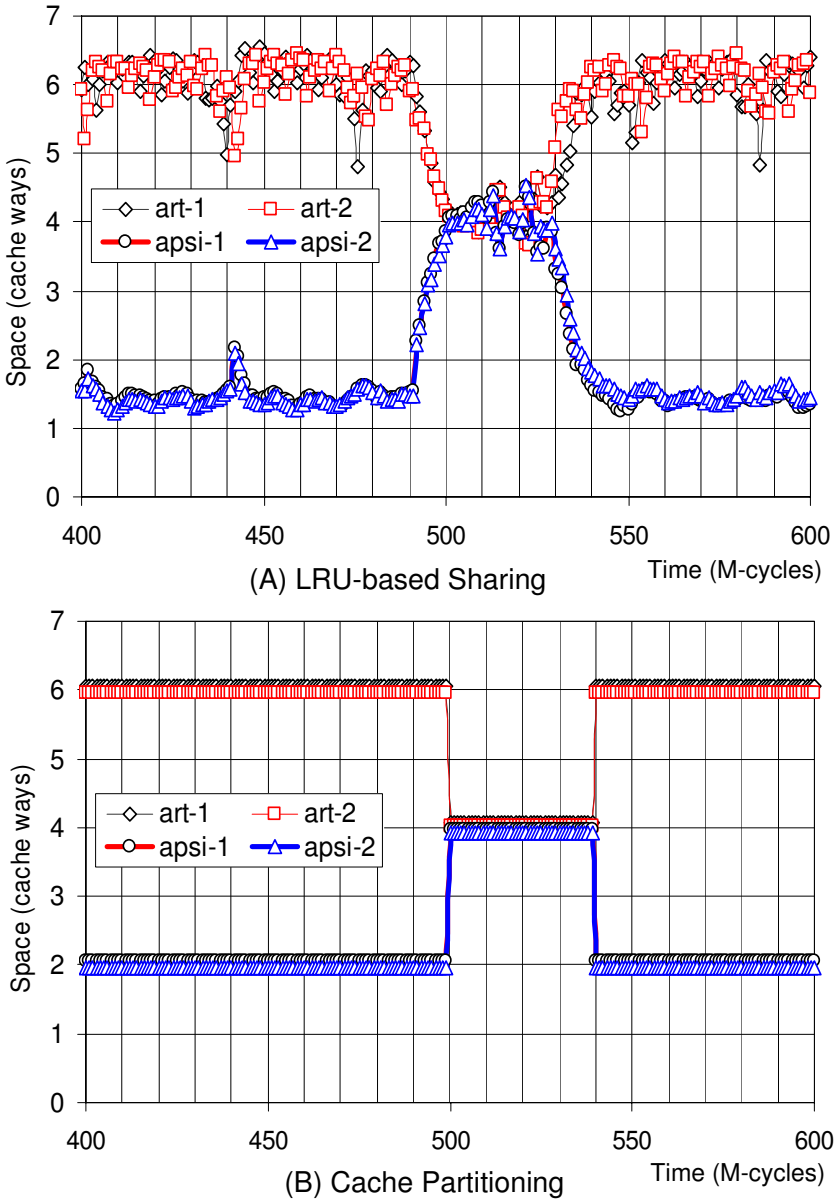


Figure 5.2: LRU vs. Cache Partitioning for 2 Copies of `art` and `apsi` (`art-art-apsi-apsi`)

capacity in cache ways and triggers repartitioning at the boundary of 10M-cycle epochs. On the other hand, LRU allocates on average 6.5 ways and 1.5 ways of capacity to `art` and `apsi` to better fit their capacity requirements. LRU also supports temporally fine-grained sharing when `apsi` enters a phase that needs more capacity (simulation time 500 to 530 million cycles), adapting to phase changes swiftly without extra support or overhead.

Due to spatially fine-grained sharing, `art` with LRU achieves 34% better speedup than with way partitioning. `Apsi`'s throughput is also slightly better than using cache partitioning because of temporally fine-grained sharing (even though its average cache capacity is lower than way partitioning). Similar observations have been made in [141], which shows that LRU can provide near-optimal cache allocation for many workloads.

Fairness and QoS Issues

Another common characteristic of prior cache partitioning schemes is that they are Single Spatial Partition (SSP) based: they all attempt to reach their optimization goals by selecting the best spatial partition and use a single spatial partition repeatedly for all epochs in a stable program phase (as previously shown in Figure 5.2 (B)). In other words, these scheme consider only space sharing among threads, but not time sharing among different spatial partitions. This implies that it may be difficult for prior proposals to simultaneously improve performance/efficiency and fairness while maintaining QoS, as it is intrinsically hard to satisfy multiple conflicting goals with a single partition. We will describe our notions and metrics for performance/efficiency, fairness and QoS in Section 5.2.1, but use the following example to briefly illustrate the related issues.

Figure 5.3 plots the normalized performance of benchmark `vpr` across a set of 4-thread multiprogrammed workloads with large working set requirements (e.g., workload `art-mcf-ammp-vpr`). The shared L2 cache is managed by one of 4 caching schemes. (1) PAR_Fair is a cache partitioning scheme that equally divides the L2 capacity among co-scheduled threads. This scheme is fair (in terms of resource allocation) and maintains QoS (i.e., providing predictable performance across different workloads). (2) LRU (often implemented as pseudo-LRU) is the caching policy for most CMP designs, which allocates cache blocks based on L2 requests. (3) PAR_WS is a cache partitioning scheme that optimizes the Weighted Speedup (WS) metric (a speedup/efficiency metric proposed for SMT architecture [134] and used in prior

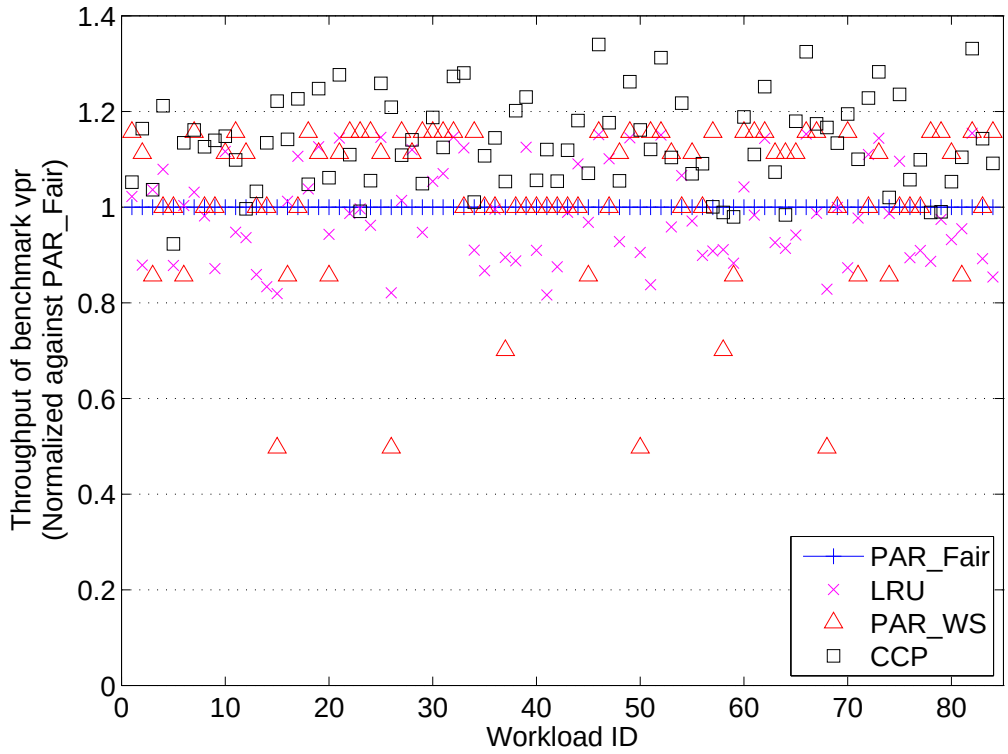


Figure 5.3: Throughput of Benchmark `vpr` in Different Workloads and Caching Schemes

CMP cache partitioning schemes [123,156]). (4) CCP is our proposed caching scheme that will be described in later sections.

Because PAR_Fair provides consistent performance for all workload combinations[2], we use its performance as a stable baseline for comparison. The LRU policy often performs worse than PAR_Fair because its demand-driven capacity allocation favors threads with frequent misses and unfairly shrinks `vpr`'s capacity. It is also hard to predict `vpr`'s performance as it is dependent on the caching characteristics of co-scheduled threads. Therefore LRU cannot maintains fairness or QoS. PAR_WS improves the performance over both PAR_Fair and LRU for many workloads by balancing the capacity allocated to different threads, but it can still unfairly shrink the capacity of `vpr` to achieve its optimization goal. Same as LRU, PAR_WS does not guarantee QoS. A better cache partitioning scheme (e.g., the CCP scheme) should be able to achieve both

[2]Contention in other shared resources, especially the memory system, can also cause destructive interference. Here we focus on the impact of destructive interference occurring in the last-level CMP caches, assuming a fair memory system (supporting min-max fairness [17]) as proposed in [114].

throughput and fairness/QoS goals.

Summary

We view the sharing of CMP cache resources among threads as a resource management problem. The properties of a good resource manager have been extensively examined by the operating system researchers (e.g., in [147]), which include (1) abilities to improve overall performance, (2) maintenance of fairness and QoS, and (3) suitability for a wide range of workloads combinations. Fairness and QoS are especially important for CMP as it is used in consolidated servers, shared computing clusters, embedded systems, and other platforms where meeting these requirements is as important as improving overall throughput.

Previous research in CMP cache management mainly focused on using cache partitioning to achieve some of these requirements [68, 74, 84, 96, 121, 123, 144, 156]. However, none of these proposals is sufficient to satisfy all CMP cache management requirements because of two limitations. (1) **Limited functionality.** Prior proposals cannot address all functional requirements, including thrashing avoidance, fairness improvement, QoS guarantee and priority support, partially due to the difficulty of satisfying multiple, often conflicting, goals in a single cache partition. (2) **Limited scope of application.** Cache partitioning can only outperform LRU-based latency-reducing CMP caching schemes for some multiprogrammed workloads. An attempt to solely use cache partitioning can cause sub-optimal performance for workloads that do not experience destructive inter-thread interference.

5.1.2 Proposed Solution

Our proposal has two aspects, each addressing one of the two limitations of cache partitioning: (1) we introduce a time-sharing based cache partitioning scheme to simultaneously improve throughput and fairness while maintaining QoS; (2) our cache partitioning scheme is integrated with CC's LRU-based capacity sharing policy (covered in Chapter 4) to support both workloads that prefer caching partitioning and workloads

that prefer LRU-based sharing.

To provide multiple functionalities, our Multiple Time-sharing Partitions (MTP) scheme makes different threads cooperatively shrink and expand their capacity allocations across multiple partitions, and schedules different partitions in a time-sharing manner. Specifically, each MTP partition improves at least one thrashing thread's throughput by temporarily shrinking the capacity of other threads to make room for it. By time sharing cache resources among multiple unfair partitions that favor different threads, the problems of fairness improvement and priority support are translated into well-studied time-sharing based scheduling problems. Fairness can thus be improved by giving different threads equal opportunity to speed up, while priority (or QoS differentiation) can be supported by allocating different numbers of time slices to different unfair partitions. The MTP partitioning algorithm further guarantees QoS by using partitions that, on average, can bound each thread's slowdown against the even partitioning baseline (PAR_Fair in Figure 5.3).

Depending on whether destructive inter-thread interference exists, a workload can either prefer cache partitioning or LRU-based sharing. In order to combine the strengths of cache partitioning and LRU, we integrate MTP with CC's baseline capacity sharing policy (1-Fwd as discussed in Section 4.2.2). The complementary benefits of these two approaches are achieved by dividing the total execution epochs into those controlled by either MTP or CC's baseline policy, according to the fraction of threads that can benefit from each of them. The integrated scheme, **Cooperative Caching Partitioning (CCP)**, can achieve robust performance for both workloads with and without destructive inter-thread interference. Furthermore, having CC as the default policy can simplify the MTP partitioning algorithm by focusing only on threads with large speedup potentials, leading to a heuristic-based algorithm that can be practically implemented.

5.2 Metrics and Methodology

In this section, we provide background information on multiprogramming metrics and evaluation methods to simplify later discussion.

5.2.1 Multiprogramming Metrics for CMP Caching

To compare the effectiveness of CMP caching schemes for multiprogramming, we first need to find proper metrics to summarize the overall performance, fairness and QoS results for a thread schedule. A multi-programmed workload's throughput can be simply measured as the sum of per-thread throughput (i.e., IPC for our workloads), but quantifying QoS and fairness can be hard, and requires an understanding of these notions in the context of CMP caching.

Our notions of performance, fairness and QoS are based on two principles: (1) proportional-share resource allocation and (2) Pareto efficiency. The first principle states that QoS and fairness is achieved when the shared resource is divided among sharers in proportion to their priorities or weights [19, 135, 151, 153][3]. Using proportional-share allocation to maintain the baseline fairness, the second principle further improve performance (efficiency) by allowing disproportional sharing if it helps some sharers without hurting the others. These principles have been used to define min-max fairness [17], which has wide applications in computer networks and scheduling policies (e.g., Generalized Processor Sharing [117]).

QoS Metric

QoS is the ability to provide a thread with guaranteed baseline performance (corresponding to a specific resource partition) regardless of the load placed on the shared resource from other co-scheduled threads [151]. We use **equal-share cache allocation** to define the performance bottom line for QoS, which corresponds to the special case of proportional-sharing when all threads have the same priority. Notice that equal-priority has been implicitly assumed by previous fair caching proposals [66, 84, 156], while our MTP scheme can also support threads with different priority levels (refer to Section 5.3.3). This baseline can be implemented either by uniform-sized private caches or an equal partitioning of shared cache capacity between on-chip cores, and it guarantees QoS because all threads get the same capacity and thus can achieve the same

[3] Contention in other shared resources, especially the memory system, can also cause destructive interference. Here we focus on the impact of destructive interference occurring in the last-level CMP caches, assuming a fair memory system as proposed in [114].

performance across different schedules. The equal-share allocation baseline also provides intuitive QoS results to multiprocessor users because it corresponds to traditional multiprocessors with private caches. For similar reasons, Yeh and Reinman [156] use this baseline implemented by private caches. Here we use the even partitioning of a shared cache as our baseline because most existing cache partitioning schemes assume a shared cache.

The QoS metric is thus defined as the sum of per-thread slowdowns (as negative percentages) over this baseline. Same as [114,156], we claim a caching scheme can guarantee QoS if this measurement is bounded within a user-defined threshold (e.g., -5%). Other ways of measuring QoS exist (e.g., reporting the maximum slowdown or the number of threads that violate QoS), but we use the total slowdown because it captures the behavior of the entire workload and thus is a more stringent criteria.

$$QoS(scheme) = \sum_{i=1}^{\#app} min(0,\ \frac{IPC_i(scheme)}{IPC_i(base)} - 1)$$

Fair Speedup Metric

According to the principle of Pareto efficiency, CMP caching schemes can further improve performance while maintaining fairness, if uneven resource allocation can speed up some threads over the equal-share allocation baseline without hurting others. Now we consider how to measure the scale of performance improvement for multiple co-scheduled threads.

Summarizing the overall performance of multiple benchmarks (co-scheduled threads in our context) has been an extensively discussed topic [79, 133]. We adopt prior wisdom and define the **Fair Speedup (FS)** metric to quantify the overall performance of co-scheduled threads. FS is calculated as the harmonic mean of per-thread speedups over the equal-share allocation baseline.

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

Using harmonic mean of speedups, FS measures the execution time reduction (more accurately, execution

resource occupation) against a baseline cache configuration that resembles traditional multiprocessors (so higher FS is better). FS is also a fair metric because using the harmonic mean (instead of the sum as used by [156]) rewards uniform speedups and penalizes slowdowns[4], which corresponds to the principle of Pareto efficiency.

The notion of fair speedup is similar to the fair slowdown metrics proposed by Kim et al. [84], which is measured against a single-thread execution baseline where one thread has exclusive use of all cache resources. Such a baseline is borrowed from SMT processors [134], where it corresponds to the single-thread execution mode that allocates all execution and cache resources to one thread. However, single-thread execution in a CMP will waste the majority of execution resources. Instead, we choose to use the equal-share allocation baseline because it has better resource utilization by supporting multiple concurrently running threads and performs similarly as in traditional multiprocessors. For the same reason, two other SMT performance metrics using a single-thread execution baseline—weighted speedup (or WS, which is the sum of speedups) [134] and harmonic mean of speedups [98]—are not used.

Metrics Comparison

The choice of optimization metrics has a significant impact on CMP caching policies. Below we use two examples to demonstrate the differences between caching schemes that optimize for different metrics.

Figure 5.4 shows the per-thread speedups of benchmarks `art` and `vpr` using two cache partitioning schemes (2 threads sharing a 2MB L2 cache). Scheme (A) maximizes weighted speedup (WS) by tripling the performance of `art`, however, its fair speedup (FS) measurement is worse than the baseline (FS = 1) due to unfair per-thread speedups. On the other hand, the fair scheme (B) optimizes FS at the cost of a lowered WS result because the low-speedup thread is given a more fair cache allocation. This example shows that (1) FS optimization has the side effect of avoiding unfair caching, (2) a scheme that optimizes FS may hurt its WS or IPC results, and vice versa.

[4] According to the power-mean inequality, the harmonic mean of a vector is maximized when all elements have the same value.

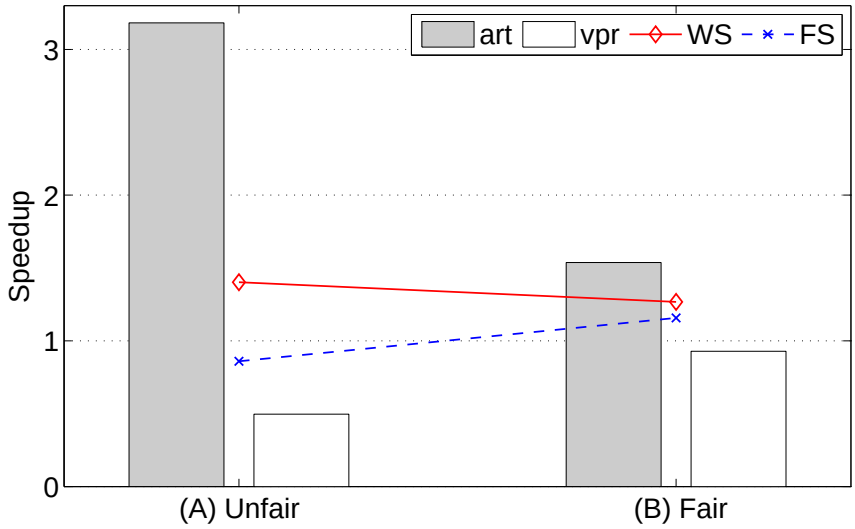


Figure 5.4: FS vs. WS for Two Example Schemes

Metrics	Scheme A	<=>	Scheme B
Per-thread Speedups	0.76 / 0.76 / 3.18 / 3.18		1.97 / 1.97 / 1.97 / 1.97
Throughput (IPC)	*0.52*	==	*0.52*
Weighted Speedup	*2.42*	==	*2.42*
QoS	**-52%**	<	**0%**
Fair Speedup	**1.22**	<	**1.97**

Table 5.2: Performance Comparison Using Different Metrics

Table 5.2 compares the performance of two cache partitioning schemes for workload `art-art-art-art`. Scheme A optimizes WS and throughput without considering its implications on fairness and QoS, while Scheme B aims to optimize FS while maintaining QoS. If only comparing throughput and WS results, the two schemes have the same performance (shown as *italic* in Table 5.2). However, our QoS and FS metrics (marked as **bold** in Table 5.2) reveal that Scheme A cannot guarantee QoS while Scheme B can, and Scheme B achieves better fairness and execution time than A. This example shows that our QoS and fair speedup metrics are able to distinguish whether a scheme can maintain QoS and fairness, but the WS and IPC metrics cannot.

To summarize, using QoS and FS metrics together, we can measure a caching scheme's effectiveness in

improving performance, fairness and QoS. We will report results using the FS and QoS metrics throughout this chapter, and provide WS and IPC results in the evaluation section for comparison.

5.2.2 Benchmark Selection and Characteristics

CMP caching schemes should be compared using a wide range of multiprogrammed workloads to evaluate their performance robustness. For evaluation purpose, we model a CMP with 4 single-threaded cores and consider all 4-thread multiprogramming combinations (repetition allowed) from 7 representative SPEC2000 benchmarks. There are 210 workloads because the number of K combinations (with repetition) selected from N objects is C_K^{N+K-1}, so selecting 4-thread combinations from 7 benchmarks can generate C_4^{7+4-1} (=210) workloads.

Figure 5.5 shows the performance of our benchmarks under different cache allocations. The IPC data are gathered using a 4MB 16-way total L2 cache. With way partitioning, cache resources are allocated in 256KB chunks. The equal-share allocation baseline (marked as the vertical line) is for each thread to

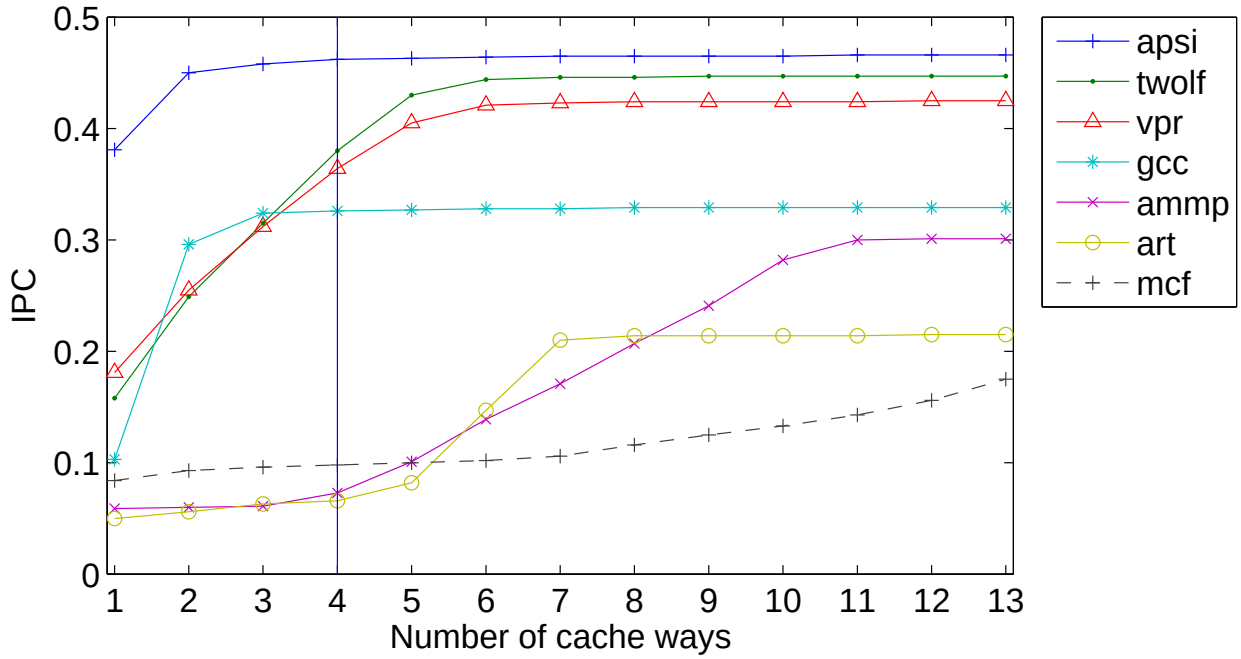


Figure 5.5: IPC Curves

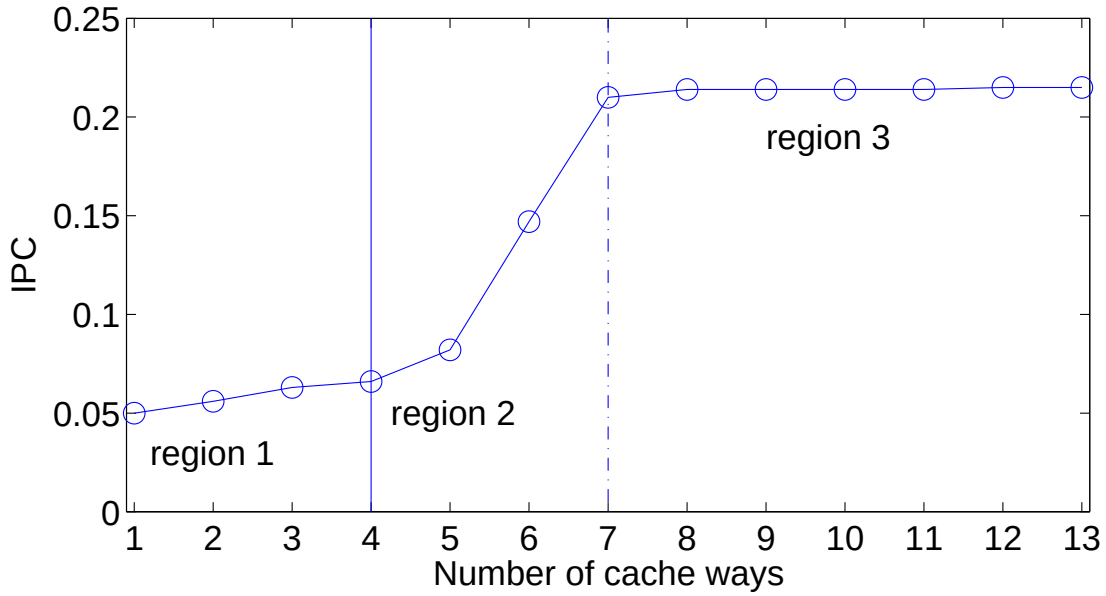


Figure 5.6: IPC of `art`

use 1MB cache. At least 1 way is allocated to each thread. With three other cores on-chip, this leaves 13 ways (13=16-3 $*$ 1), or 3.25MB, as the maximum capacity for one thread to have. This figure shows that our selected benchmarks have a wide variety of working set sizes, and IPC curve shapes, therefore their combinations are able to generate a wide range of workload behaviors.

To understand the relationship between capacity allocation and benchmark performance, Figure 5.6 breaks the IPC curve of `art` into three distinctive regions as more capacity is allocated: (1) pre-working-set region (from 1 way to 4 way) represents gradual speedups before the program's working set starts to fit into cache; (2) in-working-set region (from 5 way to 7 way) indicates dramatic throughput increases when the working set can be partly cached; (3) post-working-set region (starting from 7 way) shows saturated performance after the working set is fully cached. Except for benchmarks whose working sets are beyond the capacity of the on-chip cache (e.g., streaming benchmarks), most benchmarks demonstrate IPC curves that consist of regions with distinct slopes, albeit with different cache configurations.

According to which region intersects with the equal-share allocation (which is dependent on both benchmark characteristic and cache configuration), we classify these benchmarks into 3 categories. This classifi-

cation will be used to describe our cache partitioning heuristics.

- **Supplier benchmarks**: These benchmarks can supply some or all of their equal-share capacity for other benchmarks while still achieving the same level of performance as using all cache resources. They include workloads with very small working sets (e.g., `apsi` and `gcc` whose working set sizes are less than 1MB) and streaming benchmarks (e.g., `swim` and `facerec` which are not included in Figure 5.5).

- **Sensitive benchmarks**: `vpr` and `twolf` are benchmarks whose in-working-set regions are divided by the line of equal-share capacity. The performance of such programs changes significantly over the baseline as cache size varies, therefore judicious cache partitioning is needed when they are co-scheduled with other benchmarks.

- **Thrashing benchmarks**: `art`, `ammp` and `mcf` are benchmarks whose in-working-set regions are beyond their equal-share capacity. These benchmarks usually slow down gradually with reduced capacity, but can speed up dramatically when a certain amount of extra capacity is allocated.

A similar classification can be found in [123], according to the benefit of increased capacity (or utility). Our classification is different by separating thrashing benchmarks from sensitive benchmarks, both called high-utility programs in [123] because they can speed up with more capacity. We focus on thrashing benchmarks because they can easily benefit from our proposed cache partitioning policy.

5.2.3 Offline Analysis vs. Online Simulation

Because cache partitioning schemes are often coarse-grained, they are amenable to not only the commonly used online simulation approach, but also offline analysis [66, 84, 96]. To do offline analysis, we first gather performance profiles for all possible (benchmark, capacity) combinations. Comparing against each benchmark's baseline IPC, we can calculate the per-thread speedups for all (benchmark, capacity) com-

binations and use them to calculate metrics such as FS, WS and QoS. For a given metric, we construct the candidate cache partition space for each workload and exhaustively search in the partition space for the optimal result. Compared with online simulation, offline analysis is idealized because (1) it uses accurate measurement information and (2) it searches for all possible partitions, which can be too slow to be practically implemented.

Due to its idealized nature, offline analysis can be used to estimate the performance upper bounds for given cache partitioning policies. We will use this approach to demonstrate the advantage of our proposed MTP policy over prior cache partitioning schemes, and avoid the need to compare against realistic implementations of prior proposals. We will also compare the offline analysis results of MTP with online simulation results of LRU-based caching schemes to identify the limitation of cache partitioning schemes. However, our final scheme CCP, which integrates MTP with CC, will be evaluated using a practical implementation, online measurement information, and execution-driven simulation results.

Our offline analysis method assumes the evaluated execution interval has stable program phases (which can be composed by regular interleaving of sub-phases). For our selected benchmarks, this requirement is satisfied by using sufficiently long execution epochs to include multiple sub-phases, whose aggregate behavior is stable enough for cache partitioning. Figure 5.7 shows the phase behaviors of benchmarks `art` and `gcc` under different epoch sizes (5M, 10M, 20M, and 40M cycles). The benchmarks are allocated with either 256KB or 2.56M to demonstrate the caching behaviors under both small and large capacities. For epoch size of 5M-cycle, both benchmarks experiences irregular phase changes especially using small capacity. Because cache partitioning schemes' prediction of future execution relies on stable phases, such irregularity can lead to sub-optimal partitioning decisions. However, as the epoch size increases, irregular phase changes gradually disappear (especially beyond 20M-cycle). Based on these data, we set our epoch size to be 20M cycles[5].

[5]Notice 20M-cycle is already the length of normal operating system scheduling interval on a 2GHz machine. Due to large epoch sizes, current cache partitioning schemes cannot adapt to frequent thread-scheduling changes. Software/hardware cooperation is thus needed to support environments with frequent scheduling changes, which we leave as future work.

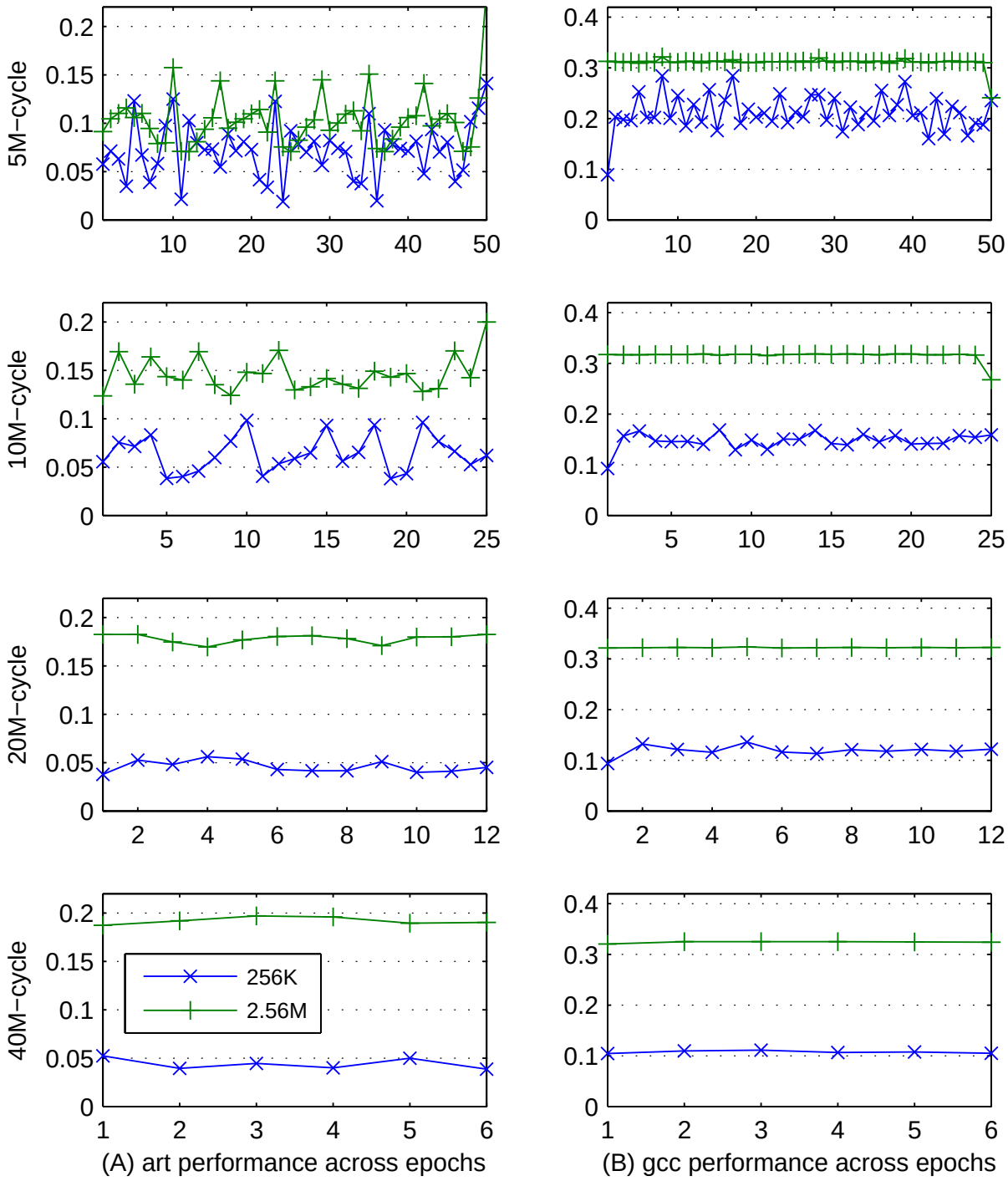


Figure 5.7: Selection of Execution Epoch Size

5.3 Multiple Time-Sharing Partitions (MTP)

Prior CMP cache partitioning policies use a single spatial partition to achieve their optimization goals. However, it is an intrinsically hard problem to satisfy multiple goals (e.g., throughput, fairness and QoS) with a single partition when conflicts arise between competing threads. In this section we add a time-sharing aspect on top of multiple spatial partitions, which uses Multiple Time-sharing Partitions (MTP) to resolve such conflicts over the long term. Below we detail the development of MTP as we add support for different cache management functionalities.

5.3.1 Thrashing Avoidance

We first discuss when cache partitioning is needed by examining when destructive interference occurs. Starting with the equal-share allocation baseline, if this configuration can satisfy the caching requirements of every co-scheduled thread, then cache partitioning is not needed because little inter-thread interference exists. Cache partitioning is needed only if some threads experience thrashing with their current capacity allocations. These threads will attempt to acquire extra cache resources from each other and from other threads, which leads to performance, fairness and QoS problems.

Thrashing is a classic virtual memory management problem [35], and can be avoided by reducing the multiprogramming level: when the number of competing programs is reduced to a point that their working sets can be cached simultaneously, they can all run much faster. In the context of CMP caching, the number of co-scheduled threads is determined by the operating system, but cache partitioning can intentionally manage capacity contention by unfairly shrinking the capacities of some thrashing threads to expand the capacities of other thrashing threads.

Consider partitioning a 4MB 16-way L2 cache between 4 co-scheduled copies of `art`. With the equal-share allocation of a 1MB L2 cache, `art` has a low IPC of 0.066 due to thrashing (over 50 off-chip misses per thousand instructions) as previously shown in Figure 5.6. As more cache resources are allocated, its

throughput increases quickly and reaches a saturating point of 0.215 IPC with 1.75MB capacity. At this point, thrashing can be avoided for 2 threads by unfairly expand each of their capacity allocate to 1.75MB, and shrink the capacities of the other threads to 256KB each (0.05 IPC). This partition doubles the total throughput ($0.215 * 2$+$0.05 * 2$=0.52, which is two times of $0.066 * 4$=0.264), but is unfair to the shrinking threads.

5.3.2 Fairness Improvement

Cache partitioning between 4 copies of `art` is an example of the throughput-fairness dilemma. When the available cache capacity can not simultaneously satisfy the working set requirements of multiple large threads, compromise has to be made within a single spatial partition. In this example, fair partitions cause thrashing for all threads, while thrashing avoidance requires unfair partitioning. Existing cache partitioning schemes all face this dilemma, but differ in the way they trade off between throughput and fairness.

We resolve this dilemma by learning from a similar example in game theory [44]. Consider two office-mates who commute to their workplace, performance is doubled when they carpool but it is unfair because the driver invests more effort and money. Not carpooling is a fair strategy, but is also inefficient. In real life, such games are played daily by the same players who often improve both performance and fairness by "taking turns" to drive when they carpool. We adopt the same cooperative policy to simultaneously improve throughput and fairness with multiple time-sharing partitions (MTP). Instead of using a single partition that is either low-throughput or unfair, multiple unfair but high-throughput partitions are used in a time-sharing manner to also improve fairness.

Specifically, individual threads are coordinated to shrink and expand their cache allocations in different cache partitions. Within a partition, the spare capacity collected from shrinking threads is used by expanding threads, and different threads are expanded in different partitions. As a thrashing thread goes through shrinking and expanding partitions, its average throughput can be much better than its baseline throughput.

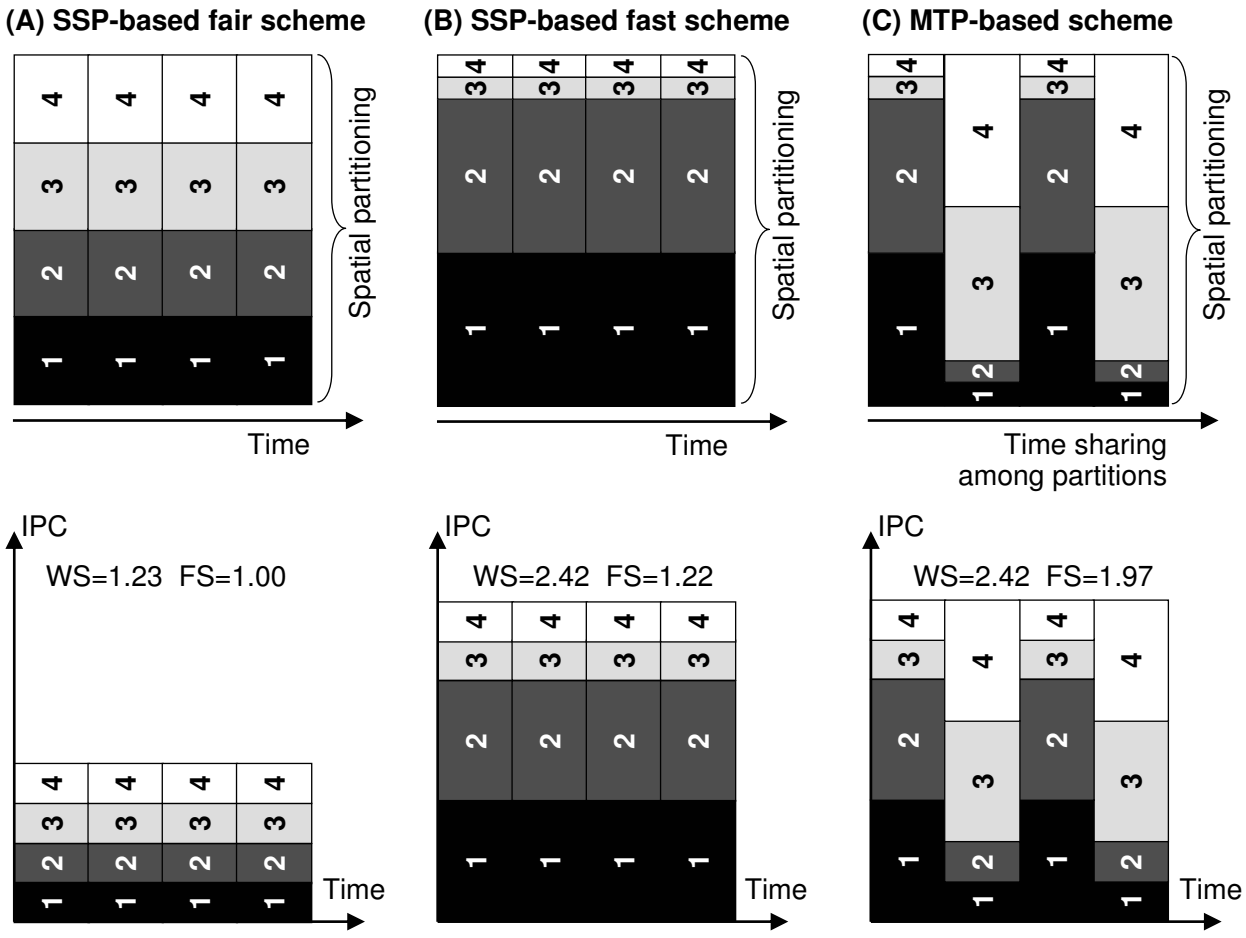


Figure 5.8: Cache Partitioning Options for a Co-schedule of 4 Copies of `art`

This is because a thrashing thread's baseline performance is already low by definition, and shrinking its capacity usually only causes insignificant slowdown. However, it can achieve dramatic speedup in one expanding partition (when the allocated cache can hold its working set) and, on average, the speedup in one expanding partition is often more than what is needed to compensate the slowdowns in multiple shrinking partitions. Overall, the multiprogrammed workload's fair speedup measurement is improved because all expandable threads get a fair chance to speedup.

Figure 5.8 compares three cache partitioning schemes for 4 copies of `art`. Single spatial partition based schemes A and B provide the most fair and fast partitions, respectively. Based on MTP, scheme C can both

maintain the same level of fairness as scheme A (by equalizing per-thread speedups) and achieve the same high throughput (IPC=0.52) and weighted speedup (WS=2.42) of scheme B. Such improvement is reflected by its high FS result (97% and 61% higher than scheme A and B, respectively), but can be overlooked by only comparing IPC or WS results.

5.3.3 Priority Support

MTP extends the option of cache partitioning from the single dimension of space-sharing into two-dimensional time-sharing between spatial partitions. The time-sharing optimization can be applied to any proportional-sharing resource partition baseline, thus supporting priority if the priority levels of co-scheduled threads are reflected in the baseline.

Priority can also be supported through time-sharing. Instead of giving different threads equal opportunity to speedup, different time-sharing priorities can be assigned to different unfair partitions to deliver differentiated levels of performance. Because time-sharing based priority support has been well understood and implemented by operating systems [62, 153], MTP can serve as the cache management primitive to the high-level software by focusing on the determination and enforcement of multiple unfair partitions.

As priority specification and interpretation are usually conducted by end-users and operating systems, we leave the development and evaluation of priority algorithms for future work and assume the co-scheduled threads have the same priority for the rest of the dissertation.

5.3.4 QoS Guarantee

QoS can be guaranteed either in a real-time manner or over the long term to meet different application's timing requirements. Real-time QoS is needed only by certain applications (e.g., real-time video playback or transaction processing systems), and is not needed for many other programs. For example, users of SPEC benchmarks, batching systems and scientific applications are mostly concerned about total execution time,

and thus long-term QoS, often measured over hundreds of millions of cycles.

To guarantee real-time QoS, fast and fair partitioning [156] reserves for each thread a **guaranteed partition**, which is the minimum amount of cache space required to achieve the same level of performance as using the equal-share cache allocation. Further speedup can be obtained by intelligently partitioning the remaining space. However, because only supplier benchmarks (defined in Section 5.2.2) can have their guaranteed partitions smaller than the equal-share capacity, the cache partitioning algorithm is often left with limited amount of space to optimize, which results in low performance cache partitions compared with schemes under no QoS constraints.

Single spatial partition (SSP) based cache partitioning schemes experience the same problem even for threads that require only long-term QoS. Because the same cache partition is used repeatedly throughout a stable program phase, these schemes have to guarantee long-term QoS by guaranteeing QoS within every cache partition. In contrast, MTP's cooperative shrink/expand model can be used to guarantee long-term QoS with little loss of performance. To meet the QoS requirement, the MTP partitioning algorithm now uses multiple partitions to maximize FS, under the constraint that each thread's average throughput across multiple partitions is no worse than the equal-share baseline throughput.

To demonstrate MTP's advantage in guaranteeing long-term QoS, Figure 5.9 and Figure 5.10 compare SSP vs. MTP based cache partitioning schemes that optimize FS under different types of QoS requirements: no QoS (SSP_{noQoS} and MTP_{noQoS}), real-time QoS (SSP_{QoS} and MTP_{rtQoS}), and long-term QoS (MTP_{ltQoS}). These results are obtained from offline analysis (described in Section 5.2.3) to demonstrate the performance potential of MTP_{ltQoS} implementation. In Sections 5.4 and 5.5, we will develop and evaluate its practical implementation.

For each scheme, we plot the percentage of workloads that can achieve various metric values. These curves are essentially Cumulative Distribution Functions (CDF) being transposed, so that a higher curve indicates a better performing scheme. For example in Figure 5.9, each point (X%, Y) on the MTP_{ltQoS}

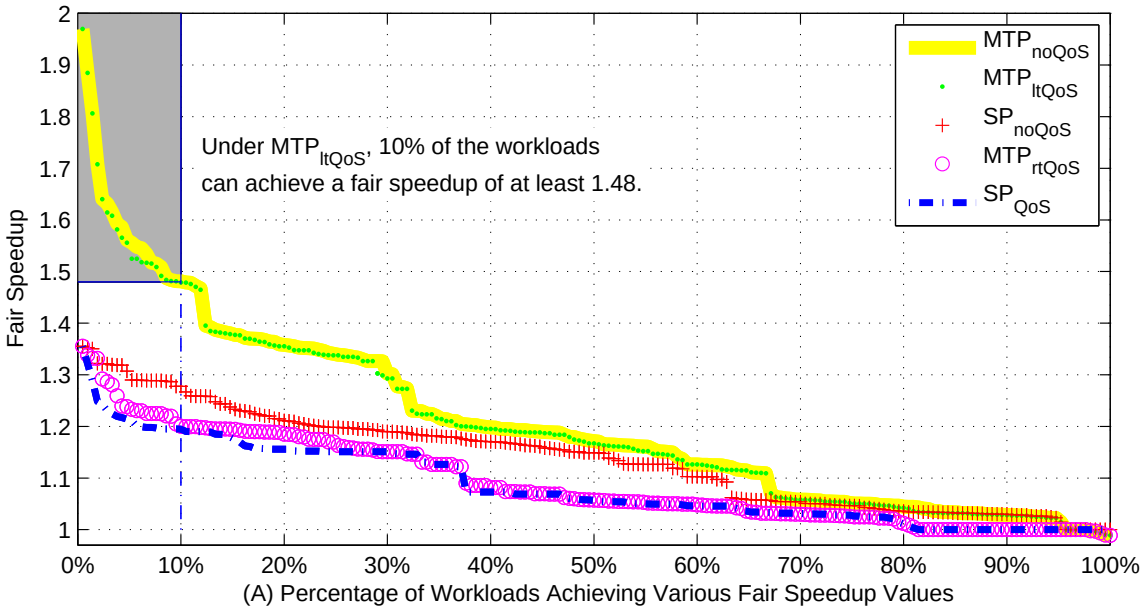


Figure 5.9: FS Comparison of SSP and MTP Under Different QoS Requirements

curve indicates that, X% of workloads have FS measurements equal to or large than ($\geq$) value Y. Notice that for the same type of QoS guarantee, the ideal SSP scheme can never outperform the ideal MTP because single spatial partitioning is a special case in the MTP model, which is also empirically shown in the figure.

Figure 5.9 shows 4 distinct curves for 5 schemes because MTP$_{ltQoS}$ and MTP$_{noQoS}$ have almost the same FS results. Similarly, the curves of the QoS guaranteeing schemes overlap in Figure 5.10. The two figures together show that (1) SSP$_{noQoS}$ and MTP$_{noQoS}$ can not bound per-thread slowdown within the user-specified threshold (-5% here); (2) SSP$_{QoS}$ and MTP$_{rtQoS}$ are the worst performing policies (their curves overlap for workloads with smaller FS values), indicating that real-time QoS guarantee can restrict performance optimization; and (3) MTP$_{ltQoS}$ can maintain long-term QoS while achieving almost the same performance as the best performing scheme MTP$_{noQoS}$.

For its performance and QoS benefits, we now use MTP$_{ltQoS}$ as the representative MTP policy, and denote it directly as MTP. MTP can be extended to support real-time QoS by reserving guaranteed partitions for real-time applications and optimizing the rest of programs with the remaining capacity.

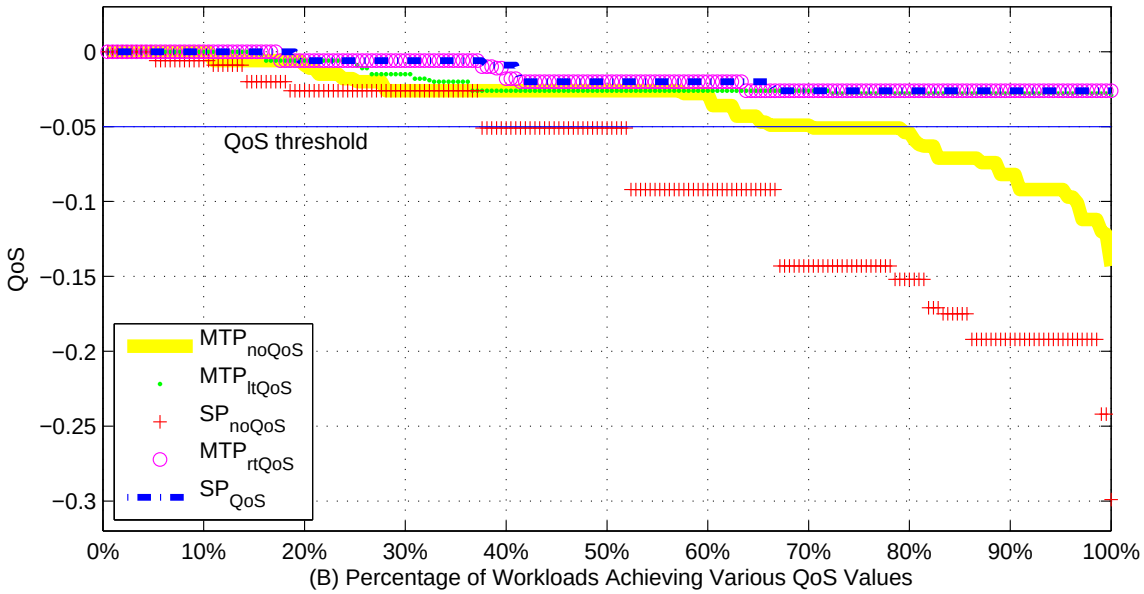


Figure 5.10: QoS Comparison of Various SSP and MTP Based Schemes

5.3.5 Summary

MTP is a high-level cache partitioning policy that extends existing proposals with time-sharing multiple cache partitions. MTP addresses four cache partitioning requirements: (1) thrashing is avoided with unfair allocation within a partition; (2) fairness is improved with fair time-sharing between unfair partitions that each favors a different subset of co-scheduled threads; (3) priority can be supported with unfair time-sharing; and (4) different types of QoS can be guaranteed by bounding per-program slowdown within each partition or across multiple partitions.

MTP can be implemented in various ways. A hardware-only solution is transparent to software but less flexible, especially considering priority support. Cooperation between hardware and software allows hardware to collect measurement and enforce partitioning decisions, and software to schedule partitions based on high-level requirements.

Our offline analysis results show that MTP can significantly outperform the best SSP based cache partitioning scheme while maintaining long-term QoS. However, in order to realize MTP's benefits, we still

need to answer two challenges. First, with both spatial partitioning and time-sharing of spatial partitions, the number of possible MTP partitions is many times more than SSP based partitions. We need to come up with an MTP partitioning algorithm that can quickly prune the large partition space and is simple enough for hardware implementation. Second, same as other cache partitioning schemes, MTP is only needed for some workloads, and not needed for workloads without destructive interference. We need to combine the strength of MTP with LRU-based sharing and dynamically choose the best policy for a given workload. In the next section, these issues are addressed by the other aspect of our proposal—integration of MTP with CC's LRU-based policies.

5.4 Integrating MTP with CMP Cooperative Caching

This section addresses another limitation of existing cache partitioning proposals—inadequacy for workloads that are well supported by LRU-based latency-reducing caching schemes, where cache partitioning can hurt. We propose a new hybrid scheme, Cooperative Cache Partitioning (CCP), that combines the advantages of MTP and CC's latency optimizations. Below, we motivate the need for the integration by showing the complementary advantages of MTP and CC on different workloads. We then develop a simple online heuristic to select MTP partitions based on the different characteristics of MTP and CC, and extend the CC design to implement the hybrid scheme.

5.4.1 Motivation

Figure 5.11 compares the fair speedup and QoS results of MTP with two LRU-based caching schemes: CC and shared cache, using the same aggregate cache size and associativity. We use scatter plots to reveal the correlation between the best performing schemes and workload characteristics. To show the advantages of MTP and CC over shared cache, we normalize performance results (FS) against the better results provided by MTP and CC (i.e., **Max[FS(MTP), FS(CC)]**). To compare between MTP and CC, we cluster the 210

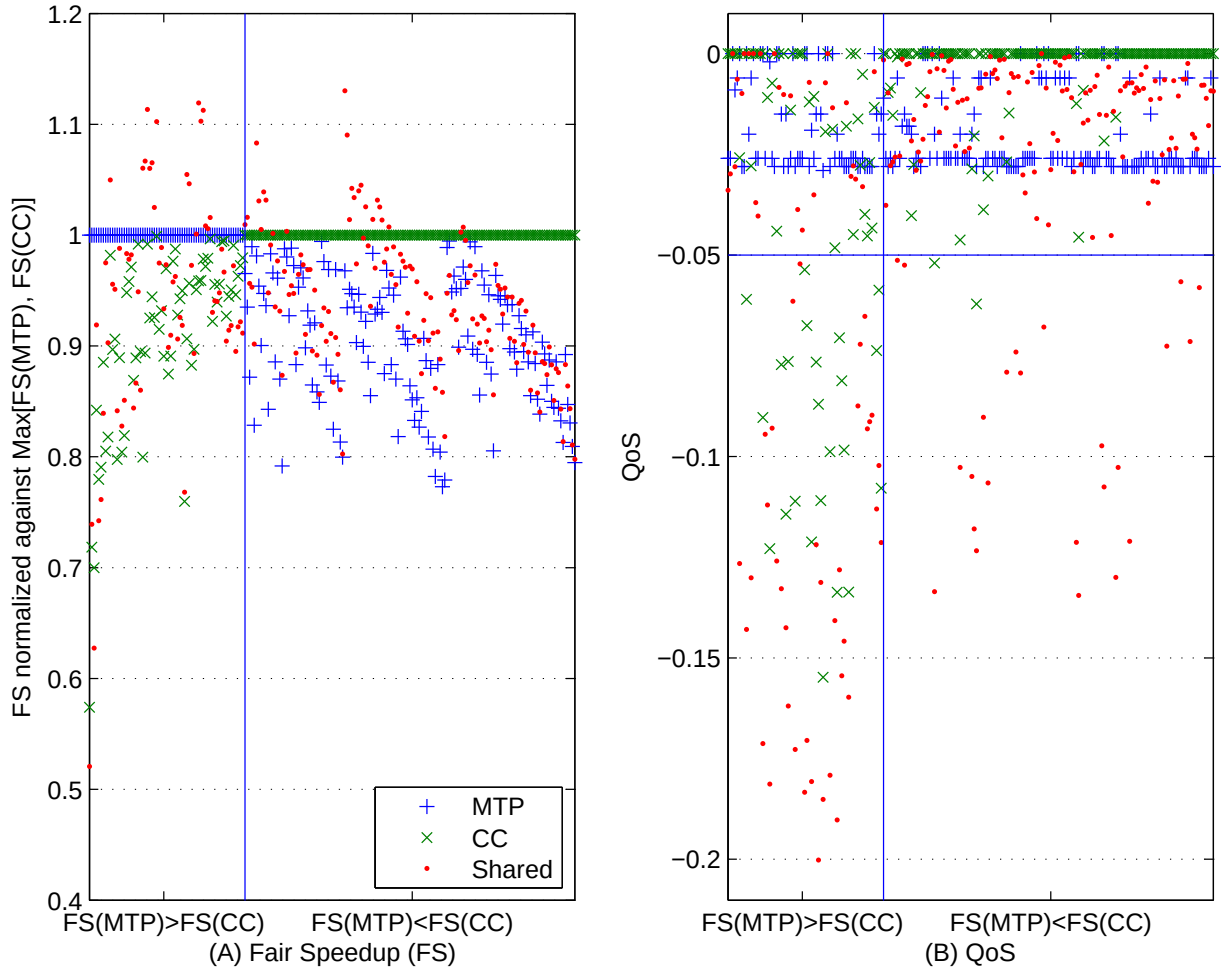


Figure 5.11: Comparing MTP with CC and Shared Cache (Each point on the X-axis represents a workload, and the corresponding Y-values are the measured results of the three schemes.)

combinations of benchmarks into two groups according to whether MTP outperforms CC (i.e., whether **FS(MTP) > FS(CC)**).

Several observations can be made from Figure 5.11. First, Figure 5.11 (A) shows that only a small number of dots (10%) are over 1 and fewer (3%) are over 1.1, indicating that a shared cache only outperforms both MTP and CC infrequently and insignificantly. Second, MTP only provides better performance than CC for 32% of the workloads (67 out of 210 workloads), indicating the limited effectiveness of cache partitioning over CC for many workloads. Third, for workloads that benefit less from MTP, CC is almost

always the best performing scheme (Figure 5.11 (A)) and can guarantee QoS (Figure 5.11 (B)), showing the complementary strengths of MTP and CC.

These observations imply that we can choose the best performing scheme (in both FS and QoS) for a given workload according to whether **FS(MTP) > FS(CC)**: if MTP provides better FS result than CC, then MTP is almost always the best scheme; otherwise, CC is the best choice. Therefore, a hybrid scheme that integrates MTP and CC can potentially provide the best performance for all workloads by simply choosing the better scheme for any given workload. Below we analyze the reasons for CC and MTP's performance advantages, in order to achieve such an integration.

Advantages of CC

Two major reasons contribute to CC's performance advantage over MTP: (1) latency optimization over shared cache and (2) LRU-based fine-grained cache sharing. The first reason is unique to CC—it is the only proposed private cache based CMP caching optimization that approximates global LRU replacement for multiprogrammed workloads; the second is supported by both CC and a shared cache.

CC reduces average cache access latency by keeping a program's data set locally in the processor's private L2 cache. Due to on-chip wire delay, local cache access latencies are much lower than remote access latencies. Comparing with a shared cache where data are distributed evenly across all banks and a large fraction of L2 accesses are to remote banks, CC has the advantage of servicing most L2 accesses locally. For threads whose working sets can be mostly satisfied by a private cache, such reduced L2 cache latencies often translate into higher performance.

Similar to a shared cache, CC supports LRU-based capacity sharing by (1) allowing a local cache's victim block to be placed in a randomly picked peer cache (i.e., spill), and (2) approximating global LRU replacement for multiprogrammed workloads via the combination of local LRU and global spill/reuse history. CC differs from a shared cache in that it allocates capacity according to the local thread's L2

reference stream and the remote threads' L2 miss streams, which are first filtered by their local L2 caches.

As discussed in Section 5.1.1, an LRU-based policy allocates cache resources in both spatially and temporally fine-grained manner, and can often outperform coarse-grained cache partitioning schemes. Programs with highly non-uniform demands across different cache sets [124] can also benefit from fine-grained sharing. For example, although `ammp` has a working set size of 1.5MB (or 6 cache ways), it can further speed up by 2X when the associativity requirements of certain hot sets are satisfied by a 16-way allocation.

Heuristics for MTP to Outperform CC

We now try to discover the characteristics of the workloads that can achieve better performance with MTP than with CC. Such characteristics will be used to develop a simple heuristic to integrate MTP with CC.

For brevity, we use expanding and shrinking partitions to denote spatial partitions in which a given thread's capacity is above and below its equal-share allocation, respectively. To simplify discussion, we assume that within one group of MTP partitions, a thread always uses the same capacity C_{expand} in all of its expanding partitions and the same capacity C_{shrink} in all shrinking partitions, thus achieving the same speedup Sp and slowdown Sd repeatedly. Offline analysis results show that this assumption has almost no performance impact on MTP. We further filter out supplier benchmarks by allocating their guaranteed partitions to them, and allocate the remaining space between other threads.

To guarantee QoS and improve performance using MTP, each thread's total speedup has to exceed its total slowdown, and at least one thread should have a much larger total speedup. This can be achieved in two ways. The first way is to have a thrashing benchmark whose dramatic Sp can compensate the total slowdown in multiple shrinking partitions. The other way for a thread to achieve speedup using MTP, without a large Sp, is to have a modest Sp, but a steep speedup curve and a gradual slowdown curve, so that the speedups accumulated in multiple expanding partitions exceed the slowdown in one shrinking partition. However, achieving a modest Sp along with a steep speedup curve requires only a small amount of extra capacity, in

which case CC is likely to achieve the same effect because the LRU policy is better at fine-turning cache allocation to achieve speedups (example shown in Figure 5.2).

The above analysis suggests that the common case for MTP to outperform CC is to have at least one thrashing benchmark, determined by whether its speedup Sp in one expanding partition is larger than the total slowdown accumulated in shrinking partitions. Here the Sp and Sd values are dependent on both the thread's IPC curve and the available capacity (which further depends on capacity allocated to co-scheduled threads). The test of a thrashing benchmark will be used as the partitioning heuristic for MTP.

The common case also explains why CC can guarantee QoS when it achieves better fair speedup than MTP (Figure 5.11 (B)). A QoS violation occurs in CC only when a thread's private cache is overly used by blocks replaced from other threads (i.e., spilled blocks). Because CC's private cache has the same capacity as the equal-share baseline used to define thrashing, the aggressive spilling implies the existence of high miss rate, and thus thrashing benchmarks. Therefore, for workloads that prefer CC, the spilling should not be too invasive to affect QoS, otherwise thrashing will occur, causing MTP to be preferred.

5.4.2 Cooperative Cache Partitioning (CCP)

We now develop cooperative cache partitioning (CCP), a heuristic-based hybrid cache allocation scheme that integrates MTP with CC. CCP consists of three components: (1) a heuristic-based partitioning algorithm to determine MTP partitions; (2) a weight-based integration policy to decide when to use MTP or CC's LRU-based capacity sharing policy (1-Fwd, discussed in Section 4.2.2); and (3) modifications to the baseline CC design to enforce fine-grained cache partitioning decisions.

CCP Partitioning and Weighting Heuristics

Before MTP partitioning, CCP first gathers each thread's L2 cache miss rates under candidate cache allocations, and uses them to estimate the IPC curve. Miss rates are collected in our simulator in dedicated,

online sampling epochs where each thread takes turns to use the maximum amount of cache. We use LRU stack hit counters to estimate miss rates under all possible cache associativities to reduce sampling overhead. Although such overhead can be avoided with the recently proposed UMON online sampling mechanism [123], we include it in our evaluation results.

Using IPC estimations, each thread's guaranteed partition (for real-time QoS guarantee) can be calculated. CCP also initializes each thread's C_{expand} to the minimum capacity needed to achieve the highest speedup, and C_{shrink} to the minimum capacity that can ensure long-term QoS when cooperating with C_{expand}. A thread is a supplier benchmark if its C_{shrink} is the same as its guaranteed partition.

The CCP partitioning algorithm (shown in Table 5.3) then returns a set of MTP partitions that are likely to outperform CC, using the test of a thrashing benchmark as a simple heuristic. This algorithm has the following three steps: (1) filtering out supplier benchmarks which will not benefit from any partitioning schemes; (2) determine MTP partitions that each favors one thrashing benchmark by starving other thrashing benchmarks with their C_{shrink} capacity; (3) for MTP partitions where one expanding thread can not use all the remaining space, expand other threads to further increase speedup. We will describe steps (2) and (3) in detail because step (1) is rather straightforward.

Step (2) determines the set of thrashing benchmarks by removing threads whose speedups are not large enough to guarantee long-term QoS. Each candidate thread is tested by the function **thrashing_test** to see whether its speedup in one expanding partition can compensate for the total slowdown accumulated in other (shrinking) partitions. The threads that fail the **thrashing_test** are assigned with their guaranteed partitions and removed from the candidate set, which will reduce the number of candidate partitions, the amount of remaining capacity and possibly remaining candidates' C_{expand} and speedups. Such tests are repeated until one of the two termination conditions is satisfied: (1) the candidate set is empty, or (2) all candidate threads pass the test. This step is guaranteed to terminate because each round of tests either reduces the candidate set size which leads to condition (1) in a finite number of steps, or satisfies condition (2).

```
Inputs: capacity C, thread set TS, sample results (IPC[i][c], guaranteed partitions g[i]);
Outputs: expanded[i], MTP partitions MTP[p][i];   /* Thread i's capacity in partition p */
/* Step 1: Filter out supplier benchmarks */
Identify supplier benchmarks SupplierTS, subtract their g[i] from C;

/* Step 2: Determine the set of thrashing benchmarks ThrashTS */
/* init stable = false; ThrashTS=TS-SupplierTS; */
while (ThrashTS is non-empty and !stable)
    stable=true;
    foreach thread i∈ThrashTS
        C_expand[i]=i's capacity usage when other threads use their C_shrink[j];
        stable &= thrashing_test(i, size(ThrashTS), C_expand[i], C_shrink[i]);

/* Step 3: Merge multiple expanding threads */
/* init p = 0; expanded[i] = false; MTP[p][j]=C_shrink[j]; */
foreach thread i∈ThrashTS, p++
    foreach thread j, start from i, in circular order
        MTP[p][j] += minimal remaining capacity for j to achieve its best speedup;
        if (MTP[p][j]≥C_expand[j]) expanded[j]=true;   /* Expanded in MTP */
thrashing_test(i, nump, expand, shrink)      /* Key heuristic */
    if (IPC[i][expand]-IPC[i][base])>(nump-1)*(IPC[i][base]-IPC[i][shrink])
        return true;      /* large speedup */
    C_shrink[i]=g[i]; C=C-g[i]; remove i from ThrashTS;
    return false;
```

Table 5.3: CCP Partitioning Algorithm

After step (2), it is possible that in an MTP partition, the expanding thread does not need all the spare space provided by other shrinking threads. Step (3) merges multiple expandable threads in such a partition to further increase speedup. To be fair, the algorithm attempts to expand different sets of threads in different partitions.

This algorithm also returns a vector $expanded$. A thread i benefits from MTP if it is allocated with C_{expand} capacity in at least one partition ($expanded$[i] is true), otherwise it is likely to benefit from CC. This observation leads to the CCP integration heuristic: the execution time is broken into epochs managed by either MTP or CC's LRU-based capacity sharing policy (1-Fwd), weighted by how many threads can benefit from them respectively. For N concurrently running threads, if M of them can be expanded by MTP partitions, then CCP will use MTP for every M out of N epochs and use CC for other epochs. A special case

is when no thread is expanded because step (2) cannot find any MTP partitions, in which case CC should be used throughout the execution.

Extending CC to Enforce Capacity Quota

CCP uses the quota-based throttling to enforce MTP partitions. Compared with way partitioning, CC's fine-grained cache-level quota enforcement can support threads with non-uniform capacity demands across different cache sets. For MTP partitions based on way partitioning miss rates, such threads only use part of its capacity quota in their expanding partitions while still achieving large speedups. CCP detects such cases and triggers the partitioning algorithm with newly collected capacity usage, which often leads to better results.

5.5 Evaluation and Results

We evaluate the effectiveness of different cache allocation schemes using the same Simics-based full system simulator as described in Chapter 4. Here the same set of cache/memory/interconnect configuration parameters as in the default setup, but execution is driven by a single-issue, in-order processor model. The simpler processor model allows us to simulate all 210 multiprogrammed workloads in a manageable time frame. We choose this methodology because, under the same simulation time, simulating a wide range of workload combinations allows us to recognize the limitations of different approaches on different workloads, which could have been missed by simulating a few combinations with a more aggressive processor model.

The same total 4MB L2 cache capacity and 16-way associativity are used for shared cache (both with and without way partitioning) and CC. Except for `art` which uses the train input, we use the reference input sets for other selected SPEC benchmarks[6]. All benchmarks are fast forwarded by 800M instructions to bypass program initialization, and simulated for 700M cycles.

[6] `Art` with reference input is a streaming (thus supplier) benchmark. We do not include streaming benchmarks because cache partitioning for them is very simple: their IPCs don't change with L2 cache allocations, so we can simply allocate the minimal capacity (e.g., 1 cache way) to them.

We compare the online simulation results of realistic CCP implementation with offline analysis results of ideal cache partitioning policies (e.g., MTP). Because the ideal MTP implementation results were shown to be the performance upper bound of existing cache partitioning schemes in Section 5.3, we do not compare CCP with realistic implementations of prior cache partitioning proposals.

We first compare CCP with its two baseline schemes CC and MTP in terms of fair speedup, followed by comparison between CCP with idealized offline analysis results on two other metrics—throughput and weighted speedup. We then evaluate the robustness of CCP by halving the total cache size. Lastly, we compare the results of using in-order vs. out-of-order processor models with a subset of workloads.

5.5.1 Effectiveness of CCP

In Section 5.4, MTP was shown to be better than CC for only a subset of workloads. Since the ideal MTP implementation represents the best cache partitioning results, we now refer to workloads that prefer CC over MTP as workloads where cache partitioning could hurt performance, and the other workloads as workloads that need the help of cache partitioning. Figure 5.12 compares the performance of CCP (realistic) with MTP (ideal) and CC (realistic) on both classes of workloads. Only FS results are reported because both CCP and MTP can guarantee QoS. Same as in Figure 5.9, we use transposed CDF curves to show the percentage of workloads that can achieve various levels of performance. Here, a higher curve indicates a better scheme because it achieves better FS measurements across different fractions of the workloads, and the gaps between curves correspond to their performance differences.

Figure 5.12 (A) shows that when cache partitioning is needed, CCP achieves comparable performance as MTP (the gap between CCP and MTP curves is small), and much better FS values than CC (the gap between CCP and CC is relatively large). The performance difference between CCP and MTP reflects the difference between our practical partitioning heuristic and a less realistic, offline, exhaustive search of MTP partitions. For workloads where cache partitioning hurts, Figure 5.12 (B) shows that CCP performs almost

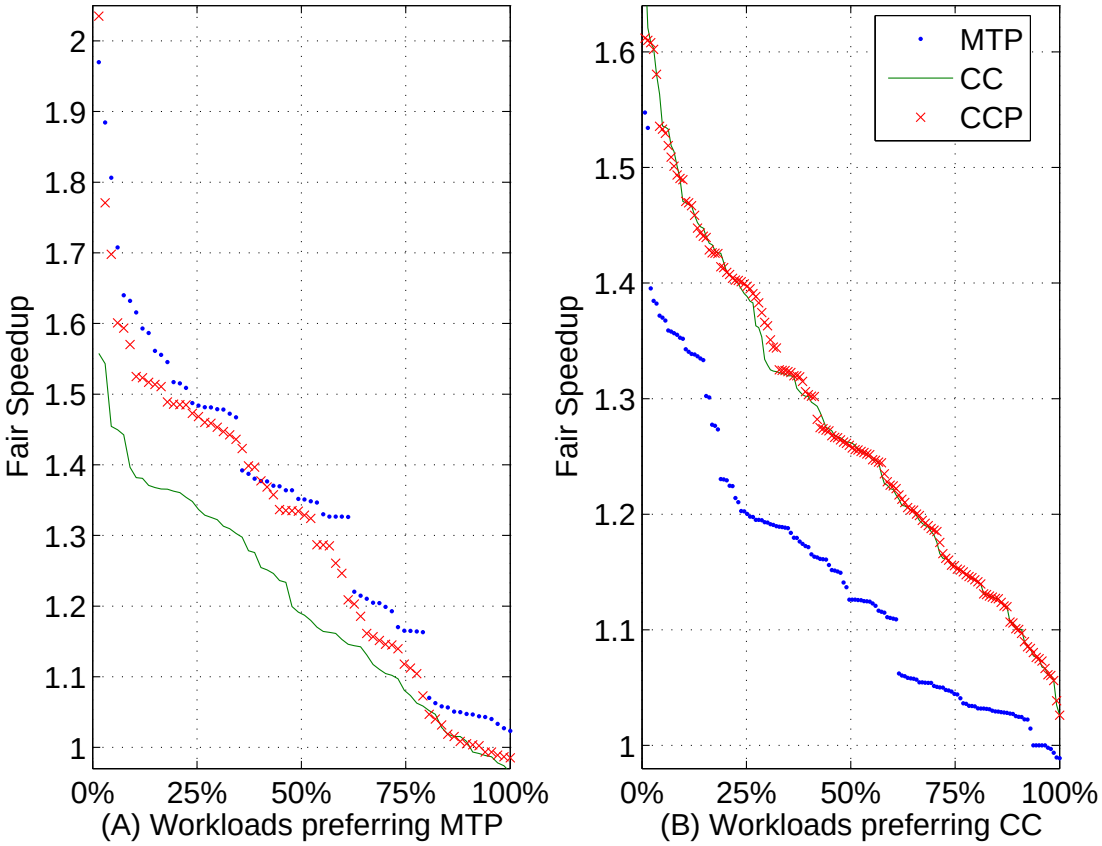


Figure 5.12: Comparing MTP, CC and CCP's FS Results

the same as CC and significantly better than MTP. Together they demonstrate that CCP effectively combines the strengths of both MTP and CC.

5.5.2 Results of Different Metrics

Besides FS, CMP caching performance can also be evaluated using other metrics. We use transposed CDF plots to compare CCP (realistic) and MTP (ideal) against two single spatial partition (SSP) based schemes IPC_{opt} and WS_{opt}, which optimize offline for throughput and weighted speedup, respectively. Focusing on workloads that need cache partitioning, Figure 5.13 compares IPC_{opt}, WS_{opt}, MTP and CCP over 4 different metrics: (A) fair speedup, (B) QoS, (C) throughput, and (D) weighted speedup.

For the first two metrics (fair speedup and QoS), MTP and CCP are both significantly better than IPC_{opt}

and WS_{opt}. This is because the SSP based schemes, when their goals conflict with fairness and QoS requirements, often optimize by favoring only a subset of threads while sacrificing the performance of other threads. For IPC and WS metrics, both IPC_{opt} and WS_{opt} are better, although the gap between different schemes are much smaller than in Figure 5.13 (A) and (B). As illustrated in the metrics comparison examples (Section 5.2.1), this is because schemes optimizing for WS, IPC and FS have different tradeoffs between performance vs. fairness.

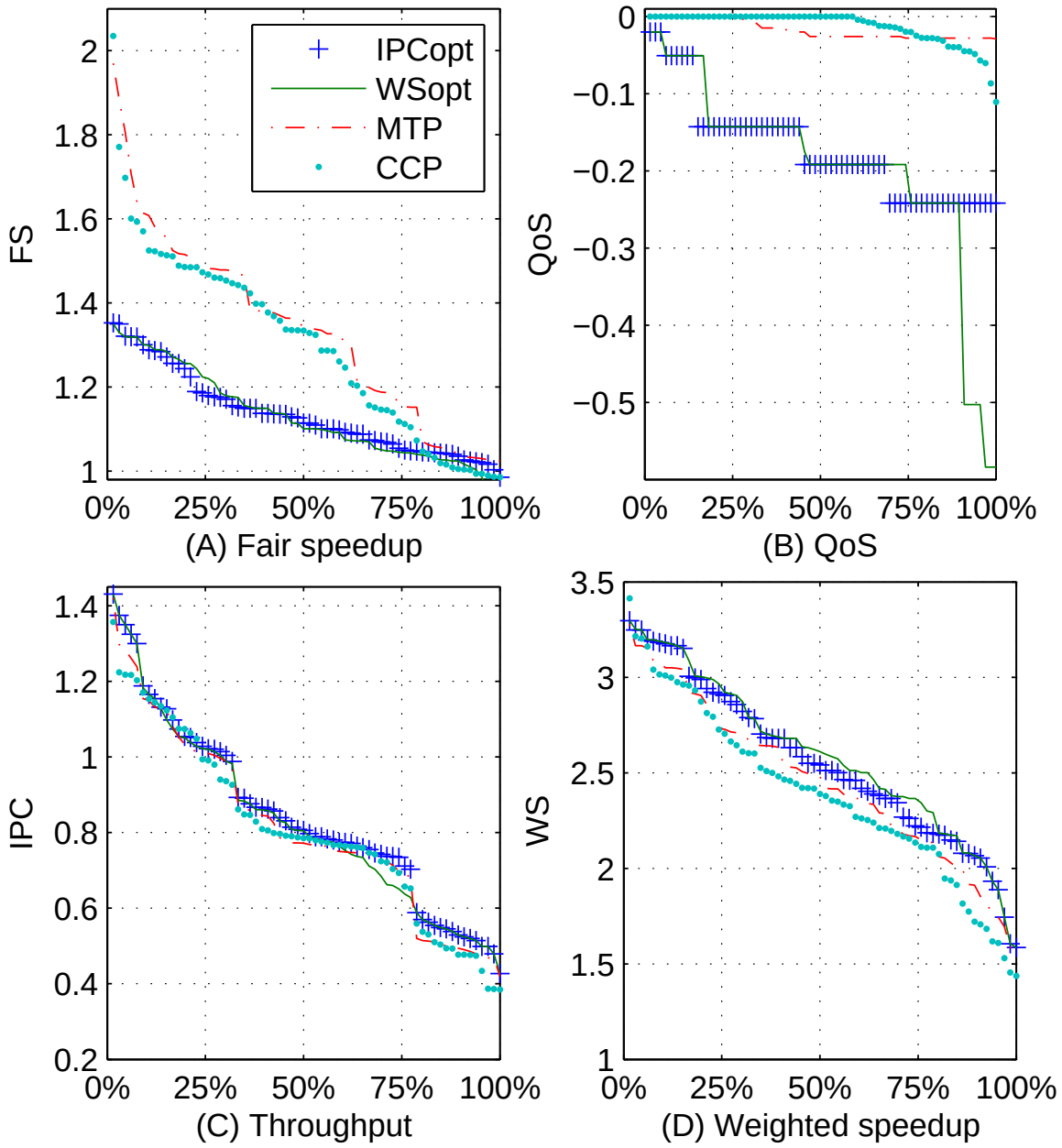


Figure 5.13: Results for Workloads that Need Cache Partitioning (4MB cache, 32% of total workloads)

Figure 5.14 summarizes the average improvement of WS_{opt}, IPC_{opt}, shared cache, CC and CCP over the equal-share baseline for different metrics.

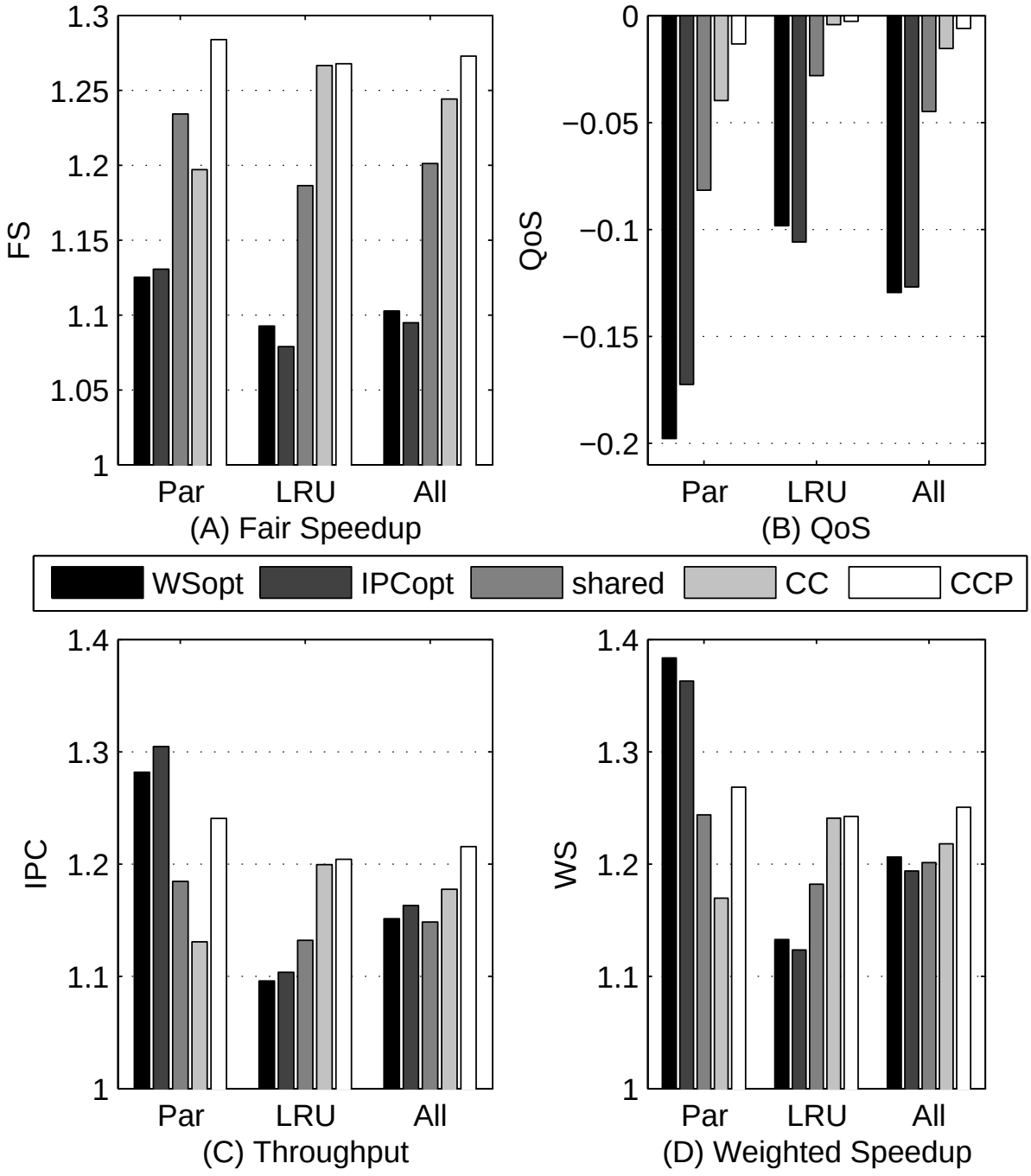


Figure 5.14: Average Improvement for 4MB L2 cache

The average improvements are calculated as geometric means of per-workload improvements[7]. The

[7]QoS results are summarized using the arithmetic mean because the QoS measurements of many workloads are zero, which causes the average results to be the same (zero) when using geometric mean.

results are summarized over three groups of workloads: “Par” represents workloads that prefer cache partitioning, “LRU” groups other workloads, while “All” covers all workload combinations. This figure shows that for workloads preferring cache partitioning (Par), CCP performs much better than a shared cache and CC, while achieving similar or much better results than the two cache partitioning schemes. Considering workloads that prefer LRU-based sharing (LRU) and all workloads (All), CCP provides the best average results on all reported metrics.

5.5.3 Results for a 2MB L2 Cache

Now we evaluate the robustness of CCP when the total L2 cache capacity is reduced to 2MB. The reduction of cache size not only increases capacity contention between threads, but also causes some benchmarks to switch their categories (e.g., from supplier benchmarks to sensitive benchmarks, or from sensitive benchmarks to thrashing benchmarks) so the performance of CCP can be tested under new scenarios.

Figure 5.15 uses transposed CDF plots to compare 4 cache partitioning schemes (IPC_{opt}, WS_{opt}, MTP and CCP) on workloads that need cache partitioning. CCP again achieves comparable FS and QoS results as the ideal MTP implementation and outperforms the two SSP-based partitioning schemes. This shows the robustness of CCP’s heuristic-based partitioning algorithm. In terms of fairness and throughput tradeoff, the weighted speedup results of MTP and CCP are similar to IPC_{opt} and WS_{opt}, while their throughput results are 10% lower. Again, QoS constraint and fair speedup optimization are the two reasons that cause MTP and CCP’s lower throughput, while IPC_{opt} and WS_{opt} can achieve better throughput without satisfying such constraints.

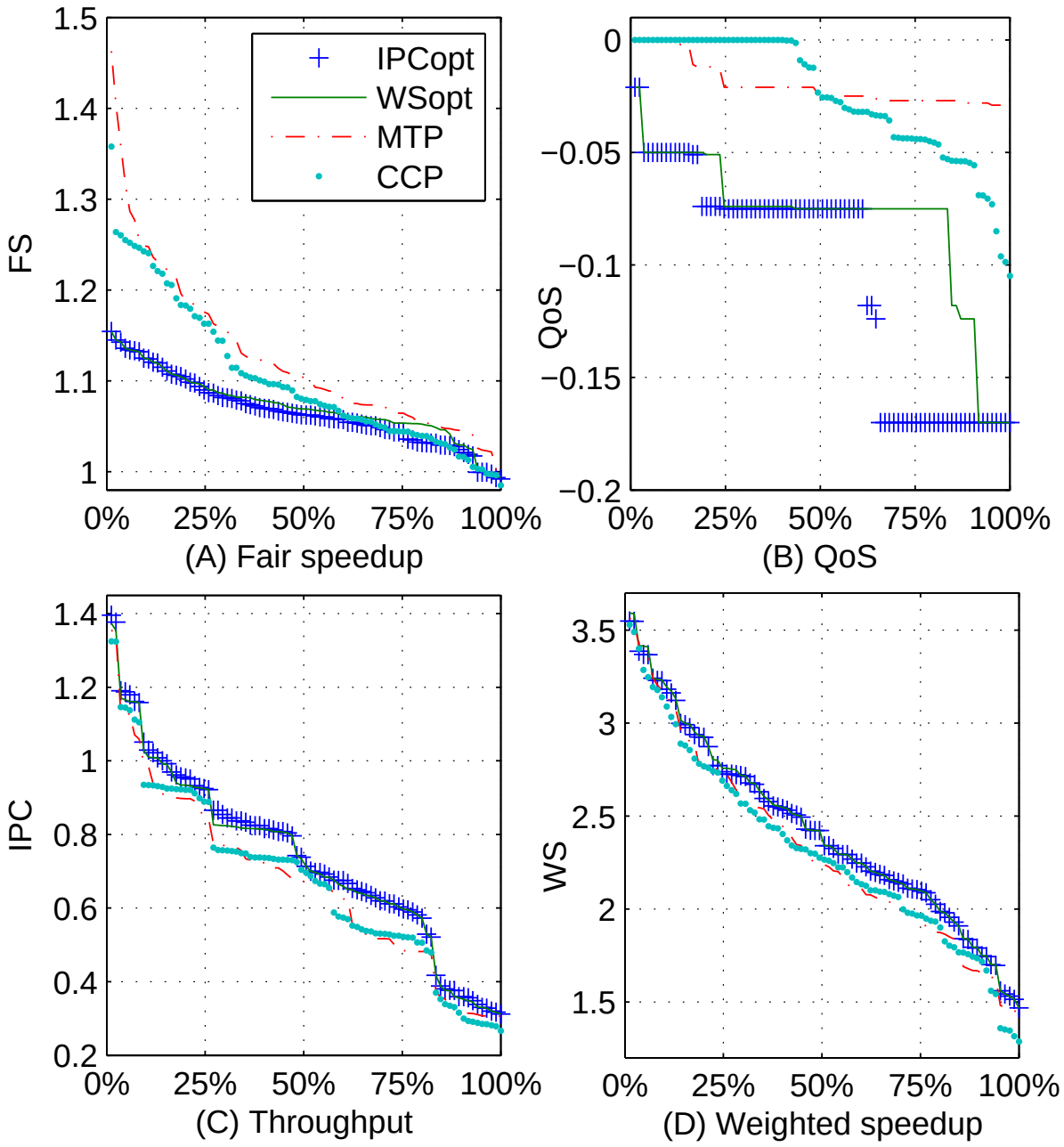


Figure 5.15: Results for Workloads that Need Cache Partitioning (2MB cache, 40% of total workloads)

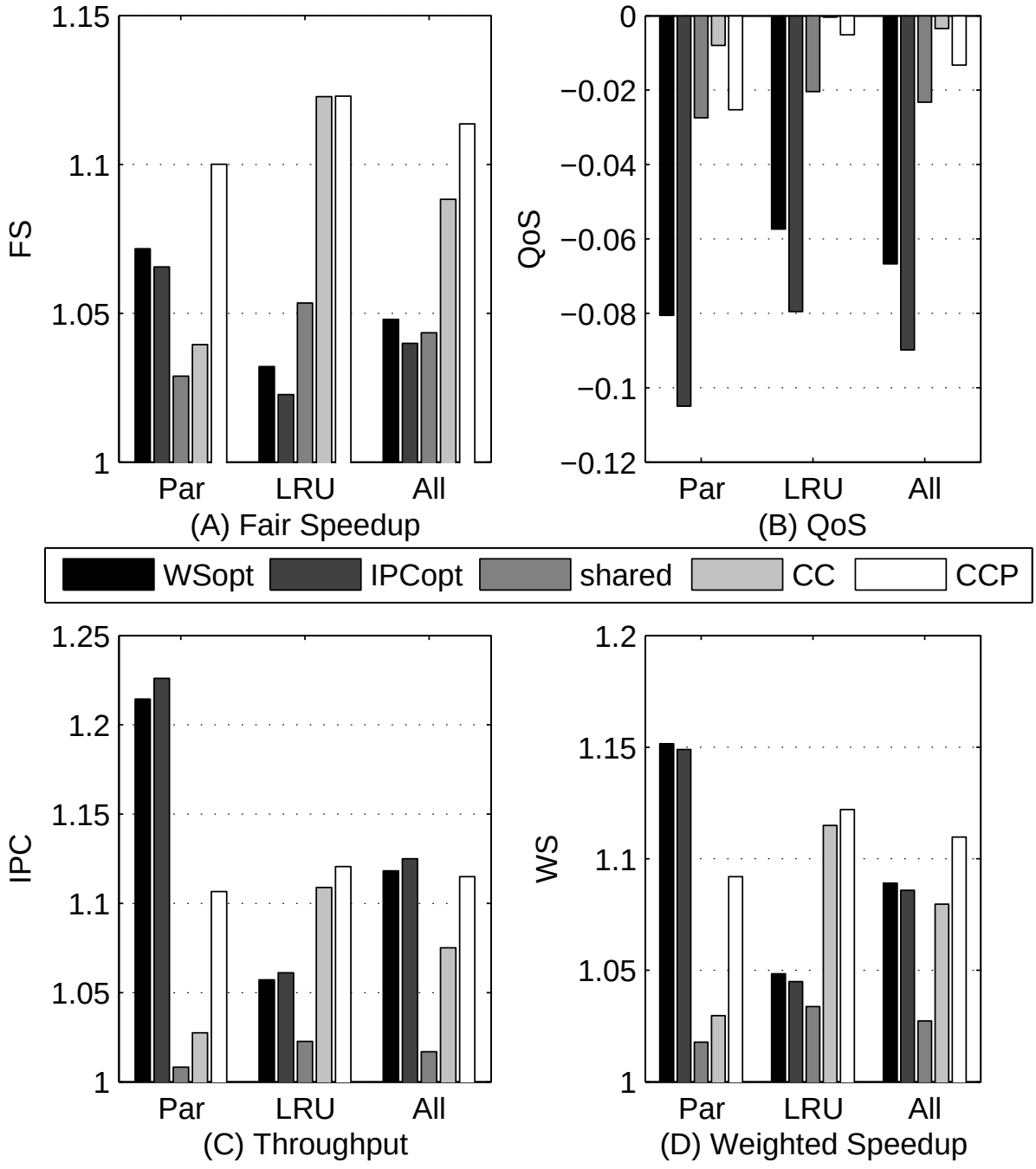


Figure 5.16: Average Improvement for 2MB L2 Cache

Figure 5.16 compares the average improvements of various schemes over the equal-share baseline. Because smaller cache size causes more capacity contention, more workloads now prefer cache partitioning (increased from 32% to 40%). As partitioning becomes more preferable, the performance of the shared cache also drops significantly and comes close to the equal-share baseline performance. In contrast, CCP still consistently outperforms other schemes for workloads where cache partitioning can hurt. However, the gap between cache partitioning schemes and CCP (as well as CC) is reduced because, due to capacity pressure, latency optimizations contribute less to the overall speedup. Averaged over all 210 workloads, CCP achieves the best results on almost all metrics (except for throughput, where CCP is 1% lower than IPC_{opt}).

5.5.4 Out-of-order Processor Results

The robustness of CCP can also be evaluated using a more aggressive processor model. Here we use the same out-of-order processor modeled in Chapter 4, but only evaluate a representative subset of workloads to shorten the simulation time. The workloads for the out-of-order processor are selected as follows. First we choose three benchmarks to represent each of the three benchmark categories discussed in Section 5.2.2: `gcc` for supplier benchmarks, `vpr` for sensitive benchmarks, and `art` for thrashing benchmarks. A total of 15 workloads can be generated by all 4-thread combinations of these benchmarks, and we only report results for these 15 workloads.

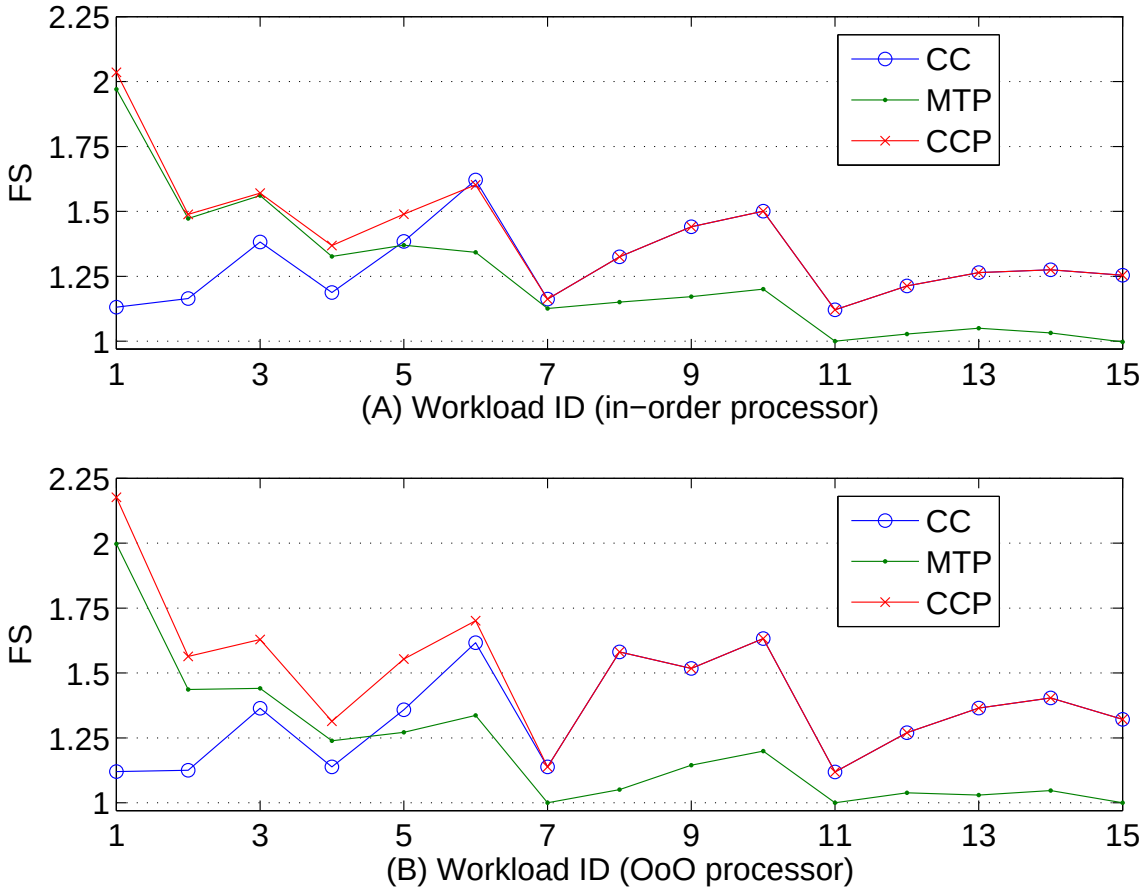


Figure 5.17: Comparison of FS Results with In-order vs. Out-of-order Processor Models

Figure 5.17 compares the FS results achieved by CC, MTP and CCP when simulated using (A) in-order vs. (B) out-of-order processor models. The key point shown in Figure 5.17 is that, across all workloads, the relative ordering of different schemes for in-order and out-of-order processors are the same. Specifically, CCP performs the best for all workloads with both processor models.

The two sets of results differ mainly in the performance gaps among different schemes. Comparing with in-order processor results, using the out-of-order processor model increases the performance advantage of CC and CCP over the shared cache (thus MTP). The reason for improved benefits is because, for the three selected SPEC benchmarks, the out-of-order processor can tolerate some of the local L2 hit latency (15-cycles) but not the local L2 miss latencies (more than 30 cycles). Because CC can significantly increase the number of local L2 hits than a shared cache, its performance advantage is magnified when using the

out-of-order processor model.

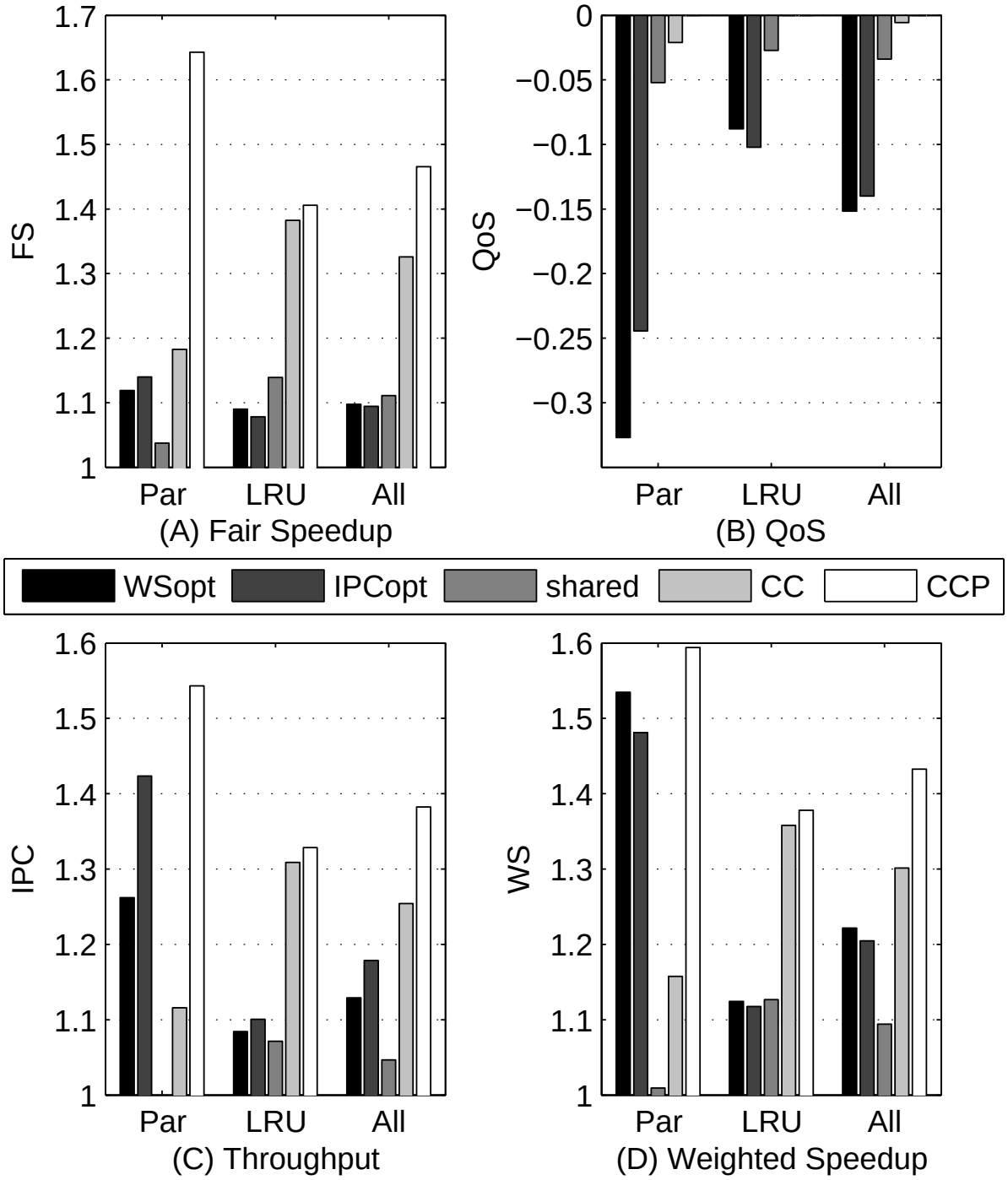


Figure 5.18: Average Improvement with Out-of-order Processor Models

Generally speaking, depending on a benchmark's cache latency tolerance capabilities under different

processor models, switching from the in-order to the out-of-order processor can either increase or decrease the benefits of CC and CCP over a shared cache. But as shown in Figure 5.17, switching to a different processor model does not change the advantage of CCP over its two baseline schemes (CC and MTP) for our evaluated workloads.

Figure 5.18 summarizes the average improvements of various schemes over the equal-share baseline when using the out-of-order processor model. Same as using the in-order processor, CCP achieves the best FS and QoS results for both PAR and LRU workloads. Different from previous results, CCP even achieves the best WS and IPC results, due to the increase performance benefits of CC and CCP with the out-of-order processor. For our evaluated workloads, CCP provides the best results in all evaluation metrics.

5.6 Conclusion

Current cache partitioning schemes have limited functionality and applicability because they can only support a subset of CMP caching requirements, and they can not compete with LRU-based latency-reducing caching schemes (e.g., CC) for many workloads. To answer these challenges, we introduce Multiple Time-sharing Partitions (MTP) to simultaneously improve throughput and fairness while guaranteeing long-term QoS. MTP is further integrated with CMP cooperative caching (CC) to exploit its latency optimizations.

The resulted CCP scheme is evaluated and shown to provide the best overall performance over 210 combinations of 7 representative SPEC2000 benchmarks under two different cache sizes. For a 4-core CMP with 4MB L2 cache, CCP not only maintains QoS, but also improves performance (measured by fair speedup) by 12% and throughput by 4.5% over the best static partitioning schemes optimizing fair speedup and throughput, respectively.

CCP takes a first step in balancing partitioning-based capacity optimizations and LRU-based latency optimizations for multiprogrammed workloads. Future research is needed to extend CCP to better adapt to phase/scheduling changes, as well as to support large-scale CMPs and CMPs with SMT cores. For

applications with real-time or response time requirements, future research can extend CCP to guarantee the minimum performance needed for such workloads while exploiting the benefits of MTP and CC for the remaining, best-effort, applications. For environments that prefer higher throughput over execution time reduction, fairness, and QoS guarantee, CCP can also be modified to use a single spatial partition that only optimizes throughput, which is left as future work.

CHAPTER 6

CONCLUSION

As CMPs become the mainstream processors, designs of on-chip cache hierarchy will play an important role to provide fast and fair data accesses for multiple, competing processor cores. To offset the negative impact of limited off-chip bandwidth, on-chip wire delay, and hardware/software design complexity, this dissertation contributes to two important goals of CMP caching—latency reduction and shared resource management—with a unified supporting framework and two sets of cooperation policies.

6.1 Key Contributions

The first contribution of the dissertation is a unified Cooperative Caching (CC) framework for efficient organization and use of the aggregate on-chip cache resources. This framework includes the following three key components.

- **Private cache based design.** CC uses private cache organization to reduce remote on-chip misses and cross-chip communication requirements (thus active power associated with remote cache accesses). The reduction of cache associativity, network bandwidth, and coherence traffic also leads to potentially simpler designs. To enable resource sharing, CC enhances the base private design with two features: (1) non-inclusion between multiple levels of caches to enable flexible data placement and (2) support of cache-to-cache transfers of clean data to avoid unnecessary off-chip misses. Among many CMP caching proposals [13, 15, 23, 27, 59, 68, 96, 138, 156, 159], CC is the only proposed design that exploits the advantages of private cache based design to optimize for both multhreaded and multiprogrammed workloads.

- **Cooperation and throttling mechanisms.** By viewing a CMP's use of the aggregate cache resources as a shared resource management problem, we have identified two sets of commonly used mechanisms to support a wide range of resource sharing behaviors. Explicit resource sharing between peer caches is carried out via placement and replacement based cooperation mechanisms, which together determine what data are kept on-chip, and specifically in which cache. To control the amount of sharing, probability based throttling can cover the whole spectrum between the baseline design and unconstrained resource sharing. By controlling which cache can use the capacity of peer on-chip caches, quota-based throttling can enforce policy-specific capacity allocations for all individual cores. The combination of private cache organization, resource sharing mechanisms and throttling mechanisms thus provides a vast space of sharing behaviors for the cooperation policies to explore.

- **Decoupling of policies, mechanisms and implementations.** With a set of commonly used mechanisms, the CC framework is extensible on both policy and implementation sides. At the implementation level, CC's cooperation and throttling mechanisms are independent of specific cache algorithms or coherence protocols, and can be adopted by various existing implementations. In contrast, most current CMP caching proposals assume or rely on particular implementations (e.g., NUCA caches [15, 68, 156], snooping protocols [27, 138], or tile-based distributed directory protocols [159]). With CC's basic mechanisms, researchers can focus on the high-level tasks of understanding caching requirements and devising innovative caching policies.

The dissertation also proposes and evaluates cache sharing policies for reducing off-chip accesses and mitigating destructive inter-thread interference.

- **Capacity improving policies.** To achieve robust performance, the disadvantage of private cache designs—high off-chip miss rate—is mitigated using cooperation policies. These policies reduce off-chip accesses by mimicking the behavior of a shared cache. Beside cache-to-cache transferring of clean data, two policies are introduced to control replication and allow an LRU-like global sharing of

cache space. The two policies improve capacity utilization of multithreaded and multiprogrammed workloads respectively, making CC more versatile than CMP caching designs that only optimize for one class of workloads [13, 15, 138, 156]. Our evaluation shows that the combination of private cache design and such cooperative resource sharing leads to robust performance for a wide range of processor, cache/memory and system configurations.

- **Cache partitioning policies.** We use Multiple Time-sharing Partitions (MTP) to achieve a broader sense of performance isolation in a multiprogramming environment. Compared with single spatial partitioning (SSP) based policies, MTP introduces a time-sharing aspect to spatial cache partitioning and has the advantage of simultaneously satisfying multiple conflicting requirements. Each MTP partition is an unfair partition that intentionally "starves" a subset of concurrent threads to avoid thrashing; throughput and fairness can be simultaneously improved by executing multiple MTP partitions in a round-robin manner, giving different threads fair opportunities to speedup; and QoS is guaranteed by orchestrating capacity shrinking/growing in different MTP partitions so that every thread's average slowdown is bounded. The time-sharing behavior introduced by MTP is a familiar concept to operating system schedulers, and can lead to simple implementations of priority scheduling.

- **Policy integration.** Motivated by the complementary benefits of the two sets of policies on two different problems, CC integrates these policies with a high-level adaptation policy. The integrated scheme—Cooperative Cache Partitioning (CCP)—divides the total execution time into epochs managed by either CC's LRU-based policies or the MTP cache partitioning policy, proportionally to the number of threads that can benefit from each of them respectively. Using CC's LRU-based sharing as the default policy simplifies MTP partitioning and allows heuristic-based, practical implementations. Our evaluation of CCP implementation shows that it effectively combines the benefits of MTP and CC: it can significantly improve the performance of workloads that need cache partitioning while

maintaining fairness and QoS, and it achieves the same benefits of LRU-based latency optimizations for workloads that do not need cache partitioning.

6.2 Future Directions

We have demonstrated the effectiveness of the CC framework with two important CMP caching applications, and we believe the cooperative sharing mechanisms and the philosophy of using cooperation for conflict resolution can be applied to other problems. Below we discuss some of the directions in extending the CC framework and its applications.

6.2.1 Better Latency Reduction and Cache Partitioning

One promising direction for latency reduction is to integrate with caching algorithms that can adapt between a recency-based replacement policy (e.g., LRU) and a frequency-based policy (e.g., LFU). Such algorithms have already been used by software caches to dynamically discriminate data streams with weak locality. Extending CC with an adaptive policy for the processor cache can narrow the performance gap between LRU and ideal cache algorithms.

New ways of characterizing and adapting to the caching requirements of individual threads are also needed to further improve cooperative cache partitioning. Currently we use epoch-based sampling to estimate such requirements, which may not adapt to phase/scheduling changes swiftly. Future research can attack this problem with program phase tracking techniques [38, 131], on-line sampling mechanisms [122], and analytical models. We believe that a timely and better understanding of program requirements (in terms of capacity, associativity [124], locality [56, 150], etc.) will lead to better assignment of cache resources among co-scheduled threads, and thus more efficient use of these resources. Such techniques can be applied to optimize not only performance, but also power, thermal and other metrics.

CC can be further extended to adaptively choose between latency reduction policies and cache partition-

ing policies, not only for multiprogrammed workloads as demonstrated by CCP, but also for multithreaded workloads and their multiprogramming combinations. The key for this policy selection is to determine whether a group of co-scheduled threads shares data, which should trigger the replication control policy. Such information can be provided either by the scheduler software, or a hardware monitor of data sharing messages.

6.2.2 Heterogeneous Processor and Cache Designs

Because private caches are interfaced in CC only via the coherence protocol, their internal organizations are encapsulated and thus can be different. This property can be exploited to support statically or dynamically created heterogeneous private caches, with different number of sets, associativities, threshold voltages, cell sizes, or even replacement and write-back policies.

One application of heterogeneous caches is to provide matching cache resources for heterogeneous processor designs without global synchronization. The challenge is to define fairness and QoS for such heterogeneous architectures with core-specific cache configurations. Future research should explore the benefits of heterogeneous cache designs and understand its implication to parallel applications, operating systems, and end-users [10].

6.2.3 Power and Reliability Optimizations

CC can be extended to support other cache optimizations, such as leakage power reduction and reliability improvement. At the architectural level, better power efficiency and reliability can be achieved by making some cache resources slow or sacrificing some capacity. For example, power can be saved by ”turning off” selected portions of the aggregate on-chip cache, or using higher threshold voltage [20] (leading to slower accesses), while reliability can be improved via spatial redundancy (e.g., ECC, which can slow down tag accesses), or using larger transistors or higher threshold voltage [33].

However, to maintain high performance, not all L2 caches can be made slow [6] or turned off, and cache management policies are needed to trade off between performance and power/reliability enhancements. CC can be extended to manage a set of cache resources with different capacity and speed characteristics (thus different power and reliability parameters), and match them with threads having heterogeneous power/reliability/performance requirements. Similar to existing cooperative policies, CC will need to identify what portion of the cache (e.g., at cache blocks, cache subarrays, or private cache level) is not used efficiently and can thus be turned off or made slow, and for how long, and how to manage the remaining cache resources to minimize the consequent performance loss. To eliminate the over-provisioning inefficiency based on worse-case power/reliability requirements [127], chip-level management policies can be explored to share available power budget or reliable banks and react to over-quota emergencies. CC can also exploit data criticality information (e.g., dirty blocks or blocks touched by kernel code) to place important data into caches with matching characteristics [107].

6.2.4 Software/Hardware Cooperative Caching

Cooperation can be beneficial not only between private caches, but also between software and hardware cache management schemes. CC can be easily augmented with cache bypassing [80] or prioritized eviction [154], and use compiler generated hints to control the placement and replacement of cache blocks. Future work can also consider software controlled cache partitioning, where MTPs, instead of thread-mixes, are explicitly scheduled to satisfy specific software requirements (e.g., throughput, fairness, or priority).

For embedded [69] or streaming applications [53], compiler can partition the computation into specialized components executed on dedicated cores, and orchestrate data communication among producer-consumer cores. Compared with conventional private and shared cache organization, CC provides better support for such orchestrated data usage and movement, because (1) its private cache organization can keep a computation task's data set close to it, and (2) its prioritized replacement and spill mechanisms together can

streamline producer-consumer communications. Similarly, virtual machine monitors that migrate threads around can also benefit from CC when collocating computation with data.

References

[1] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, D. J. Sorin, M. D. Hill, and D. A. Wood. Simulating a $2M commercial server on a $2K PC. *IEEE Computer*, 36(2):50–57, Feb. 2003.

[2] A. R. Alameldeen and D. A. Wood. Variability in Architectural Simulations of Multi-Threaded Workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*, 2003.

[3] A. R. Alameldeen and D. A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, 2006.

[4] AMD. AMD Multi-core. http://multicore.amd.com/en/, 2006.

[5] J. K. Archibald. A Cache Coherence Approach for Large Multiprocessor Systems. In *Proceedings of the 2nd International Conference on Supercomputing*, 1988.

[6] H. Asadi, V. Sridharan, M. B. Tahoori, and D. Kaeli. Reliability Tradeoffs in Design of Cache Memories. In *1st Workshop on Architectural Reliability (WAR-1)*, 2005.

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report No. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[8] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPECOMP: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Proceedings of the International Workshop on OpenMP Applications and Tools*, 2001.

[9] J.-L. Baer and W.-H. Wang. On the Inclusion Properties for Multi-level Cache Hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988.

[10] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, pages 506–517, 2005.

[11] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, 1998.

[12] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[13] B. Beckmann. *Managing Wire Delay in Chip Multiprocessor Caches*. PhD thesis, University of Wisconsin-Madison, 2006.

[14] B. M. Beckmann, M. R. Marty, and D. A. Wood. ASR: Adaptive Selective Replication for CMP Caches. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, Dec. 2006.

[15] B. M. Beckmann and D. A. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, Dec. 2004.

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

[17] D. Bertsekas and R. Gallager. *Data networks (2nd ed.)*. Prentice-Hall, Inc., 1992.

[18] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communication of ACM*, 13(7):422–426, July 1970.

[19] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *USENIX 1998 Annual Technical Conference*, 1998.

[20] J. A. Butts and G. S. Sohi. A Static Power Model for Architects. In *MICRO-33*, 2000.

[21] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, June 2005.

[22] R. W. Carr and J. L. Hennessy. WSClock - a Simple and Effective Algorithm for Virtual Memory Management. In *8th ACM Symposium on Operating Systems Principles*, 1981.

[23] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the 33th Annual International Symposium on Computer Architecture*, June 2006.

[24] J. Chang and G. S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *Proceedings of the 21th ACM International Conference on Supercomputing (ICS'07)*, 2007.

[25] A. Charlesworth, N. Aneshansley, M. Haakmeester, D. Drogichen, G. Gilbert, R. Williams, and A. Phelps. The Starfire SMP interconnect. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, 1997.

[26] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, 2003.

[27] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing Replication, Communication and Capacity Allocation in CMPs. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, June 2005.

[28] S. Cho and L. Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *MICRO-39*, 2006.

[29] D. D. Clark. The Design Philosophy of the DARPA Internet Protocols. *ACM SIGCOMM Computer Communication Review*, 18(4):106–114, 1988.

[30] CORPORATE Computer Science and Telecommunications Board. *Realizing the Information Future: the Internet and Beyond*. National Academy Press, Washington, DC, USA, 1994.

[31] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[32] M. Dahlin, R. Wang, T. E. Anderson, and D. A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, 1994.

[33] V. Degalahal, R. Ramanarayanan, N. Vijaykrishnan, Y. Xie, and M. J. Irwin. The Effect of Threshold Voltages on the Soft Error Rate. In *ISQED '04: Proceedings of the 5th International Symposium on Quality Electronic Design*, 2004.

[34] P. J. Denning. The Working Set Model for Program Behavior. *Communication of ACM*, 11(5):323–333, 1968.

[35] P. J. Denning. Thrashing: Its Causes and Prevention. In *AFIPS 1968 Fall Joint Computer Conference*, volume 33, pages 915–922, 1968.

[36] P. J. Denning. Virtual Memory. *ACM Computing Survey*, 2(3):153–189, 1970.

[37] P. J. Denning. The Locality Principle. *Communication of ACM*, 48(7):19–24, 2005.

[38] A. S. Dhodapkar and J. E. Smith. Managing Multi-configuration Hardware via Dynamic Working Set Analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[39] R. Drost and I. Sutherland. Proximity Communication and Time. In *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, 2005.

[40] H. Dybdahl and P. Stenstrom. An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In *Proceedings of the 13th Annual International Symposium on High-Performance Computer Architecture*, 2007.

[41] H. Dybdahl, P. Stenstrom, and L. Natvig. A Cache-Partition Aware Replacement Policy for Chip Multiprocessors. In *Proceedings of 13th International Conference of High Performance Computing (HiPC)*, 2006.

[42] H. Dybdahl, P. Stenstrom, and L. Natvig. An LRU-based Replacement Algorithm Augmented with Frequency of Access in Shared Chip-Multiprocessor Caches. In *Proceedings of the 6th Workshop on Memory Performance: Dealing with Applications, Systems and Architecture (MEDEA 2006)*, 2006.

[43] J. Edward G. Coffman and P. J. Denning. *Operating Systems Theory*. Prentice Hall, 1973.

[44] R. Fagin and J. H. Williams. A Fair Carpool Scheduling Algorithm. *IBM Journal of Research and Development*, 27(2):133–139, 1983.

[45] B. Falsafi and D. A. Wood. Reactive NUMA: a design for unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[46] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: a Scalable Wide-area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[47] A. Fedorova. *Operating System Scheduling for Chip Multithreaded Processors*. PhD thesis, Harvard University, 2006.

[48] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design. In *USENIX 2005 Annual Technical Conference*, April 2005.

[49] A. Fedorova, M. Seltzer, and M. D. Smith. A Non-Work-Conserving Operating System Scheduler For SMT Processors. In *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), in conjunction with ISCA-33*, 2006.

[50] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[51] I. T. R. for Semiconductors. ITRS 2005 Update. Semiconductor Industry Association, 2005.

[52] A. Glew. MLP Yes! ILP No! Memory Level Parallelism, or, Why I No Longer Worry About IPC. Wild and Crazy Ideas Session of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, 1998.

[53] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[54] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. A Novel SIMD Architecture for the Cell Heterogeneous Chip-multiprocessor. In *Hot Chips 17*, 2005.

[55] A. Gupta, W.-D. Weber, and T. C. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *ICPP*, 1990.

[56] E. Hagersten, A. Landin, and S. Haridi. DDM: A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):44–54, 1992.

[57] E. G. Hallnor and S. K. Reinhardt. A Fully Associative Software-Managed Cache Design. In *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[58] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[59] S. Harris. *Synergistic Caching in Single-Chip Multiprocessors*. PhD thesis, Stanford University, 2005.

[60] A. Hartstein and T. R. Puzak. The Optimum Pipeline Depth for a Microprocessor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[61] A. Hartstein and T. R. Puzak. Optimum Power/Performance Pipeline Depth. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, 2003.

[62] J. L. Hellerstein. Achieving Service Rate Objectives with Decay Usage Scheduling. *IEEE Transaction of Software Engineering*, 19(8), 1993.

[63] R. Ho, K. Mai, and M. A. Horowitz. The Future of Wires. In *Proceedings of the IEEE*, pages 490–504, 2001.

[64] M. S. Hrishikesh, D. Burger, N. P. Jouppi, S. W. Keckler, K. I. Farkas, and P. Shivakumar. The Optimal Logic Depth per Pipeline Stage is 6 to 8 FO4 Inverter Delays. 2002.

[65] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell. Exploring the Cache Design Space for Large Scale CMPs. *SIGARCH Computer Architecture News*, 33(4), 2005.

[66] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource. In *Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques*, 2006.

[67] J. Huh, D. Burger, and S. W. Keckler. Exploring the Design Space of Future CMPs. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[68] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA Substrate for Flexible CMP Cache Sharing. In *Proceedings of the 19th ACM International Conference on Supercomputing*, June, 2005.

[69] H. C. Hunter. *Matching On-Chip Data Storage To Telecommunication and Media Application Properties*. PhD thesis, University of Illinois, Urbana, 2004.

[70] IEEE. IEEE Standard for Scalable Coherent Interface (SCI), 1992. IEEE 1596-1992.

[71] Intel. A Platform 2015 Workload Model: Recognition, Mining and Synthesis Move the Computers to the Era of Tera. ftp://download.intel.com/technology/computing/archinnov/platform2015/download/RMS.pdf, 2006.

[72] Intel. Multi-core from Intel. http://www.intel.com/multi-core/, 2006.

[73] Intel. Tera-scale Computing. http://www.intel.com/technology/techresearch/terascale/, 2006.

[74] R. Iyer. CQoS: a Framework for Enabling QoS in Shared Caches of CMP Platforms. In *Proceedings of the 18th Annual International Conference on Supercomputing*, 2004.

[75] A. Jaleel, M. Mattina, and B. Jacob. Last-Level Cache (LLC) Performance of Data-mining Workloads on a CMP–A Case Study of Parallel Bioinformatics Workloads. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, 2006.

[76] J. Jeong and M. Dubois. Cost-Sensitive Cache Replacement Algorithms. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.

[77] S. Jiang and X. Zhang. LIRS: an Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2002.

[78] L. Jin, J. Lee, and S. Cho. A Flexible Data to L2 Cache Mapping Approach for Future Multicore Processors. In *ACM Workshop on Memory Systems Performance and Correctness (MSPC)*, 2006.

[79] L. K. John. More on Finding a Single number to Indicate Overall Performance of a Benchmark Suite. *SIGARCH Computer Architecture News*, 32(1), 2004.

[80] T. L. Johnson, D. A. Connors, M. C. Merten, and W. mei W. Hwu. Run-Time Cache Bypassing. *IEEE Transaction of Computer*, 48(12):1338–1354, 1999.

[81] H. Kannan, F. Guo, L. Zhao, R. Illikkal, R. Iyer, D. Newell, Y. Solihin, and C. Kozyrakis. From Chaos to QoS: Case Studies in CMP Resource Management. In *Proceedings of 2006 Workshop on Design, Architecture and Simulation of Chip Multi-Processors (dasCMP 2006)*, 2006.

[82] S. Kaxiras and J. R. Goodman. Improving CC-NUMA performance using Instruction-based Prediction. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, Jan. 1999.

[83] C. Kim, D. Burger, and S. 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*, 2002.

[84] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, 2004.

[85] A. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation Based Computer Architecture Research. *Computer Architecture Letters*, 2002.

[86] M. Kondo, H. Sasaki, and H. Nakamura. Improving Fairness, Throughput and Energy-Efficiency on a Chip Multiprocessor through DVFS. In *Proceedings of 2006 Workshop on Design, Architecture and Simulation of Chip Multi-Processors (dasCMP 2006)*, 2006.

[87] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, 2005.

[88] C.-C. Kuo, J. B. Carter, R. Kuramkote, and M. R. Swanson. AS-COMA: An Adaptive Hybrid Shared Memory Architecture. In *Proceedings of the 1998 International Conference on Parallel Processing*, pages 207–216, 1998.

[89] A.-C. Lai and B. Falsafi. Memory Sharing Predictor: The Key to a Speculative Coherent DSM. In *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[90] A.-C. Lai and B. Falsafi. Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction. In *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[91] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, 1997.

[92] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 48–59, June 1995.

[93] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, Mar. 1992.

[94] F. Li, C. Nicopoulos, T. Richardson, Y. Xie, V. Narayanan, and M. Kandemir. Design and Management of 3D Chip Multiprocessors Using Network-in-Memory. In *Proceedings of the 33th Annual International Symposium on Computer Architecture*, June 2006.

[95] J. S. Liptay. Structural Aspects of the System/360 Model 85, Part II: The cache. *IBM Journal of Research and Development*, 7(1), 1968.

[96] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs. In *Proceedings of the 10th IEEE Symposium on High-Performance Computer Architecture*, Feb. 2004.

[97] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 39–50, 1998.

[98] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *Proceedings of 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, 2001.

[99] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[100] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared Memory Multiprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture*, 2003.

[101] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[102] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 2005.

[103] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. K. Martin, and D. A. Wood. Improving Multiple-CMP Systems Using Token Coherence. In *Proceedings of the 11th Annual International Symposium on High-Performance Computer Architecture*, February 2005.

[104] M. R. Marty and M. D. Hill. Coherence Ordering for Ring-based Chip Multiprocessors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[105] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[106] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003.

[107] M. Mehrara and T. Austin. Reliability-aware Data Placement for Partial Memory Protection in Embedded Processors. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness (MSPC 2006)*, 2006.

[108] A. Moga and M. Dubois. The Effectiveness of SRAM Network Caches in Clustered DSMs. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture*, 1998.

[109] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, June 2005.

[110] S. S. Mukherjee and M. D. Hill. Using Prediction to Accelerate Coherence Protocols. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 179–190, June 1998.

[111] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Effective Alternative to Large Instruction Windows. *IEEE Micro*, 23(6):20–25, 2003.

[112] A. K. Nanda, A.-T. Nguyen, M. M. Michael, and D. J. Joseph. High-throughput Coherence Control and Hardware Messaging in Everest. *IBM Journal of Research and Development*, 45(2), 2001.

[113] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[114] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, 2006.

[115] H. Oi and N. Ranganathan. Utilization of Cache Area in On-Chip Multiprocessor. In *Proceedings of the Second International Symposium on High Performance Computing (ISHPC '99)*, 1999.

[116] M. S. Papamarcos and J. H. Patel. A Low-overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, 1984.

[117] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: the Single-node Case. *IEEE Transaction of Networks*, 1(3):344–357, 1993.

[118] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Micro*, 17(2):34–44, 1997.

[119] D. A. Patterson. Computer Science Education in the 21st Century. *Communication of ACM*, 49(3):27–30, 2006.

[120] C. Percival. http://www.daemonology.net/hyperthreading-considered-harmful.

[121] P. Petoumenos, G. Keramidas, H. Zeffer, S. Kaxiras, and E. Hagersten. STATSHARE: A Statistical Model for Managing Cache Sharing via Decay. In *MoBS 2006*, June 2006.

[122] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. In *Proceedings of the 33th Annual International Symposium on Computer Architecture*, June 2006.

[123] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO-39*, 2006.

[124] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-way Cache: Demand Based Associativity via Global Replacement. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[125] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural Support for Operating System-Driven CMP Cache Management. In *Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques*, 2006.

[126] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *34th International Symposium on Microarchitecture*, December, 2001.

[127] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level Power Management for Dense Blade Servers. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[128] Rapport Inc. KC256.
http://www.rapportincorporated.com/kilocore/kc256.html, 2006.

[129] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An Argument for Simple COMA. In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, 1995.

[130] T. Sherwood, B. Calder, and J. Emer. Reducing Cache Misses Using Hardware and Software Page Placement. In *Proceedings of the 13th international conference on Supercomputing*, 1999.

[131] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In *Proceedings of the 30th International Symposium on Computer Architecture*, 2003.

[132] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. Technical report, 2001.

[133] J. E. Smith. Characterizing Computer Performance with a Single Number. *Communication of ACM*, 31(10), 1988.

[134] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[135] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. In *The 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2002.

[136] K. So and R. N. Rechtschaffen. Cache Operations by MRU Change. *IEEE Transaction of Computers*, 37(6):700–709, 1988.

[137] V. Soundararajan, M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, and J. Hennessy. Flexible Use of Memory for Replication/Migration in Cache-coherent DSM Multiprocessors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[138] E. Speight, H. Shafi, L. Zhang, and R. Rajamony. Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, June 2005.

[139] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P. N. Strenski, and P. G. Emma. Optimizing Pipelines for Power and Performance. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, 2002.

[140] P. Stenström, M. Brorsson, and L. Sandberg. Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[141] H. S. Stone, J. Turek, and J. L. Wolf. Optimal Partitioning of Cache Memory. *IEEE Transaction of Computers*, 41(9):1054–1068, 1992.

[142] R. Subramanian, Y. Smaragdakis, and G. H. Loh. Adaptive Caches: Effective Shaping of Cache Behavior to Workloads. In *MICRO-39*, 2006.

[143] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-aware Scheduling and Partitioning. In *HPCA-8*, 2002.

[144] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *Journal of Supercomputing*, 28(1):7–26, 2004.

[145] D. G. Sullivan and M. I. Seltzer. Isolation with Flexibility: A Resource Management Framework for Central Servers. In *USENIX 2000 Annual Technical Conference*, June 2000.

[146] M. Takahashi, H. Takano, E. Kaneko, and S. Suzuki. A Shared-bus Control Mechanism and a Cache Coherence Protocol for a High-performance On-chip Multiprocessor. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, 1996.

[147] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems: Design And Implementation, 2/E*. Prentice Hall, 1997.

[148] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Le, and B. Sinharoy. IBM Power4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[149] K. Vardarajan, S. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, S. Makineni, and D. Newell. Molecular Caches: A caching structure for dynamic creation of application-specific Heterogeneous cache regions. In *MICRO-39*, 2006.

[150] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[151] B. Verghese, A. Gupta, and M. Rosenblum. Performance Isolation: Sharing and Isolation in Shared-memory Multiprocessors. In *Proceedings of the 8th international Conference on Architectural Support for Programming Languages and operating systems*, 1998.

[152] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[153] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *First Symposium on Operating System Design and Implementation*, 1994.

[154] Z. Wang. *Cooperative Software and Hardware Caching for Next Generation Memory Systems*. PhD thesis, University of Massachussets, Amherst, 2004.

[155] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, 1995.

[156] T. Y. Yeh and G. Reinman. Fast and Fair: Data-Stream Quality of Service. In *CASES'05*, Sep 2005.

[157] N. E. Young. On-line File Caching. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[158] M. Zhang and K. Asanovic. Victim Migration: Dynamically Adapting Between Private and Shared CMP Caches. Technical Report MIT-CSAIL-TR-2005-064, MIT CSAIL, 2005.

[159] M. Zhang and K. Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled CMPs. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, June 2005.

[160] Z. Zhang and J. Torrellas. Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, 1997.