HARDWARE-ORIENTED CACHE MANAGEMENT FOR LARGE-SCALE CHIP MULTIPROCESSORS

by

Mohammad Hammoud

BS, American University of Science and Technology, 2004

MS, University of Pittsburgh, 2010

Submitted to the Graduate Faculty of

the Department of Computer Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2010

UNIVERSITY OF PITTSBURGH

COMPUTER SCIENCE DEPARTMENT

This dissertation was presented

by

Mohammad Hammoud

It was defended on

July 1st 2010

and approved by

Rami Melhem, Professor, Department of Computer Science

Sangyeun Cho, Assistant Professor, Department of Computer Science

Jun Yang, Associate Professor, Electrical and Computer Engineering

Bruce Childers, Associate Professor, Department of Computer Science

Dissertation Advisors: Rami Melhem, Professor, Department of Computer Science,

Sangyeun Cho, Assistant Professor, Department of Computer Science

HARDWARE-ORIENTED CACHE MANAGEMENT FOR LARGE-SCALE CHIP MULTIPROCESSORS

Mohammad Hammoud, PhD

University of Pittsburgh, 2010

Abstract. One of the key requirements to obtaining high performance from chip multiprocessors (CMPs) is to effectively manage the limited on-chip cache resources shared among co-scheduled threads/processes. This thesis proposes new hardware-oriented solutions for distributed CMP caches.

Computer architects are faced with growing challenges when designing cache systems for CMPs. These challenges result from non-uniform access latencies, interference misses, the bandwidth wall problem, and diverse workload characteristics. Our exploration of the CMP cache management problem suggests a CMP caching framework (CC-FR) that defines three main approaches to solve the problem: (1) *data placement*, (2) *data retention*, and (3) *data relocation*. We effectively implement CC-FR's components by proposing and evaluating multiple cache management mechanisms.

Pressure and Distance Aware Placement (PDA) decouples the physical locations of cache blocks from their addresses for the sake of reducing misses caused by destructive interferences. Flexible Set Balancing (FSB), on the other hand, reduces interference misses via extending the life time of cache lines through retaining some fraction of the working set at underutilized local sets to satisfy far-flung reuses. PDA implements CC-FR's data placement and relocation components and FSB applies CC-FR's retention approach.

To alleviate non-uniform access latencies and adapt to phase changes in programs, Adaptive Controlled Migration (ACM) dynamically and periodically promotes cache blocks towards L2 banks close to requesting cores. ACM lies under CC-FR's data relocation category.

Dynamic Cache Clustering (DCC), on the other hand, addresses diverse workload characteristics and growing non-uniform access latencies challenges via constructing a cache cluster for each core and expands/contracts all clusters synergistically to match each core's cache demand. DCC implements CC-FR's data placement and relocation approaches.

Lastly, Dynamic Pressure and Distance Aware Placement (DPDA) combines PDA and ACM to cooperatively mitigate interference misses and non-uniform access latencies. Dynamic Cache Clustering and Balancing (DCCB), on the other hand, combines DCC and FSB to employ all CC-FR's categories and achieve higher system performance. Simulation results demonstrate the effectiveness of the proposed mechanisms and show that they compare favorably with related cache designs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

As the industry continues to shrink the size of transistors, chip multiprocessors (CMPs) are increasingly becoming the trend of computing platforms. CMPs can easily spread multiple threads of execution across various cores. Besides, CMPs scale across generations of silicon process simply by stamping down copies of the hard-to-design cores on successive chip generations [47]. A key requirement to obtaining high performance from CMPs is to effectively manage the limited on-chip cache resources. This dissertation addresses this requirement and presents a general framework for approaching CMP cache management. Per each category of the presented framework, a caching solution that satisfies performance needs required by large-scale CMPs is proposed.

1.1 CONVENTIONAL CMP CACHING SCHEMES

Exponential increase in cache sizes, bandwidth requirements, growing wire resistivity, power consumption, thermal cooling, and reliability considerations have necessitated a departure from traditional cache architectures. As such, large monolithic cache designs, referred to as uniform cache architectures (UCA) have been replaced by decomposed cache architectures, referred to as non-uniform cache architectures (NUCA). A cache (typically the L2 cache) is partitioned into multiple banks and distributed on a single die.

A traditional practice referred to as the shared scheme, logically shares the physically distributed L2 banks. Consequently, the available cache capacity is resourcefully utilized due to caching only a single copy of a shared cache block at a unique L2 bank. Nonetheless, On-chip access latencies differ depending on the distances between requester cores and target

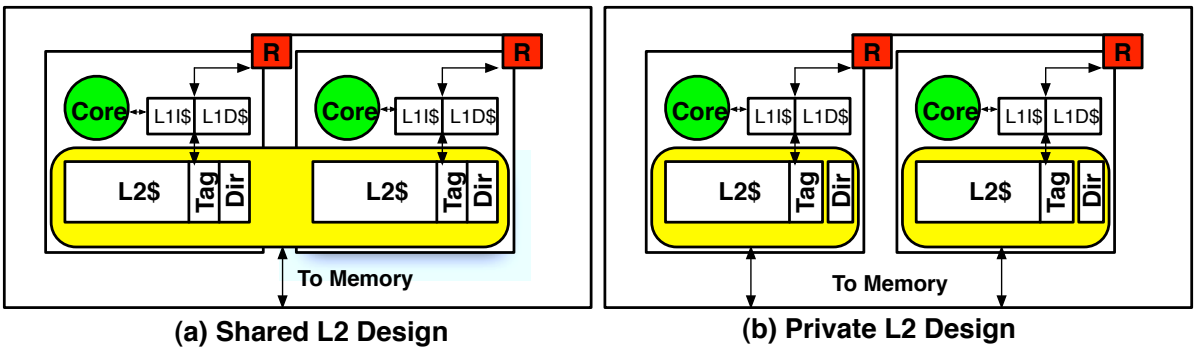


Figure 1: **Two traditional cache organizations. (a) The shared L2 design backs up all the L1 caches. (b) The private L2 design backs up only the private L1 cache on each tile. (Dir stands for directory and R for router).**

banks creating a Non Uniform Cache Architecture (NUCA) [40]. Another conventional practice referred to as the private scheme, associates each L2 bank to a single core and provides no capacity sharing among cores. However, the private scheme is characterized by the proximity of data to requester cores because each core maps and locates the requested cache blocks to and from its corresponding L2 cache bank. Fig. 1 demonstrates the two designs. For simplicity a tiled CMP architecture is shown with only two cores.

1.2 CMP CACHE MANAGEMENT CHALLENGES

Unlike conventional single-processor and multichip multiprocessor systems, CMP systems introduce new challenges and aggravate some prior inherent caching problems. The classical CMP organizations, the shared and the private paradigms, can't sufficiently and satisfactorily address all the emergent challenges. Some of the problems that are expected to be faced by computer architects in managing CMP caches are next described.

1.2.1 Growing Non uniform Access latencies

Access latencies to the cache banks are functions of proximity between distributed requesting cores and target banks. These non-uniform access latencies are expected to increase over time as chips involving a large number of cores are forthcoming. For instance, Intel's Tera-scale project is exploring ways to utilize hundreds of processors/caches tiles on a single die [31]. Thus, without careful data placement (or management), cache blocks of a small working set might map far away from a requesting core causing, consequently, remarkable deterioration of the average L2 access latency. Ongoing proposals, such as migration and replication mechanisms, have been suggested to reduce non-uniform access latencies [6, 7, 13, 16, 25, 39, 73, 74].

1.2.2 The Bandwidth Wall Problem and The Processor-Memory Speed Gap

Technology trends indicate that off-chip pin bandwidth will grow at a much lower rate than the number of processor cores on a CMP chip [11]. Emerging software inclinations also stress the memory bandwidth requirement [11]. In the traditional multichip multiprocessor systems (e.g., DSM or NUMA), each node encompasses a processor (or a group of processors), a cache hierarchy, and a local memory. Consequently, adding more nodes typically results in an increase of the aggregate bandwidth to the main memory. With CMPs, the situation is the opposite. Each node involves only a core and a cache hierarchy without a local memory. Hence, the amount of off-chip bandwidth that each core can utilize declines as the number of cores on a single die is scaled. This problem is referred to as the *bandwidth wall* problem where CMP system performance becomes increasingly limited by the amount of available off-chip bandwidth [60].

Lastly, processor and memory speeds are increasing at about 60% and 10% per year, respectively [30]. These factors combined altogether substantially increase the capacity pressure on the on-chip memory hierarchy. Efficient and intelligent CMP cache management can effectively tackle the bandwidth wall problem and bridge the increasing processor-memory speed gap.

1.2.3 Interference Misses

Off-chip memory accesses are very expensive. To mitigate the high off-chip data access latency, the microprocessor industry has classically incorporated techniques such as deep cache hierarchies, large associative last level caches (LLC), and sophisticated data prefetchers. But even with these techniques, a significant number of cache lines still miss in LLC [5]. Evaluations of 10 benchmarks from Spec2006 [63], PARSEC [8], and Splash-2 [70][1] showed that more than two thirds of the cache lines are never reused before being evicted. A similar study appeared in [51]. These cache lines are referred to as *zero reuse lines* and the problem is referred to as the zero reuse lines problem.

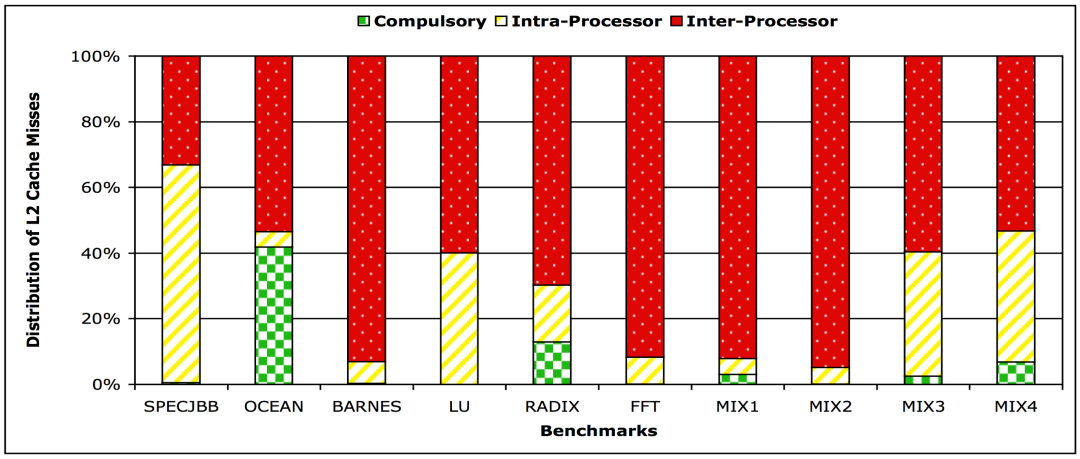
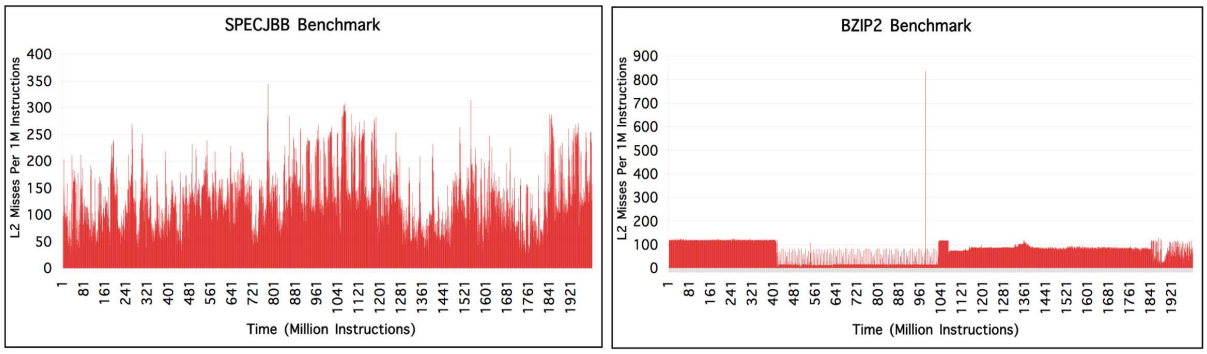


Figure 2: **Distribution of L2 cache misses (compulsory, intra-processor, and inter-processor).**

Many reasons cause the occurrence of zero reuse lines at LLC. First, memory references exhibit locality and are not evenly distributed across cache sets. This skew reduces the effectiveness of a cache and results in storing a considerable number of lines that are less likely to be re-referenced before replacement [48]. Second, the access stream visible to LLC is filtered through the higher level(s) caches on the memory hierarchy. Third, some cache lines reveal no temporal locality. Fourth, many cache lines exhibit far-flung reuses. That is, an evicted block might be used many times in the future, although not in the near-

[1]Description of the adopted CMP platform, the experimental parameters, and the benchmark programs used in this chapter can be found in chapter 2.

future [14]. Fifth, with the advent of CMPs the problem is exacerbated due to interferences among concurrently running threads/processes on an underlying shared LLC. A misbehaving application can evict useful L2 cache content belonging to other running programs. To establish such a key hypothesis, Fig. 2 demonstrates the distribution of the L2 cache misses for some simulated benchmarks. Misses in a CMP with a shared cache space can be classified into compulsory (caused by the first reference to a datum), intra-processor (a block being replaced at an earlier time by the same processor), and inter-processor (a block being replaced at an earlier time by different processor) misses [61]. For the simulated applications, on average, 6.8% of misses are compulsory, 23% are intra-processor, and 70% are inter-processor.



(a) (b)

Figure 3: **Cache demands are irregular among different applications and within the same application.**

1.2.4 Diverse Workload Characteristics

Computer applications exhibit different cache demands. Furthermore, a single application may demonstrate different phases corresponding to distinct code regions invoked during its execution [49]. A program phase can be characterized by different L2 cache misses and durations. Fig. 3 illustrates the L2 misses per 1 million instructions experienced by SPECJBB and BZIP2 from the SPEC2006 benchmark suite [63]. The behaviors of the two programs are clearly different and demonstrate characteristically different working set sizes and irregular execution phases.

Static partitioning of the available cache capacity might not tolerate the variability among different working sets and phases of a working set. For instance, a cache demanding program phase requires large cache capacity to mitigate the effect of increased cache misses. Conversely, a phase with less cache demand requires lower capacity to diminish average access latency. Static designs typically provide either fast accesses or capacity, but not both. A crucial step towards designing an efficient memory hierarchy is to provide both fast accesses and capacity.

1.3 A GENERAL CMP CACHING FRAMEWORK (CC-FR)

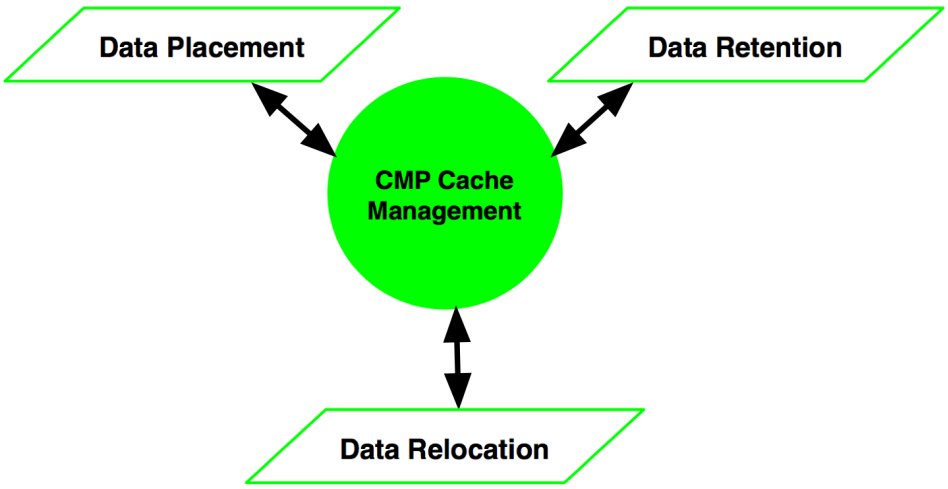


Figure 4: **General CMP Caching Framework (CC-FR).**

I suggest that the CMP cache management problem can be approached in three different ways. Fig. 4 depicts my thesis CMP caching framework (CC-FR). *Data placement* denotes the strategy to adopt on mapping cache blocks into the CMP cache space. *Data retention* indicates the strategy to follow on dealing with replaced cache blocks. *Data relocation* designates the strategy to utilize on moving (i.e., promoting and demoting) cache blocks after placed into the cache space. A CMP scheme can manage caches by targeting one or many of the suggested CC-FR's categories with an objective to reduce non-uniform access latencies, tackle the bandwidth wall problem, bridge processor-memory speed gap, alleviate

destructive interferences, and/or adapt to diverse workload characteristics.

1.4 THESIS OVERVIEW

The goal of this thesis is to effectively employ all CC-FR's approaches and address the presented CMP cache management challenges. At the very beginning, Constrained Associative-Mapping-of-Tracking-Entries (C-AMTE) is presented as a technique to enable flexible data placement and relocation CMP caching schemes. C-AMTE enables fast lookup of cache blocks in cache schemes that employ one-to-one or one-to-many associative mappings. C-AMTE stores in each core tracking data structures to avoid on-chip interconnect traffic outburst or long distance directory lookups. Simulation results show that C-AMTE achieves an improvement in the cache access latency by up to 34.4%, close to that of a perfect location strategy. Three of the proposed schemes in this thesis make use of C-AMTE to rapidly locate L2 cache blocks and effectively mitigate the average L2 access latency.

Motivated by the large non-uniform distribution of memory accesses across cache sets in different L2 banks, Pressure and Distance Aware (PDA) placement strategy is firstly described. PDA decouples the physical locations of cache blocks from their addresses for the sake of reducing misses caused by destructive interferences. Spatial pressure at the on-chip last-level cache, is continuously collected at a group (comprised of local cache sets) granularity, and periodically recorded at the memory controller(s) to guide the placement process. An incoming block is consequently placed at an underutilized cache group that is closest to the requesting core. To achieve fast lookup of cache blocks on subsequent accesses, PDA makes use of the C-AMTE location strategy. Clearly, PDA lies under CC-FR's data placement category and addresses interference misses, the bandwidth wall problem, and the processor-memory speed gap challenges. Simulation results show that PDA outperforms the baseline shared NUCA scheme by an average of 8.9% and by as much as 21.1% for the examined benchmarks. Furthermore, evaluations manifested the outperformance of PDA versus related cache designs.

Motivated by the large asymmetry in cache sets' usages within the same cache bank,

Flexible Set Balancing (FSB) is secondly proposed. FSB is a practical strategy for providing high-performance caching. The lifetime of cache lines is extended via retaining some fraction of the working set at underutilized sets to satisfy far-flung reuses. FSB promotes a very flexible sharing among cache sets, referred to as *many-from-many* sharing, providing significant reduction in interference misses. FSB targets CC-FR's data retention category and addresses interference misses, the bandwidth wall problem, and the processor-memory speed gap challenges. Simulation results demonstrate that FSB achieves an average miss rate reduction of 36.6% on multithreading and multiprogramming benchmarks from Spec2006 [63], PARSEC [8], and Splash-2 [70] suites. This translates into an average execution time improvement of 13%. Furthermore, evaluations manifested the outperformance of FSB over some recent proposals including DSBC [55] and V-WAY [53].

The third proposed scheme is the Adaptive Controlled Migration (ACM) scheme. ACM relies on prediction to collect accessibility information regarding cores that accessed a cache block B in the past, and then assuming that each of these cores will access B again in the future, dynamically migrates B to a bank that minimizes the average L2 access latency. ACM targets CC-FR's data relocation category and addresses growing non-uniform access latencies. C-AMTE is used by ACM to enable fast locating of migratory blocks on subsequent accesses. Simulation results demonstrate that ACM yields an average L2 access latency that is, on average, 20.4% better than a shared NUCA design.

Huh *et al.* [34] defined the concept of sharing degree (SD) as the number of processors that share a pool of L2 cache banks. In this terminology, an SD of 1 means that each core is assigned a single L2 bank not shared by any other core (private scheme). On the other hand, an SD of 16 means that each of the 16 cores (assuming a 16-way CMP platform) shares with all other cores the 16 L2 banks (shared scheme). Similarly, an SD of 2 means that 2 of the cores share their L2 banks. These sharing schemes with different SDs are referred to as the *fixed schemes*. It has been shown that no single fixed scheme provides the best performance for all kinds of workloads. As such, and to tolerate the variability among different working sets and phases of a working set, the Dynamic Cache Clustering (DCC) scheme is presented. DCC constructs a cache cluster (comprised of a number of L2 cache banks) for each core and expands/contracts all clusters dynamically to match each core's cache demand. The basic

Proposed Scheme	CC-FR's Category	Challenges Addressed
Pressure and Distance Aware Placement (PDA)	Data Placement	Interference Misses/ Processor-Memory Speed Gap/ Bandwidth Wall
Flexible Set Balancing (FSB)	Data Retention	Interference Misses/ Processor-Memory Speed Gap/ Bandwidth Wall
Adaptive Controlled Migration (ACM)	Data Relocation	Non-Uniform Access Latencies
Dynamic Cache Clustering (DCC)	Data Placement/ Data Relocation	Diverse Workload Characteristics/ Non-Uniform Access Latencies
Dynamic Pressure and Distance Aware Placement (DPDA)	Data Placement/ Data Relocation	Interference Misses/ Non-Uniform Access Latencies/ Processor-Memory Speed Gap/ Bandwidth Wall
Dynamic Cache Clustering and Balancing (DCCB)	Data Placement/ Data Retention/ Data Relocation	Interference Misses/ Diverse Workload Characteristics/ Non-Uniform Access Latencies/ Processor-Memory Speed Gap/ Bandwidth Wall

Table 1: **CC-FR's categories and challenges that each proposed scheme lies under and addresses.**

trade-offs of varying the on-chip cache clusters are average L2 access latency and L2 miss rate. DCC uniquely and efficiently optimizes both metrics and continuously tracks a near-optimal cache organization from the many possible configurations. DCC lies under CC-FR's data placement and relocation categories and addresses diverse workload characteristics and growing non-uniform access latencies challenges. Simulation results show that DCC improves the average L1 miss time by as much as 21.3% (10% execution time) versus previous static (fixed) designs.

Deciding upon the placement of a cache block from the first touch, as employed by PDA, might be sub-optimal. Virtually, multiple threads can compete for shared cache blocks. Ideally, we want to place a cache block at an L2 bank that best optimizes the overall access latencies from all the sharing cores. The best location for a shared block can't be known until runtime. As discussed earlier, ACM synergistically monitors the access patterns of cores and periodically migrates blocks to banks that minimize the access time for the sharing cores. As such, and to make PDA more practical, I suggest combining PDA and ACM in

one scheme referred to as the Dynamic Pressure and Distance Aware (DPDA) placement scheme. DPDA lies under CC-FR's data placement and relocation categories and addresses interference misses, growing non-uniform access latencies, the bandwidth wall problem, and the processor-memory speed gap.

Finally, and to implement all CC-FR's approaches and address all the presented CMP caching challenges, I propose combining DCC and FSB together in one scheme referred to as the Dynamic Cache Clustering and Balancing (DCCB) scheme. DCC increases the aggregate cache footprint via allowing replication of shared cache blocks at multiple clusters. Furthermore, DCC already implements two of CC-FR's components (i.e., data placement and relocation) and requires only a retention strategy to fully incorporate all CC-FR's categories. Clearly, augmenting FSB with DCC would fulfill the objective. However, with DCCB, it has been observed that DCC's interference misses were not significantly diminished. FSB demonstrated more effectiveness under the nominal shared NUCA organization than under DCC. Two main conclusions were drawn: (1) implementing more components of CC-FR might not necessarily correlate to a monotonic improvement in system performance, and (2) the additional obtained performance improvement from a combined scheme that implements all CC-FR's components (e.g., DCCB) might not even justify the incurred hardware overhead. Table 1 shows the CC-FR's categories and the caching challenges that each proposed scheme in this thesis targets and addresses.

1.5 CONTRIBUTIONS

The major contributions of my thesis are as follows:

- A general CMP Caching Framework (CC-FR) that defines three main management approaches: (1) *data placement*, (2) *data retention*, and (3) *data relocation*. I claim that any proposed CMP caching scheme would lie under one or more of CC-FR's categories.
- A new technique, Constrained Associative-Mapping-of-Tracking-Entries (C-AMTE), that enables flexible data placement and relocation CMP caching schemes. This location strategy can, in fact, be generally applied to cache organizations that extend the conventional

private and shared designs. Furthermore, it opens opportunities for architects to propose creative cache management strategies with no necessity to stick to either private or shared traditional paradigms.

- A Pressure and Distance Aware (PDA) scheme that implements CC-FR's data placement component.
- A Flexible Set Balancing (FSB) scheme that implements CC-FR's data retention component.
- An Adaptive Controlled Migration (ACM) scheme that implements CC-FR's data relocation component.
- A Dynamic Cache Clustering (DCC) scheme that implements CC-FR's data placement and relocation components.
- A combined scheme referred to as Dynamic Pressure and Distance Aware Placement (DPDA) that targets CC-FR's data placement and relocation approaches via combining PDA and ACM in one paradigm.
- A combined scheme referred to as Dynamic Cache Clustering and Balancing (DCCB) that targets all CC-FR's approaches via combining DCC and FSB in one paradigm.
- Two main observations that implementing more components of CC-FR might not necessarily correlate to a monotonic improvement in system performance, and that the additional obtained performance improvement from a combined scheme that targets all CC-FR's categories (e.g., DCCB) might not even justify the incurred hardware overhead for implementing that scheme.

1.6 ROADMAP

The rest of the thesis is organized as follows. Chapter 2 presents prior work and the evaluation methodology. Chapter 3 discusses the Constrained Associative-Mapping-of-Tracking-Entries (C-AMTE) strategy. In Chapter 4 I motivate and present the Pressure and Distance Aware Placement (PDA) scheme. The Flexible Set Balancing (FSB), the Adaptive Controlled Migration (ACM), and the Dynamic Cache Clustering (DCC) schemes are described in Chapters 5, 6, and 7, respectively. Dynamic Pressure and Distance Aware (DPDA) place-

ment and Dynamic Cache Clustering and Balancing (DCCB) are discussed in Chapter 8. Finally, conclusions and future work are given in Chapter 9.

2.0 RELATED WORK AND EVALUATION METHODOLOGY

2.1 BASELINE PROCESSOR ARCHITECTURE

Economic, manufacturing, and physical design considerations promote tiled CMP architectures (e.g., Tilera's Tile64 and Intel's Teraflops Research Chip) that co-locate distributed cores with distributed cache banks in tiles communicating via a network on-chip (NoC) [29]. Besides, tiled architectures have been recently advocated as a scalable processor design approach [56]. A tile typically includes a core, private L1 caches (I/D), and an L2 cache bank. Fig. 5 displays a typical 16-tile CMP architecture with a magnified single tile to demonstrate the incorporated components. I assume a 16-tile CMP model with a 2D mesh NoC in my experimental studies.

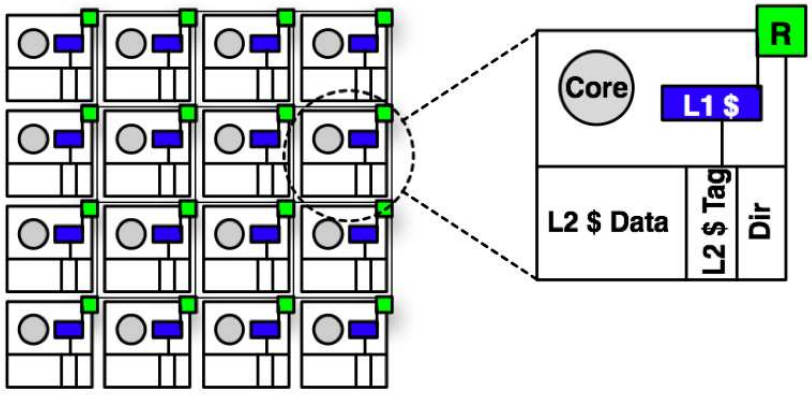


Figure 5: **Tiled CMP architecture (Figure not to scale).**

As described in Chapter 1, the distributed L2 cache banks can be either allocated one bank for one core (private scheme), or one bank for all cores (shared scheme). The private scheme replicates shared cache blocks at the L1 and L2 caches. As such, an engine is required

to maintain coherence at both levels (typically by using a distributed directory protocol. See Fig. 5 (a). Dir stands for directory). In contrast, the shared scheme requires an engine to maintain coherence at only the L1 level because no replication of shared cache blocks is allowed at the L2 cache. A core maps and locates a cache block, B, to and from a target L2 bank at a tile referred to as the *static home tile* (SHT) of B. The SHT of B stores B itself and its coherence state. The SHT of B is determined by a subset of bits (denoted as *home select* bits or HS bits) from B's physical address (e.g., block interleaved). This thesis assumes a shared NUCA design and employs a distributed directory protocol for coherence maintenance.

2.2 RELATED WORK

Much work has been done to effectively manage single-core as well as multi-core caches. Many proposals advocate CMP cache management at either fine (block) or coarse (page) granularities and base their work on either the nominal shared or private schemes. Furthermore, each proposal addresses reducing NUCA latencies, zero reuse lines and interference misses, bandwidth wall problem, and/or diverse workload characteristics. Each proposal lies under one or more of CC-FR's categories. I discuss below some of the prior related work that are most relevant to my proposed schemes and categorize them into single-core (mainly those that target interference misses challenge), CMP page-granular, or CMP block-granular caching schemes. Table 2 shows taxonomy of all the CMP discussed schemes in the context of my CC-FR framework.

2.2.1 Single-Core Caching Schemes

Reducing interference misses in uniprocessor caches has been for decades a hot topic of research [19, 37, 45, 65, 67, 72]. In summary, two main directions have been proven to reduce conflict misses effectively: (1) higher set associativity and (2) victim caching [37, 45].

A recent study, namely Dynamic Set Balancing Cache (DSBC) [55], suggests reducing

Scheme	Baseline	Granularity	CC-FR's Category	Addressed Challenges
Adaptive Set Pinning (ASP) [61]	Shared	Block	Retention/ Placement	IM/ BW and PMSG
Cooperative Caching (CC) [12]	Private	Block	Retention	IM/ BW and PMSG
Dynamic Spill-Receive (DSR) [50]	Private	Block	Retention	IM/ BW and PMSG
Hardware and Software Page Placement [58]	Shared	Page	Placement	IM/ BW and PMSG
OS-Based Page Allocation [17]	Shared	Page	Placement	IM/ BW and PMSG
Victim Replication (VR) [74]	Shared	Block	Retention	NUCA Latencies
CMP-NuRAPID [16]	Private	Block	Retention/ Placement/ Relocation	NUCA Latencies/ IM/ BW and PMSG
Adaptive Selective Replication (ASR) [6]	Private	Block	Retention	NUCA Latencies/ IM/ BW and PMSG/ DWC
Dynamic NUCA (DNUCA) [7]	Shared	Block	Relocation	NUCA Latencies
Nahalal [22]	Shared	Block	Placement/ Relocation	NUCA Latencies
Migration-Based NUCA [39]	Shared	Block	Relocation	NUCA Latencies
Victim Migration (VM) [73]	Shared	Block	Retention Relocation	NUCA Latencies
NUCA CMP Substrate [34]	Shared	Block	Placement Relocation	NUCA Latencies
PageNUCA [13]	Shared	Page	Relocation	NUCA Latencies
Dynamic Page Placement [3]	Shared	Page	Placement/ Relocation	NUCA Latencies/ DWC
R-NUCA [29]	Shared	Page	Placement/ Relocation	NUCA Latencies
Virtual Hierarchies (VHs) [44]	Shared	Page	Placement	NUCA Latencies/ DWC

Table 2: **Taxonomy of some CMP related work (IM = Interference Misses, PMSG = Processor-Memory Speed Gap, DWG = Diverse Workload Characteristics).**

interference misses via mitigating the large asymmetry in cache sets' usages. DSBC associates every two cache sets within a single cache structure, making the capacity of an underutilized set available for a pressured one. Once a set reaches a saturation level (set's miss rate hits a maximum value of $2K-1$ where K is the associativity of the cache) it requests a free (not associated yet) underutilized set. If such a set is detected, both sets, the highly pressured one (or referred to as *source*) and the underutilized one (or referred to as *destination*), are associated. As long as the two sets are associated, the source is allowed to *retain* its lines at the destination but not the reverse (i.e., unidirectional retention).

Variable-Way Set Associative Cache (V-WAY) [53] addresses the problem of workload imbalance among sets via varying the associativity of a cache by increasing the number of tag-store entries relative to the number of data lines. The tag and data stores are decoupled. As such, and to associate tags and data lines, forward and reverse pointers are used. The extra tag-store entries are added as additional sets rather than as additional ways in order to keep the number of tag comparisons required on each access unchanged. Besides, the data-store is structured as one large piece and a *global* frequency based replacement policy, referred to as *Reuse Replacement* is employed in order to achieve better replacements. In reverse, the tag-store keeps a conventional set granular (local) replacement strategy (e.g. LRU). The global replacement policy is triggered only if an invalid tag entry is found upon a miss. Otherwise, V-WAY bypasses the global policy, identifies a tag victim using an LRU local policy, and uses the tag's associated forward pointer to index the corresponding data line for replacement.

Many proposals suggest alternative indexing functions to achieve a more uniform distribution of memory accesses. Predictive Sequential Associative-Cache [9], Column Associative Cache [2], and Hash-Rehash [1] are proposed in the context of direct-mapped caches. They provide the capability of mapping a cache line at an alternative pre-determined (using different hash functions) cache frame in order to provide performance similar to that of 2-way caches (schemes referred to as skewed caches). In [57], an in-depth analysis of the pathological behaviors of cache hash functions is presented. Based on that analysis, the authors propose prime modulo and prime displacement hash functions resistant to pathological behaviors. Rolán *et al.* [55] started, in fact, with a skewed set associative cache, referred to as

static set balancing cache (SSBC), and found it impractical. Consequently, they proposed DSBC as a superior scheme.

Utility Based Cache Partitioning (UCP) [52] partitions at a way-granularity the last level shared cache among concurrently running applications depending on how much each application is likely to benefit from the cache (i.e., utility) rather than the application's demand for the cache. UCP suggests cost-effective monitoring circuits (UMON) to collect information about applications' utilities of cache resources. The collected information is then utilized by a partitioning algorithm to effectively decide the amount of cache allocated to each application.

Dynamic Insertion Policy (DIP) [51] makes a key observation that a large number of cache lines become dead on arrival. Thus, a Bimodal Insertion Policy (BIP) is proposed to insert incoming lines frequently in the LRU positions and infrequently (with a low probability) in the MRU positions. Lines inserted at the LRU positions are only promoted to the MRU positions upon hits while residing in the LRU positions. For LRU-friendly workloads (i.e., favoring MRU insertions), however, the changes to the insertion policy might become detrimental to cache performance. As such, a *Set Dueling* mechanism is proposed to select among BIP and LRU depending on which policy incurs fewer misses. Set Dueling dedicates a few sets of the cache to each of the two competing policies and uses the winner policy on the dedicated sets for the remaining follower sets.

Pseudo-Last-In-First-Out (Pseudo-LIFO) [14] proposes a family of replacement policies that manages each cache set as a fill stack. The replacement activities are restrained within a set to the upper part of the fill stack as much as possible. The lower part of the fill stack is left undistributed to extend the lifetime of the resident blocks. Among three members of the Pseudo-LIFO family, namely dead block prediction LIFO (dbpLIFO), probabilistic escape LIFO (peLIFO), and probabilistic counter LIFO (pcounter-LIFO), peLIFO is central. peLIFO synergistically learns the probabilities of experiencing hits beyond each of the fill stack positions and a set of highly preferred eviction positions is then deduced (based on this probability function) in the upper part of the fill stack.

Finally, Scavenger [5] partitions the total storage budget at the last level cache (LLC) into a conventional cache and a novel victim file (VF). Block addresses missing at the LLC

are prioritized based on the number of times they have been observed in the LLC miss stream. This is accomplished by incorporating a skewed bloom filter and a pipelined heap with the VF architecture. If a block is evicted from the conventional part of the cache and indicates a high priority (i.e., frequently missed in the recent past), it gets stored in the VF.

2.2.2 CMP Page-Granular Caching Schemes

Many researchers examined reducing interference misses at coarser (page) granularity [58, 36, 17, 3]. Sherwood et al. [58] proposed reducing misses using hardware and software page placement. Their software page placement algorithm performs a coloring of virtual pages using profiles at compile time. The generated colored pages can be used by the OS to guide the allocation of physical pages. Their hardware approach works by adding a page remap field to the TLB. This field is used as part of the index to the L2 cache (instead of the physical page number) thus allowing a page to be remapped to a different color in the cache while keeping the same physical page in memory. Cho and Jin [17] proposed an OS-based page allocation algorithm that maps cache blocks to the L2 cache space using a simple interleaving on page frame numbers.

To reduce non-uniform access latencies, Chaudhuri [13] proposed data migration at a page granularity. Access patterns of cores are dynamically monitored and pages are migrated to banks that minimize the access time for the sharing cores. Awasthi et al. [3] proposed re-coloring pages at runtime (via elegant use of shadow addresses to rename pages) then moving them to the center of gravity from various requester cores. Hardvellas et al. [29] proposed reactive NUCA (R-NUCA) that relies on OS to classify cache accesses onto either private, shared, or instructions. R-NUCA then places private pages into the local L2 cache banks of the requesting cores, shared ones into fixed address-interleaved on-chip locations, and instructions into non-overlapping clusters of L2 cache banks. Marty and Hill [44] proposed imposing a two-level virtual coherence hierarchy on a physically flat CMP that harmonizes with virtual machines (VMs) assignments.

2.2.3 CMP Block-Granular Caching Schemes

Dynamic Insertion Policy (DIP) [51] discussed above, uses a single policy (LRU or BIP) for *all* the concurrently running applications. A subsequent proposal, namely Thread-Aware Dynamic Insertion Policy (TADIP) [35], extends DIP to use a single policy for *each* application in the context of chip multiprocessors. Promotion/Insertion Pseudo-Partitioning (PIPP) [71] combines dynamic insertion and probabilistic promotion policies to provide the benefits of cache partitioning, adaptive insertion, and capacity stealing all with a single mechanism. Adaptive Set Pinning (ASP) [61] associates processors to cache sets and solely grant them permissions to evict blocks from their sets on cache misses. Therefore, references that may potentially cause inter-processor misses are no more allowed to interfere with each other even if they index the same set. Blocks that could lead to inter-processor misses are redirected to small processor owned private (POP) caches.

Based on nominal private caching, Chang and Sohi [12] proposed cooperative caching (CC) that creates a globally managed shared aggregate on-chip cache. CC employs spilling (instead of evicting) singlet blocks (blocks that have no replicas at the L2 cache space) to random L2 banks seeking to reduce intra-processor misses. Qureshi [50] proposed dynamic spill-receive (DSR) that improves upon CC by allowing private caches to either spill or receive cache blocks, but not both at the same time.

While all of the above CMP schemes attempt to reduce interference misses, many others explored reducing non-uniform access latencies. Zhang and Asanović [74] proposed victim replication (VR) that mitigates the average on-chip access latency via keeping replicas of local primary cache victims within the local L2 cache banks. Chisti et al. [16] proposed CMP-NuRAPID that controls replication based on usage patterns. Beckmann et al. [6] proposed adaptive selective replication (ASR) that dynamically monitors workloads behaviors to control replication on the private cache organization.

Beckmann and Wood [7] studied generational promotion and suggested *Dynamic NUCA* (DNUCA) that migrates blocks towards banks close to requesting processors. Guz et al. [22] presented a new CMP architecture that utilizes migration to divert only shared data to cache banks at the center of the chip close to all cores. Kandemir et al. [39] proposed a mechanism

Component	Parameter
Number of Tiles	16
Network On-Chip	2D Mesh
Cache Line Size	64 B
L1 I/D-Cache Size/Associativity	32KB/2way
L1 Hit Latency	1 cycle
L1 Replacement Policy	LRU
L2 Cache Size/Associativity	512KB per L2 bank or 8MB aggregate/16way
L2 Bank Access Penalty	12 cycles
L2 Replacement Policy	LRU
Latency Per NoC Hop	3 cycles
Memory Latency	320 cycles

Table 3: **System parameters.**

Name	Input
SPECJbb	Java HotSpot (TM) server VM v 1.5, 4 warehouses
Bodytrack	4 frames and 1K particles (16 threads)
Fluidanimate	5 frames and 300K particles (16 threads)
Swaptions	64 swaptions and 20K simulations (16 threads)
Barnes	64K particles (16 threads)
Lu	2048×2048 matrix (16 threads)
MIX1	Hmmer (reference) (16 copies)
MIX2	Sphinx (reference) (16 copies)
MIX3	Barnes, Ocean(1026×1026 grid), Radix (3M Int), Lu, Milc (ref), Mcf (ref), Bzip2 (ref), and Hmmer (2 threads/copies each)
MIX4	Barnes, FFT (4M complex numbers), Lu, and Radix (4 threads each)

Table 4: **Benchmark programs.**

that determines suitable locations for data blocks at runtime, and then promotes these blocks to these calculated locations. Zhang and Asanović [73] examined direct migration and promoted cache blocks straightforwardly from their home tiles to the tiles of the initial requesters (i.e., a first touch migration policy). Huh et al. [34] proposed a spectrum of degrees of sharing to manage NUCA caches and suggested generational migration to reduce their negative latency effects.

2.3 EVALUATION METHODOLOGY

Throughout this thesis I use the following evaluation methodology unless otherwise specified. All results are presented based on detailed full-system simulation using Virtutech's Simics 3.0.29 [68]. I use my own CMP cache modules fully developed in-house. I implement the XY-

routing algorithm and accurately model congestion for both coherence and data messages. A tiled CMP architecture comprised of 16 UltraSPARC-III Cu processors is simulated running with Solaris 10 OS. Each processor uses an in-order core model with an issue width of 2 and a clock frequency of 1.4 GHz. The tiles are organized as a 4×4 grid connected by a 2D mesh NoC. A 3-cycle latency (in addition to the NoC congestion delay) per hop is incurred when a datum traverses through the mesh network [74, 73]. Each tile encompasses a switch, 32KB I/D L1 caches, and a 512KB L2 cache bank. A distributed MESI-based directory protocol is employed. Table 3 shows my configuration's experimental parameters.

To study the proposed and related schemes, a mixture of multithreaded and multiprogramming workloads is utilized. For multithreaded workloads I use the commercial benchmark SpecJBB [63], five shared memory programs (Ocean, Barnes, Lu, Radix, and FFT) from the SPLASH-2 suite [70], and three applications (Bodytrack, Fluidanimate, and Swaptions) from the PARSEC suite [8]. Besides, six applications (Parser, Art, Equake, Mcf, Ammp, and Vortex) from SPEC2K [63] and five programs (Hmmer, Sphinx, Milc, Mcf, and Bzip2) from SPEC2006 [63] are used. I compose multiprogramming workloads using different programs from SPLASH-2, SPEC2K, and SPEC2006 benchmarks. Table 4 shows the data sets and other important features of all the examined benchmark programs. The workloads are fast forwarded to get past of their initialization phases. After various warm-up periods, each SPLASH-2 and PARSEC benchmark is run until the completion of its main loop, and each of SpecJBB, MIX1, MIX2, MIX3, and MIX4 is run for 8 billion user instructions.

3.0 CONSTRAINED ASSOCIATIVE-MAPPING OF TRACKING ENTRIES

In this chapter I describe Constrained Associative-Mapping-of-Tracking-Entries (C-AMTE), a scalable mechanism to facilitate flexible and efficient distributed cache management in large-scale chip multiprocessors (CMPs). C-AMTE enables data placement and relocation in CMP caching schemes. In Section 3.1 I define the *location problem* in CMP cache management and outline my proposed solution. Section 3.2 describes C-AMTE mechanism. An evaluation of C-AMTE applied to a classical CMP caching scheme is presented in Section 3.3 and conclusions are given in Section 3.4.

3.1 PROBLEM DEFINITION AND PROPOSED SOLUTION

3.1.1 Problem Definition

The nominal private scheme replicates cache blocks at the L2 banks of the requesting cores. Hence, an effective cache associativity which equates the aggregate associativity of the L2 cache banks is provided [12]. For instance, 16 private L2 banks with 8-way associativity effectively offer 128-way set associativity. A cache block can map to any of the 128-way entries, and if shared amongst cores, can reside in multiple L2 banks. A high bandwidth on-chip directory protocol can be employed to keep the multiple L2 banks coherent. The directory can be held as a duplicate set of L2 tags distributed by set index across tiles [4, 74]. I generally refer to a mapping process that exploits the aggregate associativity of the L2 cache banks as an *associative mapping* strategy. In particular, I designate the mapping strategy of the private scheme as *one-to-many associative mapping* because a single block can be

mapped to multiple L2 banks.

In contrast to the private design, the nominal shared scheme maintains the exclusiveness of cache blocks at the L2 level. A core maps and locates a cache block, B, to and from a target L2 bank at a tile referred to as the *static home tile* (SHT) of B. The SHT of B is determined by a subset of bits denoted as *home select* bits (or HS bits) from B's physical address. As such, the shared strategy requires maintaining coherence only at the L1 level. The SHT of B can store B itself and a bit vector indicating which cores had cached copies of B in their L1 private caches. This on-chip coherence practice is referred to as an in-cache coherence protocol [10, 74]. In this thesis I refer to an entry that tracks copies (either at L1 or L2) of a certain cache block as a *tracking entry*. I, furthermore, identify a mapping process that maps an entry (block or tracking) to a fixed tile as a *fixed mapping* strategy (e.g., the shared design employs fixed mapping).

Recent research work on CMP cache management has recognized the importance of the shared scheme [61, 22, 34, 64, 39]. Besides, many of today's multi-core processors, the Intel Core^{TM}2 Duo processor family [54], Sun Niagara [41], and IBM Power5 [59], have featured shared caches. A shared design, however, suffers from a growing on-chip delay problem. Access latencies to L2 banks are non-uniform and proportional to the distances between requester cores and target banks. This drawback is referred to as the *NUCA problem*.

To mitigate the NUCA problem, many proposals have extended the nominal basic shared design to allow associative mapping (i.e., leveraging the aggregate associativity of the L2 cache banks). For instance, block migration [7, 22, 34, 39, 73] exploits associative mapping by moving frequently accessed blocks closer to requesting cores. I denote such a strategy as *one-to-one associative mapping* due to the fact that the exclusiveness of cache blocks at the L2 level is still preserved (only a single copy of a block is promoted along identical sets over different banks). In contrast to migration, replication duplicates cache blocks at different L2 banks [16, 12, 74]. Accordingly, a replication scheme is said to adopt *one-to-many associative* mapping.

A major shortcoming of using associative mapping for blocks in any CMP cache management scheme is the location process. For example, a migration scheme that promotes a cache block B to a tile different than its home tile, denoted as the *current host* of B, can't use

anymore the HS bits of B's physical address to locate B. Consequently, different strategies for the location process need to be considered. A tracking entry can always be retained at a centralized directory or at B's home tile (if the underlying directory protocol is distributed) to enable tracking B after promotion. Hence, if a core requests B, the repository of the tracking entries is reached first then the query is forwarded to B's host tile to satisfy the request. The disadvantage of this option is the arousal of 3-way cache-to-cache communication which can degrade the average L2 access latency. An alternative location strategy could be to broadcast queries to all the tiles assuming no tracking entry for B is kept at a specific repository. Such a strategy can, however, burden the NoC and potentially degrade the overall system performance.

3.1.2 Proposed Solution

I propose Constrained Associative-Mapping-of-Tracking-Entries (C-AMTE), a mechanism that flexibly accelerates cache management in CMPs. In particular, C-AMTE presents *constrained associative mapping* that combines the effectiveness of both, the associative and fixed mapping strategies and applies that to tracking entries to resolve the challenge of locating cache blocks without broadcasting and with minimal 3-way communications.

To summarize, the contributions of C-AMTE are as follows:

- It enables fast location of cache blocks without swamping the NoC.
- It can be applied whenever associative mapping is used for cache blocks, either in case of one-to-one (i.e, migration) or one-to-many (i.e, replication).
- It can be generally applied to cache organizations that extend the conventional private or shared schemes. Furthermore, it opens opportunities for architects to propose more creative cache management designs with no necessity to stick to either private or shared traditional paradigms.

3.2 THE CONSTRAINED ASSOCIATIVE-MAPPING-OF-TRACKING-ENTRIES (C-AMTE) MECHANISM

Constrained Associative-Mapping-of-Tracking-Entries (C-AMTE) is not an autonomous CMP cache organization that can run by itself but rather a mechanism that can be applied to CMP cache designs that employ one-to-one (i.e., migration) or one-to-many (i.e., replication) associative mappings. A shared NUCA architecture maps and locates a cache block, B, to and from a *home tile* determined by a subset of bits (home select or HS bits) from B's physical address. Accordingly, B might be mapped to a bank far away from the requester core, causing the core significant latency to locate B. Such a problem is referred to as the *NUCA problem*. Migration and replication have been suggested as techniques to alleviate the NUCA problem. To save latency on subsequent requests to B, migration and replication relocate and replicate, respectively B at a tile different than its home tile, denoted as the *host tile* of B, closer to requesting cores. Consequently, B can have, in addition to the home tile, one or more host tiles. To locate B at a host tile, the HS bits of B's physical address can't be used anymore. C-AMTE offers a robust and versatile location strategy to locate B at host tiles.

Assuming a distributed directory protocol, C-AMTE supports storing one tracking entry corresponding to a block B at the home tile of B. I refer to this tracking entry as the *principal* tracking entry. The principal tracking entry points to B and can always be checked by any requester core to locate B at its current host. The principal tracking entry is stored using a fixed mapping strategy because the home tile of B is designated by the HS bits of B's physical address. C-AMTE also supports storing another type of tracking entries for B at requester tiles. I refer to this type of tracking entries as *replicated* tracking entries. A replicated tracking entry at a requester tile also points to the current host of B but can be rapidly checked by a requester core to directly locate B (instead of checking with B's home tile to achieve that). The idea of replicating tracking entries at requester tiles capitalizes on the one-to-many associative mapping strategy traditionally applied for cache blocks. C-AMTE combines associative and fixed mapping strategies and applies that to

	Block Mapping	Tracking Entries Mapping
Private Scheme (P)	Associative (at requesting tiles)	Fixed (at home tiles)
Shared Scheme (S)	Fixed (at home tiles)	Fixed (at home tiles)
Scheme With C-AMTE	Associative (one-to-one or one-to-many depending on the underlying cache scheme)	Constrained=Fixed (at home tiles) + Associative (at requesting tiles)

Table 5: **Mapping strategies of private and shared CMP caches and the hybrid mapping approach of C-AMTE.**

tracking entries in order to efficiently solve the location problem. Table 5 illustrates the hybrid approach adopted by the C-AMTE mechanism. I refer to such a hybrid mapping process as a *constrained associative mapping* strategy.

Based on the above discussion, per each tile, T, a principal tracking entry is kept for each cache block B whose home tile is T but had been mapped/promoted to another tile. Besides, replicated tracking entries are retained at T to track the locations of other corresponding cache blocks that have been recently accessed by T but whose home tile is not T. Though both, principal and tracking entries essentially act as pointers to the current hosts of cache blocks, I differentiate between them for consistency and replacement purposes (more on this shortly). I can add two distinct data structures per each tile to store the two types of the tracking entries. A data structure, referred to as the principal tracking entries (PTR) table, can hold principal tracking entries, and a data structure, referred to as the replicated tracking entries (RTR) table, can hold replicated ones. Alternatively, a single table, could be referred to as the tracking entries (TR) table, can be added to hold both classes of tracking entries pertaining that a hardware extension (i.e. an indicative bit) is engaged to distinguish between the two entries.

Assume a CMP organization with PTR and RTR tables. Whenever a core issues a request to a block B, its RTR table is checked first for a matching replicated tracking entry. C-AMTE then proceeds as follows:

- On a miss at the RTR table, the home tile of B is reached and its PTR table is looked up.
  - If a miss occurs at the PTR table, B is fetched from the main memory and mapped to a tile T specified by the underlying cache scheme. If T is not B's home tile, principal and replicated tracking entries are stored at the PTR table of B's home tile and the RTR table of the requester core, respectively. If, in contrary, T is B's home tile, no tracking entries are kept at either PTR or RTR tables (B can be located directly using the HS bits of B's physical address).
  - If, on the other hand, a hit occurs at the PTR table, B is located at its current host tile and a replicated tracking entry is stored at the requester's RTR table.
- On a hit at the RTR table, B is located directly at its current host designated by the matched replicated tracking entry.

Therefore, upon a hit to the requester's RTR table, a 3-way cache-to-cache communication, which would have been incurred if I had to approach B's home tile to locate B, is avoided. Similar logic applies if C-AMTE assumes a single TR table instead of two distinct PTR and RTR ones.

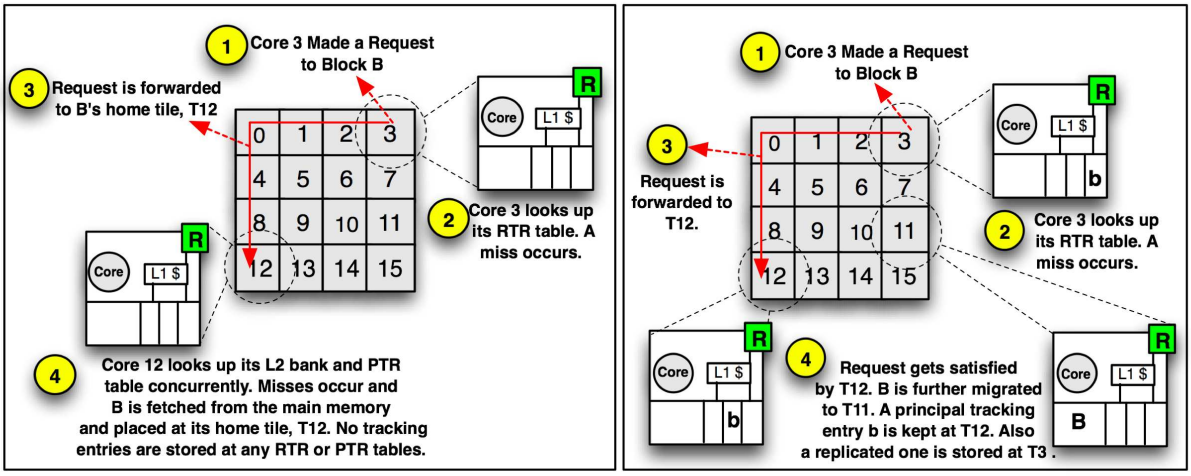


Figure 6: **A first example on locating a migratory block B using the C-AMTE mechanism.**

Fig. 6 demonstrates an example of the C-AMTE mechanism on a tiled CMP platform, assuming an underlying shared scheme and a migration policy that promotes cache blocks towards requesting cores. Fig. 6(a) shows a request made by core 3 to a cache block, B. Core 3 looks up it local RTR table. I assume a miss occurs and the request is subsequently forwarded to B's home tile, T12. The PTR table and the regular L2 bank at T12 are looked up concurrently. I assume misses occur at both. Consequently, B is fetched from the main memory and mapped to B's home tile, T12 (following the mapping strategy of the nominal shared scheme). As such, no tracking entries are retained at either PTR or RTR tables. Fig. 6(b) shows a subsequent request made by core 3 to B. B is located at its home tile, T12. Assume after that hit, B is migrated to T11 (closer to T3). Thus, corresponding principal and replicated tracking entries are stored at T12 and T3, respectively. If at any later time core 3 requests B again, a hit will occur at its RTR table (as long as the entry has not been replaced yet) and B can be located straightforwardly at T11 avoiding thereby 3-way cache-to-cache transfers. Lastly, note that if any other core requests B, T12 can always indirectly satisfy the request and a corresponding tracking entry can be stored at the new requester's RTR table.

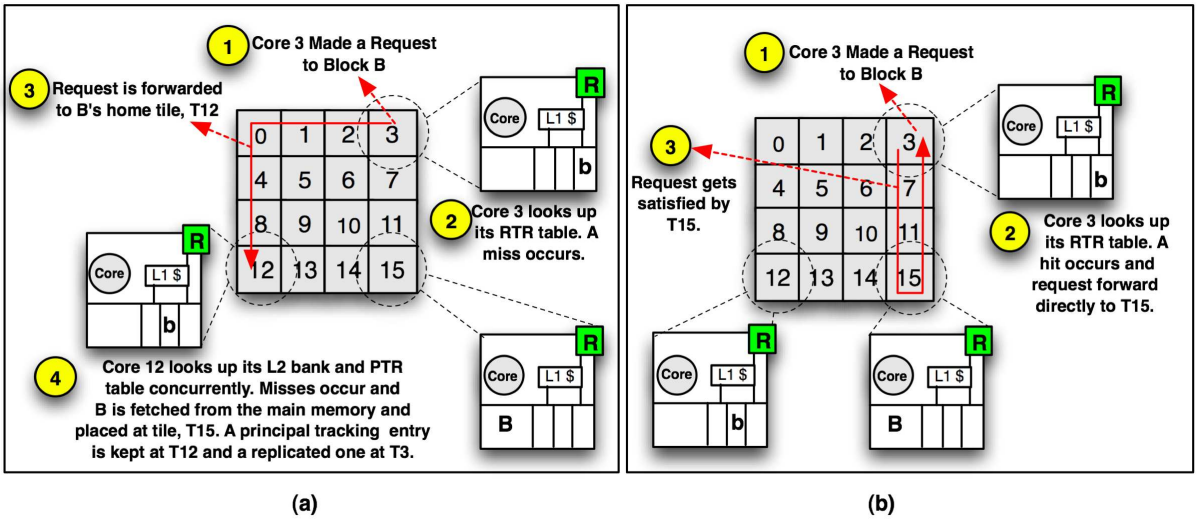


Figure 7: **A second example on locating a block B using the C-AMTE mechanism.**

Fig. 7 demonstrates C-AMTE in operation assuming a cache scheme that might map cache blocks to tiles different than their home tiles. Fig. 7(a) shows a request made by core 3 to a cache block B. Core 3 looks up it local RTR table. I assume a miss occurs and the request is subsequently forwarded to B's home tile, T12. The PTR table and the regular L2 bank at T12 are looked up concurrently and misses are then incurred. Consequently, B is fetched from the main memory and mapped to T15 (determined by the mapping strategy of the cache scheme). As such, principal and replicated tracking entries are kept at T12 and T3, respectively. Fig. 7(b) shows a request made again by core 3 to B. A hit occurs at T3's RTR table. Consequently, B is directly located at T15. Clearly, the two examples shown in Figures 6 and 7 reveal the efficiency and versatility of C-AMTE as a strategy that exploits distance locality. C-AMTE, in fact, opens opportunities for architects to propose creative migration, replication, and placement CMP strategies with the required location process being on-hand.

The principal and replicated tracking entries need to be kept coherent. I accomplish this by embedding a bit vector with each principal tracking entry at the PTR tables to indicate which cores had cached related replicated tracking entries at their RTR tables (similar to the in-cache coherence protocol in [10]). For instance, given the example depicted in Fig. 6, each time B is migrated to a different tile, the principal and the replicated tracking entries that correspond to B are updated to point to the new host of B. Besides, C-AMTE can easily preclude potential false misses that can occur when L2 requests fail to hit on cache blocks because they are in transit between L2 banks. When migration is to be performed, a copy, B', of the cache block B is kept at the current bank so as if an L2 request arrives while B is in transit, the request is immediately satisfied without incurring any delay. When B reaches the new host, an acknowledgement message is sent back to the old host to discard B'. The old host keeps track of any tile that accesses B', and when receiving the acknowledgment message, sends an update message to the new host to indicate the new sharers that requested B while it was in transit. The directory state entry of B is consecutively updated. Clearly, enforcing coherence among tracking entries and precluding false misses impose traffic overhead on the network on-chip. Section 3.3.1 demonstrates the increase in message hops per 1K instructions incurred by the C-AMTE mechanism.

To that end, each principal tracking entry would include: (1) The tag of the related block (typically 22 bits), (2) a bit vector that acts as a directory to keep the principal and the replicated tracking entries coherent (16 bits for a 16-tile CMP model), and (3) an ID that points to the tile that is currently hosting the block (4 bits for a 16-tile CMP model). On the other hand, a replicated tracking entry would include only the tag of the related block and the ID of the current host tile. In contrast, in case of a single TR table, both, the principal and the replicated tracking entries would each encompass a tag, a bit vector, an ID, and an indicative bit to distinguish between the two types of entries (required for replacement purposes). Clearly, the bit vector added to each replicated entry becomes in this case redundant. Thus, splitting TR table into RTR and PTR might be preferable for reducing storage overhead.

Finally, PTR and RTR tables can employ the LRU replacement policy. However, in case of a single TR table, it is wise to never evict a principal tracking entry in favor of a replicated one (this is the reason of why I suggested distinguishing between the two entries). An eviction of a principal tracking entry causes evictions to the corresponding cache block and all the related replicated tracking entries. Therefore, the TR replacement policy should replace the following three classes of entries in an ascending order: (1) an invalid entry, (2) the LRU replicated tracking entry, (3) and the LRU principal tracking entry. Besides, upon storing a replicated tracking entry, only the first two classes are considered for replacement. If no entry belonging to one of these two classes is detected, a replicated tracking entry is not retained.

3.3 QUANTITATIVE EVALUATION

I study C-AMTE with an implementation of the DNUCA scheme [34, 7]. I employ DNUCA on my tiled CMP architecture (see Section 2.1) via allowing migration in vertical and horizontal directions seeking to reduce hit latencies. Each cache block is augmented by four 2-bit saturation counters in correspondence to the four plausible ways: north, south, west, and east. Once a counter saturates, its value is cleared and the block is migrated towards

Name	Input
SPECJbb	Java HotSpot (TM) server VM v 1.5, 4 warehouses
Ocean	514×514 grid (16 threads)
Barnes	32K particles (16 threads)
Lu	2048×2048 matrix (16 threads)
Radix	3M integers (16 threads)
Bodytrack	4 frames and 1K particles (16 threads)
Fluidanimate	5 frames and 300K particles (16 threads)
Swaptions	64 swaptions and 20K simulations (16 threads)
MIX1	Hmmer (reference) (16 copies)
MIX2	Sphinx (reference) (16 copies)
MIX3	Barnes, Ocean, Radix, Lu, Milc (ref), Mcf (ref), Bzip2 (ref), and Hmmer (2 threads/copies each)
MIX4	Barnes, FFT (4M complex numbers), Lu, and Radix (4 threads each)

Table 6: **Benchmark programs.**

the indicated direction (i.e., promoted up, down, left, or right one tile upon the saturation of the north, south, west, or east counter, respectively). To locate cache blocks after migration, C-AMTE is utilized. I refer to this DNUCA implementation with C-AMTE being incorporated as DNUCA(C-AMTE).

To demonstrate the potential performance gain from C-AMTE I compare DNUCA(C-AMTE) against the baseline shared (S) scheme and three other DNUCA implementations that are only different in their location processes. First, I consider DNUCA with a broadcast location strategy. That is, queries to all tiles are sent upon every L2 request to locate the required block. I denote this implementation as DNUCA(B). Second, a 3-way cache-to-cache transfer strategy is employed similar to the one in [73]. This implementation is designated as DNUCA(3W). Lastly, I consider DNUCA with an ideal location strategy to set an upper bound for C-AMTE and see how close it draws to a perfect approach. The ideal strategy assumes that cores have oracle knowledge about the on-chip residences of blocks. Hence, every L2 request is routed directly to the correct L2 bank. I refer to such an implementation as DNUCA(Ideal).

The evaluation methodology I use in this chapter is the one described in Section 2.3. In addition, a tracking table (TR) with 16K entries is incorporated to each tile. The latency to lookup a TR table is hidden under the delay to enqueue the request in the port scheduler of the local switch [13]. Lastly, Table 6 illustrates the simulated programs as they are a little different than the ones shown in Table 4. After various warm-up periods, each SPLASH-2 and PARSEC benchmark is run until the completion of its main loop, and each of SpecJBB, MIX1, MIX2, MIX3, and MIX4 is run for 20 billion user instructions.

3.3.1 Results

Fig. 8 demonstrates the average L2 access latency (AAL) of S, DNUCA(B), DNUCA(3W), DNUCA(C-AMTE), and DNUCA(Ideal) schemes normalized to S. The incurred latency per L2 access is defined depending on three scenarios. First, it can involve only the L2 access time. This happens when a hit occurs to a local L2 bank from a requesting core. Second, it can incorporate distance latency (computed in terms of the number of hops traversed

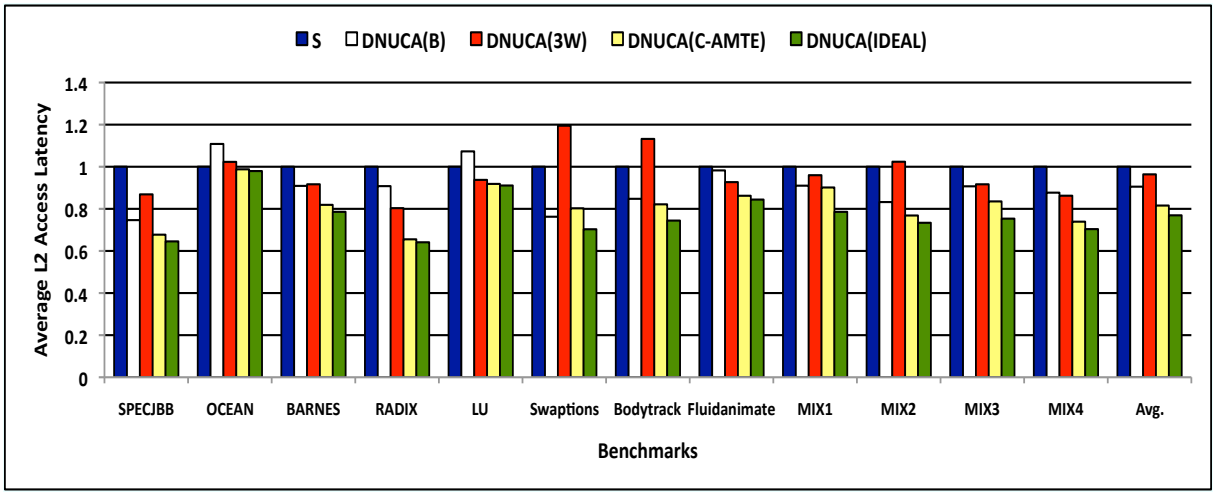


Figure 8: **Average L2 access latency of the baseline shared scheme (S), DNUCA(B), DNUCA(3W), DNUCA(C-AMTE), and DNUCA(Ideal) normalized to S (B= Broadcast, 3W = 3 Way).**

	S	DNUCA(B)	DNUCA(3W)	DNUCA(C-AMTE)	DNUCA(IDEAL)
SPECjbb	5.3	87.5	5.7	4.8	2.4
Ocean	2.5	35.8	3	3.1	2.4
Barnes	3.6	55.1	4.4	4	2.9
Radix	6.9	136.4	9.8	13.5	9.4
Lu	70	905.4	78.3	76	70.5
Swaptions	4.8	64.3	6.6	7.2	3.2
Bodytrack	5.2	95	8.5	11.3	4.9
Fluidanimate	11.3	174.9	11.88	11.82	10.3
MIX1	35.5	573.8	37.3	37.6	27.4
MIX2	22.1	370.2	32.6	47	19
MIX3	11.6	168	16.4	14.9	10.3
MIX4	50.8	691.7	54.8	80	26

Table 7: **Message-Hops per 1K insructions**

between the requester and the target tiles and the observed NoC congestion delay) and the L2 access time. This occurs upon a hit to a remote L2 bank. Third, it can involve memory latency because of a miss on L2. DUNCA(C-AMTE) achieves AAL improvement over S by an average of 18.4%, and by as much as 34.4% for Radix. This makes DNUCA(C-AMTE) significantly close to DNUCA(Ideal) which accomplishes, in contrast, an average AAL improvement of 23%. DNUCA(C-AMTE) doesn't come closer to DNUCA(Ideal) because of two main reasons: (1) misses to TR tables by requester cores and (2) overhead to keep the principal and the replicated tracking entries coherent after blocks' migrations. Consequently, DNUCA(C-AMTE) generates a higher NoC traffic which causes more NoC delay and, subsequently, inferior AAL. Table 7 shows the number of message-hops per 1K instructions experienced by all the studied schemes for the examined benchmark programs. A message-hop is defined as one message travelling one hop on a router in the 2D mesh NoC.

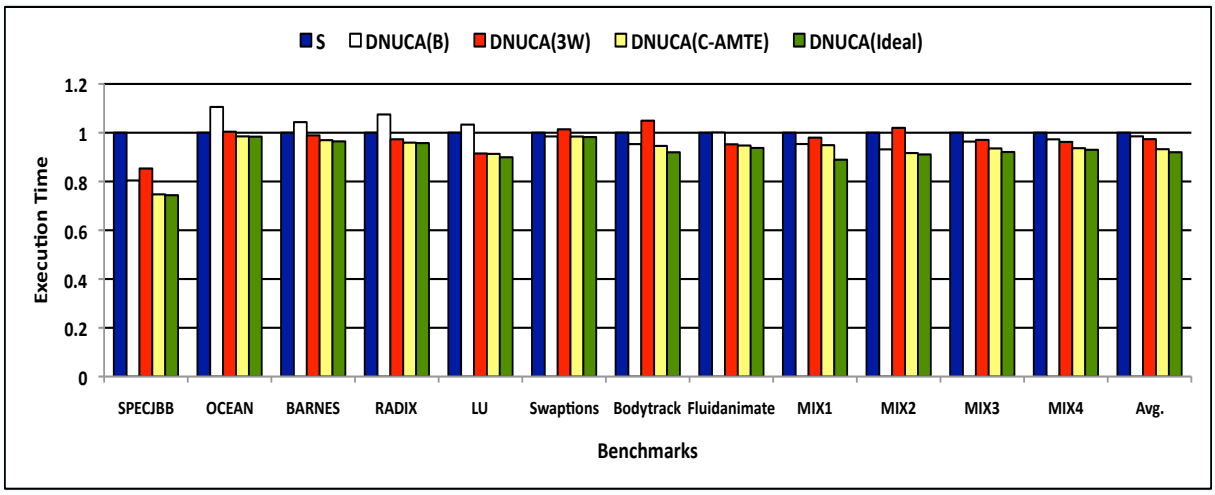


Figure 9: **Execution times of the baseline shared scheme (S), DNUCA(B), DNUCA(3W), DNUCA(C-AMTE), and DNUCA(Ideal) normalized to S (B= Broadcast, 3W = 3 Way).**

As shown in Fig. 8, DNUCA(B) and DNUCA(3W) provide AAL improvements over S by averages of 9.4% and 3.6%, respectively. DNUCA(B) is similar to DNUCA(Ideal) in that it offers a direct locations for cache blocks. However, DNUCA(B) profoundly outbursts the NoC with superfluous queries. This causes more NoC delay which translates to a lower

AAL improvement. Two factors determine the eligibility of an application to accomplish a higher or a lower AAL under DUNCA(B): (1) the gain, G, out of direct locations to cache blocks and (2) the loss, L, from congestion delay. When L is offset by G, DNUCA(B) improves AAL (e.g., SpecJBB), otherwise, a degradation over S is observed (e.g., Ocean). DNUCA(3W), on the other hand, fails to exploit distance locality and is expected, accordingly, not to surpass S. Nonetheless, most of the applications experience AAL improvement under DNUCA(3W) (SpecJBB, Barnes, Radix, Lu, Fluidanimate, MIX1, MIX3, MIX4). This improvement comes, in fact, from the fewer off-chip accesses attained by DNUCA. Computer programs exhibit large asymmetry in cache sets' usages [55, 53]. DNUCA inadvertently equalizes the non-uniformity across cache sets via the employment of the one-to-one associative mapping.

To that end, Fig. 9 presents the execution times of S, DNUCA(B), DNUCA(3W), DNUCA(C-AMTE), and DNUCA(Ideal) normalized to S. Across all benchmarks, DNUCA(B), DNUCA(3W), DNUCA(C-AMTE), and DNUCA(Ideal) outperform S by averages of 1.4%, 2.6%, 6.7%, and 8%, respectively. Though DNUCA(B) accomplished 9% and 9.2% AAL reductions over S for Barnes and Radix respectively, this didn't effectively translate to an improvement in the overall system performance.

3.4 SUMMARY

Cache management in CMP is crucial to fuel its performance growth. I propose C-AMTE, a mechanism that effectively simplifies the process of locating cache blocks in CMP caching schemes that employ either one-to-one or one-to-many associative mappings. C-AMTE stores tracking entries that correspond to cache blocks at per-core data structures for direct locations at subsequent accesses. From the perspective of my CC-FR framework, C-AMTE enables data placement and relocation CMP caching schemes. I demonstrated the effectiveness of C-AMTE by applying it to the DNUCA [7, 34] migration scheme (i.e., a scheme that adopts one-to-one associative mapping). A performance improvement of up to 25.2% has been achieved, close to that of a perfect location strategy.

4.0 PRESSURE AND DISTANCE AWARE PLACEMENT

In this chapter I describe Pressure and Distance Aware (PDA) Placement, a novel mechanism that involves cache pressure as well as distance locality in its placement algorithm. PDA targets CC-FR's data placement category and addresses interference misses, growing non-uniform access latencies, the bandwidth wall problem, and the processor-memory speed gap challenges. In Section 4.1, I provide a motivational study and outline my proposed solution. I detail PDA in Section 4.2. A quantitative evaluation of PDA and related designs is presented in Section 4.3 and conclusions are given in Section 4.4.

4.1 MOTIVATION AND PROPOSED SOLUTION

4.1.1 Motivation

As discussed earlier in Section 3.1.1, recent research work on CMP cache management has recognized the importance of the NUCA shared design. However, a NUCA shared organization suffers from a growing on-chip delay as well as interference misses. An application can evict useful L2 cache content belonging to other co-scheduled programs. Thus, a program that exposes temporal locality can experience high cache misses caused by interferences. (I showed in Section 1.2.3 that 69.5% of misses on a 16-way tiled shared CMP platform are inter-processor).

I primarily correlate the interference misses problem to the *root* of CMP cache management, the cache placement algorithm. Fig. 10 demonstrates the number of misses per 1 million instructions experienced by cache sets across L2 cache banks for two benchmarks,

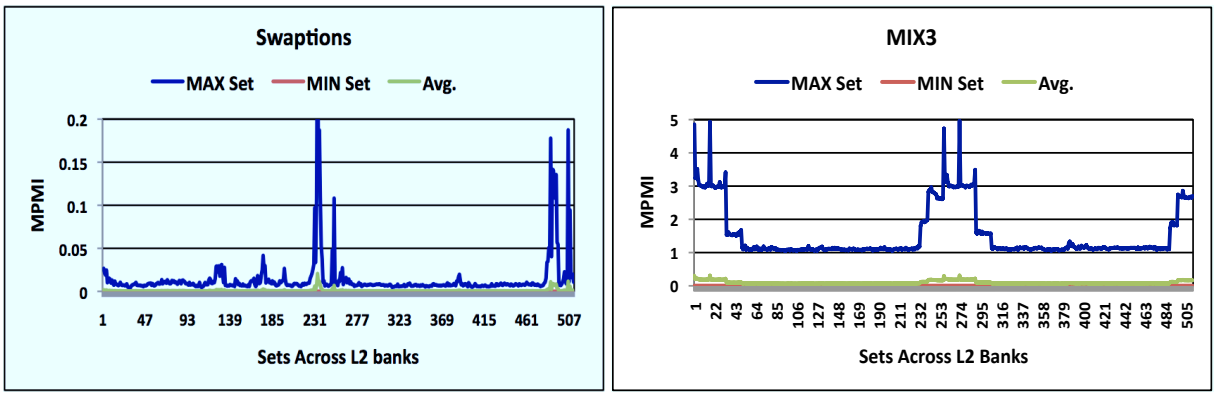


Figure 10: **Number of misses per 1 million instructions (MPMI) experienced by two local cache sets (the ones that experience the max and the min misses) at different sets across L2 banks for two benchmarks, Swaptions and MIX3.**

Swaptions (from the PARSEC suite [8]) and MIX3 (see Section 4.3 for details on these benchmarks). A set across L2 banks is formally $\bigcup_{k=1}^{n} set_{ki}$ where set_{ki} is the *local* set with index i at bank k. Again, I assume a 16-way tiled CMP platform with physically distributed, logically shared L2 banks. I only show results for two local sets (sets on the same L2 bank) that exhibit the maximum and the minimum misses, in addition to the average misses, per each set across L2 cache banks. Clearly, we can see that memory accesses between sets across L2 banks are asymmetric. A placement strategy that is aware of the current pressure at each bank can reduce the workload imbalance among sets across L2 banks by preventing placing an incoming cache block at an exceedingly pressured local set. This can potentially minimize interference misses and maximize system performance.

4.1.2 Proposed Solution

Traditionally, cache blocks are stored at cache locations solely based on their physical addresses. This makes the placement process unaware of the disparity in the hotness of the shared cache sets and the distances from requesting cores. I identify two main requirements for enabling pressure and distance aware placement strategies. First, the physical location of

a cache block has to be decoupled from its address. A block can thereby be placed at any tile independent of its address. This allows flexibility on the placement process as it effectively transforms the cache associativity of the L2 cache to equate the aggregate associativity of the L2 cache banks. For instance, 16 L2 banks with 8-way associativity would offer 128-way set associativity and a requested cache block can be placed at any of these 128-way entries. Second, by having a pressure and distance aware placement algorithm, a location strategy capable of rapidly locating cache blocks would be required.

In this chapter, I explain the importance of incorporating pressure and distance aware placement strategies to improve CMP system performance. I propose pressure and distance aware placement (PDA), a novel mechanism that involves a low-hardware overhead framework to monitor the L2 cache banks at a *group* (comprised of local cache sets) granularity and periodically record pressure information at an array embedded within the memory controller. The collected pressure information is utilized to guide the placement process. Upon fetching a block, B from the main memory, PDA looks up the pressure array at the memory controller, identifies the *underutilized* banks, and places B in an underutilized bank that is *closest* to the requesting core. PDA is said, accordingly, to exploit *distance locality* (i.e., attempting to place blocks in close proximity to requesting cores).

Typically, the pressure at an L2 bank can be measured in terms of cache misses or hits. However, it is not possible to measure cache misses in a meaningful way at L2 banks when a pressure and distance aware placement strategy is employed. Unlike an address-based placement strategy, on an L1 miss to a block B, there is no address that dictates the bank responsible for caching B. Besides, B might map to any bank (versus mapping only to the SHT on the nominal shared). As such, a reported L2 miss can't be correlated to any specific L2 bank but rather to the whole L2 cache space. Hence, I don't use misses to collect pressures at L2 banks but rather hits. More specifically, I quantify a pressure value as the number of *unique* lines that yield cache hits during a time interval, referred to as an *epoch*, and designate that as *spatial pressure*.

4.2 THE PRESSURE AND DISTANCE AWARE PLACEMENT MECHANISM

4.2.1 A Pressure Limit and Manhattan Distance

I define the distance between a requester core, C, and a tile, T, that hosts a cache block requested by C as the Manhattan distance between C and T. Besides, I define a high pressure limit (HPL) to designate the highly and the underutilized pressured L2 banks. A bank that exhibits a pressure that is below HPL is deemed underutilized; otherwise, it is highly pressured. HPL is defined in equation (1). The range of underutilized and highly pressured banks can be expanded or contracted by altering α. The max and min parameters are simply the maximum and minimum pressures on banks.

$$HighPressureLimit(HPL) = max - (\alpha \times (max \text{ - } min)) \qquad \textbf{(1)}$$

4.2.2 Pressure and Distance Aware Placement

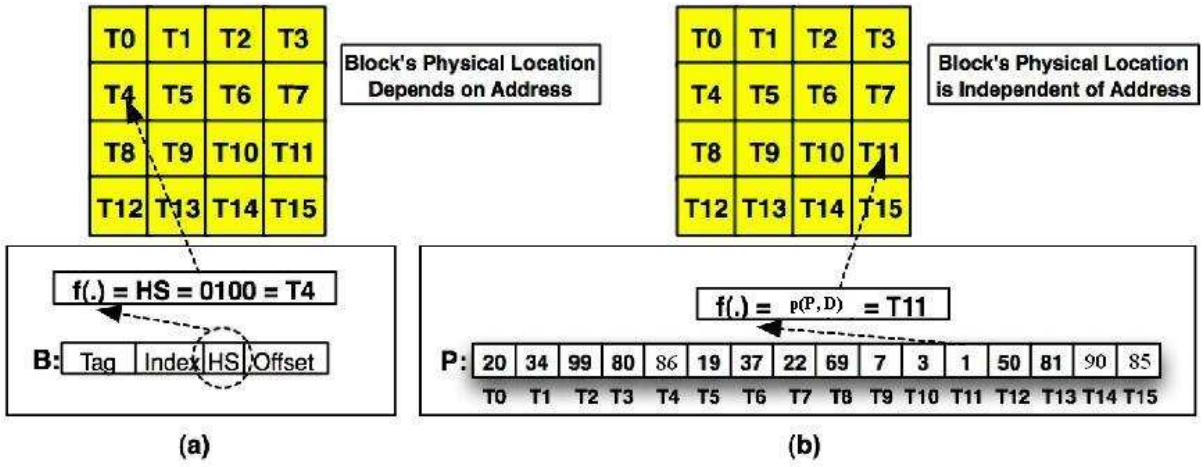


Figure 11: **Address-based versus pressure and distance aware placements. (a) The nominal shared scheme placement strategy. (b) The PDA strategy. (T15 is the requesting core, *f(.)* denotes the placement function, HS is the home select bits of block B, and P is the pressure array)**

I propose a pressure and distance aware placement strategy that maps cache blocks to the L2 cache space depending on the observed pressures at the L2 cache banks (refined shortly to groups of local cache sets) and the distances from requesting cores. The pressure at each L2 bank can be collected at run time, stored, and utilized to guide the placement process. Specifically, a pressure array is maintained at the memory controller(s) of the CMP system. Each slot on the array corresponds to an L2 bank and represents the pressure on that bank. For instance, for 16 banks (assuming a 16-tile CMP) the pressure array would consist of 16 slots. On a miss to L2, the main memory is accessed and the pressure array is probed. I select the bank that is *closest* to the requesting core and is *underutilized* to host the fetched cache block. Fig. 11 demonstrates a descriptive comparison between the placement strategies of the nominal shared NUCA design and my proposed scheme. I assume that T15 is the requesting core and $\alpha = 0.25$. As described earlier in Section 2.1, by using the shared scheme's placement strategy, a subset of bits (the HS bits) from the physical address of a requested block, B, is utilized to map B to its static home tile (SHT). Assuming the HS bits of B are 0100, B is accordingly placed at tile T4. Alternatively, by using PDA, the pressure array at the memory controller is inspected before B is mapped to L2. The closest underutilized tile to the requesting core is tile T11 (the pressure array indicates that T11's pressure is less than HPL), thus selected.

PDA doesn't rely on prior knowledge of the program but on hardware counters. A saturating counter per bank (or group of local sets as will be discussed shortly) can be installed at each tile to count the number of successful accesses to that bank (group) during an epoch. At the end of every epoch the values of the counters are copied from the local tiles to the pressure array at the memory controller(s). Besides, in order to allow PDA to adapt to phase changes of applications, at the copy time I keep only 0.25 of the last epoch's pressure values (by shifting each value 2 bits to the right) and add to them the newly collected ones.

Finally, by having a pressure distance aware placement algorithm, a location strategy capable of rapidly locating cache blocks at the L2 cache space is required. PDA adopts C-AMTE (see Chapter 3) to achieve fast location of L2 cache blocks. In summary, upon placing a cache block, B, at an L2 bank using PDA, C-AMTE stores two corresponding tracking entries, replicated and principal, in special tracking entries (TR) tables at the requesting tile

and B's SHT, respectively. Subsequently, when the requesting core requests B and misses at L1, its TR table is looked up and if a hit is obtained, B is located directly at the L2 bank designated by the matched tracking entry at TR. Furthermore, if any other sharer core requests B, the SHT of B can be always approached and its TR table can be looked up to locate B at its current L2 bank. If no matching entry is found in SHT's TR table, an L2 miss is reported and the request is satisfied from the main memory.

4.2.3 Group-Based Pressure Collection

Collecting pressures at a bank granularity might be relatively imprecise. We can gather more detailed, and thus more accurate, pressures from individual sets or *groups* of sets. A cache bank can be divided into a number of groups. I denote a group size as the number of local sets (sets on the same bank) that a group can include. As such, the upper bound on the number of groups per bank is equal to the number of sets per bank (as a group can't consist of less than one set). The lower bound, conversely, is 1 (as a group can include all the cache sets at an L2 bank). The dimension of the pressure array (rows vs. columns) at the memory controller changes depending on the number of groups per bank (n-group per bank) and the number of banks/tiles (p-bank). With n-group and p-bank the pressure array would consist of n rows and p columns. Therefore, a 1-group (i.e., bank) granularity indicates a linear pressure array and can be probed straightforwardly (as described in the previous subsection). With finer granularities, however, we need to select the row first (denoting the group number of an incoming cache block K) and then the column (denoting the bank that exhibits low pressure for the selected group). The group number (GN) of a block, K, can be simply determined by dividing the index of K by the group size.

Fig. 12 demonstrates PDA using different granularities. For intuitive presentation, I assume a simplified 2-tile (T0 and T1) CMP with two logically shared, physically distributed L2 cache banks and show only the L2 banks referred to by the names of the tiles. Each bank is 2-way associative and has space for 8 cache blocks thus encompassing 4 cache sets. I further assume that T0 is the requesting core and $\alpha = 0.25$. Fig. 12(a) illustrates PDA operating at 1-group granularity. I start with a pressure array of zero values and assume that each

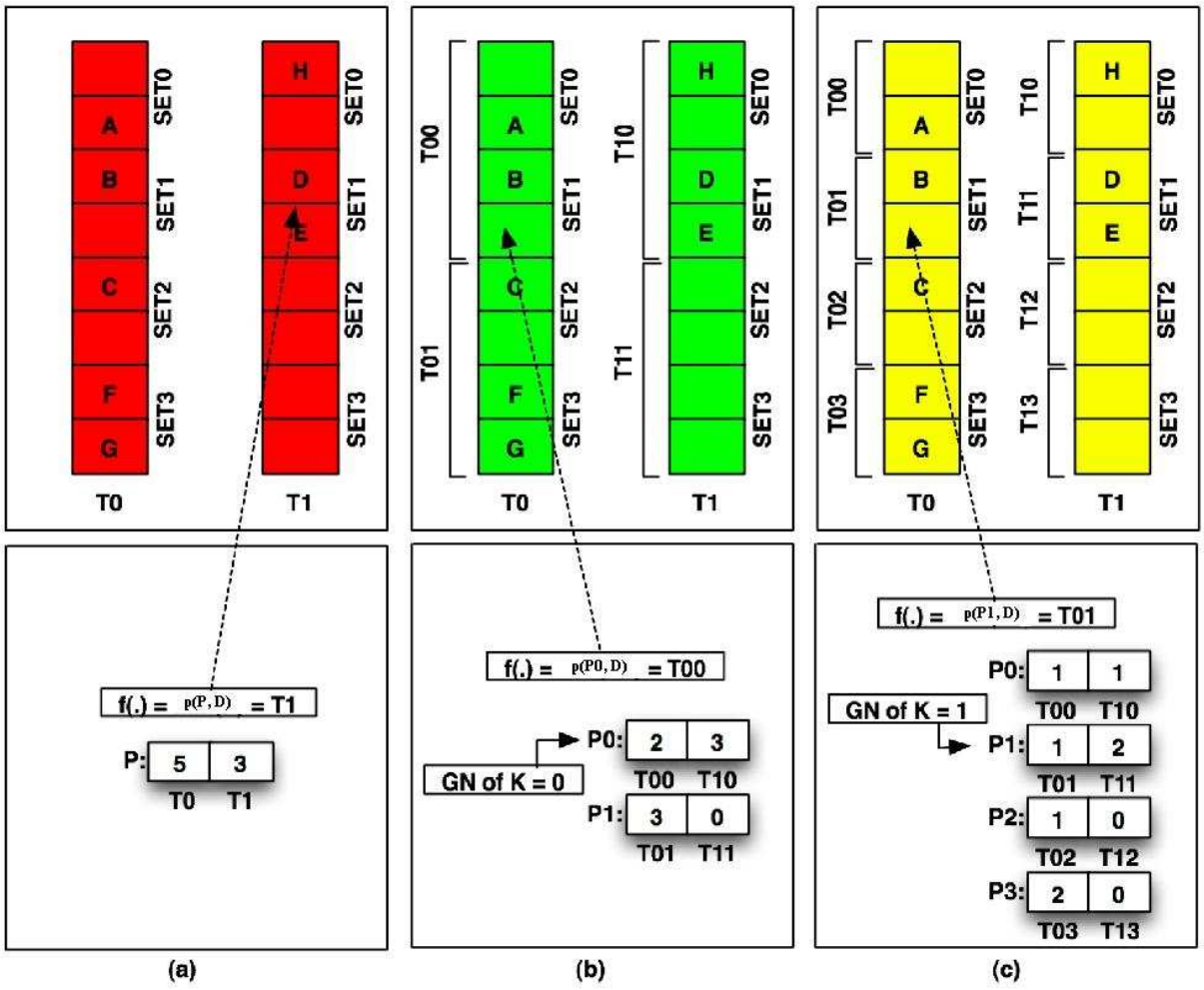


Figure 12: **Placing block K (with index = 1) using PDA with various granularities. (a) 1-group. (b) 2-group. (c) 4-group. (GN is the group number)**

of the blocks on the banks has been successfully accessed for only one time during the last epoch (this describes the numbers displayed in the array). By inspecting the pressure values stored at the array, bank T1 (the closest underutilized bank to T0) is selected to host an incoming block K. Assuming that the index of K is 01, K is mapped subsequently to set1 of bank T1. As a consequence, a conflict miss occurs. Had bank T0 (though exposing higher pressure as indicated by the pressure array) been selected, no conflict miss would have been incurred (because set1 of bank T0 has a free space for an incoming block) and more access latency will be additionally saved (i.e., bank To is local to the requesting core). This explains the rationale behind collecting pressures at finer granularities for the sake of a more precise behavior.

Fig 12(b) demonstrates PDA operating at a 2-group granularity. Given that the index of the incoming block K is 01, GN of K is accordingly 0 (index/group size = 1/2). Hence, row 0 is investigated. Group T00 at bank T0 exhibits a lower pressure than group T10 at bank T1 (though both are underutilized) and is closer to the requesting core at T0. It is, accordingly, selected to host K. Compared to a 1-group operating PDA (illustrated in Fig. 12(a)), no conflict miss is incurred and more latency is expected to be saved on subsequent accesses to K from T0. In Fig. 12(c) I refine the granularity more, specifically to 4-group. GN of K is now 1, and row 1 is therefore explored. Again, group T00 at bank T0 reveals a lower pressure than group T11 at bank T1 and is closer to the requesting core at T0 thus selected. Note that the placement strategy with a 4-group and a 2-group granularities demonstrate a similar behavior for K. This hints to the fact that we might not need hitting the upper bound in refining the group granularity in order to attain the most accurate behavior.

4.3 QUANTITATIVE EVALUATION

The evaluation methodology and the benchmark programs I use in this chapter are the ones described in Section 2.3. Besides, after every 20 million instructions, I keep only 0.25 of the pressure values (see Section 4.2.2). Finally, I compare PDA to the nominal shared (S) baseline architecture, the private scheme (P), and two related proposals; victim caching

(VC) [37], and cooperative caching (CC) [12].

4.3.1 Comparing Against the Shared NUCA Design

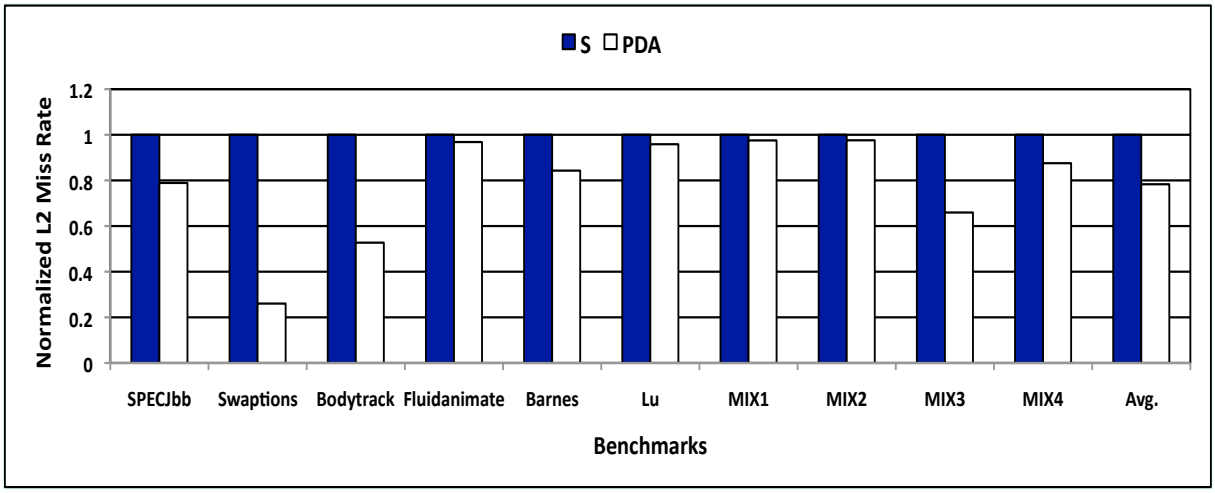


Figure 13: **L2 miss rates of PDA and shared (S) schemes (normalized to S).**

Let me first compare PDA against the baseline shared (S) scheme. I assume a high pressure limit (HPL) with $\alpha = 0.25$. In Section 4.3.3 I offer a sensitivity study on different α values. Besides, I consider a tracking entries (TR) table with 16K entries. The overhead incurred by the TR table is justified in Section 4.3.6. Each access to a TR table requires 1.35ns estimated using CACTI v5.3 [32]. Fig. 13 shows the L2 miss rates of S and PDA normalized to S. As discussed in Section 4.2.3, PDA can run with different granularities (varying from 1-group to 512-group given our employed number of sets per L2 bank). All the results shown in this subsection are for PDA with 32-group granularity. Dividing a bank into only 32 groups (i.e., a counter per each group of 16 sets) provides, in fact, close benefits to dividing it into 512 groups (i.e., a counter per each set). Section 4.3.2 shows results for PDA with all the possible granularities. On average, PDA achieves an L2 miss rate reduction of 21.6% over S, and by as much as 73.9% for the Swaptions program.

In addition to reducing interference misses, PDA seeks to alleviate the growing non-uniform access latencies exposed by S via attempting to place cache blocks fetched from

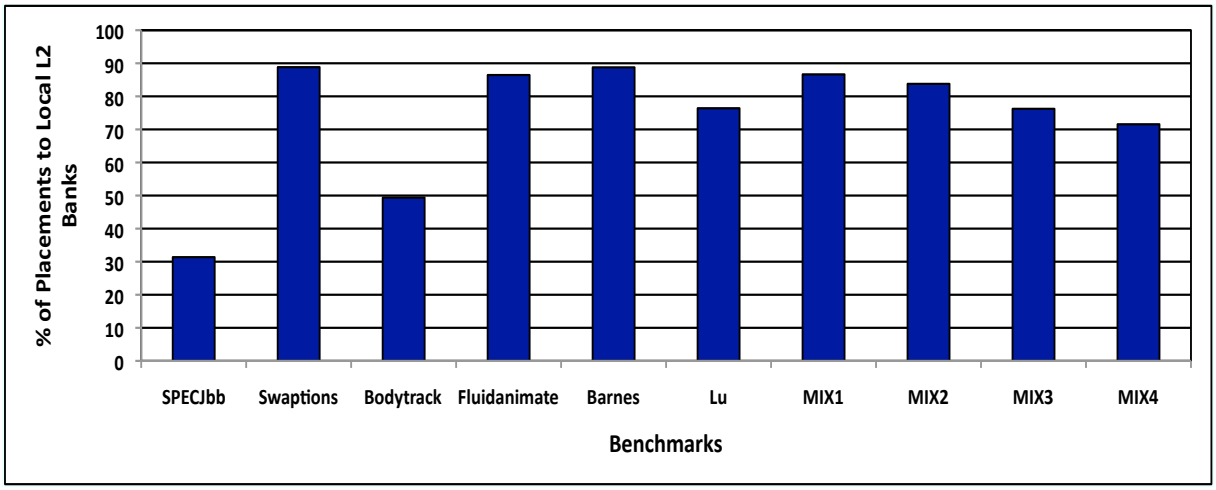


Figure 14: **Percentage of placements to local L2 banks under PDA.**

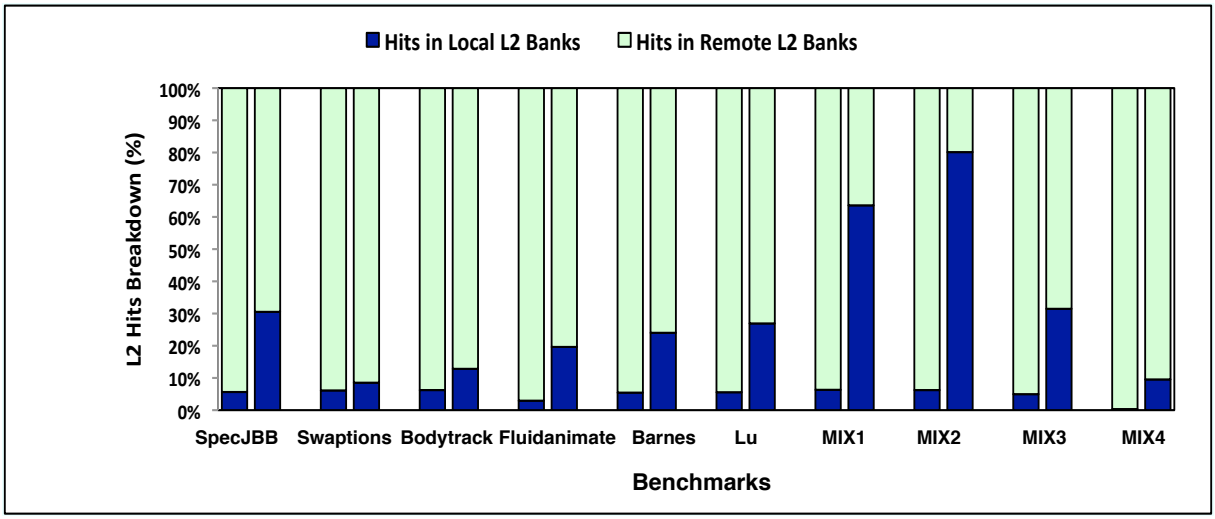


Figure 15: **L2 hits breakdown. Moving from left to right, the 2 bars for each benchmark are for S and PDA schemes, respectively.**

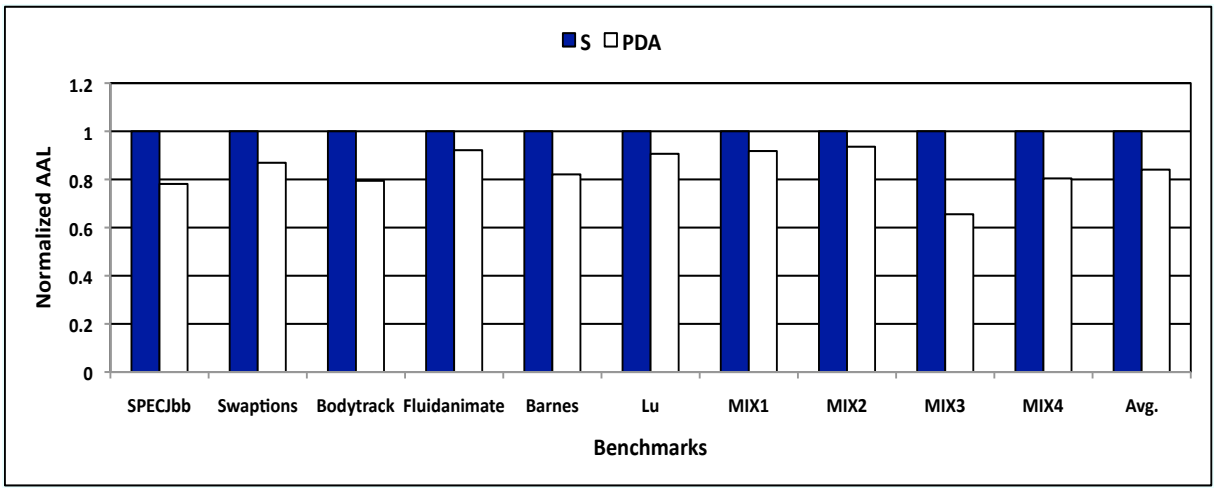


Figure 16: **Average L2 Access Latencies (AALs) of PDA and shared (S) schemes (normalized to S).**

the main memory in underutilized L2 banks that are closest to the requesting cores. Fig. 14 shows the percentage of blocks' placements to local L2 banks accomplished by PDA. Clearly, we can see that for many of the examined benchmarks, the local L2 banks (the closest to the requesting cores) are most of the time underutilized and are, as such, selected to host the incoming cache blocks. I note, however, that placing a large number of blocks at local L2 banks might exacerbate the L2 miss rate especially when the local banks get dwarfed by the quantity of the mapped blocks. Evidently, PDA controls the capacity pressure via involving HPL and, accordingly, precludes such a scenario from occurring. This fact is corroborated in Fig. 13.

To illustrate the contributions of local placements to the reduction in the average L2 access latency (AAL) and, consequently, the system performance, Fig. 15 depicts the L2 hits breakdown. For all the simulated programs, we observe an increase in hits to local L2 banks. Lastly, Fig. 16 demonstrates AAL of S and PDA normalized to S. PDA improves the AAL of S by an average of 15.9%, and by as much as 34.4% for MIX3.

The L2 miss rate and AAL reductions provided by PDA come at the advantage of lower

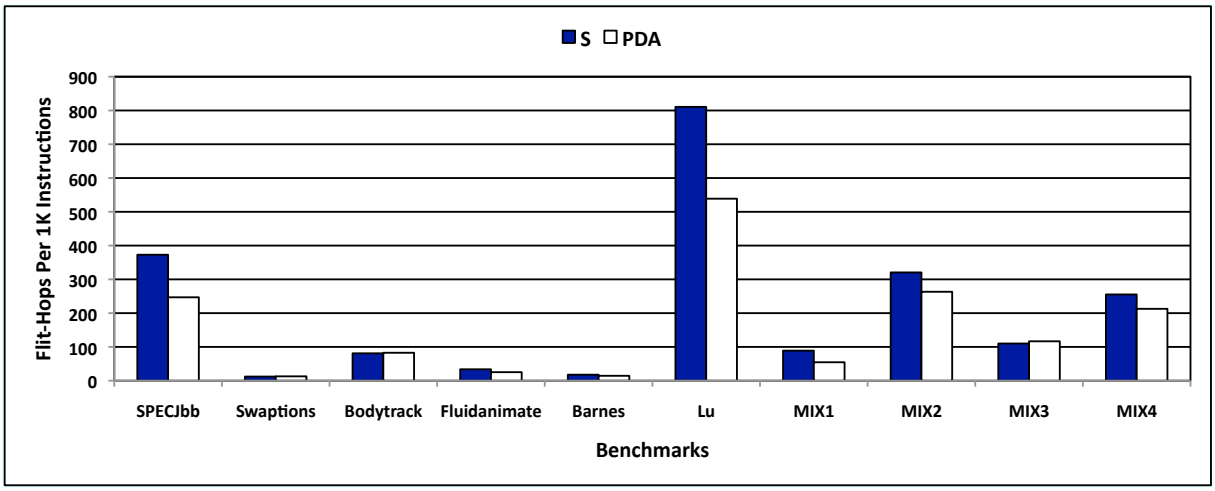


Figure 17: **On-chip network traffic.**

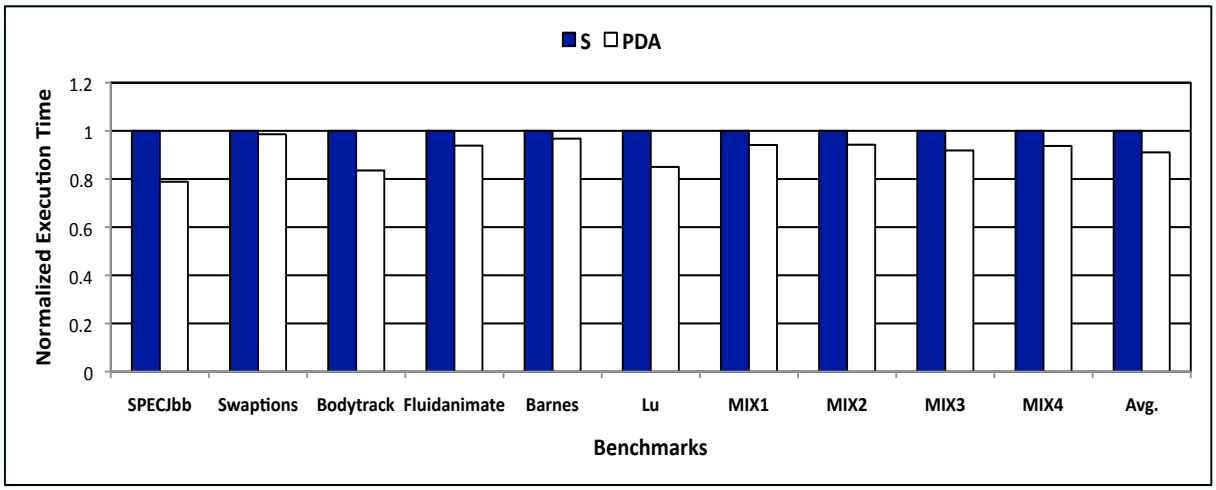


Figure 18: **Execution times of PDA and shared (S) schemes (normalized to S).**

network on-chip (NoC) traffic. Fig. 17 shows the number of flit-hops per 1K instructions experienced by S and PDA. A flit-hop is defined as one flit traveling one hop on a router in the 2D mesh NoC. On average, PDA reduces the NoC traffic of S by 16.9%. For some benchmarks (Swaptions, Bodytrack, and MIX3) PDA, however, degrades the interconnect traffic versus S. This degradation is correlated to the use of the C-AMTE location strategy. C-AMTE introduces more coherence messages (mainly control messages) on the NoC for maintaining consistency among principal and replicated tracking entries. To the contrary, PDA reduces the inter-tile communications via placing cache blocks in close proximity to requesting cores. If the gain from assuaging the inter-tile communications offsets the loss from the incurred C-AMTE interconnect traffic, PDA diminishes the flit-hops number over S, otherwise, PDA aggravates against S. Note that such a little degradation doesn't, in fact, affect AAL. For instance, although MIX3 shows a 5.9% increase in flit-hops per 1K instructions under PDA as compared to S, PDA accomplishes an AAL improvement of 34.4% over S for MIX3. To that end, Fig. 18 presents the execution times of S and PDA normalized to S. Across all benchmarks, PDA outperforms S by an average of 8.9%, and by as much as 21.1% for SPECJbb.

4.3.2 Sensitivity of PDA to Different Group Granularities

I demonstrate PDA's behavior with all possible group granularities. Fig. 19 plots the outcome. For each program I show cycles per instruction (CPI). As explained in Section 4.2.3, collecting pressures at a more refined granularity might make PDA perform better (e.g., MIX1) but not necessarily until striking the upper bound (e.g., Barnes). Besides, I note that many programs show irregularities in performance (e.g., MIX4) as we proceed in refining group granularities. This is due to a skew in pressure values at the array in the memory controller when compared to the actual pressures at cache groups. Actual pressures might deviate (e.g., as a consequence of phase changes or nondeterministic behaviors of programs) some time before the end of an epoch (the time at which I update the array at the memory controller) causing the array to be a little biased in representing actual pressures at cache groups.

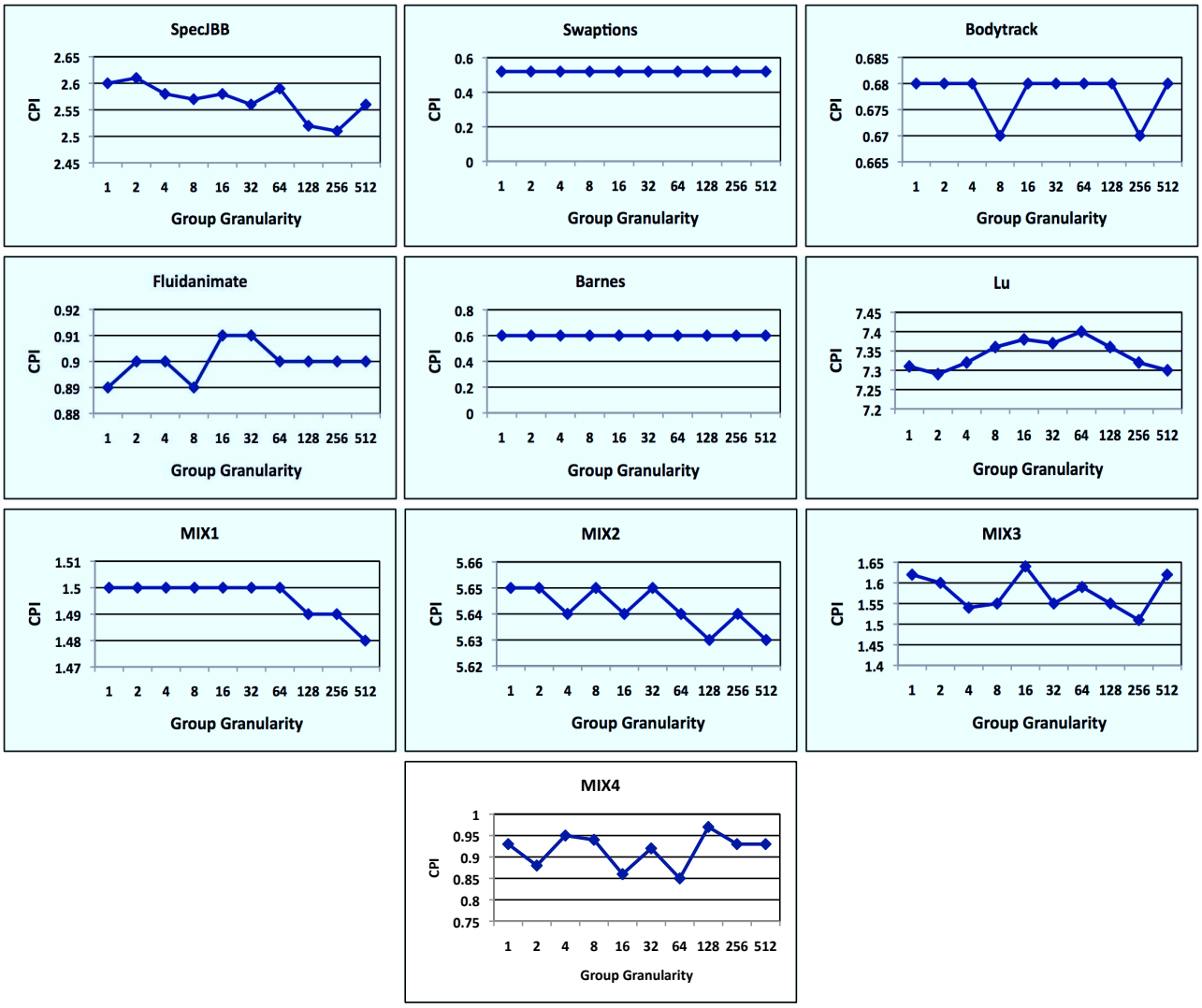


Figure 19: **The PDA behavior with different granularities (varying from 1-group to 512-group).**

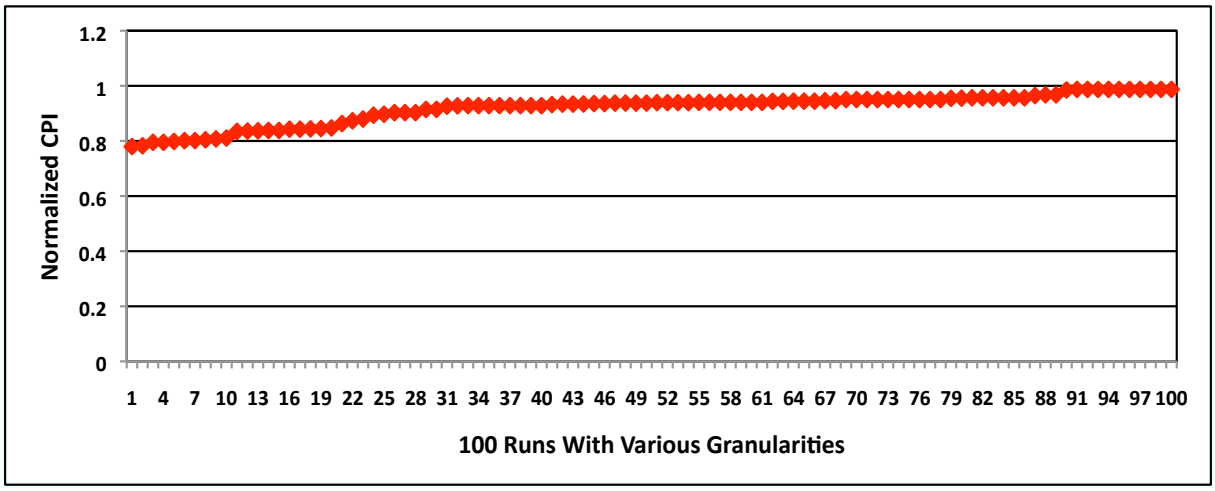


Figure 20: **S-Curve for CPI improvement of PDA over S.**

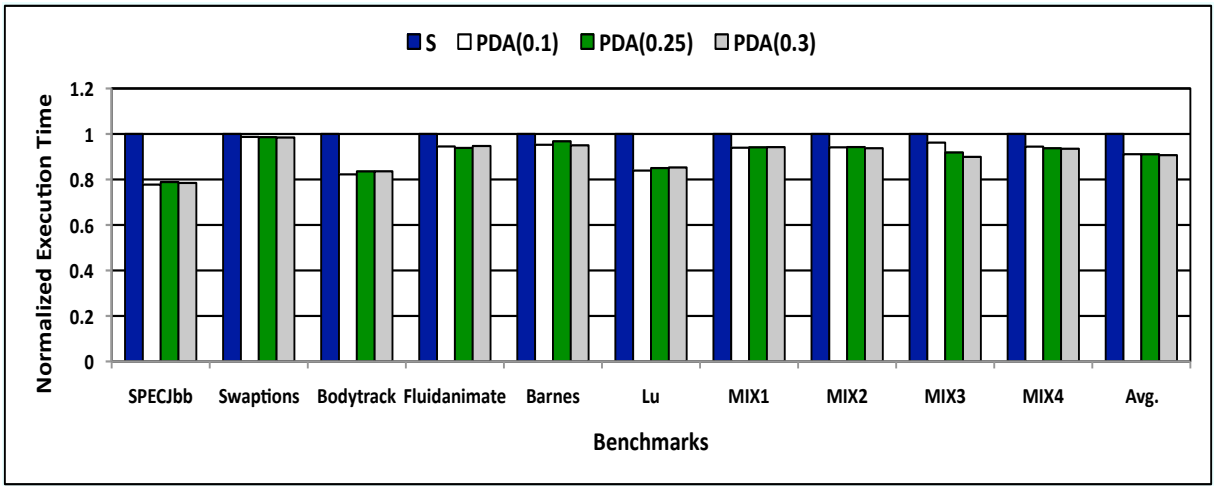


Figure 21: **S-Curve for CPI improvement of PDA over S.**

Finally, we observe that for all the examined programs, PDA always provides a robust performance versus S. That is, none of the programs, running under any group granularity, shows performance degradation against S. Fig. 20 demonstrates the *S-Curve*[1] of the CPI improvement provided by CE for the 100 runs (10 workloads each with 10 group granularities).

4.3.3 Sensitivity of PDA to HPL

So far, I have been using $\alpha = 0.25$ for the high pressure limit, HPL. As Section 4.2.1 describes, by altering α, the range of underutilized and highly pressured banks can be expanded or contracted. I tested PDA with two more α values, particularly 0.1 and 0.3 for HPL. Fig. 21 shows the results. PDA(0.1), PDA(0.25), and PDA(0.3) denote utilizing HPL with α values of 0.1, 0.25, and 0.3, respectively. As demonstrated in the figure, PDA(0.1), PDA(0.25), and PDA(0.3) outperform S by averages of 8.8%, 8.9%, and 9.3%, respectively. PDA(0.3) surpasses PDA(0.1) and PDA(0.25) (by a little margin) as it allows more blocks to be placed in the vicinity of requesting cores without largely affecting the L2 miss rate. We observe that PDA is not very sensitive to the examined α values. As such, 0.25 can be utilized as a default value for α and is important because it is power of 2. With this setting, a multiplier is not needed to compute HPL but a simple shifter.

4.3.4 Accounting for the Overhead of the Location Strategy

Cache performance can be improved not only via efficient cache management but also by increasing cache size and associativity. To justify the overhead incurred by C-AMTE as a location strategy adopted by PDA, I add to each cache set of S two more ways. Given my system parameters, each L2 bank encompasses 512 sets and each cache line is 64 byte. Therefore, each L2 bank is augmented by an additional 64KB cache area. I refer to this configuration as S(2W). Moreover, I examine S's performance by doubling the size of each L2 bank (i.e., from 512KB to 1MB). I refer to the latter configuration as S(D). Fig. 22 shows the L2 miss rates of S, S(2W), S(D), and PDA normalized to S. S(2W), S(D), and PDA

[1] An S-Curve is plotted by sorting the data from lowest to highest. Each point on the graph represents one data-point from this sorted list [50].

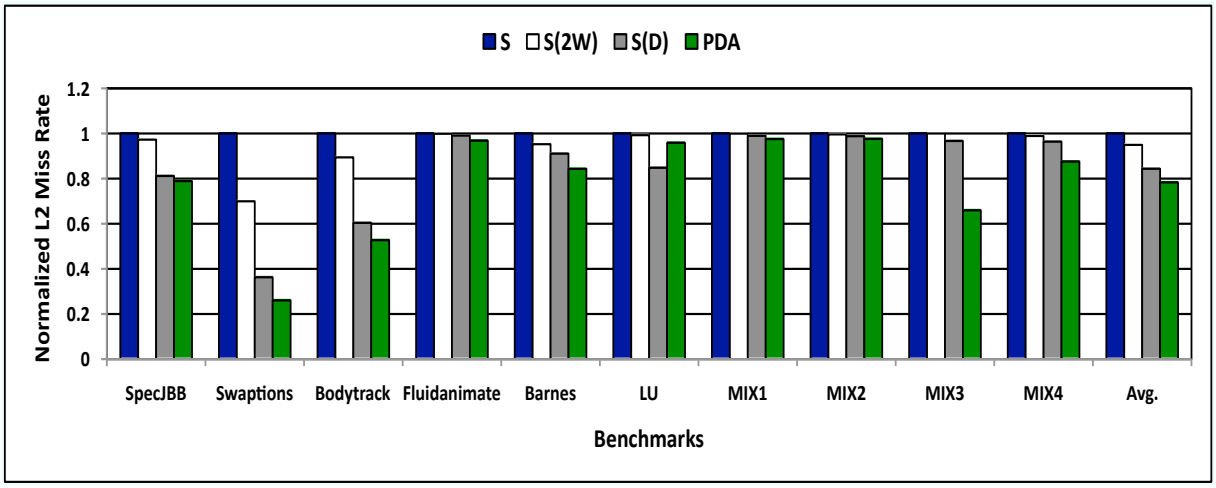


Figure 22: **L2 miss rates of S, S with two more ways added (S(2W)), S with double sized cache (S(D)), and PDA (all normalized to S).**

reduce L2 miss rates over S by averages of 5.1%, 15.6%, and 21.6%, respectively. I conclude that PDA is quite attractive as with small design and storage overhead (i.e., 1.4MB increase in aggregate) it provides more miss rate reduction benefits over S with twice its cache size (i.e., 8MB increase in aggregate).

4.3.5 Scalability

The storage overhead incurred by PDA is mainly from the utilization of C-AMTE as a location strategy to rapidly locate L2 cache blocks after placement. C-AMTE incurs at least one principal and one replicated tracking entries per each cache block, B, when placed at a tile different than its static home tile. On the other hand, C-AMTE incurs at most $N-1$ tracking entries (one principal and the rest are replicated) per B with N being the number of tiles on the CMP platform. The worst case scenario occurs only when B exhibits a sharing degree of N. As discussed earlier in Section 3.2 each principal tracking entry includes: (1) the tag of B (typically 22 bits), (2) a bit vector that acts as a directory to keep the principal and the replicated tracking entries coherent (e.g., 16 bits for a 16-tile CMP model), and

(3) an ID that points to the tile that is currently hosting B (e.g., 4 bits for a 16-tile CMP model). On the other hand, a replicated tracking entry includes only B's tag and the ID to B's current host tile.

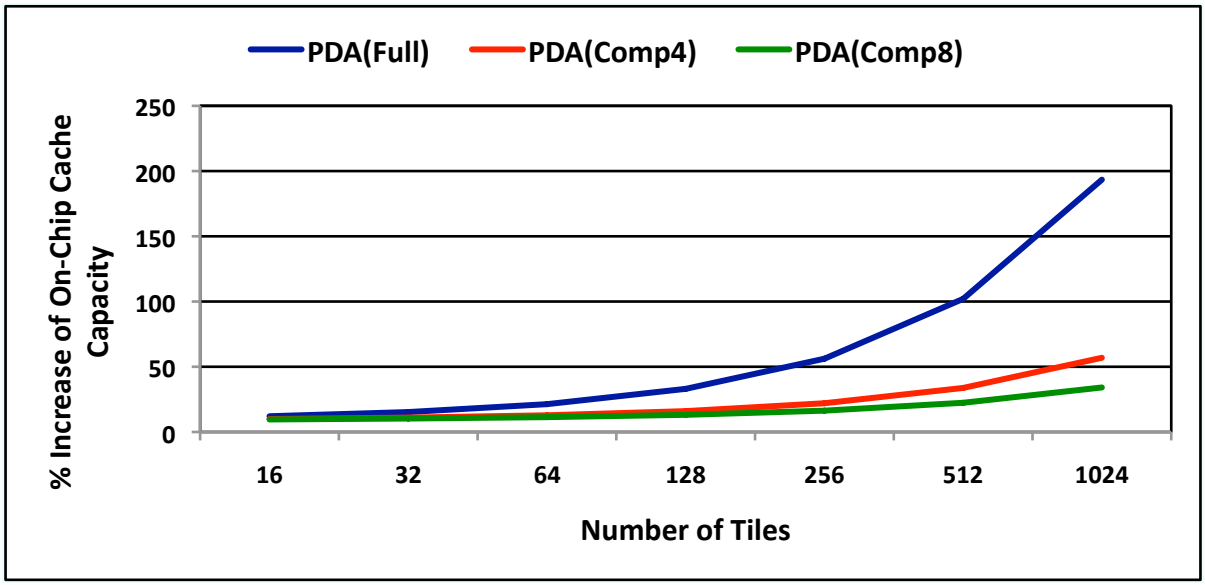


Figure 23: **Storage requirements of PDA with a full-map bit vector (PDA(Full)), a compact vector with 1 bit for every 4 cores (PDA(Comp4)), and a compact vector with 1 bit for every 8 cores (PDA(Comp8)).**

Assuming split tracking entries tables, a principal tracking table (PTR) and a replicated tracking table (RTR) each with 8K entries per tile, PDA demonstrates a 12% increase of on-chip cache capacity (Section 4.3.4 justifies such an incurred overhead). To illustrate how the area overhead of PDA scales, Fig. 23 shows the storage requirements of PDA under 16-tile, 32-tile, 64-tile, 128-tile, 256-tile, 512-tile, and 1024-tile platforms. The figure shows that PDA with full-map bit vector (one bit for every core) per each principal tracking entry (PDA(Full)) scales poorly especially after involving more than 64 cores on a single chip. Clearly, what makes PDA non-scalable to large number of tiles is the bit vector associated with each principal tracking entry. PDA, however, needs not incorporate full-map vectors. Similar to sparse directories [23] and SGI Origin style design [42], PDA can involve more compact (coarse) vectors to improve upon the poor scalability at a moderate bandwidth

increase. For instance, a bit vector can contain one bit for every four cores (PDA(Comp4)), or one bit for every eight cores (PDA(Comp8)) and rely on a broadcast or multicast protocol to track replicated tracking entries. PDA engages distance locality in its placement process and was shown to effectively reduce network on-chip (NoC) traffic and the average L2 access latency (AAL). As such, PDA optimizes NoC bandwidth utilization and can potentially offset the bandwidth increase resulted from the usage of compact vectors. Note, furthermore, that with a scaled number of tiles, PDA is expected to improve system performance more due to the higher exposure of the NUCA problem which PDA attempts to tackle by placing blocks close to requesting cores.

4.3.6 Comparing with Related Designs

In addition to comparing with the nominal shared (S) scheme, I compare PDA against the nominal private (P) design, victim caching (VC) [37], and cooperative caching (CC)) [12]. VC effectively extends the associativity of hot sets in the cache and reduces conflict misses. For fair comparison, I choose the size of an L2 victim cache per tile to approximately match the area increase in PDA (the utilized TR table consists of 16K entries which translates to an 88KB area overhead). Consequently, I set the size and associativity of each victim cache per tile to 64KB and 16-way, respectively. The time to access a victim cache is set to 4.3 ns (or 6 cycles) estimated using CACTI v5.3 [32]. The CC design attempts to reduce L2 (intra-processor) misses. The performance of CC is highly dependent on the cooperation throttling probability [50]. Hence, I evaluate two configurations of CC, one with probability of 100% (CC(100%)) and another with probability of 70% (CC(70%)).

Fig. 24 depicts the execution times of all the compared schemes normalized to S. First, P shows an average performance degradation of 0.6% versus S. Second, when multiple hot sets compete for a victim cache space, the victim cache is flushed quickly and fails subsequently to reduce conflict misses appreciably (e.g., MIX3). VC degrades S by an average of 0.9%. Third, CC spills cache blocks to neighboring L2 banks without knowing if spilling helps or hurts cache performance [50]. As such, CC sometimes degrades performance (e.g., MIX2) while in some other times demonstrates improvement (e.g., SpecJBB). On average,

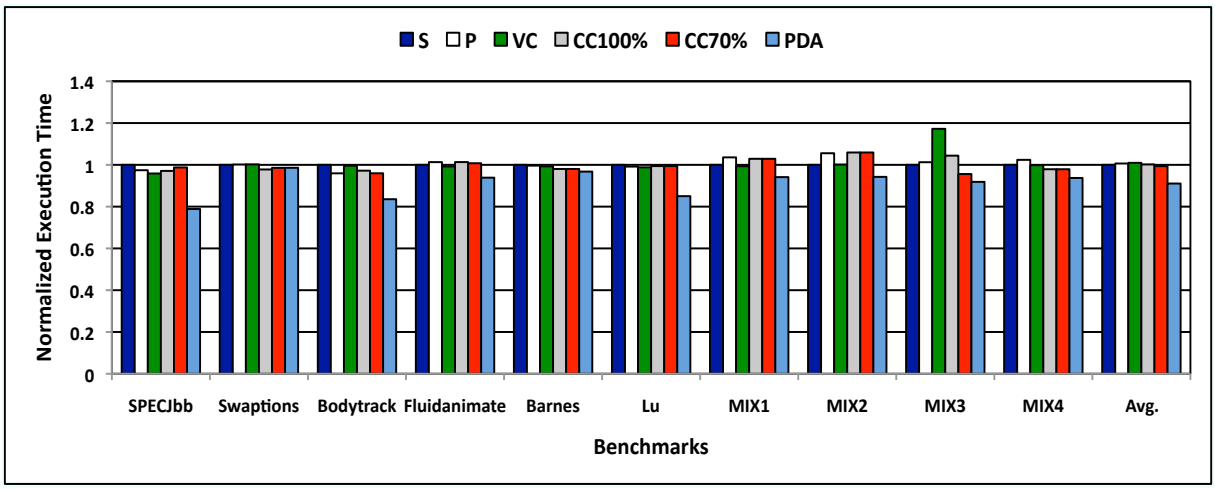


Figure 24: **Execution times of shared (S), private (P), victim caching (VC), cooperative caching 100% (CC(100%)), cooperative caching 70% ((CC(70%)), and PDA schemes (normalized to S).**

CC(100%) reveals a performance degradation versus S by an average 0.1%. On the other hand, CC(70%) outperforms S by an average of only 0.6%. In summary, PDA outperforms S, P, VC, CC(100%), and CC(70%) by averages of 8.9%, 9.5%, 9.6%, 9%, and 8.3%, respectively.

4.4 SUMMARY

This chapter investigates the interference misses and the growing non-uniform access latencies problems inherent in distributed shared CMP caches and proposes pressure and distance aware placement (PDA), a novel strategy that mitigates L2 misses and latencies. I indicate the significance of applying a pressure and distance aware placement strategy on a shared CMP organization to achieve high system performance. Spatial pressure information is collected at a group granularity and recorded in an array at the memory controller. On an incoming cache block, PDA inspects the pressure array, identifies the underutilized groups,

and maps the block to the group that is closest to the requesting core. Clearly, PDA implements CC-FR's data placement component. Simulation results using a full system simulator demonstrate that PDA reduces the cache misses and the average L2 access latency of a shared NUCA design by averages of 21.6% and 15.9%, respectively. Furthermore, results show that PDA outperforms the nominal private design, victim caching [37], and cooperative caching [12] by averages of 9.5%, 9.6%, and 8.3%, respectively.

5.0 FLEXIBLE SET BALANCING

In this chapter I describe Flexible Set Balancing (FSB) that targets CC-FR's data retention category and addresses interference misses, the bandwidth wall problem, and the processor-memory speed gap challenges. In Section 5.1 I provide a motivational study and outline my proposed solution. Section 5.2 details FSB mechanism. A quantitative evaluation of FSB and related designs is presented in Section 5.3 and a conclusion is given in Section 5.4.

5.1 MOTIVATION AND PROPOSED SOLUTION

5.1.1 Motivation

As discussed earlier in Section 1.2.3, more than two thirds of cache lines placed on a last level cache (LLC) logically shared by 16 CMP cores remain unused between placement and eviction. Therefore, these lines don't contribute to good utilization of the silicon estate devoted to the caches. One reason for this phenomenon (referred to as the *zero reuse lines* problem) is that cache lines might be re-referenced at distances greater than the cache associativity [51]. The problem is magnified on CMPs that share caches as on-chip lifetimes of cache lines can become shorter due to the increasing interferences between concurrently running threads/processes. Cache performance can be improved by retaining some fraction of the working set long enough to provide cache hits for future reuses [51, 35].

Computer programs exhibit a non-uniform distribution of memory accesses across different cache sets [55, 53]. Fig. 25 demonstrates this fact by showing the number of misses experienced by cache sets at different physically distributed, logically shared L2 banks on a

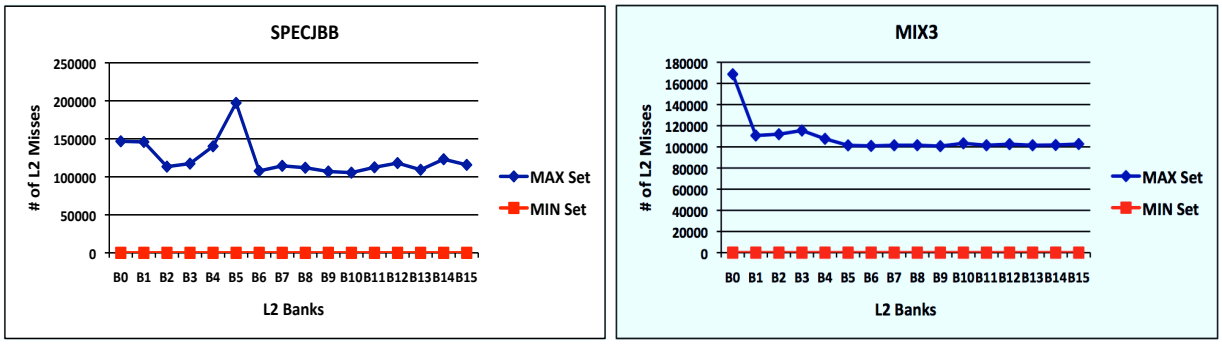


Figure 25: **Number of misses experienced by two cache sets at different L2 banks for two benchmarks, SPECJBB and MIX3 (MAX Set = the set that experiences the maximum misses and MIN Set = the set that experiences the minimum misses).**

16-way tiled CMP for two benchmarks, SpecJBB and MIX3 (experimental parameters and the benchmark programs used are described in Section 5.3) Only the sets that exhibit the maximum and the minimum misses are shown. Clearly, we can see that some sets suffer from large local miss ratios while some others remain underutilized. My work extends the lifetime of cache lines by exploiting this phenomenon via *flexibly* retaining cache lines evicted from highly pressured sets at underutilized sets.

5.1.2 Dynamic Set Balancing Cache and Inherent Shortcomings

A recent study, namely Dynamic Set Balancing Cache (DSBC), suggests extending the lifetime of some cache lines by exploiting the asymmetry in cache sets' usages [55]. Specifically, DSBC proposes shifting lines within the same L2 cache bank from sets with high local miss rates to sets with low local miss rates where they can be found later. Once a set reaches a saturation level (set's miss rate hits a maximum value of $2K - 1$ where K is the associativity of the cache) it requests a free (not associated yet) underutilized set. If such a set is detected, both sets, the highly pressured one (or referred to as *source*) and the underutilized one (or referred to as *destination*), are associated. As long as the two sets are associated, the

source is allowed to retain its lines at the destination but not the reverse (i.e., unidirectional retention).

The association between the source and the destination sets can be simply broken upon the eviction of the last retained line at the destination set (the destination set includes only retained lines from the source set). DSBC maintains a table with one entry per set called the Association Table (AT). AT stores in the i-th entry AT(i).index which corresponds to the index of the set associated with set i. Besides, AT stores a source/destination (s/d) bit (AT(i).s/d) that indicates whether the set is associated or not. Each AT entry can have three different values. First, if a cache set is not associated, its corresponding AT entry stores the set's index and s/d = 0. Second, if a set is a source set, its corresponding AT entry stores the destination index and s/d = 1. Lastly, if a set is a destination set, AT stores the source index and s/d = 0. When a certain request misses at a source set, the destination set is looked up for either a secondary hit or a definitive miss.

DSBC has a number of shortcomings. First, once a destination set, D, is designated, it will continue receiving retained lines from a source set, S, until the association is broken. This overlooks the fact that D's pressure progressively increases while receiving more lines from S. Nevertheless, after association, a new program phase can start where S might remain pressured (and still associated with D) and D becomes highly pressured (due to receiving lines not only from S but further from a new large working set which maps now originally onto it). As a result, S and D compete on only D's resources causing significant thrashing. I illustrate this basic problem by an example.

Consider a 2-way set associative cache shown in Fig. 26. For simplicity I represent a cache by a linear array consisting of only 4 sets. Assume first (Fig. 26(a)) that a working set A with reference pattern $[a0, a1, a2, a3]$ maps to set 3 and has been observed twice by a program. The sequence of references of A can't co-reside in set 3. Accordingly, DSBC selects an underutilized set, say set 0, in the cache and displaces the evicted blocks from set 3 to set 0. Fig. 26(a) shows the final residences of lines in the cache after the completion of the program. A's resultant misses and hits are, consequently, 4 and 4, respectively (the cache is assumed to be initially empty). If the traditional caching strategy is to be followed 4 more misses will be incurred.

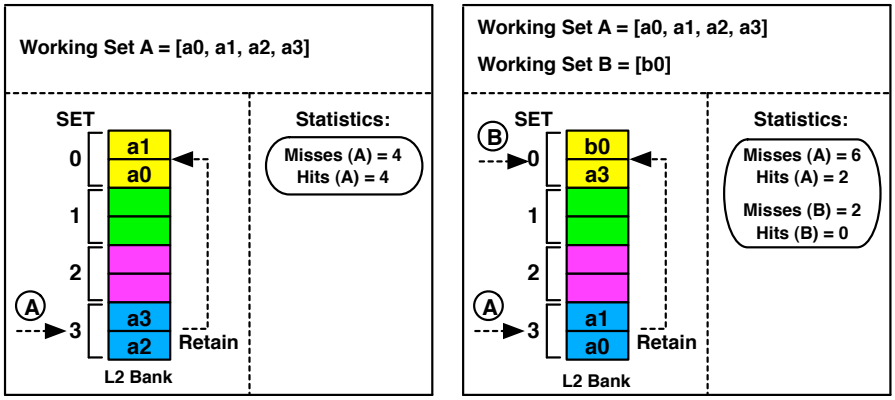


Figure 26: **DSBC in operation. (a) A maps originally to set 3. The program executes A's references in the order of A, A. DSBC is able to save much A's interference misses. (b) A and B map originally to sets 3 and 0, respectively. The program executes A's and B's references in the order of A, B, A, B. DSBC is incapable of adapting to the phase change in the program.**

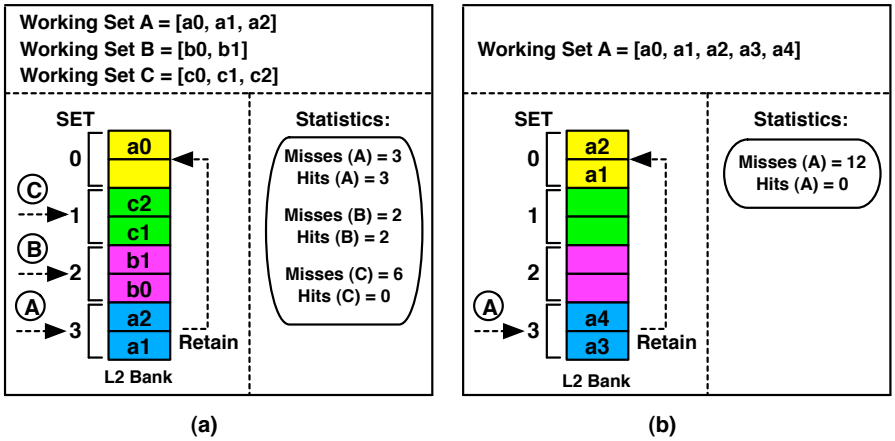


Figure 27: **DSBC in operation. (a) The program executes A's, B's, and C's references in the order of A, B, C, A, B, C. DSBC doesn't allow one-from-many sharing. (b) The program executes A's references twice. DSBC doesn't allow many-from-one sharing.**

In Fig. 26(b), presumably at a different phase in the program, a new working set B with reference pattern $[b0]$ is considered and assumed to map to set 0. As in Fig. 26(a), working set A still maps originally to set 3 and acts as a source set associated with set 0 as a destination set. I assume that the program executes A's and B's references in the order of A, B, A, B. The figure shows the final residences of lines after the completion of the program. A's resultant misses and hits are, consequently, 6 and 2, respectively. B, on the other hand, experienced 2 misses and got no hits. Note that DSBC didn't even attempt to break the association between sets 0 and 3 during the program's execution because there was always at least one retained block at set 0. *If DSBC would rather adapt to the phase change in the program, during the first execution of B's references, the evicted blocks from set 0 (i.e., $a0$) can be retained at another underutilized set (say set 1) so that in the second execution of A's and B's references no misses will be incurred.*

I refer to the sharing policy employed by DSBC among cache sets as *one-from-one* sharing. That is, a destination set is shared by only a single source set. Fig. 27(a) shows three working sets A, B, and C with reference patterns $[a0, a1, a2]$, $[b0, b1]$, and $[c0, c1, c2]$, respectively. I assume that A, B, and C map originally to sets 3, 2, and 1, respectively. The figure demonstrates two issuances of A's, B's, and C's reference patterns in the order of A, B, C, A, B, C. A's lines can't all co-reside in set 3 and DSBC selects set 0 as a destination set for set 3. Also, C's lines can't all co-exist in set 1. However, DSBC doesn't select any destination set for set 1 because no set that is both underutilized and not associated yet is found. As a result, C's references will experience *zero* hits during their two issuances (with this cache topology C is said to experience far-flung reuses). The cache depicts the final residences of all the cache lines after the completion of the program. The misses and hits counts of A are 3 and 3, respectively. On the other hand, B's references miss twice and hit twice. Lastly, C's references miss 6 times and get no hits. *If DSBC would allow set 0 to be shared by both sets, 3 and 1, C' misses will be avoided when issued on the second time. I refer to this kind of flexible sharing as* ***one-from-many*** *sharing. That is, a single destination set can be shared by multiple source sets.*

Finally, as a consequence of the adopted one-from-one sharing strategy, DSBC doesn't allow a source set S to retain blocks in more than one destination set D. As such, if the

working set that maps to S is large enough that both S and D are incapable of providing enough capacity as required, many conflict misses can be incurred. Fig. 27(b) assumes a working set A with reference pattern $[a0, a1, a2, a3, a4]$ that maps to set 3. The program issues A's references twice. DSBC selects an underutilized set; say set 0, where evicted lines from set 3 can be retained. The cache in the figure depicts the final residences of A's lines after the completion of the program. The final misses and hits counts are 12 and 0, respectively (assuming that the cache was initially empty). In this case, DSBC didn't provide any benefit for A. *If DSBC would allow more than one destination set to be shared by set 3; A's misses will be avoided when issued on the second time. I refer to this kind of flexible sharing as* ***many-from-one*** *sharing. That is, many destination sets can be shared by a single source set.*

5.1.3 Proposed Solution

I propose Flexible Set Balancing (FSB), a caching strategy that adapts to phase changes in programs and allows *many-from-many* sharing among cache sets. The difference in this work compared to DSBC are two key insights: (1) retention should be efficiently and dynamically allowed at any point during the program's execution in any direction looking for spare space to effectively minimize interference misses, (2) one-from-many and many-from-one (i.e., many-from-many) sharing should be allowed among cache sets for high flexibility. I demonstrate my solution with an example.

Fig. 28(a) demonstrates the same example shown in Fig. 26(b) but with FSB being incorporated instead of DSBC. Again, A and B are assumed to map to sets 3 and 0, respectively. The program executes A's and B's references in the order of A, B, A, B. In the first issuance of A's references, FSB selects set 0 as a destination set for set 3. Afterwards, when B is issued, line $a0$, which has been already retained at set 0, is evicted again. FSB doesn't discard $a0$ but yet retain it again at a new underutilized set, say set 1. In the second issuance of the working sets, A's and B's references hit on all their cache lines. As such, misses and hits outcomes become 4 and 4 for A, and 1 and 1 for B. Therefore, FSB saves 3 misses as compared to DSBC. The figure shows the final residences of all the cache lines after the

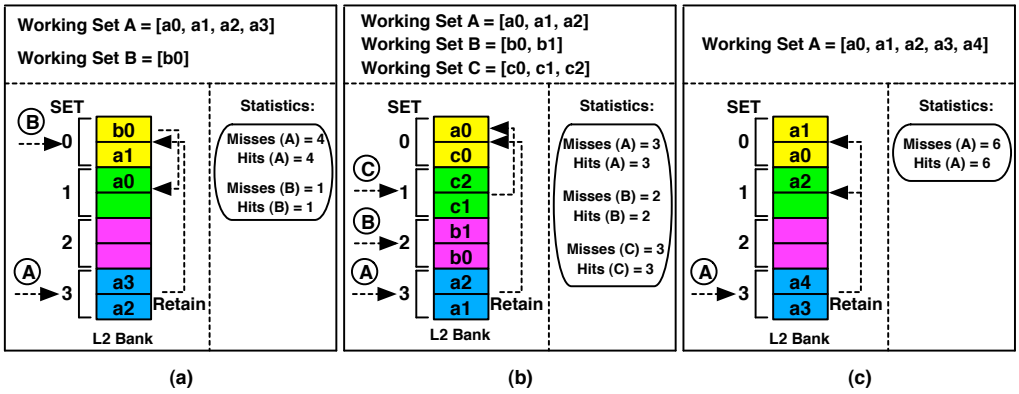


Figure 28: **My solution. (a) The program executes A's, and B's references in the order of A, B, A, B. I adapt to the phase change in the program. (b) The program executes A's, B's, and C's references in the order of A, B, C, A, B, C. I allow one-from-many sharing. (c) The program executes A's references twice. I allow many-from-one sharing.**

program's completion. Clearly, this example illustrates FSB's capability to adapt to phase changes in programs.

Fig. 28(b) shows the same example illustrated in Fig. 27(a). Again, I assume that A, B, and C map to sets 3, 2, and 1, respectively and that the program observes A's, B's, and C's references in the order of A, B, C, A, B, C. FSB allows set 0 to be shared by many source sets. As such, in the first iteration of the working sets when lines of C can't all co-reside in set 1, FSB retains $c0$ at set 0 (the current least pressured set available). In the second iteration, the references of A, B, and C hit on all their cache lines. Misses and hits outcomes become, accordingly, 3 and 3 for A, 2 and 2 for B, and 3 and 3 for C. As compared to DSBC, FSB saves the three misses incurred by C in Fig. 27(a). The cache array in the figure displays the final residences of all the lines after the program's completion. Clearly, this example demonstrates FSB's efficiency in reducing conflict misses by allowing one-from-many sharing among cache sets.

Lastly, Fig. 28(c) illustrates the same example demonstrated in Fig. 27(b). Again, I assume that A maps to set 3 and that the program issues A's sequence of references twice.

FSB allows many sets to be shared by a source set. As such, in the first issuance of A, FSB selects an underutilized set, say set 0, and retains $a1$ and $a0$ at, then selects another underutilized set, say set 1, and retains $a2$ at. In the second issuance, all references of A hit in the cache. FSB, consequently, saves the 6 misses incurred by DSBC in Fig. 27(b). Clearly, this example magnifies the potential of FSB in reducing interference misses by employing many-from-one sharing among cache sets.

5.2 FLEXIBLE SET BALANCING (FSB) MECHANISM

Flexible Set Balancing (FSB) regulates cache allocation by flexibly retaining a fraction of a working set at underutilized cache sets to minimize interference misses and maximize system performance. FSB is extensible and practical in that it can be employed on single-core as well as multi-core architectures. FSB is oriented towards last level caches (in my case L2). FSB requires three main capabilities: (1) deciding upon source and destination sets, (2) retaining working sets of source sets at destination sets in a many-from-many sharing fashion, and (3) locating retained blocks on destination sets upon future reuses. I next describe each capability in turn and close with an analysis on FSB's hardware storage, area, energy, and latency requirements.

5.2.1 Retention Limits

FSB is a pressure-aware strategy where lines evicted from highly pressured sets (source sets) are retained at low pressured sets (destination sets). The pressure at a cache set can be measured in terms of cache misses or hits. In this work I adopt cache misses as a pressure function but provide in Section 5.3.2 a study on a variety of pressure functions. The pressure information can be recorded in an array embedded within the L2 controller of a cache bank. Each cache set corresponds to an entry in the pressure array and the indexes of the cache sets are used to index the array. Each time a miss occurs at a certain set, the array can be updated accordingly (by incrementing the corresponding array slot). In order to allow the

array to accurately represent pressures at sets, after every time interval, I keep only part of the pressure values (e.g., 0.25 of values by shifting each value 2 bits to the right). That permits FSB to adapt to undergoing phase changes in programs. The collected pressures can be utilized to guide the retention process.

Clearly, the set that corresponds to the maximum value in the pressure array is the most highly pressured set. In contrast, the lowest pressured set is the one that corresponds to the minimum value in the array. In this work I define two limits, the low pressure limit (LPL) and the high pressure limit (HPL), to allow a *range* of highly pressured sets to retain their blocks at a range of low pressured sets. A range can encompass one or many sets. When the pressure of a set is below LPL, the set is deemed to be within the limit of the destination sets and can receive lines from *any* source set. In contrast, when the pressure of a set is above HPL, the set is considered to be within the limit of source sets and is permitted, accordingly, to retain its lines at *multiple* destinations sets. Clearly, this allows many-from-many sharing among cache sets. LPL and HPL are defined in equations (1) and (2). The range of source and destination sets can be expanded or contracted by altering α. The max and min parameters are the maximum and minimum pressures on the pressure array.

$$LowPressureLimit(LPL) = min + (\alpha \times (max \text{ - } min)) \quad \textbf{(1)}$$

$$HighPressureLimit(HPL) = max - (\alpha \times (max \text{ - } min)) \quad \textbf{(2)}$$

5.2.2 Retention Policy

FSB maintains a small retention table (RT) per each L2 bank. Each cache set has a corresponding RT entry. As such, the number of entries in RT equals the number of cache sets in the L2 bank. RT can store in the i-th entry *many* RT(i).index values, each pointing to a destination set with a different index. In Section 5.3.1, I empirically show that four RT(i).index pointers are enough to attain an efficient FSB. RT(i).index pointers can be used by FSB to locate retained blocks upon future reuses (more on this shortly).

When an LRU line, L, is evicted from a set i, my retention policy proceeds as follows:

1. The i's corresponding pressure value in the pressure array is looked up, minimum (MIN)

and maximum (MAX) values are generated, and HPL and LPL are calculated.

2. If i's pressure is greater than HPL, i becomes a source set and L is deemed eligible for retention. Otherwise, L is evicted.
3. In parallel, RT(i) entry is looked up. If L is eligible for retention and RT(i) entry has no pointers to destination sets, MIN is checked if less than LPL. If satisfied, L is retained at the cache set corresponding to MIN and an equivalent RT(i).index pointer is created. Otherwise, L is evicted.
4. If RT(i) entry, on the other hand, has pointers (or at least one pointer), these pointers are used to index the pressure array, generate the minimum value out of the indexed values, and compare it against LPL. If satisfied, L is retained at the corresponding cache set and no RT(i).index pointer is created. Otherwise, an invalid RT(i).index is checked if exists.
5. If an invalid RT(i).index is found and MIN satisfies LPL, L is retained at the corresponding set and an equivalent RT(i).index pointer is created. Otherwise, L is discarded.

Note that upon retention, L is inserted as the most recently used (MRU) line in the selected destination set. The LRU line evicted at the destination set, to make room for L, is discarded simply because the destination set doesn't satisfy HPL. As such, FSB avoids ripple effects.

The LRU evicted line, L, at the source set can be either *native* or *retained*. If L is native, FSB simply proceeds with the retention process. Otherwise, L is checked if *active*. L is defined to be active if at least one core on the CMP platform had cached a copy of L (in its L1). This can be easily determined from L's associated directory bit vector. I assume that an active L is currently in use by the caching core(s) and, accordingly, attempt to retain it again. If L is retained and not active, I assume that it has been kept long enough in the cache without providing a cache hit, and, as such, avoid retaining it over again (although eligible for retention).

The pressure array is updated not only at a miss/hit but further when retaining a line at a destination set. When a destination set receives a retained line, its corresponding pressure value is incremented. This is critical so as to reflect the progressive increasing pressure on a destination set each time it receives a retained line. This makes FSB very flexible and

attentive as it allows selecting a different destination set once the pressure of the current destination set surpasses LPL.

Retaining cache lines at destination sets requires extending lines' tags. This is due to the fact that a cache line must have a one-to-one correspondence with a unique address. For instance, assume a line E is retained at a destination set S and that S has a line F which has an identical tag field as E. E and F addresses are, in fact, only distinct because they differ in their index fields. Now E and F co-reside at S and thus become indistinguishable. Nevertheless, this suggests a simple solution. That is, augmenting each line's tag with the index field. Section 5.2.4 describes FSB's storage, area, energy, and latency requirements.

Finally, upon discarding a retained line, R, from a destination set, D, R's augmented index j is matched with the augmented indexes of D's resident lines. A "no match" outcome means that R is the last retained line at D from the source set j. Consequently, I index RT(j) entry and invalidate the RT(j).index pointer that points to D. To that end, I note that the retention process is activated in parallel with the resolution of a definitive miss (which usually takes hundreds of cycles to fetch the requested line from the main memory).

5.2.3 Lookup Policy

Upon a request to a cache line, L, the cache starts always looking up the set i that L's index designates. RT(i) entry is also looked up concurrently. If a hit occurs at set i, the request is satisfied and the pressure array is updated (only if the pressure function involves hits). If, on the other hand, a miss occurs at set i, the cache sets identified by the RT(i).index pointers (if any) are *serially* looked up until either a secondary hit is acquired or a definitive miss is proclaimed. Set lookups are serialized to keep FSB simple, avoid port contention, and reduce power dissipation[1]. Section 5.3.1 demonstrates that such a serial policy doesn't hurt performance because the gain from hits on retained lines exceedingly offsets the loss from sequential lookups. Upon a secondary hit, the request is satisfied and the pressure array is updated (only if the pressure function involves hits). If a definitive miss is asserted, the

[1]Prior research has made use of serialization to increase flexibility and improve performance in large caches [15, 24]. Existing processors have also adopted serialization for looking up tag and data arrays seeking to reduce power dissipation [21, 69].

Component	Bits Per Entry	K Entries	K Bytes Per Tile
RT Entry	9	2	2.3
Augmented Bits Per an L2 Line	9	8	9.2
Total KBytes			11.5
% Increase of On-Chip Cache Capacity			1.9%

Table 8: **FSB storage overhead.**

pressure array is updated at slot i (if the pressure function involves misses), the retention policy is triggered and, in parallel, the requested cache line is fetched from the main memory and inserted in set i.

FSB doesn't swap retained lines upon hits to return them to their original sets for many reasons. First, this simplifies management. Second, FSB is oriented towards last level caches. As such, once a hit is obtained on a retained line, the line is moved to the upper level of the memory hierarchy where successive accesses can find it. Third, swapping is undesirable because it requires four accesses to the tag-store, consumes energy, and increases port contention [53].

5.2.4 FSB Cost

FSB comes at a little storage, area, latency, and energy overheads. In this work I assume a 32 KB 2-way associative I/D L1 caches and a 512KB 16-way associative L2 bank (i.e., encompassing 512 cache sets) per each CMP tile. Section 5.3.1 shows that 4 pointers per each RT entry are enough for an effective FSB. Each RT pointer requires 9 bits. Table 8 shows that less than 2% storage overhead is required by FSB.

To model area and energy I use CACTI v5.3 [32]. I assume a 45nm technology. Table 9 demonstrates the area and energy per access required for both a baseline L2 bank and an L2 bank with FSB being incorporated. The TR table, in addition, requires 0.14 mm^2 and 0,015 nJ area and energy per access, respectively. Note that the energy savings due to reducing

Technology	Baseline Energy	FSB Energy	Baseline Area	FSB Area
45nm	1.23nJ	1.26nJ	5.36mm^2	5.47mm^2

Table 9: **Baseline and FSB required energy and area in a 512KB/16-way/64B/LRU L2 bank.**

off-chip accesses is not considered. Such savings are expected, in fact, to counterbalance my calculated energy overhead and further provide advantages as chip crossings are one of the greediest energy consumers [33]. Finally, and due to augmenting lines' tags by indexes, FSB incurs a negligible increase in latency (only 0.02 ns) per each L2 bank access.

5.2.5 Scalability

As FSB alleviates the workload imbalance across cache sets within each L2 cache bank irrespective of the number of tiles, it becomes very scalable to large-scale CMP platforms. The discussed cost in Section 5.2.4 pertains to each L2 bank and is completely independent of the number of banks. Furthermore, the performance is expected to scale successfully with larger number of tiles as that might lead to a higher exposure of the interference misses problem which FSB attempts to tackle.

5.3 QUANTITATIVE EVALUATION

The evaluation methodology and the benchmark programs I use in this chapter are the ones described in Section 2.3. Besides, after every 20 million instructions, I keep only 0.25 of the pressure values (see Section 5.2.1).

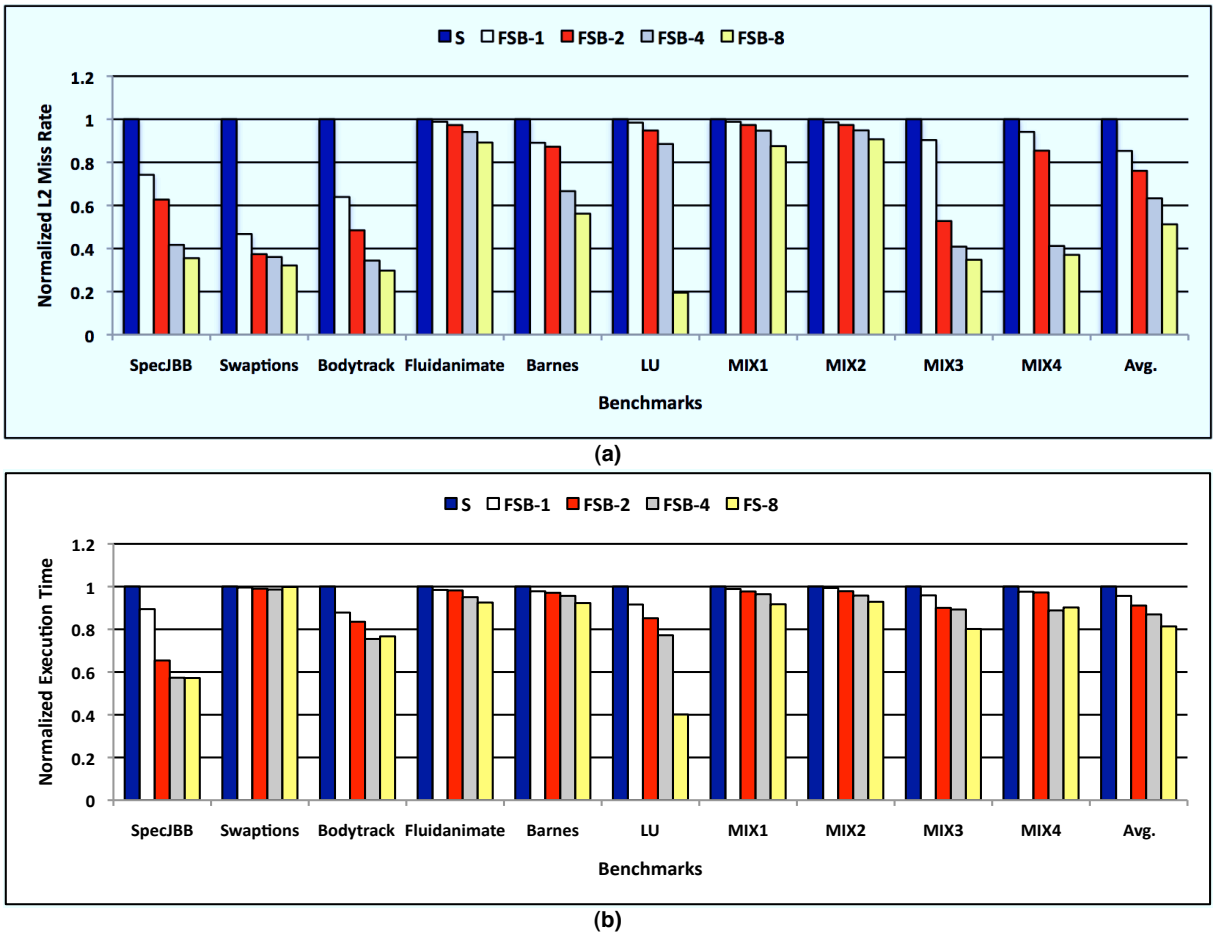


Figure 29: **L2 miss rates and execution times of the baseline shared scheme (S), FSB-1, FSB-2, FSB-4, and FSB-8 (all normalized to S).**

5.3.1 Comparing FSB against Shared Baseline

Let me first compare FSB against the baseline shared (S) scheme. Fig. 29 (a) shows the L2 miss rates of S and four FSB configurations normalized to S. I denote FSB with retention tables (RT) storing 1, 2, 4, and 8 RT(i).index pointers per each entry i as FSB-1, FSB-2, FSB-4, and FSB-8, respectively. Furthermore, I assume a low pressure limit (LPL) and a high pressure limit (HPL) each with $\alpha = 0.2$. Section 5.3.3 offers a sensitivity study on different α values. I adopt cache misses as a pressure function but Section 5.3.2 provides a study on a variety of other functions. The figure demonstrates that as the number of pointers per an RT entry increases, FSB achieves higher L2 miss rate reductions. This behavior is apparent on all the examined benchmark programs. FSB centers around the flexible many-from-many sharing policy. More pointers indicate more exploitation to the many-from-many sharing strategy and, consequently, more alleviation to the imbalance across sets. On average, FSB-1, FSB-2, FSB-4, and FSB-8 accomplish average miss rate reductions of 14.6%, 23.9%, 36.6%, and 48.7%, respectively.

FSB strategy adopts a serial lookup policy (see Section 5.2.3 for more details). Upon a miss on the original set i, RT(i).index pointers (if any) are utilized to serially index and lookup corresponding L2 cache sets. Only the tag-stores are looked up until either a secondary hit is obtained or a definitive miss is asserted. Each tag-store access takes less than 0.68 ns, estimated by CACTI v5.3 [32] assuming a 45nm technology. This incurs a higher latency per each L2 access that misses at the original set. As such, although more RT(i).index pointers result in more L2 miss rate reductions, a latency cost is to be paid. Fig. 29 presents the execution times of S, FSB-1, FSB-2, FSB-4, and FSB-8 normalized to S. A main observation is that as I proceed through FSB configurations (FSB-1 to FSB-8), the performance of each application monotonically improves until FSB-8 is reached. Under FSB-8 the case changes and programs are split into three categories: (1) no benefit is accomplished (e.g., SpecJBB), (2) a benefit is achieved (e.g., Fluidanimate, Barnes, Lu, MIX1, MIX2, and MIX3), and (3) a degradation is observed versus FSB-4 (e.g., Swaptions, Bodytrack, and MIX4).

Two factors define the eligibility of applications for accomplishing higher or lower per-

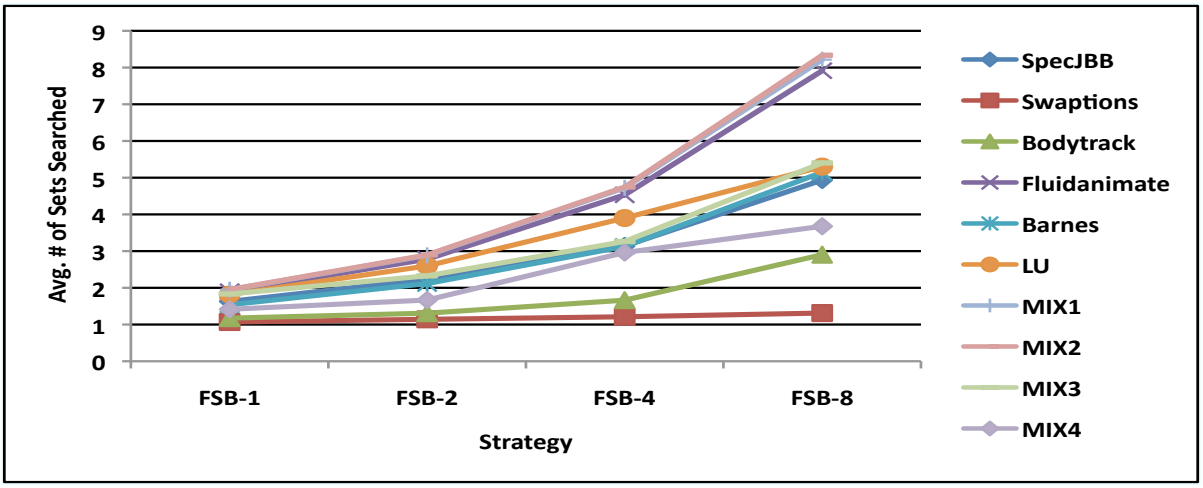


Figure 30: **The average number of L2 cache sets searched under FSB-1, FSB-2, FSB-4, and FSB-8.**

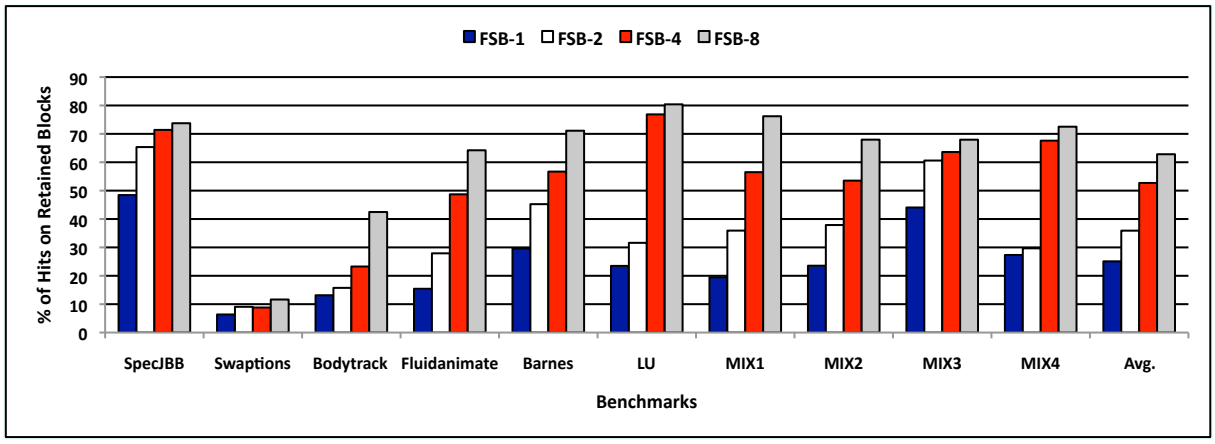


Figure 31: **The percentage of hits on retained cache lines under FSB-1, FSB-2, FSB-4, and FSB-8.**

formance when switching between FSB's configurations: (1) the gain, G, from miss rate reduction and (2) the loss, L, from increased access latency. Let $\triangle_i$ be defined as $G - L$ for FSB-i. When $\triangle_8$ exceeds $\triangle_4$, the performance of the application improves by switching from FSB-4 to FSB-8, otherwise, it degrades. Swaptions, Bodytrack, and MIX4 achieve miss rate reductions of 6.1%, 7%, and 7%, respectively after increasing RT pointers from 4 to 8. In fact, under FSB-8, these three applications reduce the L2 miss rates the least as compared to the other examined programs (see Fig. 29 (a)). Clearly, $\triangle_4$ of each of Swaptions, Bodytrack, and MIX4 overpasses $\triangle_8$, thus they degrade under FSB-8 in comparison to FSB-4. FSB-1, FSB-2, FSB-4, and FSB-8 outperform S by averages of 4.3%, 8.8%, 13%, and 18.6%, respectively. Although FSB-8, on average, surpasses the remaining FSB's configurations, I consider FSB-4 more desirable for two main reasons. First, FSB-4 doesn't observe any degradation in performance for any application when compared against the preceding configurations. Second, FSB-4 offers a better tradeoff between hardware complexity, power dissipation, and performance.

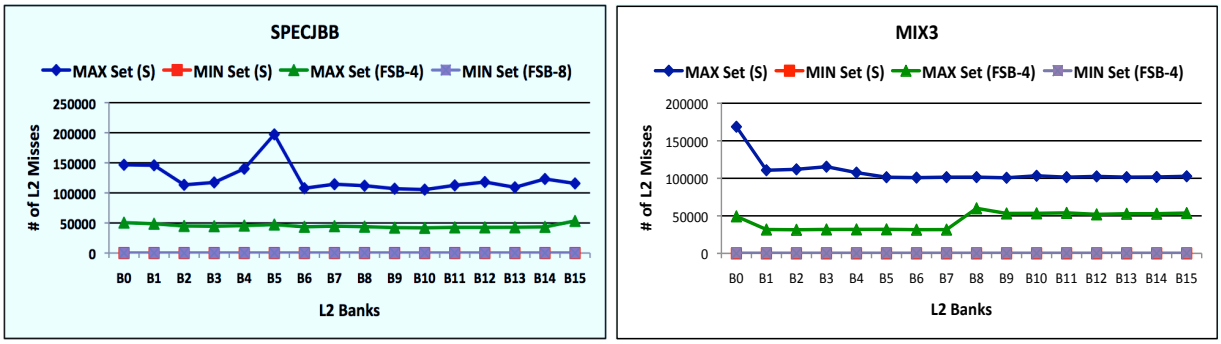


Figure 32: **The number of L2 misses experienced by cache sets at different L2 banks for SpecJBB and MIX3 programs under the baseline shared scheme (S) and FSB-4. Only the sets that exhibit the maximum (MAX Set) and the minimum (Min Set) misses are shown.**

To that end, Fig. 30 depicts the average number of L2 cache sets searched for all the applications under FSB-1, FSB-2, FSB-4, and FSB-8. Furthermore, Fig. 31 displays the percentage of hits on retained cache lines for each program. In fact, the latter figure explores FSB's efficiency in satisfying far-flung reuses after retaining some fraction of the working

set at underutilized sets. *With FSB-4, more than half of the hits are satisfied by retained lines.* On average, the percentage of hits on retained lines provided by FSB-1, FSB-2, FSB-4, and FSB-8 are 25%, 35.8%, 52.6%, and 62.8%, respectively. Finally, Fig. 32 explores FSB's effectiveness in mitigating non-uniformity across sets by showing the number of misses experienced by cache sets at different L2 banks for two benchmarks, a multithreading one (i.e., SpecJBB) and a multiprogramming one (i.e., MIX3). I present only the sets that exhibit the maximum and the minimum misses for the baseline shared, S, and FSB-4.

5.3.2 Sensitivity to Different Pressure Functions

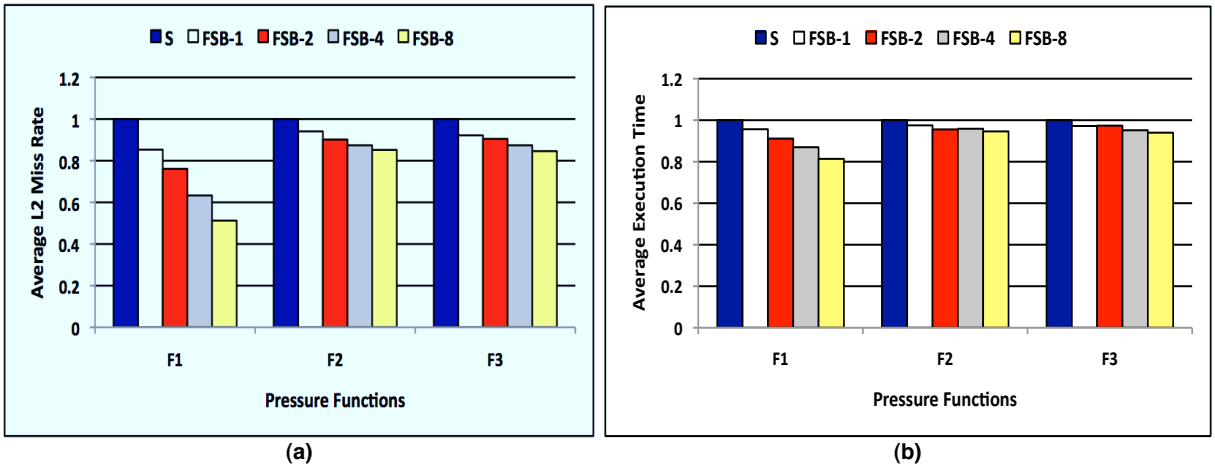


Figure 33: **Average L2 miss rates and execution times of all the benchmark programs under the baseline shared scheme (S), FSB-1, FSB-2, FSB-4, and FSB-8 (all normalized to S) (F1, F2, and F3 are pressure functions that involve misses, hits, and spatial hits, respectively).**

In the previous section I utilized cache misses as a pressure function. I tested other functions that can be used to measure pressures at cache sets. Fig. 33 plots the results for only three functions F1, F2, and F3 which denote functions with *misses* only, *hits* only, and *spatial hits*, respectively. I assume a low pressure limit (LPL) and a high pressure limit (HPL) each with $\alpha = 0.2$. The spatial hits function simply updates the pressure array with different values upon hits depending on lines' frames. That is, upon a hit on a line, L_{mru},

which exists at the MRU position, the function increments the bucket that corresponds to L_{mru}'s set by 1. However, upon a hit on a line, L_{mru} -1, next to L_{mru}, the function increments the corresponding bucket by 2, and so on. The idea stems from the fact that a single highly contended line (say a lock) could result in a very high hit count at a particular set when, in fact, the pressure of lines competing for that set is very low. As depicted in Fig. 33 (b), on average, F2 produces performance improvements of 2.4%, 4.4%, 4.1%, and 5.4% for FSB-1, FSB-2, FSB-4, and FSB-8 over the baseline shared (S) scheme, respectively. F3, on the other hand, offers average performance improvements of 2.7%, 2.6%, 4.8%, and 6% for FSB-1, FSB-2, FSB-4, and FSB-8 over S, respectively. Lastly, F1 surpasses both, F2 and F3, and provides average performance improvements of 4.3%, 8.8%, 13%, and 18.6% for FSB-1, FSB-2, FSB-4, and FSB-8 versus S, respectively. For the examined benchmarks, I conclude that cache misses is preferable among the tested functions to represent pressures at cache sets. More comprehensive functions can be considered in a future work.

5.3.3 Sensitivity to LPL and HPL

So far, I have been using $\alpha = 0.2$ for the low and the high pressure limits, LPL and HPL. As Section 5.2.1 describes, by altering α, the range of source and destination sets can be expanded or contracted. I tested FSB-1, FSB-2, FSB-4, and FSB-8 with two more α values, particularly 0.1 and 0.3 for both LPL and HPL. Fig. 34 shows the results. RL1, RL2, and RL3 denote the retention limits (i.e., LPL and HPL) with α values of 0.1, 0.2, and 0.3, respectively. As demonstrated in Fig. 34(a), on average, RL1 provides L2 miss rate reductions of 14.4%, 21.3%, 35%, and 48.3% for FSB-1, FSB-2, FSB-4, and FSB-8 against the baseline shared (S) scheme, respectively. RL2, on the other hand, offers a little more enhancement and produces 14.6%, 23.9%, 39.4%, and 48.7% L2 miss rate reductions for FSB-1, FSB-2, FSB-4, and FSB-8 versus S, respectively. Finally, RL3 achieves 15.2%, 24.3%, 36.2%, and 49.8% miss rate reductions for FSB-1, FSB-2, FSB-4, and FSB-8 over S, respectively. Fig. 34 (b) depicts the performance outcome. For the simulated benchmarks, I conclude that FSB shows low sensitivity to the examined α values .

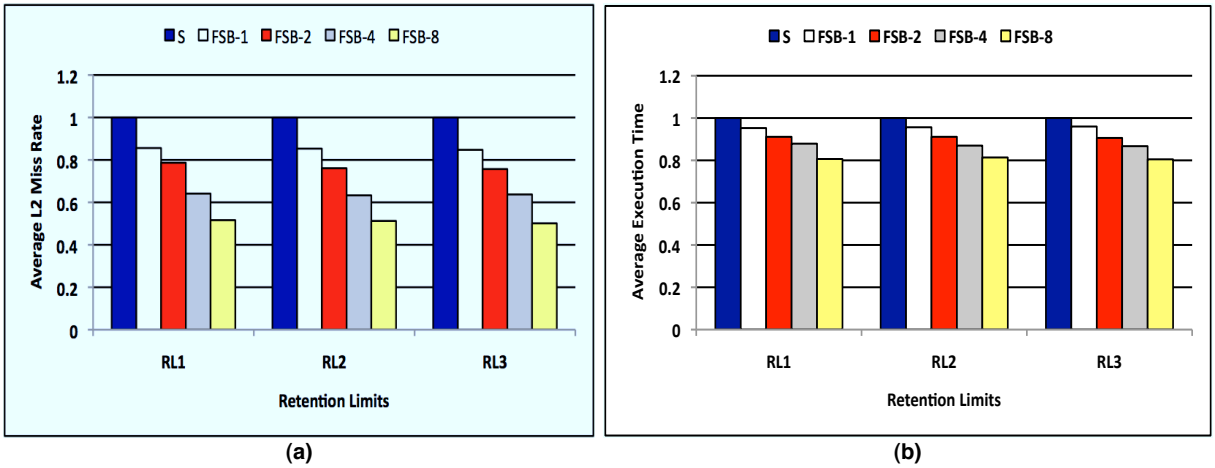


Figure 34: **Average L2 miss rates and execution times of all the benchmark programs under the baseline shared scheme (S), FSB-1, FSB-2, FSB-4, and FSB-8 (all normalized to S) (RL1, RL2, and RL3 are the Retention Limits- HPL and LPL- with $\alpha = 0.1$, $\alpha = 0.2$, and $\alpha = 0.3$, respectively).**

5.3.4 Impact of Increasing Cache Size and Associativity

We can improve cache performance not only by efficient cache management but also via increasing cache size and associativity. In this section I consider only FSB-4 (see Section 5.3.1 for a discussion on FSB's configurations). FSB-4 requires 11.5KB storage overhead per tile (see Table 8). To justify FSB-4's incurred overhead, I optimistically augment each cache set of the baseline shared scheme, S, with two more ways. In total, this adds to each L2 bank a 64KB more capacity. I refer to this configuration as S(2W). Moreover, I examine S with a double sized cache (i.e., 1MB instead of 512KB). I denote this latter configuration by S(D). Fig. 35 shows the L2 miss rates of S, S(2W), S(D), and FSB-4 normalized to S. The figure demonstrates that doubling the size of the cache results in a greater miss reduction than increasing associativity by two ways. Nonetheless, FSB-4 surpasses S(D) for all the examined programs except Lu. On average, S(2W), S(D) and FSB-4 achieves L2 miss rate reductions of 5.1%, 15.6%, and 36.6%, respectively. I conclude that FSB-4 is quite attractive as with small design and storage overhead it provides more than $2x$ miss rate reduction over

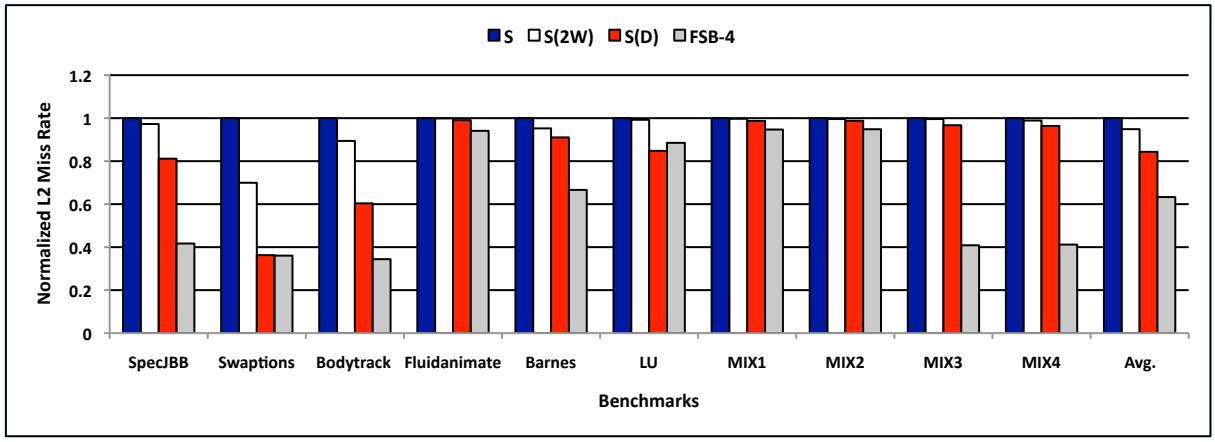


Figure 35: **L2 miss rates of the baseline shared scheme (S), S with two more ways added (S(2W)), S with double sized cache (S(D)), and FSB-4 (all normalized to S).**

S(D) which incurs 88.8% increase in the on-chip cache capacity.

5.3.5 FSB versus Victim Caching

In this section I compare FSB against victim cache (VC) [37]. Again, I contrast only against FSB-4. VC effectively extends the associativity of hot sets in the cache to reduce conflict misses. For a fair comparison, I consider a fully associative 16KB VC per tile to approximately match the storage overhead incurred by FSB-4. I, furthermore, optimistically assume only a 6 cycle access time to VC after each miss on an L2 bank. Fig. 36 depicts the execution times of S, VC, and FSB-4 normalized to S. VC outperforms S by an average of 6.3%. In contrast, FSB-4 improves upon S and VC by averages of 13% and 7.2%, respectively.

5.3.6 FSB versus DSBC and V-WAY

In addition to comparing with victim caching, I compare FSB against the closely related dynamic set balancing cache (DSBC) [55] and variable-way set associative cache (V-WAY) [53]

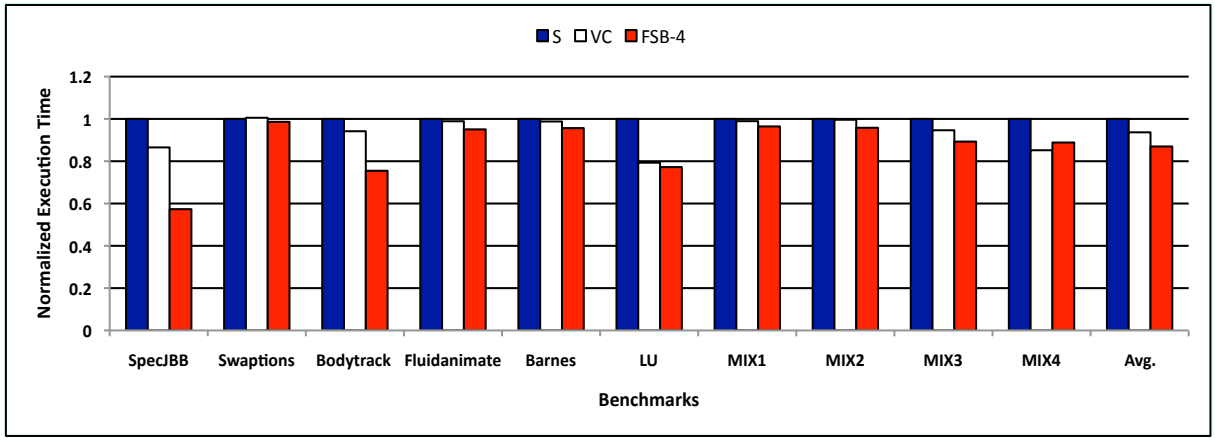


Figure 36: **Execution times of the baseline shared scheme (S), victim cache (VC), and FSB-4 (all normalized to S).**

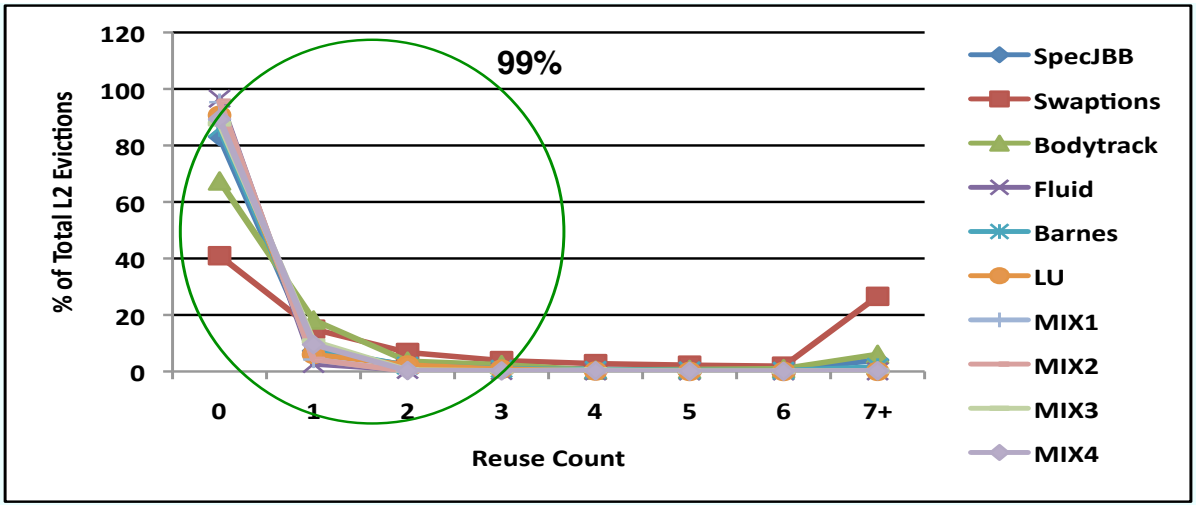


Figure 37: **Distribution of L2 cache lines' reuses before evicted from L2 (Reuse Count = the number of L2 accesses to a cache line after its initial fill).**

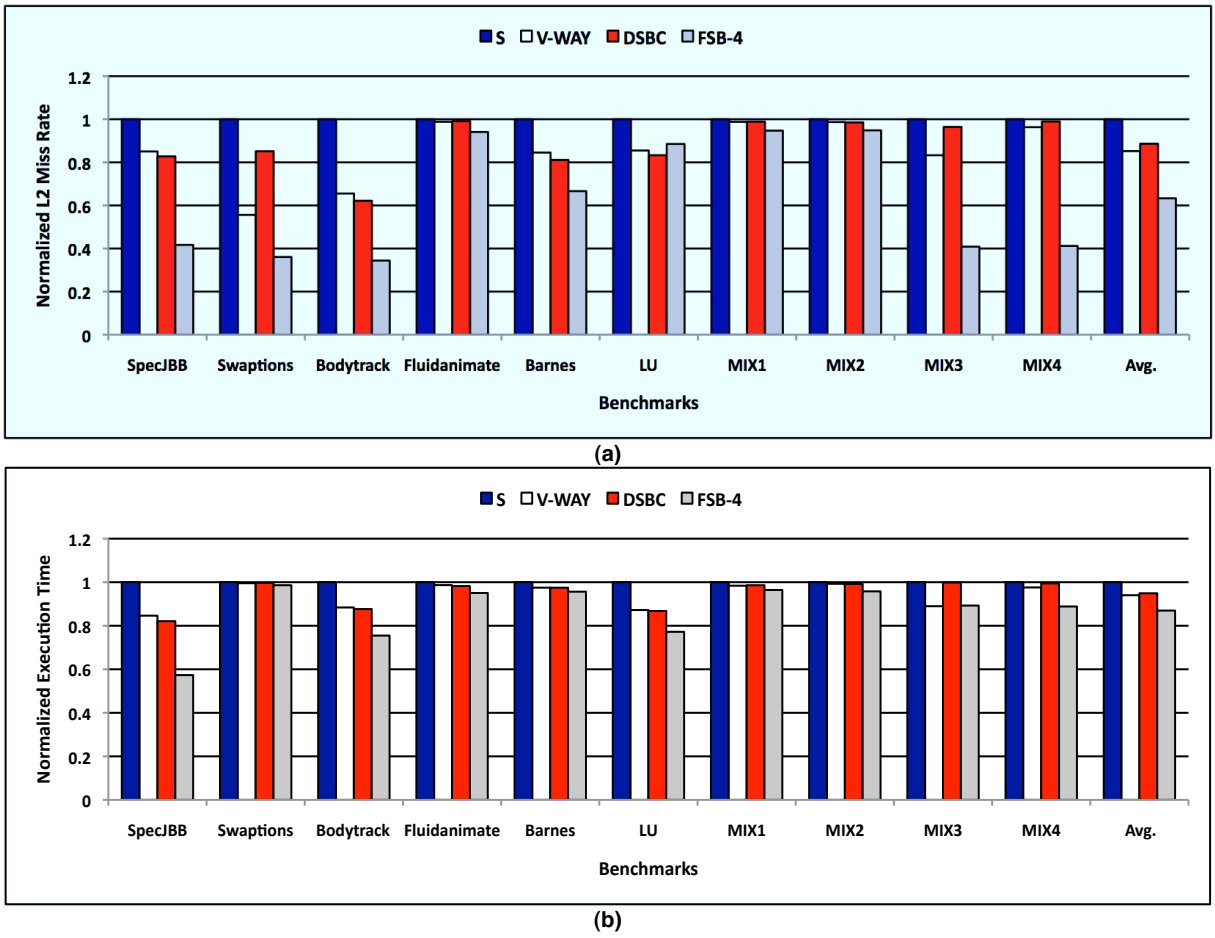


Figure 38: **L2 miss rates and execution times of the baseline shared scheme (S), variable-way set associative cache (V-WAY), dynamic set balancing cache (DSBC), and FSB-4 (all normalized to S).**

designs. Similar to FSB, both DSBC and V-WAY are directly extensible to CMPs. Sections 5.1.2 and 2.2 detail DSBC and V-WAY, respectively.

To elaborate more on V-WAY's reuse replacement policy, each data line in the cache is associated with a *reuse counter*. A reuse count is defined as the number of L2 accesses to a cache line after its initial fill. Upon replacement, a line with a reuse counter equal to zero is replaced. An engine tests and decrements each non-zero reuse counter until one with a zero value is detected. The reuse replacement policy is critical to V-WAY. Hence, to decide upon the number of bits required for reuse counters, I conducted a study to scrutinize the distribution of reuse counts for all evicted L2 cache lines from my benchmark programs. Fig. 37 explores such a distribution. I observe that 99% of L2 cache lines are reused three or fewer times. Consequently, I choose to use two-bit saturating reuse counters.

Fig. 38(a) depicts the L2 miss rates of S, V-WAY, DSBC, and FSB-4 normalized to S. On average, V-WAY and DSBC achieve miss rate reductions of 14.7% and 11.3%, respectively. FSB-4 surpasses V-WAY and DSBC by averages of 27.2% and 29.2%, respectively. Fig. 38(b) shows the execution time results. V-WAY and DSBC outperform S by averages of 5.9% and 5%, respectively. FSB-4, however, improves upon V-WAY and DSBC by averages of 7.8% and 8.8%, respectively.

5.4 SUMMARY

Memory accesses are not evenly distributed across cache sets. Such a skew in sets' usages reduces the effectiveness of the conventional cache designs and cache lines become less likely to be re-referenced before eviction. I propose Flexible Set Balancing (FSB), a strategy that exploits the demand imbalance across sets to retain cache lines evicted from highly pressured sets at underutilized sets so as to satisfy far-flung reuses. FSB adapts to phase changes in programs and promotes a very flexible sharing among cache sets. An underutilized set is allowed to share its space by any stressed set during any point in a program's execution, a policy that I refer to as *one-from-many* sharing. Besides, many sets are allowed to share their capacities with a highly utilized set, a policy that I refer to as *many-from-one* sharing. FSB

targets CC-FR's data retention category and addresses interference misses, bandwidth wall problem, and processor-memory speed gap challenges. FSB incurs a little storage, area, and energy overheads. Simulation results show that FSB achieves an average miss rate reduction of 36.5% for the benchmark programs I examined. This produces an average execution time improvement of 13%. Furthermore, evaluations manifested the outperformance of FSB over some relevant designs including DSBC [55] and V-WAY [53].

6.0 ADAPTIVE CONTROLLED MIGRATION

In this chapter I describe Adaptive Controlled Migration (ACM), a novel relocation mechanism that builds on an area-efficient shared cache design and dynamically migrates cache blocks to cache banks that best minimize the average L2 access latency. ACM targets CC-FR's data relocation category and addresses the growing non-uniform access latencies challenge. In Section 6.1 I provide a motivational study and outline my proposed solution. I then detail the ACM mechanism in Section 6.2. A quantitative evaluation of ACM and a related design is presented in Section 6.3 and conclusions are given in Section 6.4.

6.1 MOTIVATION AND PROPOSED SOLUTION

6.1.1 Motivation

As described in Section 1.2.1, the nominal shared NUCA caches have a latency problem. Block replication and migration have been suggested as techniques for shared caches to tackle this latency problem by frequently copying or moving accessed blocks to cache banks closer to the requesting processors [40, 18, 59, 38, 6, 16, 12, 74, 43]. Replication in general results in reduced cache hit latencies but may degrade cache hit rate. In fact, blind replication can be detrimental since the capacity occupied by replicas could increase significantly resulting in performance degradation [6]. Migration, on the other hand, maintains the exclusiveness of cache blocks on chip and preserves the high utilization of the caching capacity. Furthermore, it maintains the simplicity of the underlying cache coherence protocol. However, migration has been shown to be less effective for CMP caches than for uniprocessors

[7, 40]. The challenge in the CMP domain is that migration in multiple directions can cause migration conflicts, with shared blocks ping-ponging between processors [34]. Besides, locating migratory blocks in bank sets may turn out to be very expensive to an extent that it might offset the benefits offered by the migration technique.

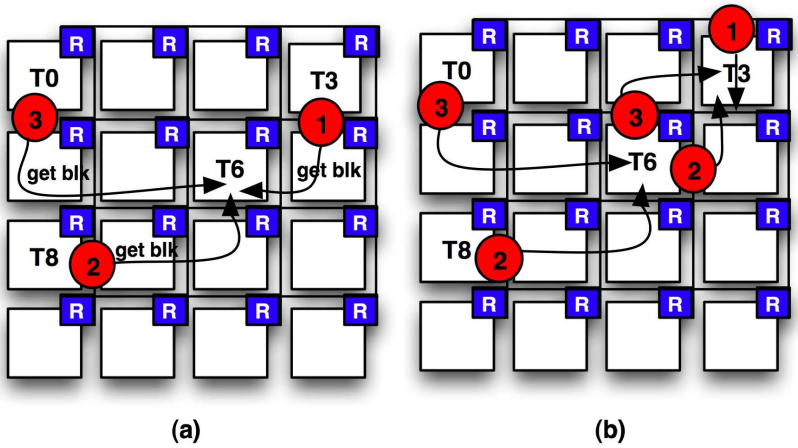


Figure 39: **(a) The Original Shared CMP Scheme. (b) A Simple Migration Example.**

I demonstrate through an example the difficulty behind block migration and the inefficiency it may cause with shared L2 cache design in the CMP context. Figure 39(a) illustrates a 16-core tiled CMP. Again, I assume a shared scheme where L2 cache slices are logically shared among all tiles. Upon an L2 miss, a line is fetched from the main memory and placed at its static home tile (SHT) determined by a subset of bits (the HS bits) of the line's physical address. The figure shows a case where a block, B, has been originally requested by tile 3 and mapped to tile 6. Later tiles 0 and 8 request the same cache block B. Tile 3 incurs 6 network hops, computed as twice the Manhattan distance between the requester and the target tiles (dimension-ordered routing [46]), to reach B's SHT, T6, and satisfy its request. Tiles 0 and 8 incur 8 hops each to satisfy their requests.

Figure 39(b) illustrates a naïve first touch migration policy that directly migrates B to the original requester. Employing that, tile 3 will save the 6 network hops when touching B for the second time, assuming that it checks its local L2 tags before accessing B's SHT. Block B, however, has been pulled away from the other two sharers incurring additional 6

hops for each one to locate the block on its new host tile. Consequently, even though one tile made a gain, in total there is a loss of 6 network hops. Besides, the on-chip network traffic increases due to the three-way cache-to-cache communications to satisfy sharers' requests. Specifically, when tile 0 (or 8) requests B, it has to check with tile 6 first, which is B's SHT, before its request is redirected to tile 3.

Though the above example shows a naïve migration policy, it highlights the intuition that applying migration in a non-controlled fashion to NUCA designs can lead to performance degradation. The problem, in fact, is that the best host of a cache block is not known apriori. Consequently, migration or replication can interfere to rectify the situation. It would be highly beneficial if there is a mechanism that can dynamically and adaptively locate the best host on chip for each cache block, and move consecutively the block to that host without incurring undesirable implications.

6.1.2 Proposed Solution

This chapter grants a fresh thought to the data migration technique as a way to manage shared NUCA caches and studies its effectiveness in tiled CMPs. I propose a novel hardware-based *adaptive controlled migration* (ACM) mechanism that relies on prediction to collect information about which tiles have accessed a block and then, assuming that each of these tiles will access the block again, dynamically migrates the block to a tile that minimizes the overall number of network hops needed. Simulation results demonstrate the effectiveness, scalability, and stability of the proposed scheme using a variety of workloads that exhibit *no-sharing*, *little sharing*, or *sharing* of cache blocks.

In summary, the contributions of the ACM mechanism are as follows:

- It demonstrates that migration, if done in an adaptive controlled fashion, yields an average L2 access latency that is on average 20.4% better than the nominal shared cache scheme, and 20.8% better than a conventional replication strategy for the simulated benchmarks.
- It demonstrates the effectiveness of migration in the CMP domain and opens new research opportunities and directions for computer architects.

- It avoids replication of cache blocks and reduces the overall L2 cache access latency without degrading the cache miss rate.

6.2 THE ADAPTIVE CONTROLLED MIGRATION (ACM) MECHANISM

6.2.1 Predicting Optimal Host Location

Keeping a block in its home tile is often sub-optimal. Ideally, we want to place a cache block in the tile that best optimizes the overall latencies. However, the best host tile for a block is not known until runtime because many cores may compete for that block. Consequently, a dynamic adaptive mechanism that monitors the runtime accessibility of a block and makes a decision about the best location for the block is needed.

I propose a simple location algorithm that attempts to locate the optimal host of a cache block at runtime and designates it as its new *host tile*. It computes the *total latency cost* for a given cache block, B, on each of the potential hosts and chooses the *minimum*. In order to achieve this, the algorithm keeps some runtime information, particularly, a pattern for the accessibility of B. The pattern is essentially a bit vector to indicate whether B has been accessed by a specific core or not. It can be built at run time with different *migration frequency levels*. The migration frequency level is the number of times B is accessed before attempting to migrate it. Whenever a core accesses B, its corresponding bit is set in the associated bit vector and a *use counter* associated with B is incremented. This continuously shapes up an accessibility pattern for B and provides the aspired runtime information. When the use counter reaches the specified migration frequency level, the location algorithm interferes and selects a new host for B that minimizes the total L2 access latency for all B's sharing cores.

After migrating B, its corresponding pattern and use counter are cleared so as to initiate a new pattern construction. As such, ACM pertains a simple prediction scheme that depends on the past to predict the future. A core that accessed B in the past is likely to access it again in the future. Because ACM's algorithm makes its decision based on this pattern, I refer to the located host a *predicted optimal host*. Lastly, by having a migration algorithm

that relocates blocks from their SHTs, a location strategy capable of rapidly locating cache blocks at the L2 cache space is required. ACM adopts C-AMTE (see Chapter 3) to achieve fast location of L2 cache blocks.

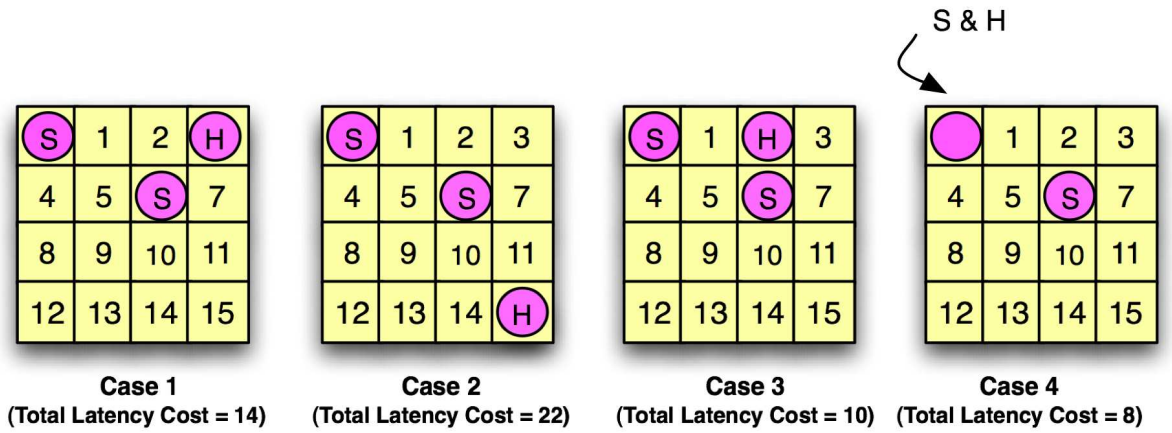


Figure 40: **An Example of How ACM Works (S = Sharer, H = Host).**

To exemplify how the ACM mechanism works, Figure 40 portrays 4 different cases for potential hosts that a cache block may migrate to. S stands for a sharer and H for the current host. A sharer is a tile that accessed the block in the past. A host is a tile that is currently hosting the cache block. The designated block has two sharers, tiles 0 and 6. The block can be potentially hosted by any of the 16 tiles, but in Figure 40 I depict only four cases where the tiles 3, 15, 2, and 0 host the block. Total latencies of 14, 22, 10, and 8 network hops are incurred by the sharers to locate the data block in these four designated hosts respectively. Among these, host 0 gives the minimum aggregate latency and is selected by the ACM mechanism to be the predicted optimal host.

6.2.2 Replacement Policy Upon Migration: Swapping the LRU Block with the Migratory One

After the location algorithm designates a new host for a block, B, and the migration is to be performed, a decision must be made about which block to replace in the new host, T, located for B. If there is no invalid block in the target set at T, a naïve approach would replace the

LRU block, say, D. However, because cache accesses might not be well distributed over the cache sets, there could be a capacity pressure at T, and D could be requested again. Hence, I try not to discard block D but to swap it with B so as to maintain the copy on chip.

Blocks B and D can be migratory or non-migratory blocks. Migratory blocks are those that have been already migrated out of their SHTs while non-migratory ones are those that have not been migrated yet. If B is non-migratory, a principal tracking entry should be allocated at its SHT in the TR table. If no entry is found to be replaced at the TR table, as planned by the C-AMTE replacement policy, migration is not performed. If B could be migrated and D is a migratory block, then they are simply swapped and D's associated tracking entries are all updated to reflect the change of the host location. If D is non-migratory, the TR table at its SHT is checked for a valid tracking entry to replace. If a valid entry is found, a corresponding principal tracking entry is allocated and B and D are swapped. If no valid entry is found then D is simply discarded and B is migrated to the new located host. Of course, if B also is a migratory block then when migrated, all its associated tracking entries are updated to denote its new host.

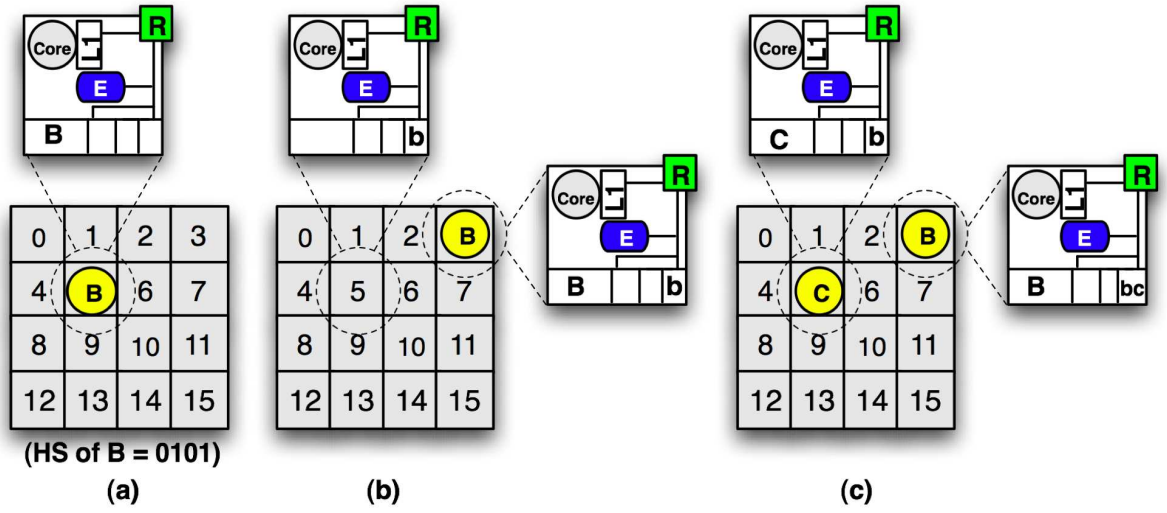


Figure 41: **An Automatic Data Attraction Case offered by ACM.**

The above swapping policy is, in fact, very effective and robust that it makes my scheme applicable even to workloads that don't share cache blocks. I illustrate this effectiveness and

robustness via an example. Figure 41 depicts a case for a single thread that exhibits *no sharing* at all and runs on tile 3 (the microarchitecture of the portrayed tiles is discussed in Section 3). I assume that the thread's working set is too large to fit entirely in the L2 cache bank of tile 3. To reduce L2 access latency, and with migration frequency level of 1, my location algorithm will choose to migrate all requested blocks to the L2 bank of tile 3 after accessing each for the second time. Figure 41(a) depicts the placement of block B after it has been requested by tile 3 and mapped to B's SHT, tile 5 (HS of B = 0101). Figure 41(b) demonstrates the migration of B to the L2 bank of tile 3 after tile 3 accessed B for the second time. Note that a principal tracking entry, b, is allocated in the TR table at B's SHT and a replicated one allocated at tile 3, as planned by the C-AMTE mechanism.

The case shows the capability of the ACM mechanism to automatically attract data to local tiles, thus allowing cores to access blocks very fast for subsequent requests. However, ACM is robust enough that it doesn't allow such *automatic data attraction* to continue freely and blindly, potentially causing increase in L2 miss rate. The *swapping with the LRU* policy suggests that when no invalid block at the target set of the located host T is found (capacity pressure), an attracted block B is swapped with the LRU block at T, thus avoiding increase in L2 miss rate. Figure 41(c) assumes that when B has been migrated to tile 3 it produces capacity pressure. The LRU block, C, at tile 3 is accordingly swapped with B as planned by the swapping with LRU policy, thus maintaining C on chip and accordingly avoiding any increase in the L2 miss rate. Note that a principal tracking entry, c, is allocated in the TR table at tile 3 as planned by the C-AMTE mechanism.

Essentially, the case demonstrates some resemblance to the victim replication strategy [74]. Victim replication also automatically attracts data to local tiles upon L1 evictions in order for subsequent accesses to save latency. However, it doesn't provide control on the capacity usage and can greatly reduce the available caching space. Section 6.3 presents a comparison between ACM and the victim replication scheme.

Finally, different cache blocks may experience entirely different degrees of sharing over time and demonstrate diverse access patterns. The ACM algorithm collects those non-uniform access patterns and based on them, finds a suitable location for data blocks that best minimize the L2 access latency. Thus ACM inherently doesn't prefer any specific on-

chip tile over the other. However, if at any course of execution, a cache bank receives a capacity pressure more than other banks (similar to the above case), ACM robustly relaxes the pressure via the swapping with the LRU policy.

6.3 QUANTITATIVE EVALUATION

In this section, I evaluate the ACM mechanism and alternative cache designs. I compare ACM against the nominal shared (S) and the victim replication (VR) schemes [74]. The rationale behind comparing against VR is to demonstrate that migration, if done in an adaptive-controlled fashion, generates favorable results compared to replication. Replication has been considered useful for tackling the NUCA problem [6, 16]. Migration, on the other hand, hasn't been proved to be highly effective in the context of CMPs [7, 73]. By showing that ACM outperforms the shared design and one of the conventional replication strategies, I thereby demonstrate that migration is in fact an effective technique for managing L2 caches in CMPs.

6.3.1 Experimental Methodology

In this chapter I adopt a little different evaluation methodology than the one described in Section 2.3, thus I detail it. As throughout the whole thesis, my evaluation employs a detailed full system simulator built on Simics 3.0.29 [68]. I simulate a 16-way tiled CMP architecture organized as a 4×4 2D mesh grid and runs under the Solaris 10 OS. Programs are compiled for an UltraSPARC-III Cu processor. Each core runs at 1.4 GHz, uses in-order issue, and has a 16KB I/D L1 cache and a 512KB L2 cache with the LRU replacement policy. The aggregate L2 cache is consequently 8MB for the 16-tiled CMP model. Each L1 cache is 4-way set associative with 1-cycle access time and 64 byte line. Each L2 cache is 16-way set associative with 6-cycle access time and 64 byte line. A 5-cycle latency per hop, based on a recent processor from Intel [66], is incurred when a datum traverses through the mesh network including both, a 3-cycle switch [17, 73, 74] and a 2-cycle link latencies. The 4-GB

off-chip main memory latency is set to 300 cycles.

ACM, S, and VR are studied using a mixture of single-threaded, multithreaded, and multiprogramming workloads. For multithreaded workloads, I use the commercial benchmark SPECjbb, and four other shared memory benchmarks from the SPLASH2 suite [70] (Ocean, Barnes, LU, Radix). For single-threaded workloads I use six programs from SPEC2K [63], three integers (vortex, parser, mcf) and three floating-points (art, equake, ammp) with the reference data sets. These benchmarks were chosen because they demonstrate different access patterns and different working set sizes [63, 20]. Two multiprogramming workloads, MIX1 (vortex, ammp, mcf, equake) and MIX2 (art, equake, parser, mcf), are constructed from the selected 6 SPEC2K programs. Initialization phases of applications are skipped using magic breakpoints from Simics. For the single-threaded and multiprogramming workloads, a detailed simulation is run for each benchmark until at least one core completes 1 billion instructions. Table 10 summarizes all the simulated benchmarks. Last but not least, I fix the migration frequency level to 10 throughout the simulation.

6.3.2 Comparing Schemes, Single-threaded and Multiprogramming Workloads

Multiprogramming workloads tend to have very little sharing among the different threads [12, 73]. Single-threaded benchmarks represent the *no-sharing* case. VR is very appealing in this situation because it can automatically attract data blocks to the only tile running the thread, thus supposedly reducing access latency by decreasing inter-tile accesses from replica hits. However, this may make the tile running the thread experience some high capacity demand. This may result in poor utilization of the on-chip cache capacity. If the scheme fails to offset the increased miss rate then this could lead to performance degradation. This intuition is confirmed by the results shown in Fig. 42. The L2 miss rates of all the single-threaded benchmarks shown for VR are all much larger than that for S. However, across all the workloads VR successfully offsets the miss rate from fast replica hits. Contrary to that, VR fails to offset the increase in L2 miss rate for the multiprogramming workloads. Clearly, the SPEC2k applications have small working sets that more or less fit in L1 and L2 caches as they expose negligible L2 miss rates as is shown in the figure for the S scheme. The L2 miss

Name	Input
SPECjbb	Java HotSpot (TM) server VM v 1.5, 4 warehouses
lu	1024×1024 matrix (16 threads)
ocean	514×514 grid (16 threads)
radix	2M integers (16 threads)
barnes	16K particles (16 threads)
parser	reference
art	reference
equake	reference
mcf	reference
ammp	reference
vortex	reference
MIX1	reference for all (vortex, ammp, mcf, and equake)
MIX2	reference for all (art, equake, parser, mcf)

Table 10: **Benchmark programs.**

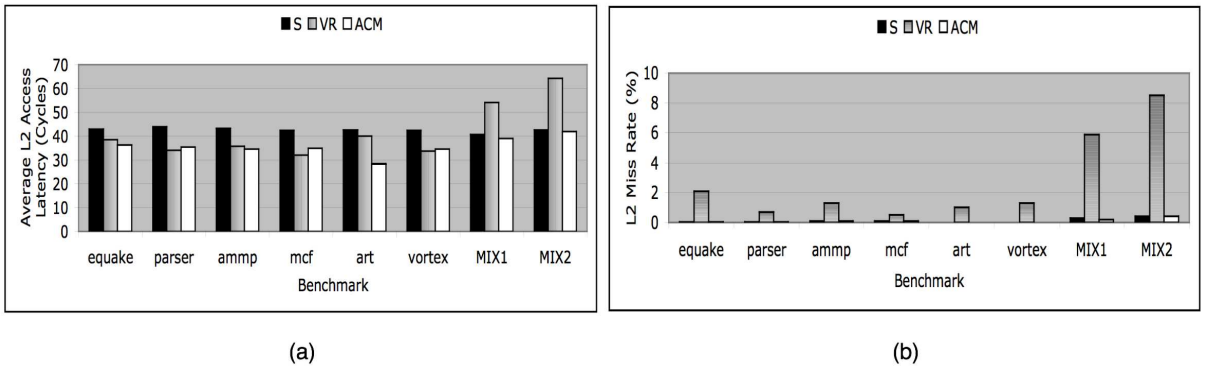


Figure 42: **Single-threaded and Multiprogramming Results (S = Shared, VR = Victim Replication).**

rate for the 6 single-threaded benchmarks is on average 0.04%. As the memory footprint of the benchmark decreases, the space made available to replicas increases and accordingly more performance improvement can be achieved. For the multiprogramming workloads, 4 benchmarks are now sharing the L2 cache space. Hence, the memory footprint of the workload has been increased. The L2 miss rate for MIX1 and MIX2 is now 0.3% on average, or 8.7x more than that of the single-threaded workloads. VR failed to offset this increase and produced 41.8% and 46.0% AAL degradation over S and ACM schemes respectively.

Contrary to that, ACM still offers this automatic data attraction functionality suggested by the VR scheme but in a very controlled fashion that it can efficiently customize allocation of on-chip capacity via the swapping with LRU policy as is discussed in Section 2. Consequently, it successfully generated AALs that are on average 20.5% and 3.7% better than S and VR respectively for the single-threaded workloads, and 2.8% and 31.3% better than S and VR respectively for the multiprogramming ones. VR performs better than ACM only for the benchmarks vortex, parser, and mcf. It has been observed that 81%, 50%, and 57% of the cache blocks of the vortex, parser, and mcf benchmarks respectively are accessed for less than 10 times (the specified migration frequency level). As a result, for all these cache blocks, the ACM mechanism didn't even attempt to migrate them to better hosts so as to minimize the L2 access latency. This is because I fixed the migration frequency level throughout simulations. Migration to any cache block is triggered upon being accessed for the number of times that the migration frequency level specifies. For that reason, ACM is not exhibiting its full ability to exploit the optimum performance though it still on average greatly surpasses both of the schemes, S and VR. With an adaptive tunable migration frequency level, the ACM mechanism would hit its optimum and consequently provide larger performance improvements. This is to be explored in future research work.

As clearly shown in Fig. 42, ACM maintains the L2 miss rates of S for all the simulated single-threaded and multiprogramming benchmarks. Moreover, it optimizes some of them because of the swapping with the LRU policy and generates on average 2x and 1.5x reductions in L2 miss rates over S for the single-threaded and the multiprogramming benchmarks respectively. Finally, Fig. 43 shows the average memory access cycles per 1K instructions experienced by all the simulated benchmarks. VR performs on average 15.1% better than

S and 38.4% worse than S for the single-threaded and the multiprogramming benchmarks respectively. ACM, on the other hand, performs on average 18.6% and 2.6% better than S, and 3.4% and 29.4% better than VR for the single-threaded and the multiprogramming benchmarks respectively.

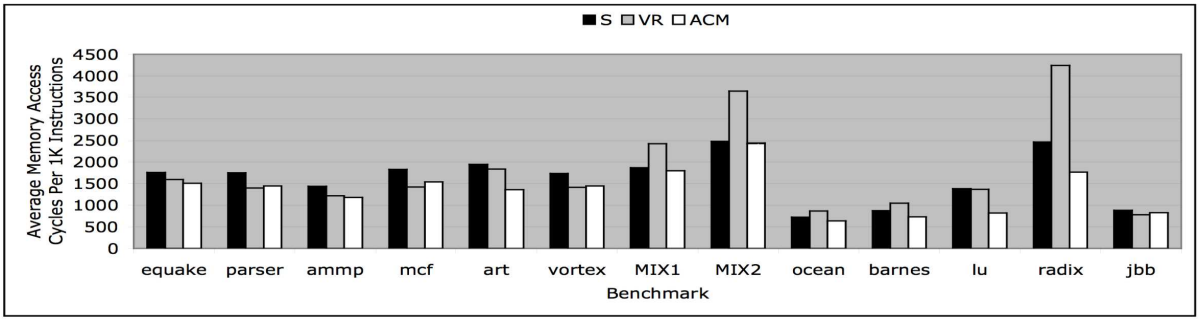


Figure 43: **Average Memory Access Cycles Per 1K Instructions Results (S = Shared, VR = Victim Replication).**

6.3.3 Comparing Schemes, Multithreaded Workloads

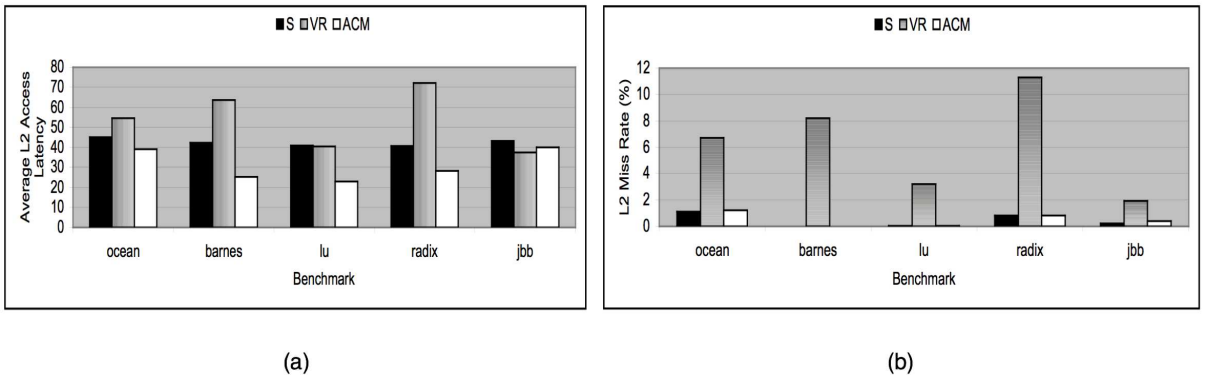


Figure 44: **Multithreaded Results (S = Shared, VR = Victim Replication).**

The multithreading workloads expose different degrees of sharing among threads and accordingly allow us to study the efficiency of the ACM mechanism with such a case. Fig. 44

depicts AALs and the L2 miss rates of the multithreading workloads compared to S and VR. ACM exhibits AALs that are on average 27.0% and 37.1% better than S and VR respectively. VR reveals 26.7% worse AAL than S for all the simulated benchmarks. This is due to the fact that VR has a static replication policy that depends on the blocks' sharing behaviors. An increase in the degree of sharing suggests that the capacity occupied by replicas could increase significantly leading to a decrease in the effective L2 cache size. As such, if replicas displace too much of the L2 cache capacity, the L2 miss rate could increase considerably, degrading thereby the average L2 access latency. This was clearly illuminated by the behaviors of the Ocean, Barnes, and Radix benchmarks where reduction in latencies via replica hits failed to offset the excessive latencies deduced by the increased miss rate. Barnes for instance utilizes a tree data structure that exhibits a sharing degree of 71% [7] and accordingly incurs a significant increase in capacity pressure when VR is used. Note that such an inferred VR behavior is more elucidated in this work than in the original evaluation [74] as the L2 cache has been downsized to half. VR was successful in offsetting the impact of increased offchip accesses for the LU and JBB workloads.

On the other hand, ACM, a pure migration technique, maintains the exclusiveness of cache blocks on chip and consequently preserves the L2 miss rates of S for the three benchmarks Barnes, Lu, and Radix. Only Ocean and JBB reveal a small increase in the L2 miss rate for ACM over S. This is because when some block, B1, is to be migrated to a new host, H, no valid entry for the LRU block, B2, that is to be swapped with B1, is found in the MT table of H, and accordingly discarded as planned by the swapping with the LRU policy. That discarded block, B2, can be requested again by some other threads. VR only performs better than ACM for the JBB benchmark. The reason is the fixed migration frequency level that I assume throughout the simulation process. I ran JBB with doubling and tripling the migration frequency and respectively obtained 3.7% and 6.7% more AAL improvements over the base run with a migration frequency of 10. Lastly, Fig. 43 shows the average memory access cycles per 1K instructions. For the multithreading benchmarks, VR performs on average 19.6% worse than S. ACM, in contrary, performs on average 20.7% and 29.7% better than S and VR respectively.

6.3.4 On-Chip Network Traffic

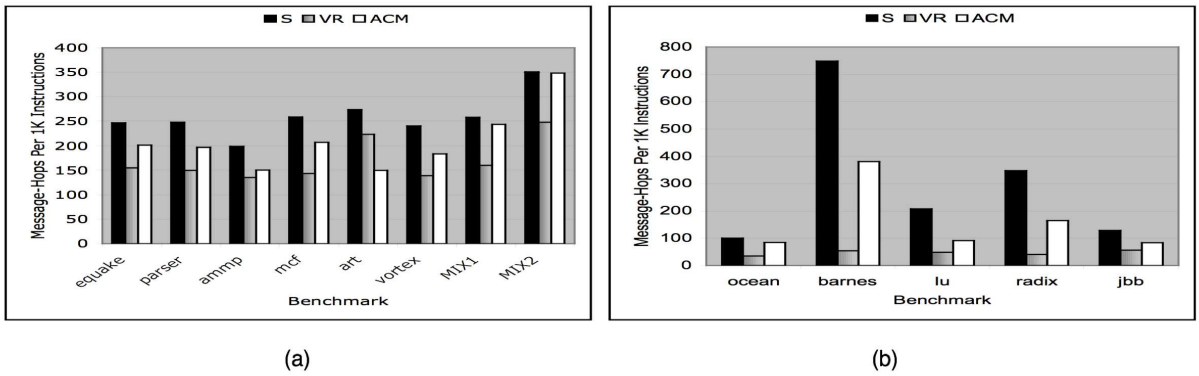


Figure 45: **On-Chip Network Traffic Comparison. (a) Single-threaded Workloads (b) Multithreaded Workloads**

A supplementary advantage of the ACM mechanism is the reduced on-chip network traffic that it offers. Fig. 45 depicts the number of message-hops per 1k instructions that the three schemes, S, VR, and ACM exhibit for the single-threaded, multiprogramming, and multithreaded workloads. The ACM scheme offers 25.3%, 3.0%, and 41.6% on-chip network traffic reduction over S for the single-threaded, multiprogramming, and multithreaded workloads respectively. The VR scheme, on the other hand, offers 35.6%, 33.6%, and 75.7% on-chip network traffic reduction over S for the three workloads respectively. Consequently, VR offers more on-chip network reduction over S than what ACM does because it decreases more inter-tile accesses from replica hits. Though the ACM mechanism bears some resemblance to the VR strategy for the single-threaded workloads as discussed in Section 2, but with a fixed migration frequency level of 10, a tile waits for 10 accesses to the block to attract it to its local L2 bank. Therefore, it incurs more inter-tile accesses compared to VR that tries to attract the block to its local L2 bank immediately after being evicted from L1. If VR fails to replicate cache blocks while ACM succeeded to attract blocks to its local L2 bank, ACM will surpass VR. The Art benchmark is the only case that exhibits such a situation. Finally, I studied the increase in network congestion for ACM over S without employing the C-AMTE mechanism. I found on average an increase of 17.6%, 4.1%, and 1% over S for

the single-threaded, multiprogramming, and multithreaded workloads respectively. Clearly, this demonstrates the decisiveness and usefulness of such a mechanism when applying block migration in CMPs.

6.3.5 Scalability

The area scalability presented in Section 4.3.5 applies to ACM because, similar to PDA, ACM utilizes C-AMTE to rapidly locate migratory L2 cache blocks. Furthermore, ACM demonstrates an effective adaptability to upcoming futuristic CMP models. ACM always selects a host for a cache block, B, that minimizes the total L2 access latency for B's sharing cores irrespective of the number of cores on the CMP platform. However, more exposure to the NUCA problem translates effectively to a larger benefit from the ACM scheme. I demonstrate such a pro of ACM via extending my CMP model to 32 tiles and running simulations for three selected benchmarks. Each tile still maintains, as with the 16-tiled CMP model, a 16KB I/D L1 cache and a 512KB L2 cache bank. The three benchmarks that have been chosen to conduct the study are, Ocean, ammp, and MIX1, from the multithreaded, single-threaded, and multiprogramming workloads, respectively. These benchmarks revealed the highest L2 miss rates among the others in their sets; hence, selected.

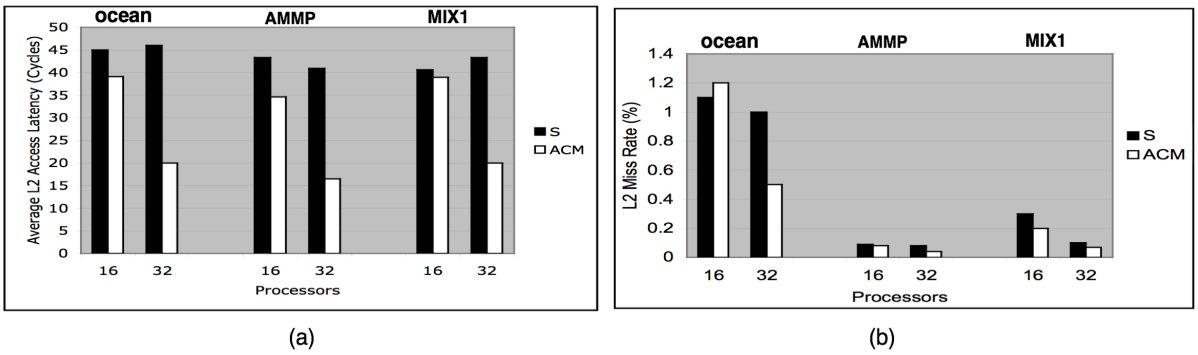


Figure 46: **Results for CMP Systems with 16 and 32 Processors. (a) Average L2 Access Latencies (b) L2 Miss Rate**

Fig. 46 depicts the AALs and the L2 miss rates of the selected benchmarks. For 16 tiles, ACM shows AAL improvements of 13.1%, 20.0%, and 1.7% for Ocean, ammp, and

MIX1 over S, respectively. However, for 32 tiles, ACM shows AAL improvements of 56.5%, 59.6%, and 53.8% over S, respectively. On average, ACM exhibits 11.6% and 56.6% AAL improvements over S for the 16-tile and 32-tile models, respectively.

6.3.6 Sensitivity and Stability Studies

So far I have assumed for simplicity that the size and associativity of the TR tables are identical to that of the L2 caches. There is nothing, in fact, that prevents this data structure from being of a smaller size or associativity. To study the sensitivity of the ACM mechanism to this component, I ran simulations for the three benchmarks, Ocean, ammp, and MIX1 with TR table sizes reduced to *half* (50%) and *quarter* (25%) the size of the *base* cache, and with a 16-way set associativity. With half and quarter configurations, I got AAL increases of 5.9% and 11.3% respectively over the base one, but still improvements of 7.6% and 2.9% respectively over S. The highest contribution for the AAL increases was from the ammp benchmark. The ammp benchmark shows alone 19.9% AAL increase, averaged for both the half and quarter configurations, over the base, though still 6.1% better than S. It was observed that 60.7% of the cache blocks in ammp are accessed at least 10 times (the specified migration frequency level) before getting evicted from L2, consequently triggering migrations. To decrease the pressure on the TR table, I ran simulations with migration frequency of 20 rather than 10. Compared to the 19.9% AAL increase, I obtained only 12.3% increase, averaged for both the half and quarter configurations, over the base one and consequently 10.1% on average better than S.

Finally, and to demonstrate the stability of the ACM scheme to different cache sizes, I simulated the LU benchmark on my 16-tiled CMP model with the L2 cache being reduced to *half* its size for the three different schemes: S, VR, and ACM. VR failed to demonstrate stability and showed AAL degradation of 37.8% over S, while ACM maintained AAL improvement of 39.7% over S. This is because the ACM mechanism maintains the exclusiveness of cache blocks on chip, while VR demands more capacity to store replicas. Clearly, this reveals the effectiveness of the migration technique in the CMP domain, and particularly that of the proposed ACM mechanism.

6.4 SUMMARY

Managing L2 caches in chip multiprocessors is essential to fuel its performance growth. This chapter studied a strategy to manage non-uniform shared caches in CMP by dynamically migrating cache blocks to optimal locations that provide the minimal L2 access latency. The proposed mechanism optimizes the L2 miss rate via maintaining the uniqueness of cache blocks on chip. Clearly, ACM lies under CC-FR's data relocation category. Simulation results demonstrated the robustness, scalability, and stability of ACM. Unlike previously studied migration strategies in CMP literature, the proposed mechanism revealed and confirmed the usefulness of data migration in chip multiprocessors.

7.0 DYNAMIC CACHE CLUSTERING

This chapter proposes DCC (Dynamic Cache Clustering), a novel distributed cache management scheme for large-scale chip multiprocessors. Using DCC, a per-core cache cluster is comprised of a number of L2 cache banks and cache clusters are constructed, expanded, and contracted dynamically to match each core's cache demand. The basic trade-offs of varying the on-chip cache clusters are average L2 access latency and L2 miss rate. DCC uniquely and efficiently optimizes both metrics and continuously tracks a near-optimal cache organization from many possible configurations. DCC employs CC-FR's data placement and relocation approaches and addresses diverse workload characteristics and growing non-uniform access latencies challenges. Section 7.1 provides a motivational study and outline the proposed solution. I give a brief background on some of the fixed (static) cache designs in Section 7.2. The DCC mechanism is detailed in Section 7.3. A quantitative evaluation of DCC and a related design is presented in Section 7.4 and conclusions are given in Section 7.5.

7.1 MOTIVATION AND PROPOSED SOLUTION

7.1.1 Motivation

As described earlier in Section 1.2.4, computer applications exhibit different cache demands. The traditional private and shared designs are subject to a principal deficiency. They both entail static partitioning of the available cache capacity and don't tolerate the variability among different working sets and phases of a working set. For instance, a program phase with high cache demand would require enough cache capacity to mitigate the effect of high

cache misses. On the other hand, a phase with less cache demand would require smaller capacity to mitigate the NoC communications. Static designs provide either fast accesses or capacity but not both. A crucial step towards designing an efficient memory hierarchy is to offer both fast accesses and capacity.

7.1.2 Proposed Solution

This chapter sheds light on the irregularity of working sets and presents a novel dynamic cache clustering (DCC) scheme that can synergistically react to programs' behaviors and judiciously adapt to their different working sets and varying phases. DCC suggests a mechanism to monitor the behavior of an executing program, and based upon its runtime cache demand makes related architecture-adaptive decisions. The tension between higher or lower cache demands is driven by optimizing the L2 miss rate (MR) versus the average L2 access latency (AAL) metrics. Each core is initially started up with an allotted cache resource, referred to as its *cache cluster*. Subsequently, after every re-clustering point on a time interval, the cache cluster is dynamically contracted, expanded, or kept intact, depending on the cache demand. The CMP cores cooperate to attain fast accesses (i.e, better AAL) and efficient capacity usage (i.e, better MR).

In summary, this chapter makes the following contributions:

- I propose DCC, a hardware mechanism that detects non-uniformity amongst working sets, or phases of a working set, and provides a flexible and efficient cache organization for CMPs.
- I introduce novel mapping and location strategies to manage dynamically resizable cache configurations on tiled CMPs.
- I demonstrate that DCC improves the average L1 miss time by as much as 21.3% (10% execution time) versus previous static designs.

7.2.1 Fixed Cache Schemes

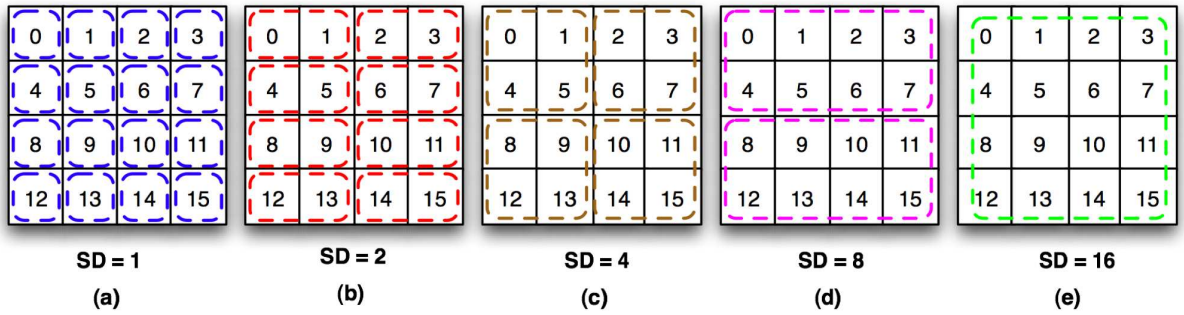


Figure 47: **Fixed Schemes (FS) with different sharing degrees (SD). (a) FS1 (b) FS2 (c) FS4 (d) FS8 (e) FS16**

The physically distributed L2 cache banks of a tiled CMP can be organized in different ways. At one extreme, each L2 bank can be made private to its associated core. This corresponds to contracting a traditional multi-chip multiprocessor onto a single die. At the other extreme, all the L2 banks can be aggregated to form one logically shared L2 cache (shared scheme). Alternatively, the L2 cache banks can be organized at any point in between private and shared. More precisely, [34] defines the concept of *sharing degree* (SD) as the number of processors that share a pool of L2 cache banks. In this terminology, an SD of 1 means that each core maps and locates the requested cache blocks to and from its corresponding L2 bank (private scheme). An SD of 16, on the other hand, means that each of the 16 cores shares with all other cores the 16 L2 banks (shared scheme). Similarly, an SD of 2 means that 2 of the cores share their L2 banks. Fig. 47 demonstrates five sharing schemes with different sharing degrees (SD= 1, 2, 4, 8, and 16) as implied by my 16-tile CMP model. I refer to these sharing schemes as Fixed Schemes (FS) to distinguish them from my proposed dynamic cache clustering (DCC) scheme.

7.2.2 Fixed Mapping and Location Strategies

At an L2 miss, a cache block, B, is fetched from main memory and mapped to an L2 cache bank. A subset of bits from the physical address of B, denoted as the home select (HS) bits, can be utilized and adjusted to map B as required to any of the shared regions of the aforementioned fixed schemes. If B is a shared block, it might be mapped to multiple shared regions. However, as the sharing degree (SD) increases, the likelihood that a shared block maps within the same shared cache region increases. As such, FS16 maps each shared block to only one L2 bank. I identify the tile at which B is mapped to, as a *dynamic home tile* (DHT) of B. For any of the above defined fixed schemes, the utilized HS bits depend on SD. Furthermore, the function that uses the HS bits of B's physical address to designate the DHT of B can be used to subsequently locate B.

7.2.3 Coherence Maintenance

The fixed scheme FS16 maintains the exclusiveness of shared cache blocks at the L2 level. Thus, FS16 requires maintaining coherence only at the L1 level. However, for the other fixed schemes with lower SDs, each L2 shared region might include a copy of a shared block. This, consequently, requires maintaining coherence at both, the L1 and the L2 levels. To achieve such an objective, two options can be employed: a centralized and a distributed directory protocols. The work in [34] suggests maintaining the L1 cache coherence by augmenting directory status vectors in the L2 tag arrays. A directory status vector associated with a cache block, B, designates the copies of B at the private L1 caches. For the L2 cache coherence, [34] utilizes a centralized engine. A centralized coherence protocol is deemed non-scalable especially with the advent of medium-to-large scale CMPs and the projected industrial plans [56]. A high-bandwidth distributed on-chip directory can be adopted to accomplish the task [56, 74].

By employing a distributed directory protocol, directory information can be decoupled from cache blocks. A cache block B can be mapped to its DHT, specified by the underlying cache organization. On the other hand, directory information that corresponds to B can be mapped independently to a potentially different tile, referred to as the static home tile (SHT)

of B. The SHT of B is typically determined by the home select (HS) bits of B's physical address (see Chapter 2, Section 2.1) [1]. For the adopted 16-tile mesh-based CMP model, a duplicate tag embedded with a 32-bit directory status vector can represent the directory information of B. For each tile, one bit in the status vector indicates a copy of B at its L1, and another bit indicates a copy at its L2 bank. To reduce off-chip accesses, Dir (see Fig. 5) can always be checked by any requester core to locate B at its current DHT, using 3-way cache-to-cache transfers.

7.3 THE DYNAMIC CACHE CLUSTERING(DCC) MECHANISM

This section begins by analytically analyzing the major metrics that are involved in managing caches in CMPs, then moves to define the problem on-hand, and finally describes the proposed DCC scheme.

7.3.1 Average Memory Access Time (AMAT)

Given the 2D mesh topology and the dimension-ordered XY routing algorithm being employed by my CMP model, upon an L1 miss, the L2 access latency can be defined in terms of the congestion delay, the number of network hops traversed to satisfy the request, and the L2 bank access time. The basic trade-offs of varying the sharing degree of a cache configuration are the average L2 access latency (AAL) and the L2 miss rate (MR). The average L2 access latency increases strictly with the sharing degree. That is, as the sharing degree increases, the Manhattan distance between a requester core and a DHT tile also increases. The L2 miss rate, on the other hand, is inversely proportional to the sharing degree. As the sharing degree decreases, shared cache blocks occupy more cache capacity and potentially cause the L2 miss rate to increase. Thus AAL and MR are in fact two conflicting metrics.

Besides, an improvement in AAL doesn't necessarily correlate to an improvement in the overall system performance. If the sharing degree, for instance, is decreased to a level that

[1] The SHT and the DHT of a cache block are identical for the maximum sharing degree (Max SD = 16 for 16-tile CMP)

doesn't satisfy the cache demand of a running process, then MR can significantly increase. This would cause performance degradation if the cache configuration fails to offset the incurred latency of the larger MR from the saved latency of the smaller AAL. Equation (1) defines a metric, referred to as the average L1 miss time (AMT_{L1}), that combines both AAL and MR. The Average Memory Access Time (AMAT) metric defined in equation (2) combines all the main factors of system performance. An improvement in AMAT typically translates into an improvement in system performance. However, as L1 caches are kept private and have fixed access time, an improvement in the AMT_{L1} metric also typically translates into an improvement in system performance.

$$AMT_{L1} = AAL_{L2} + MissRate_{L2} \times MissPenalty_{L2} \quad \textbf{(1)}$$

$$AMAT = (1 - MissRate_{L1}) \times HitTime_{L1} + MissRate_{L1} \times AMT_{L1} \quad \textbf{(2)}$$

7.3.2 The Proposed Scheme

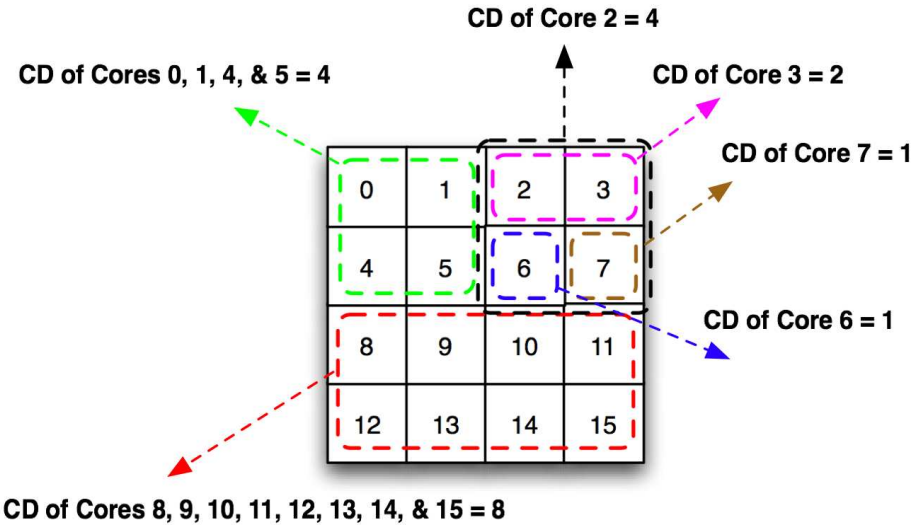


Figure 48: **A possible cache clustering configuration that the DCC scheme can select dynamically at runtime.**

This chapter suggests a cache design that can dynamically tune the AAL and MR metrics with the objective of providing a good system performance. Let us denote the L2 cache banks

that a specific CMP core, i, can map cache blocks to, and consequently locate them from, as the *cache cluster* of core i. Let us further denote the number of banks that the cache cluster of core i consists of as *cache cluster dimension* of core i (CD_i). In a 16-tile CMP, the value of CD_i can be 1, 2, 4, 8, and 16, thus generating cache clusters encompassing 1, 2, 4, 8, or 16 L2 banks, respectively. I seek to improve system performance by allowing cache clusters to independently expand or contract depending on cache demands of the working sets. I note that, for a certain working set, even the best performing of the 5 static cache designs (FS1, FS2, FS4, FS8, and FS16) could fail to hit optimal system performance. This is due to the fact that all CMP cores in these designs have the same sharing degree SD_i equal to either 1, 2, 4, 8, or 16. That is, two cores can't have different cluster dimensions. A possible optimal configuration (cache clustering) at a certain runtime point could be similar to the one shown in Fig. 48 or to any other eligible cache clustering configuration. A key feature of my DCC scheme is that it synergistically selects at run time a cache cluster for a core i that appears optimal to the currently undergoing cache demand of a program running on top of i. As such, DCC keeps seeking a *just-in-time* (JIT) near optimal cache clustering organization from amongst all the possible configurations. To the best of my knowledge, this is the first proposal to suggest such a fine-grained caching solution for the CMP cache management problem.

Given N executing processes (threads) on a CMP platform, I define the problem on-hand as the one of deciding the best cache cluster *for each single core* to minimize the overall AMAT of the N running processes. Let CC_i denote the current cache cluster of the i-th core, and $AMAT_i$ denote the Average Memory Access Time produced by a thread running on the i-th core. CC_i is allowed to be dynamically resized. Let the time at which CC_i is checked for an eligibility to be resized be referred to as a potential *re-clustering* point of CC_i. A potential re-clustering point occurs every fixed period of time, T. Although I use a 16-tile CMP model in this paper, in general, the *cache clustering* of n CMP cores over a period time T can be represented by the set $\{CC_0, \ldots CC_i, \ldots CC_{n-1}\}$. An optimal cache clustering for a CMP platform would minimize the following expression:

$$\textit{Total AMAT over time period } T = \Sigma_{i=0}^{n-1} AMAT_i$$

7.3.3 DCC Mapping Strategy

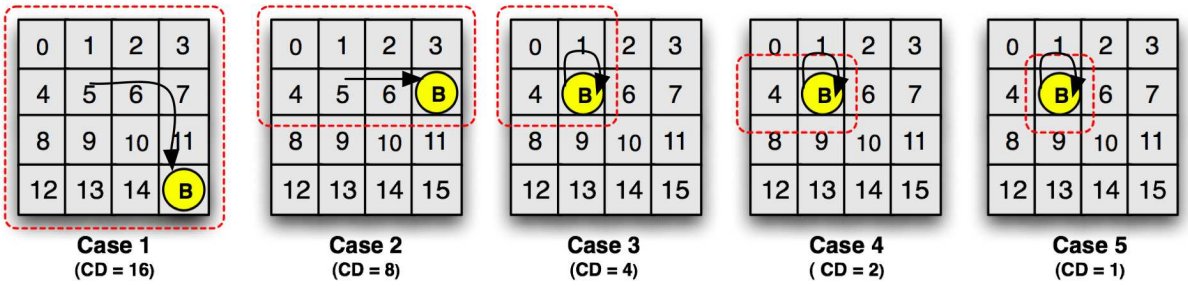


Figure 49: **An example of how the DCC mapping strategy works. Each case depicts a possible DHT of the requested cache block B with HS = 1111 upon varying the cache cluster dimension (CD) of the requester core 5 (ID = 0101).**

Varying the cache cluster dimension (CD) of each core over time, via expansions and contractions, would require a function to map cache blocks to cache clusters exactly as required. I propose a function that can efficiently fulfill this objective for n = 16, however, that function can be easily extended to any n that is a power of 2. Furthermore, appropriate functions can be obtained for any n value. Assume that a core i requests a cache block B. If CD_i is smaller than 16, B is mapped to a dynamic home tile (DHT) different than the static home tile (SHT) of B. As described earlier, the SHT of B is simply determined by the home select (HS) bits of B's physical address (4 bits for my 16-tile CMP model). On the other hand, the DHT of B is selected depending on the cluster dimension, CD_i, of the requester core i. Thus, with CD_i smaller than 16 only a subset of bits from the HS field of B's physical address need to be utilized to determine B's DHT. Specifically, 3 bits from HS are used if $CD_i = 8$, 2 bits if $CD_i = 4$, 1 bit if $CD_i = 2$, and no bits are used if $CD_i = 1$. More formally, the following function determines the DHT of B:

$$DHT = (HS\&MB) + (ID\&\overline{MB}) \qquad \textit{(3)}$$

Cache Cluster Dimension (CD)	Masking Bits (MB)
1	0000
2	0001
4	0101
8	0111
16	1111

Table 11: Masking Bits (MB) for a 16-tile CMP Model.

where ID is the binary representation of i, MB is a mask specified by the value of CD_i as illustrated in Table 11, $\overline{MB}$ is the complement of MB, and & and + are the bit-wise AND and OR operations, respectively. Fig. 49 illustrates an example for a cache block B with HS = 1111 requested by core 5. The figure depicts the 5 cases of the 5 possible CDs of core 5 (1, 2, 4, 8, and 16). The DHT of B for each of the possible CDs is determined using equation (3). For instance, with CD = 16, core 5 maps B to DHT 15. Again, note that when CD = 16, the SHT and the DHT of B are the same. Similarly, with CDs of 8, 4, 2, and 1, core 5 maps B to DHTs 7, 5, 5, and 5 respectively.

7.3.4 DCC Algorithm

The AMAT metric defined in equation (2) could be utilized to judiciously gauge the benefit of varying the cache cluster dimension of a certain core, i. I suggest a runtime monitoring mechanism that can infer enough about a running process behavior and feed the collected information to an algorithm that can make related architecture-adaptive decisions. In particular, a process P starts running on core i with an initial cache cluster (i.e., $CD_i = 16$). After a period time T, the $AMAT_i$ experienced by P is evaluated and stored, and the cache cluster of core i is contracted (or expanded if chosen so and CD_i has started from a value smaller than 16). This is the initial $AMAT_i$ of P. At every potential *re-clustering* point a new $AMAT_i$ ($AMAT_i$ current) is evaluated and deducted from the previously stored $AMAT_i$ ($AMAT_i$ previous). Suppose, for instance, that a contraction action has been initially taken.

Accordingly, a resultant positive value of the difference means that $AMAT_i$ has degraded after contracting the cache cluster of core i. As such, we infer that P didn't actually benefit from the contraction process. On the other hand, a negative outcome means that $AMAT_i$ has improved after contracting the cache cluster of core i and we infer that P benefited in fact from the contraction process. Let $\triangle$ be defined as follows:

$$\triangle_i = AMAT_{i,current} - AMAT_{i,previous} \qquad (4)$$

The DCC Algorithm Executed on Each Core *i*

$\Delta_i = AMAT_{i,current} - AMAT_{i,previous}$
if $(\Delta_i > 0 \,\&\, |\Delta_i| > T_l)$
  $Cluster\_Dimension = Cluster\_Dimension \times 2$
else
if $(\Delta_i < 0 \,\&\, |\Delta_i| > T_g)$
  $Cluster\_Dimension = Cluster\_Dimension / 2$

Figure 50: **The dynamic cache clustering algorithm.**

Therefore, a positive $\triangle_i$ indicates a *loss* while a negative one indicates a *gain*. At every re-clustering point, the value of $\triangle_i$ is fed to the DCC algorithm executing on core i (the DCC algorithm is local to each CMP core). The DCC algorithm makes in return some architecture-adaptive decisions. Specifically, if the gain is less than a certain threshold, T_g, the DCC algorithm decides to keep the cache cluster as it is for the next period time T. However, if the gain is above T_g, the DCC algorithm decides to contract the cache cluster a step further, predicting that P is likely to gain more by the contraction process. On the other hand, if the loss is less than a certain threshold, T_l, the DCC algorithm decides to keep the cache cluster as it is for the next period time T. If the loss is above T_l, the DCC algorithm

decides to expand the cache cluster to its previous value (one step backward) assuming that P is currently experiencing a high cache demand. Fig. 50 shows the suggested algorithm.

7.3.5 DCC Location Strategy

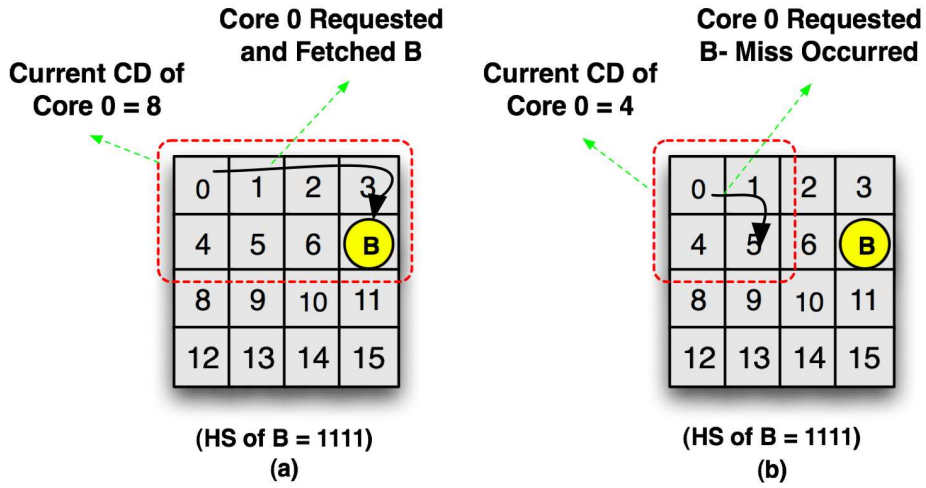


Figure 51: **An example of the DCC location strategy using equation (3). (a) Core 0 with current CD = 8 requesting and mapping a block B to DHT 7. (b) Core 0 missed B after contracting its CD from 8 to 4 banks.**

A core i can contract or expand its cache cluster at every re-clustering point. Hence, the generic mapping function defined in equation (3) can't be utilized straightforwardly to locate blocks that have been previously mapped by core i to the L2 cache space. Fig. 51(a) illustrates an example of core 0 (with CD = 8) fetched and mapped a cache block B (with HS=1111) to DHT 7 determined by equation (3). Fig. 51(b) demonstrates a scenario with core 0 contracting its CD from 8 to 4 and subsequently requesting B from L2. With current CD = 4, equation (3) designates tile 5 to be the current DHT of B. However, if core 0 simply sends its request to tile 5, a false L2 miss will occur. After a miss to tile 5, B's SHT (tile 15), which keeps B's directory information, can be accessed to locate B at tile 7 (assuming this is the only tile currently hosting B). This is a quite expensive process as it requires multiple inter-tile communications between tiles 0, 5, 15, 7 again, and eventually 0 to fulfill the request. A better solution could be to straightforwardly send the L2 request to B's SHT

instead of sending it first to B's current DHT and then possibly to B's SHT. This still might not be acceptable because it entails 3-way cache-to-cache communications between tiles 0, 15, and a prospective host of B. Such a strategy fails to exploit *distance locality*. That is, it incurs significant latency to reach the SHT of B though B resides in close proximity. A third possible solution could be to re-copy all the blocks that correlate to core 0 to its updated cache cluster upon every re-clustering action. Clearly, this is a costly and complex process because it will heavily burden the NoC with superfluous data messages.

A better solution to the location problem is to send simultaneous requests to *only* the tiles that are potential DHTs of B. The possible DHTs of B can be easily determined by varying MB and $\overline{MB}$ of equation (3) for the range of CDs, 1, 2, 4, 8, and 16. As such, the maximum number of possible DHTs, or the upper bound, would be 5, manifested when HS of B equals to 1111. On the other hand, the lower bound on the number of L2 accesses required to locate B at a DHT is 1. This would be accomplished when both, the HS of B and ID are equal to 0000 (If ID $\neq$ 0, number of L2 accesses $\neq$ 1). In general, the lower and upper bounds on the number of accesses that our proposed DCC location strategy requires to satisfy an L2 request from a possible DHT are $\Omega(1)$ and $O(log_2(NumberofTiles)) + 1$, respectively.

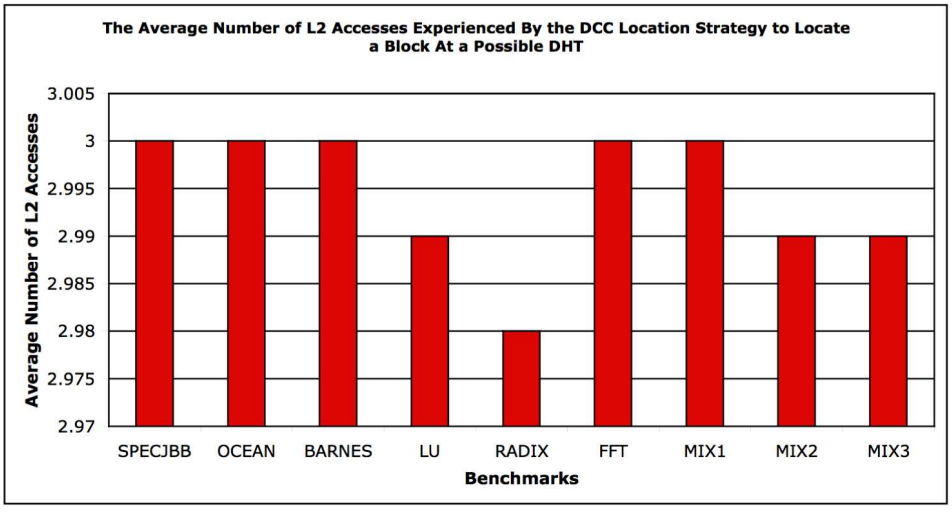


Figure 52: **The average behavior of the DCC location strategy.**

Given that the number of possible DHTs for a given block, B, depends on the HS bits

of B's physical address, it would be interesting to determine the *average* number of possible DHTs for all the blocks in the address space. To derive this number, let $AV(d)$ denote the average number of possible DHTs for all the blocks in the address space corresponding to cluster sizes 2^0, 2^1, ..., 2^d. If we add 2^{d+1}, half of the blocks in the address space will have a new DHT, while the new DHT of the other half of the blocks will coincide with the DHT of these blocks in the cluster of size 2^d. In other words,

$$AV(d+1) = \tfrac{1}{2}AV(d) + \tfrac{1}{2}(AV(d)+1) = AV(d) + \tfrac{1}{2} \qquad \textbf{(5)}$$

If CD = 1, each block has only one DHT, that is,

$$AV(1) = 1 \qquad \textbf{(6)}$$

Solving the recursive equations (5) and (6) yields,

$$AV(d) = 1 + \tfrac{1}{2}d \qquad \textbf{(7)}$$

For a CMP with n tiles, the number of possible cluster dimensions is $ln(n)$. Hence, the average number of possible DHTs is $1+\frac{1}{2}ln(n)$. Specifically, for n = 16, the average number of possible DHTs is $1+\frac{1}{2}ln(16) = 3$. Fig. 52 shows simulation results for the average number of L2 accesses experienced by the DCC location strategy using 9 benchmarks (details about the benchmarks and the utilized experimental parameters are described in Section 7.4). Clearly, the results confirm our theoretical analysis.

Multiple copies of a cache block B can map to multiple cache clusters of multiple cores. As such, a request from a core C to a block B can hit at multiple possible DHTs. However, if a miss occurs at the DHT of B that corresponds to the current cache cluster dimension of C (current DHT), though a hit occurs at some other possible DHT, a decision is to be made of whether to copy B to B's current DHT or not. If *none* of the possible DHTs that host B resides currently inside the cache cluster of C, B is copied to its current DHT, otherwise, it is not. The rationale behind this policy is to minimize the average L2 access

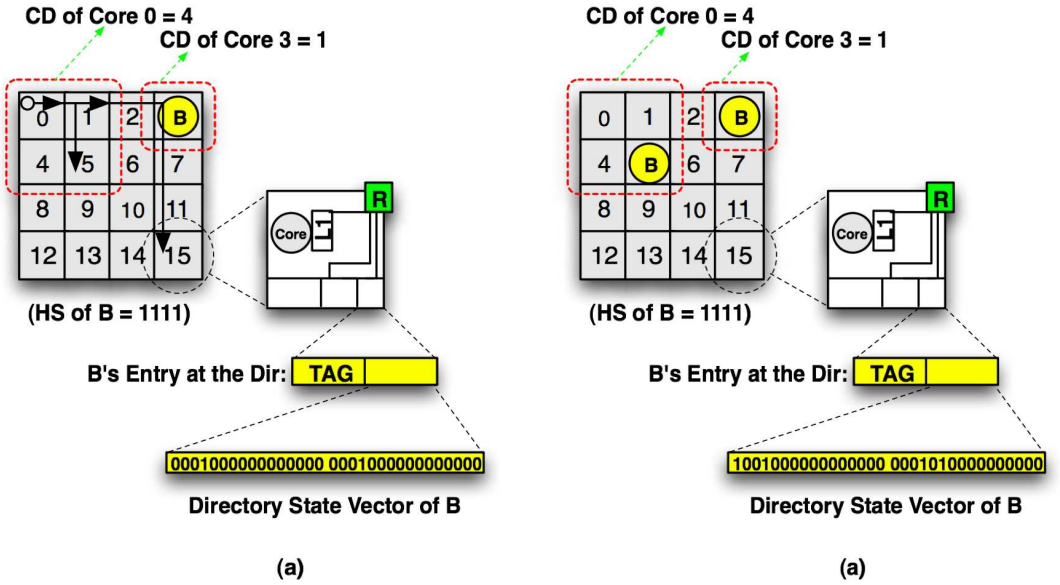


Figure 53: **A demonstration of an L2 request satisfied by a neighboring cache cluster. (a) Core 0 issued an L2 request to block B. (b) Core 3 satisfied the L2 request of Core 0 after re-transmitted to it by B's SHT (tile 15).**

latency. Specifically, a possible DHT hosting B and contained inside C's cache cluster is always *closer* to C than is the current DHT. Thus, B is not copied from that possible DHT to its current DHT. The decision of whether to copy B to its current DHT can be made by B's SHT. The SHT of B retains B's directory information and is always accessed by my location strategy (B's SHT is a possible DHT).

Finally, after inspecting B's SHT, if a copy of B is located on-chip (i.e, mapped by a different core with different CD) and none of the possible DHTs is found to host B, the SHT satisfies the request from the host that is *closest* to C (in case many hosts are located). Fig. 53(a) illustrates a scenario where core 0 with CD = 4 issues a request to cache block B with HS= 1111. Simultaneous L2 requests are sent to all the possible DHTs of B (tiles 0, 1, 5, 7, and 15). Misses occur at all of them. The directory table at B's SHT (tile 15) is inspected. A copy of B is located at tile 3 indicated by the corresponding bit within the directory status vector of B. Fig. 53(b) depicts B's directory state and residences after it has been forwarded from tile 3 to its current DHT (tile 5) and to the L1 cache of the requester

core 0. The figure depicts only copies at the L2 banks within tiles. However, the shown directory status vector reflects the presence of B at the L1 cache of core 0.

7.3.6 Scalability

DCC utilizes the AMAT metric defined in equation (2) to judiciously gauge the benefit of varying the cache cluster dimension of each core. The AMAT metric optimizes both, the L2 miss rate and the average L2 access latency. Each cluster consists of tiles that are spatially close to each other. Furthermore, most data sharing among threads will occur per each cluster, C, and even when none of the possible DHTs that host a requested block, B, resides currently inside C, B is copied to C's current DHT. In addition, the number of messages required by the DCC's location strategy to locate cache blocks scales with log the number of cores as defined in equation (7). Therefore, as the number of cores per chip is increased, the performance of DCC is expected to scale successfully. With larger number of tiles, the NUCA and the interference misses problems which DCC attempts to tackle controllably are exposed more, and accordingly, larger benefit from DCC is anticipated.

7.4 QUANTITATIVE EVALUATION

The evaluation methodology I use in this chapter is the one described in Section 2.3. However, the system parameters are a little different. Table 12 shows a synopsis of the main architectural parameters. I compare the effectiveness of the DCC scheme against the 5 alternative static designs, FS1, FS2, FS4, FS8, and FS16, detailed in Section 7.2.1, and the cooperative caching scheme [12]. Cache modules with a distributed MESI-based directory protocol for all the evaluated schemes have been developed and plugged into Simics.

I use a mixture of multithreaded and multiprogramming workloads that is also a little different than the one presented in Table 4. Table 13 shows the data set and other important features of each of the 9 workloads I examine. Lastly, I ran Ocean, Barnes, Radix, and FFT in full and stopped the remaining benchmarks after a detailed simulation of 20 Billion

Component	Parameter
Cache Line Size	64 B
L1 I-Cache Size/Associativity	16KB/2way
L1 D-Cache Size/Associativity	16KB/2way
L1 Read Penalty (on hit per tile)	1 cycle
L1 Replacement Policy	LRU
L2 Cache Size/Associativity	512KB per L2 bank/16way
L2 Bank Access Penalty	12 cycles
L2 Replacement Policy	LRU
Latency Per Hop	3 cycles
Memory Latency	300 cycles

Table 12: System parameters

Name	Input
SPECjbb	Java HotSpot (TM) server VM v 1.5, 4 warehouses
Ocean	514×514 grid (16 threads)
Barnes	64K particles (16 threads)
Lu	2048×2048 matrix (16 threads)
Radix	3M integers (16 threads)
FFT	4M complex numbers (16 threads)
MIX1	Hmmer (reference) (16 copies)
MIX2	Sphinx (reference) (16 copies)
MIX3	Barnes, Lu, 2 Milc, 2 Mcf, 2 Bzip2, and 2 Hmmer

Table 13: **Benchmark programs**

Instructions.

7.4.1 Comparing With Fixed Schemes

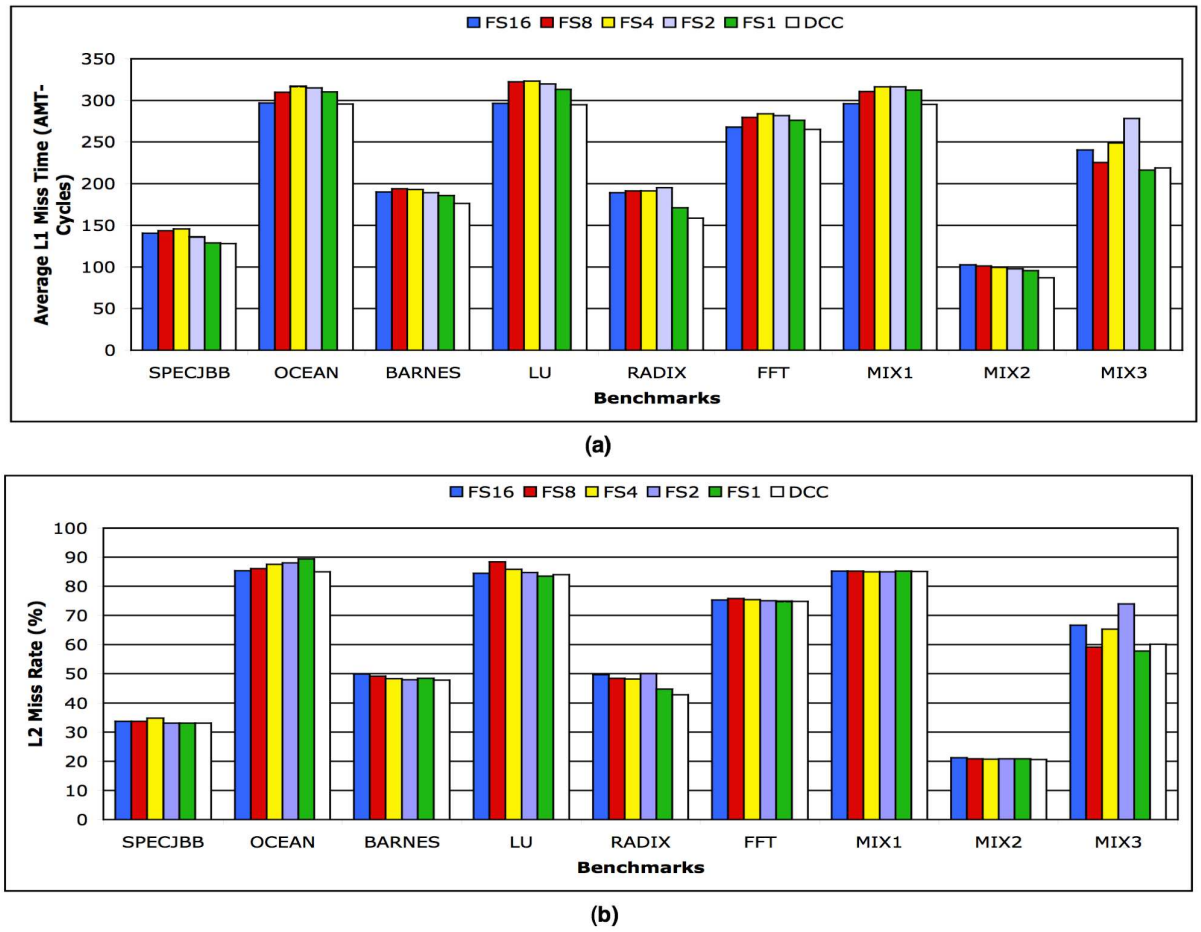


Figure 54: **Results for the simulated benchmarks. (a) Average L1 Miss Time (AMT) in cycles. (b) L2 Miss Rate.**

This section presents the experimental evaluation of the DCC scheme against the 5 alternative static designs, FS1, FS2, FS4, FS8, and FS16. The set of parameters, the period time T, the loss and gain thresholds T_l and T_g ($\{T, T_l, T_g\}$) utilized by the DCC algorithm are different for each simulated benchmark and selected from amongst 10 sets presented in the next subsection. The sensitivity analysis in Section 5.3 shows that the results are not

much dependent on the value of parameters $\{T, T_l, T_g\}$. First of all, I study the effect of the average L1 miss time (AMT), defined in equation (1), across the compared schemes. Fig. 54(a) portrays the AMTs experienced by the 9 simulated workloads. A main observation is that no single static scheme provides the best AMT for all the benchmarks. For instance, Ocean and MIX1 are best performing under FS16. On the other hand, SPECjbb and Barnes perform superlative under FS1. As such, a single static scheme fails to adapt to the varieties across the working sets. The DCC scheme, however, dynamically adapts to the irregularities exposed by different working sets and always provides performance comparable to the best static alternative. Besides, the DCC scheme sometimes even surpasses the best static option due to the fine-grained caching solution it offers (see Subsection 4.2). This is clearly exhibited by SPECjbb, Ocean, Barnes, Radix, and MIX2 benchmarks. Fig. 54(a) illustrates the outperformance of DCC over FS16, FS8, FS4, FS2, and FS1 by an average of 6.5%, 8.6%, 10.1%, 10%, and 4.5% respectively across all benchmarks, and to an extent of 21.3% for MIX3 over FS2. In fact, DCC surpasses FS1, FS2, FS4, FS8, and FS16 for all the simulated benchmarks except one. That is, MIX3 running under FS1. The current version of the DCC algorithm doesn't adaptively select an optimal set of thresholds $\{T, T_l, T_g\}$. Thus, I expect that such a diminutive superiority (1.1 %) of FS1 over DCC for MIX3 is simply because of that reason. Nevertheless, DCC always favorably converges to the best static option.

The DCC scheme manages to reduce the L2 miss rate (MR) as it varies cache clusters per cores depending on their L2 demands. Fig. 54(b) illustrates the MR produced by each of the 6 compared schemes for the simulated benchmarks. As described earlier, when the sharing degree (SD) amongst the static designs decreases, MR increases. This is because the likelihood that a shared cache block maps within the same shared cache region decreases. For instance, the L2 miss rate of Ocean increases monotonically as SD decreases. On the other hand, the L2 miss rate of Radix outshines with FS1. This is due to the fact that additional cache resources might not always correlate to better L2 miss rates [**?**]. A workload might manifest poor locality, and cache accesses could sometimes be ill-distributed over sets. I observed that Radix has a great deal of L2 misses produced by heavy interferences of cores on cache sets (inter-processor misses). The DCC scheme, however, efficiently resolves this

problem and resourcefully exploits the available cache capacity. DCC improves the Radix L2 miss rate by 4.2% and generates 7.3% better AMT.

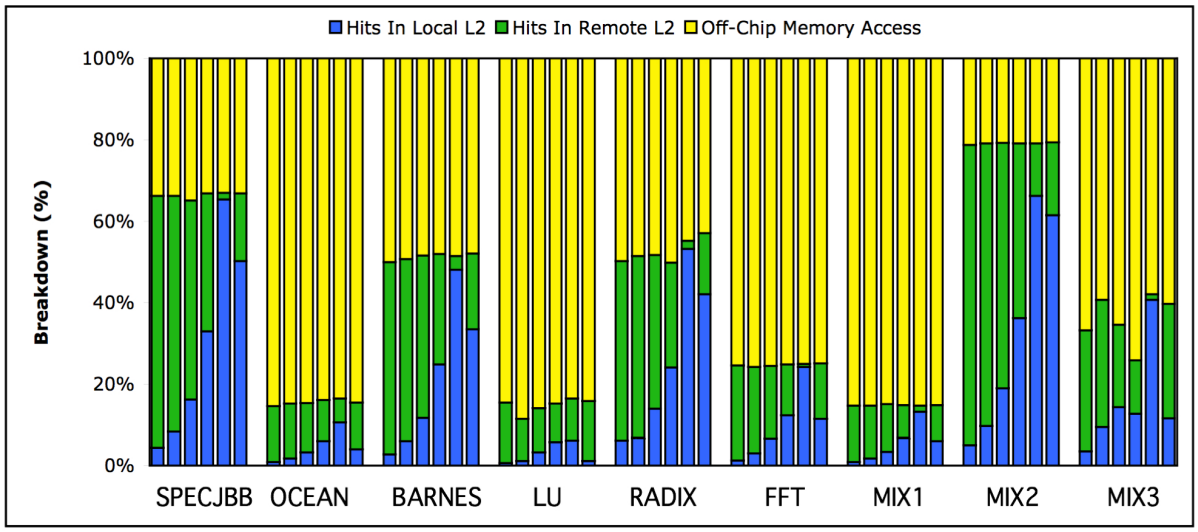


Figure 55: **Memory access breakdown. Moving from left to right, the 6 bars for each benchmark are for FS16, FS8, FS4, FS2, FS1, and DCC schemes, respectively.**

As the sharing degree (SD) of the static designs and the cache cluster dimension (CD) of the DCC scheme change, the hits to local L2 and to remote L2 banks also change. The hits to local L2 banks monotonically increase as SD decreases. This is revealed in Fig. 55 that depicts the data accesses breakdown of all the simulated benchmarks. Increases in hits to local L2 banks improves the average L2 access latency (AAL) as it decreases inter-tile communications, but, on the other hand, it might exacerbate MR thus causing both, AAL and MR to race in conflicting directions. For instance, though FS1 produces the best local L2 hits for Ocean, Fig. 54(b) shows that Ocean has the worst MR. Increasingly mapping cache blocks to local L2 banks can boost capacity misses, and if the gain acquired from higher local hits doesn't offset the loss incurred from higher memory accesses, performance will degrade. This explains the AMT behavior of Ocean under FS1. DCC, however, increases hits to local L2 banks but in a controlled and balanced fashion that it doesn't increase MR to an extent that ruins AMT. Thus, for instance, DCC degrades hits to local L2 banks of Ocean by 62.3%

over FS1 but improved in return its MR by 4.9%. As a result, DCC generated 4.7% better AMT for Ocean as compared to FS1. This reveals the robustness of DCC as a mechanism that tunes up AAL and MR so as to obtaining high performance from CMP platforms.

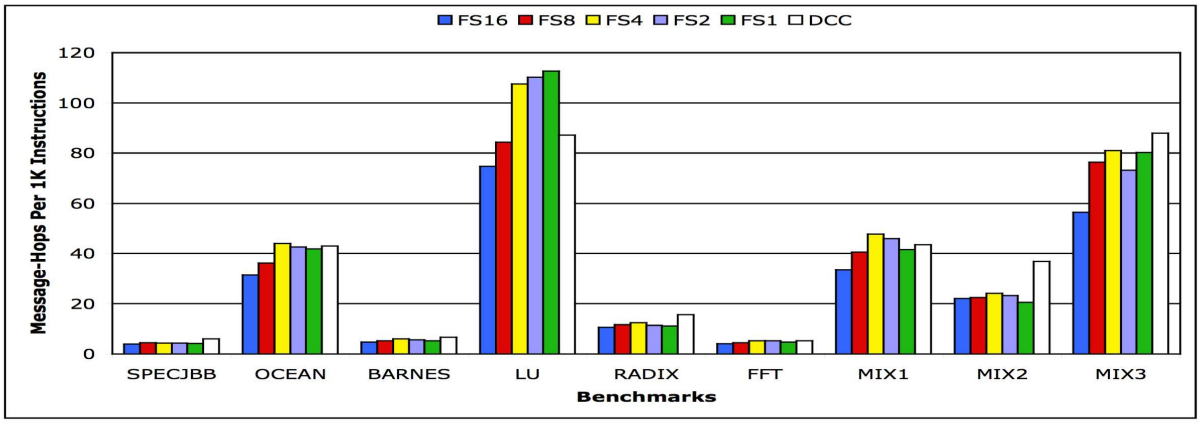


Figure 56: **On-Chip network traffic comparison.**

Fig. 56 depicts the number of message-hops (including both data and coherence) per 1k instructions for all the simulated applications with the 6 compared schemes. FS16 offers the preeminent on-chip network traffic savings (except for MIX2) as compared to other schemes. For each L1 miss, FS16 issues always one corresponding L2 request and that is to the static home tile (SHT) of the requested cache block, B. In contrary, the number of L2 requests issued by the remaining static designs depends on the access type. For a write request, B's SHT is accessed (in addition to accessing the shared region of the requester core) in order for the requester core to obtain an exclusive ownership on B. Besides, for a read request which misses at the shared region, B's SHT is also accessed to check if B resides on-chip (on some other shared regions) before an L2 miss is reported. However, for read requests that hit in the shared regions, an L1 miss corresponds always to a single L2 request. As such, if the message-hops gain (G) from read hits surpasses the message-hops loss (L) from read misses and writes, the interconnect traffic outcome of either FS1, FS2, FS4, or FS8 will improve over FS16. This explains the behavior of MIX2 with FS1. On the other hand, if L surpasses G, the interconnect traffic outcome of FS16 will improve over the 4 alternative static schemes. This explains the behavior of the remaining benchmarks. Finally, DCC

results in increased traffic due to multicast location requests (on average 3 per request). On average, DCC increases interconnect traffic by 41.1%, 24.7%, 11.7%, 16.6%, and 21.5% over FS16, FS8, FS4, FS2, and FS1, respectively. This increase in message-hops doesn't effectively hinder DCC from outperforming the static designs as demonstrated in Fig. 54(a).

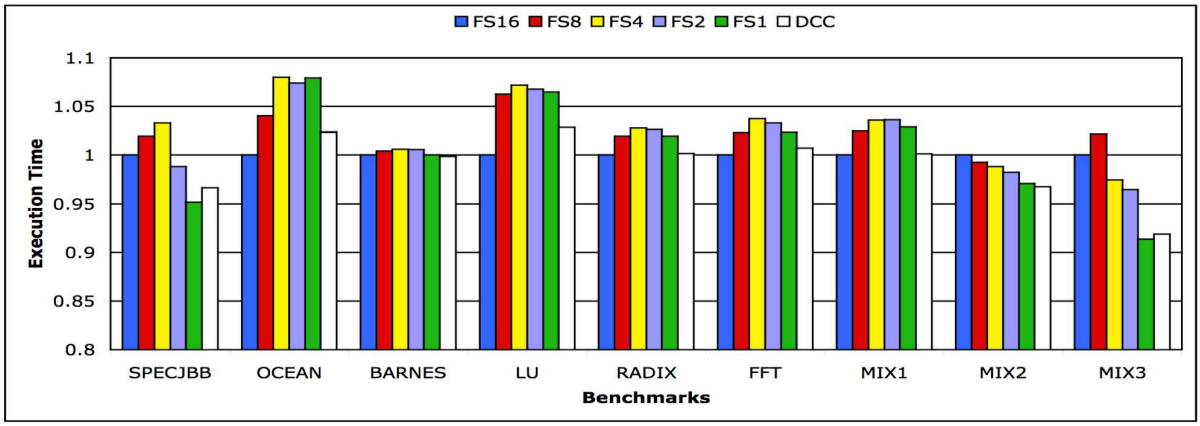


Figure 57: **Execution time (Normalized to FS16).**

Lastly, Fig. 57 presents the execution time of the compared schemes, all normalized to FS16. For Barnes, Radix, MIX1, MIX2, and MIX3, the superiority of DCC in AMT over the static designs translates to better overall performance. However, diminutive AMT improvements of DCC by 0.6% over FS1, 0.5% over FS16, 0.6% over FS16, and 0.9% over FS16 for SPECjbb, Ocean, Lu, and FFT, respectively didn't translate to an effectively better overall performance. Nonetheless, the main objective of DCC is still successfully met. DCC performs favorably comparable to the best static alternative. DCC outperforms FS16, FS8, FS4, FS2, FS1 by an average of 0.9%, 3.1%, 3.6%, 2.8%, and 1.4%, respectively across all benchmarks, and to an extent of 10% for MIX3 over FS8. DCC is expected to way surpass all static strategies had it adaptively selected $\{T, T_l, T_g\}$ parameters. As such, DCC could provide more accurate estimations regarding expansions and contractions. Having established the effectiveness of DCC as a scheme that can synergistically adapt to irregularities exposed by different working sets and within a single working set, proposing an adaptive mechanism for selecting the $\{T, T_l, T_g\}$ thresholds is an obvious next step.

7.4.2 Sensitivity Study

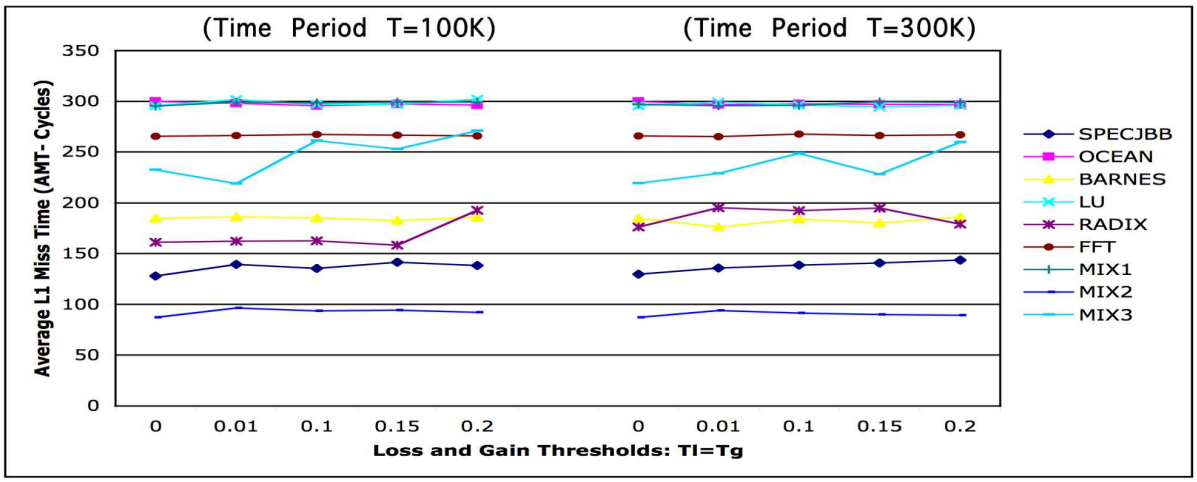


Figure 58: **DCC sensitivity to different T, Tl, and Tg values.**

The DCC algorithm utilizes the set of parameters $\{T, T_l, T_g\}$ to controllably tune up cache clusters and avoid potential noise that might hurt performance. As the current version of the algorithm assumes a fixed set of these parameters, I offer a study of DCC sensitivity to different $\{T, T_l, T_g\}$ values. Ten sets have been simulated, five with $T = 10,000$ (T1), and another five with $T = 300,000$ (T2) instructions. T_l and T_g were assigned values 0, 0.01, 0.1, 0.15, and 0.2 and ran with both, T1 and T2. Fig. 58 portrays the study outcome. A main conclusion is that no single fixed set of parameters provides superlative AMT for all the simulated benchmarks. For instance, SPECjbb performs best with T1 and $T_l = T_g = 0$. On the other hand, Barnes performs best with T2 and $T_l = T_g = 0.01$.

Overall, the DCC results with T1 are better than those with T2. Essentially, performance deteriorates when the partition period is too short or too long. Short partitions can hurt the accuracy of an estimation regarding a working set phase change. Long partitions, in contrary, can delay a detection of a phase change. The DCC algorithm doesn't expand or contract cache clusters upon every possible re-clustering point. It just checks the eligibility of an expansion or contraction step, and if found beneficial takes the action. Thus, DCC takes re-clustering actions only safely. Fig. 59 demonstrates a time varying graph that shows

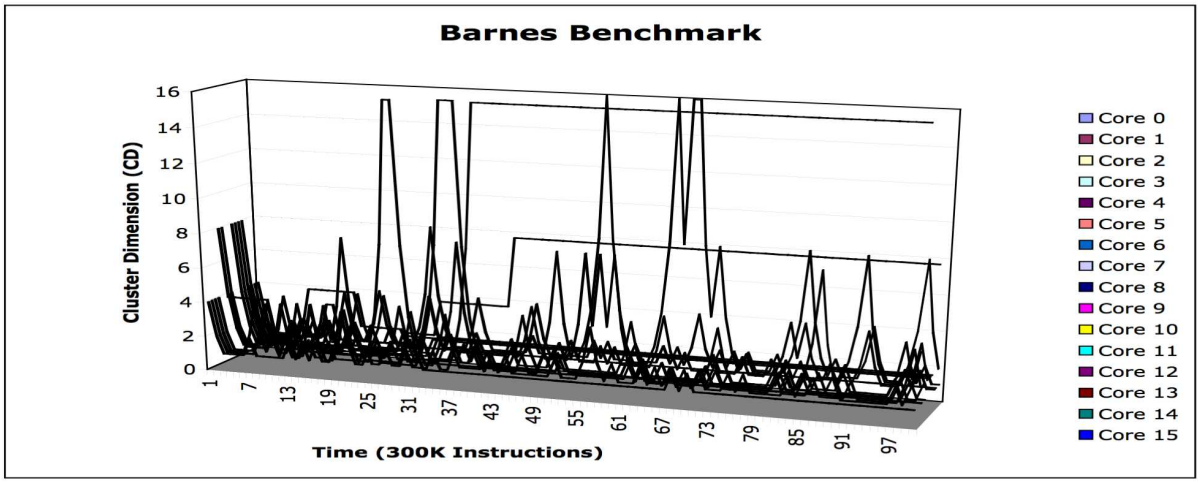


Figure 59: **Time varying graph showing the activity of the DCC algorithm.**

the activity of Barnes for 100 consecutive re-clustering points run under DCC with T2 and $T_l = T_g = 0.01$. A computation overhead of the DCC scheme at every re-clustering point is mainly that of computing the $\triangle$ metric, defined in equation (4). A performance overhead, on the other hand, can occur only if estimations about re-clustering actions fail. This is assumed, however, to be relatively little because of how the DCC algorithm inherently makes the architecture-adaptive decisions. This essentially explains why T1 yielded overall better DCC results than T2. The T1 moderate period of time attempts safely to capture a potential change in a program phase as soon as it emerges. I expect that with a time period smaller than T1, the information to feed to the DCC algorithm can be potentially skewed. As such, the estimations concerning program phases might possibly fail, and performance might, accordingly, degrade.

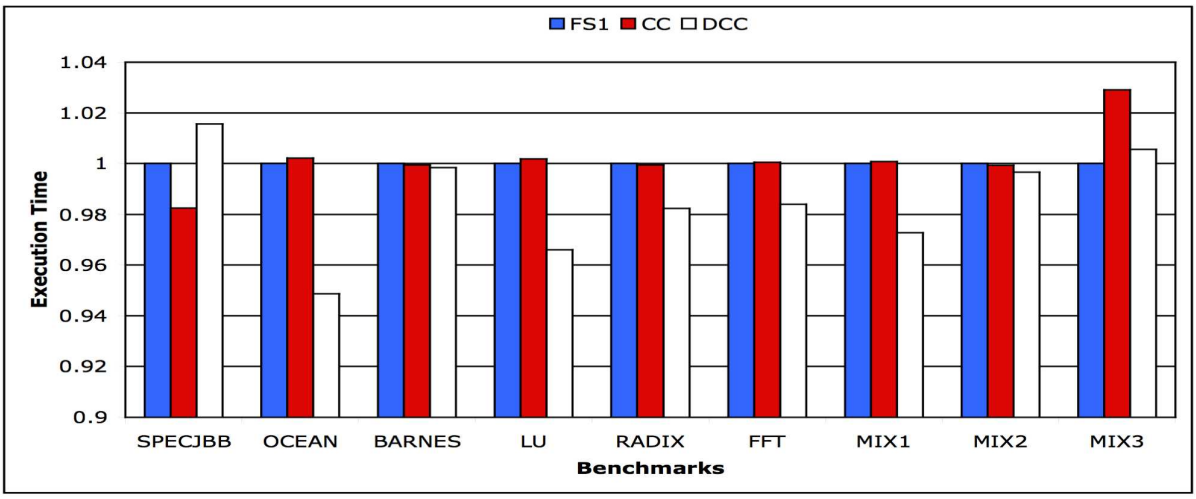


Figure 60: **Execution times of FS1, cooperative caching (CC), and DCC (normalized to FS1).**

7.4.3 Comparing With Cooperative Caching

This section presents a comparison between DCC and the related work, cooperative caching (CC) [12]. CC dynamically manages the aggregate on-chip L2 cache resources by combining the benefits of the private and shared schemes, AAL and MR, respectively. CC approaches the CMP cache management problem by basing its framework on the nominal private design and seeks to alleviate its implied capacity deficiency. If a block B is the only on-chip copy, CC refers to it as a *singlet*, otherwise as a *replicate* (because replications exist). To improve cache capacity, CC prefers to evict the following three classes of blocks in descending order: (1) an invalid block, (2) a replicate block, (3) and a singlet block. As such, CC refines cache capacity by reducing replicas as much as possible. Furthermore, CC employs spilling a singlet block from an L2 bank into another L2 bank for expected future usage. Fig. 60 demonstrates the execution time results of DCC and CC, both normalized to FS1. The shown CC is the default cooperative caching scheme that uses 100% cooperation probability (allows always the collection of the CC mechanisms to be used to optimize capacity). DCC always performs competitively, if not better, than the best static alternative. Thus DCC

performs sometimes equivalently to FS1 and sometimes surpasses it (in case it is not the best caching option). On the other hand, across all the simulated benchmarks, CC outperforms only SPECjbb by 1.7%. Surprisingly, CC degrades FS1 performance by 0.16%, on average. The reason is that CC uses the minimum replication level for each benchmark thus heavily affecting the average L2 access latency (AAL). Replication typically mitigates AAL if done controllably [6, 16].

7.5 SUMMARY

As the realm of CMP is continuously expanding, the pressure on the memory system to sustain the memory requirements of the wide variety of applications also expands. This chapter investigates the main problem with the current fixed CMP cache schemes as being unable to adapt to workloads variations, and proposes a robust alternative, the dynamic cache clustering (DCC) scheme. DCC suggests a mechanism that monitors the behavior of an executing program, and based upon its runtime cache demand makes related architecture-adaptive decisions. A per-core cache cluster comprised of a number of L2 banks can be constructed and dynamically expanded or contracted so as to tune the average L2 access latency and the L2 miss rate. Compared to static designs, the DCC scheme offered an average of 7.9% cache access latency improvement.

8.0 COMBINED SCHEMES

In this chapter I describe Dynamic Pressure and Distance Aware (DPDA) Placement and Dynamic Cache Clustering and Balancing (DCCB) schemes. DPDA augments PDA and ACM in one scheme so as to implement CC-FR's data placement and relocation components and address both interference misses and growing non-uniform access latencies challenges. In contrary, DCCB combines DCC and FSB to implement all CC-FR's approaches and address all the caching challenges presented in this thesis. In Section 8.1 I provide a motivational study and outline my proposed solution. I detail DPDA and DCCB in Section 8.2. A quantitative evaluation of DPDA and DCCB is presented in Section 8.3 and conclusions are given in Section 8.4.

8.1 MOTIVATION AND PROPOSED SOLUTION

8.1.1 Motivation

8.1.1.1 PDA and ACM Deciding upon the placement of a cache block from the first touch might be sub-optimal. For instance, though PDA places blocks at underutilized L2 banks that are closest to requesting cores, multiple other sharing cores can compete for the blocks after placed. Thus, PDA saves latency only for the initial requester and might, in contrary, degrade the average access latency for the subsequent sharing requesters. Ideally, we want to place a cache block at an L2 bank that best optimizes the overall access latencies from all the sharing cores. Clearly, the best host bank for a shared block can't be known until runtime.

Clearly, ACM is a strategy that can by employed to select better host banks for shared blocks at runtime. ACM synergistically monitors the access patterns of cores and periodically migrates blocks to banks that minimize the access time for the sharing cores. However, migrating a block to a host that best minimizes the overall access latency might be a little tricky especially if the selected host is highly pressured. Consequently, the L2 miss rate (MR) might be negatively affected in our attempt to save the average L2 access latency (AAL). As a result a performance degradation might be experienced. In summary, we want to save both AAL and MR controllably and effectively.

8.1.1.2 DCC and FSB None of the schemes that have been discussed until now implement all CC-FR's components. It is, in fact, desirable to explore the case of implementing all CC-FR's components and addressing all the caching challenges presented in this thesis. My suggested FSB scheme, which implements CC-FR's retention category, is orthogonal to all my proposed schemes and can be employed on top of any easily and straightforwardly so as to reduce interference misses. On the other hand, the DCC strategy implements two of CC-FR's components, data placement and relocation. However, DCC increases the aggregate cache footprint via allowing replication of shared cache blocks at multiple clusters. Replication in general results in reduced cache hit latency but may detrimentally affect MR if the capacity occupied by replicas increases significantly. FSB can be directly agumented to DCC in order to leverage cache capacity more effectively.

.

8.1.2 Proposed Solution

8.1.2.1 Dynamic Pressure and Distance Aware (DPDA) To make PDA a more practical scheme that can save latency, not only for the initial requesters but also for all the sharing cores, and controllably manage cache capacity, I propose combining ACM and PDA in one scheme and refer to it as the Dynamic Pressure and Distance Aware (DPDA) placement scheme. As a result, DPDA would implement CC-FR's data placement and relocation categories and addresses both interference misses and growing non-uniform access

latencies challenges.

8.1.2.2 Dynamic Cache Clustering and Balancing (DCCB) To explore the case of implementing all CC-FR's components and address all the caching challenges presented in this thesis, I suggest combining DCC and FSB together in one scheme and refer to it as the Dynamic Cache Clustering and Balancing (DCCB) scheme. Clearly, FSB is a pressure-aware strategy that seeks to reduce interference misses and can be applied to any of the proposed schemes. However, as ACM and PDA both apply only one of CC-FR's approaches, augmenting FSB to any of them would not fulfill the objective. On the other hand, DPDA do apply two of CC-FR's approaches and augmenting FSB to it would actually satisfy the goal. DPDA, however, already involves a pressure-aware strategy to reduce misses. In addition, DPDA involves various hardware complexities and requirements pertaining to the usage of the PDA scheme, the C-AMTE location strategy, and the ACM algorithm all in one. As such, applying FSB further to DPDA might not be justifiable. DCC, in contrary, is a unique in that by itself (as one single scheme with no already other added schemes) adopts two of CC-FR's approaches (i.e., placement and relocation). Besides, DCC allows replication which might negatively affect the cache capacity usage. FSB's job is to effectively leverage the cache capacity usage and accordingly reduce replication effects. Furthermore, FSB satisfies the third required CC-FR's component (i.e., retention) for DCC to become addressing all and can be simply and directly agumented; hence, I choose to apply FSB to DCC.

8.2 THE COMBINED SCHEMES

8.2.1 THE DYNAMIC PRESSURE AND DISTANCE AWARE (DPDA) PLACEMENT MECHANISM

As discussed earlier in Chapter 6, ACM computes the total latency cost for a given cache block on each of the potential hosts (L2 banks) and chooses the minimum. In order to

DPDA's Relocation Algorithm: Locate Host

```
min = Total latency cost of the current host
predicted_optimal = current host
for( i = 0 to i < number_of_tiles)
{
        cost = 0
        if( pressure(i) < LPL )
        {
            for( j = 0 to j < number_of_sharers)
                    cost += 2 * Manhattan_distance[j][i]

            if( cost < min )
                predicted_optimal = i
        }
}
```

Figure 61: **The DPDA relocation algorithm to locate a better host for a cache block.**

achieve this, ACMs algorithm keeps for each block, B, an access pattern that designates the sharing cores and after every migration frequency level (MFL) selects a new host for B that minimizes the overall L2 access latency. ACM can be straightforwardly augmented to PDA and B can be synergistically monitored and periodically migrated to a better host bank that optimizes the on-chip access latency. However, to control the cache capacity, B shouldn't be migrated to a highly pressured bank. In order to meet such an objective, we can utilize the low pressure limit (LPL) defined in Chapter 5, Section 5.2.1. A bank is not considered a potential host unless it is underutilized. A bank is underutilized if its pressure is below LPL. Consequently, DPDA's migration algorithm would compute the total latency cost for a given cache block on each of the underutilized banks and chooses the minimum. Fig. 61 shows DPDA's migration algorithm. The placement process of DPDA remains unaltered (i.e., as PDA's one).

By incorporating cache pressure with DPDA's migration algorithm, we need to have the

pressure values of banks (groups) available on chip. The baseline of DPDA, PDA maintains the pressure array at the memory controller(s). Nevertheless this suggests a simple solution. PDA quantifies pressures using hardware counters during a time interval, referred to as an epoch. At the end of every epoch on a tile, T, the values of the counters are copied from T to the pressure array at the memory controller(s). At the copy time, a bitmap can be populated and sent back to T indicating the banks (groups) that are underutilized. Corresponding to each bank (group) is a bit in the bitmap with 0 indicating a highly pressured bank and 1 indicating an underutilized one (or vice versa). A bitmap consists of n rows $\times$ p columns with n-group and p-bank pressure array (see Section 4.2.3 for details on collecting pressures at a group granularity). Note that the process of setting bitmaps is completely hidden under the copy time of pressure values from tiles. Besides, at the startup of programs a fully reset bitmap per tile is assumed (i.e., no migration is performed out of a tile until its corresponding bitmap is updated after copying pressure values collected during the first epoch).

8.2.2 THE DYNAMIC CACHE CLUSTERING AND BALANCING (DCCB) MECHANISM

The dynamic cache clustering and balancing (DCCB) scheme combines DCC and FSB. FSB is oriented towards last level caches and can, in fact, be applied straightforwardly to any caching scheme. When added to DCC, FSB can simply and as usual retain blocks evicted from highly pressured sets at underutilized sets within an L2 bank. In addition, FSB is triggered on DCC when a block is copied to the current dynamic home tile of the requesting core (see Section 7.3.5).

8.3 QUANTITATIVE EVALUATION

The evaluation methodology and the benchmark programs I use in this chapter are the ones described in Section 2.3. For DPDA, after every 20 million instructions, only 0.25 of the pressure values is kept (see Section 4.2.2). For DCCB, the parameters used by DCC (see

Section 7.3.4), T, T_l, and T_g are set to 10000, 0.01, and 0.01, respectively, corroborated by the sensitivity study in Section 7.4.2. Besides, augmented with DCC is FSB-4 with $\alpha = 0.25$ substantiated by studies in Sections 5.3.1 and 5.3.3. Both DPDA and DCCB are compared against the nominal shared (S) scheme. Furthermore, DPDA and DCCB are compared versus their baselines PDA and DCC, respectively.

8.3.1 Comparing DPDA Against the Shared NUCA and the PDA Designs

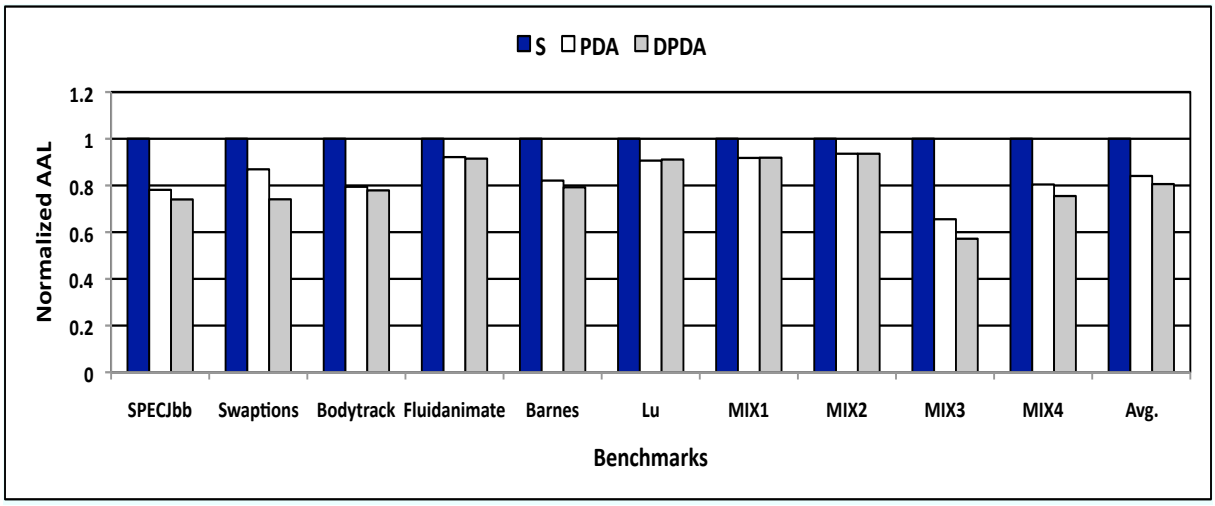


Figure 62: **Average L2 Access Latencies (AALs) of PDA, DPDA, and shared (S) schemes (normalized to S).**

Let me first compare DPDA against the baseline shared (S) and the PDA schemes. I assume a high pressure limit (HPL) and a low pressure limit (LPL) each with $\alpha = 0.25$ (corroborated by the sensitivity study in Section 4.3.3). Besides, I consider a tracking entries (TR) table with 16K entries. Each access to a TR table requires 1.35ns estimated using CACTI v5.3 [32]. Both PDA and DPDA are run with a 32-group granularity (Section 4.3.2 shows that dividing a bank into only 32 groups provides close benefits to dividing it into 512 groups). Finally, the migration frequency level (MFL) of DPDA is set to 6. Section 8.3.2 provides a sensitivity study of DPDA to different MFLs. Fig. 62 shows the average L2 access latency (AAL) of S, PDA, and DPDA normalized to S. On average, DPDA achieves an AAL

reduction of 19.4% over S, and by as much as 42.8% for the MIX3 workload. However, compared to PDA, DPDA provides an AAL improvement of only 4.4%, and by as much as 14.7% for the Swaptions program.

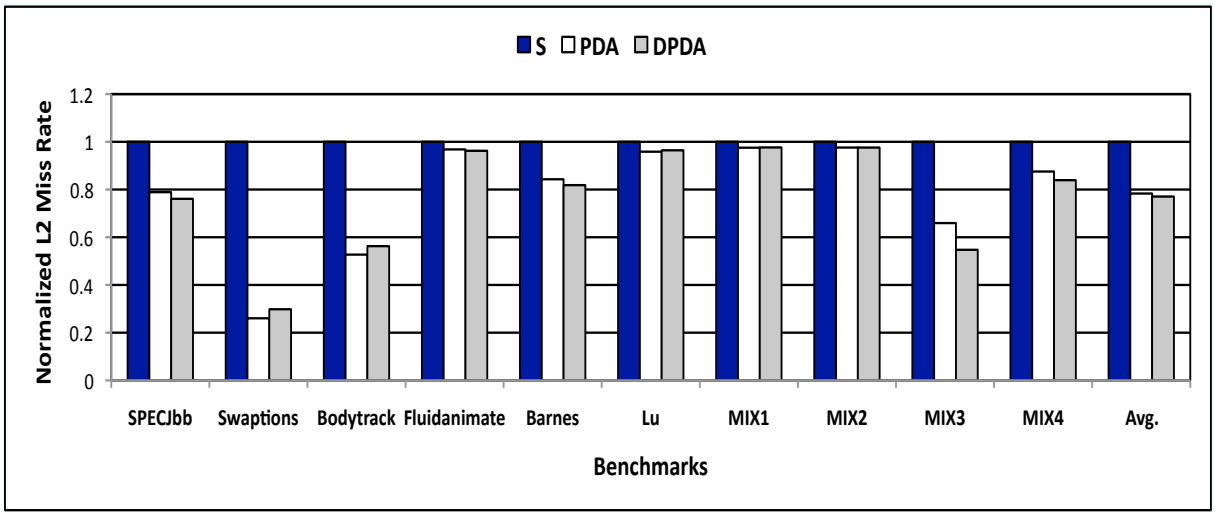


Figure 63: **L2 miss rates of PDA, DPDA, and shared (S) schemes (normalized to S).**

PDA alleviates AAL as it places cache blocks fetched from the main memory at underutilized L2 banks that are closest to the requesting cores. DPDA attempts to improve upon PDA by dynamically relocating the blocks to better hosts at runtime seeking to save more AAL while keeping cache capacity under control. Fig. 63 shows the L2 miss rates of S, PDA, and DPDA normalized to S. Clearly, for some benchmarks (e.g., MIX2) DPDA maintains to a very close extent the L2 miss rate provided by PDA. Nevertheless, for some other workloads (e.g., Barnes) DPDA improves upon PDA while for some others it shows a little degradation (e.g., Lu). Relocating blocks to different L2 banks might decrease or increase the L2 miss rate depending on whether the receiving banks are overall less or more pressured than the source banks (all the receiving banks are underutilized though). On average, DPDA achieves its L2 miss rate goal very successfully via preserving the reductions provided by PDA. DPDA produces further a miniature improvement over PDA (by an average of 0.6%).

The main goal of DPDA, however, is to mitigate AAL. Fig. 64 illustrates the reasons

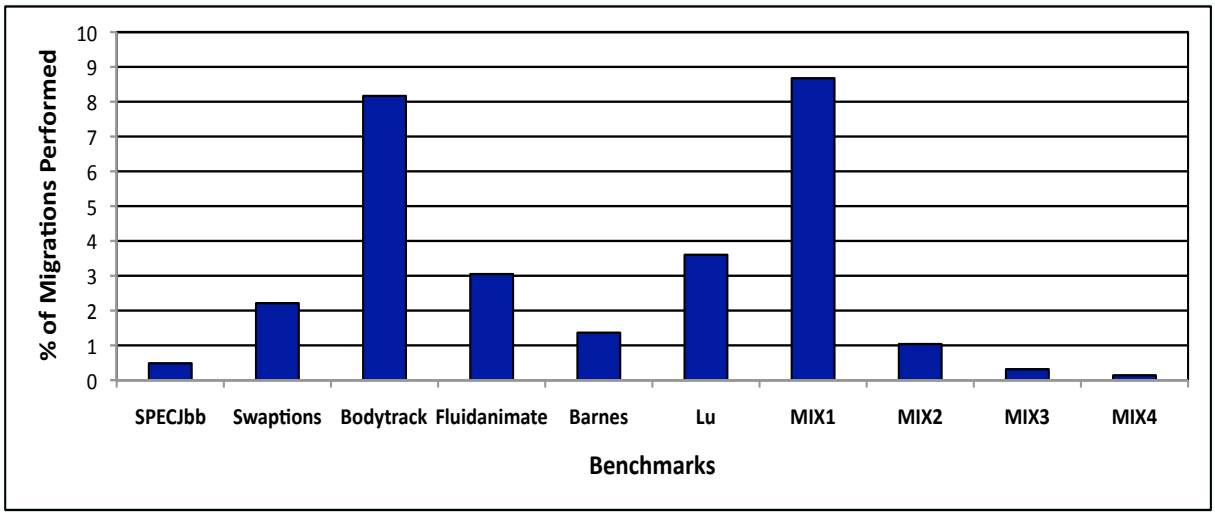


Figure 64: **The percentage of migrations performed by DPDA.**

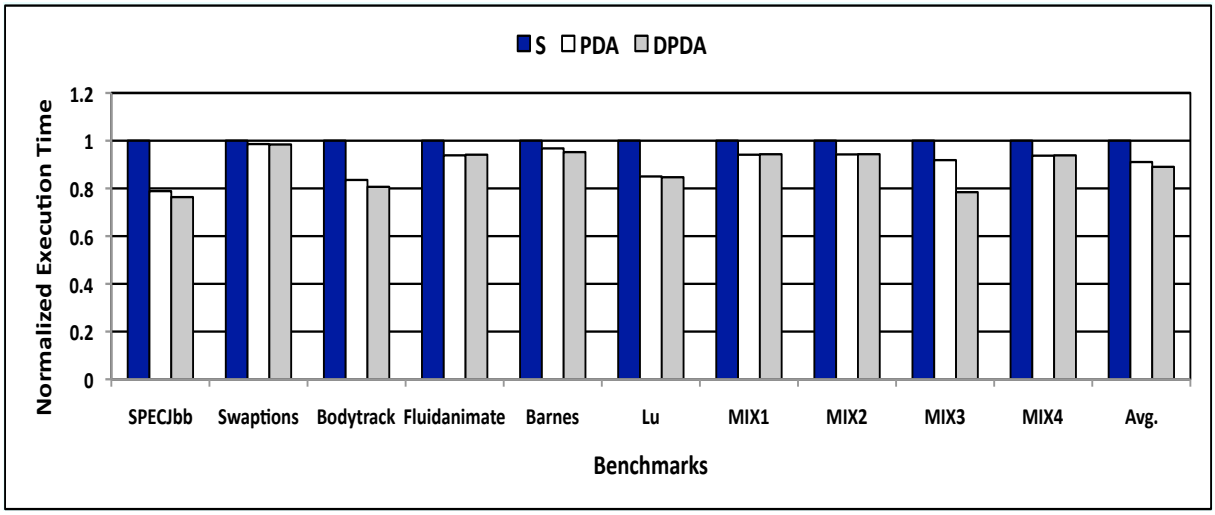


Figure 65: **Execution times of PDA, DPDA, and shared (S) schemes (normalized to S).**

for the diminutive AAL improvement accomplished by DPDA over PDA. Clearly, for all the benchmark programs DPDA finds in almost all the times (more than 90% of the times) that the cache blocks are either at the best L2 banks as predicted by the PDA or can't be migrated to banks closer to requesting cores due to exhibiting high pressures. Besides, some benchmarks (e.g., SPECJbb or MIX2) expose a little or no sharing degrees. Consequently, DPDA doesn't need to relocate blocks to the center of gravity from all the requesting cores. The initial requester core, C, might be the sole requester for a block, B, and PDA might have already successfully placed B in the vicinity of C (not successful for MIX1 as it is for MIX2). To that end, Fig. 65 presents the execution times of S, PDA, and DPDA normalized to S. Across all benchmarks, DPDA outperforms S and PDA by averages of 10.9% and 2.2%, respectively.

8.3.2 Sensitivity of DPDA to Different Migration Frequency Levels

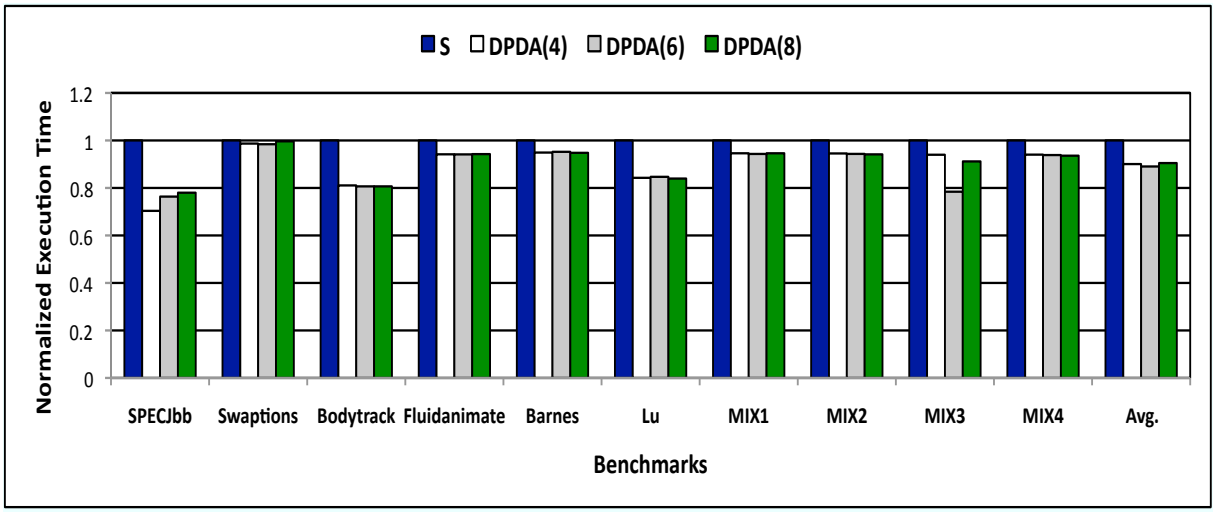


Figure 66: **The DPDA behavior with different migration frequency levels (MFLs). DPDA(4), DPDA(6), and DPDA(8) stand for DPDA with MFL values of 4, 6, and 8, respectively.**

So far, I have been using a migration frequency level (MFL) of 6 with DPDA. I tested DPDA with two more MFL values, particularly 4 and 8. Fig. 66 shows the outcome.

DPDA(4), DPDA(6), and DPDA(8) denote utilizing MFL values of 4, 6, and 8, respectively. As demonstrated in the figure, DPDA(4), DPDA(6), and DPDA(8) outperform S by averages of 9.9%, 10.9%, and 9.5%, respectively. DPDA(6) surpasses a little DPDA(4) and DPDA(8). With higher MFL values more blocks (blocks that are accessed at L2 for less than the MFL value before getting replaced) might miss the opportunity to be migrated to better hosts and, accordingly, result in less average L2 access latency savings. On the other hand, with lower MFL values more blocks might be relocated earlier on time before corresponding access patterns are accurately shaped. Overall, for the examined MFL values, we observe that DPDA performs best with an MFL value of 6.

8.3.3 Comparing DCCB Against the Shared NUCA and the DCC Designs

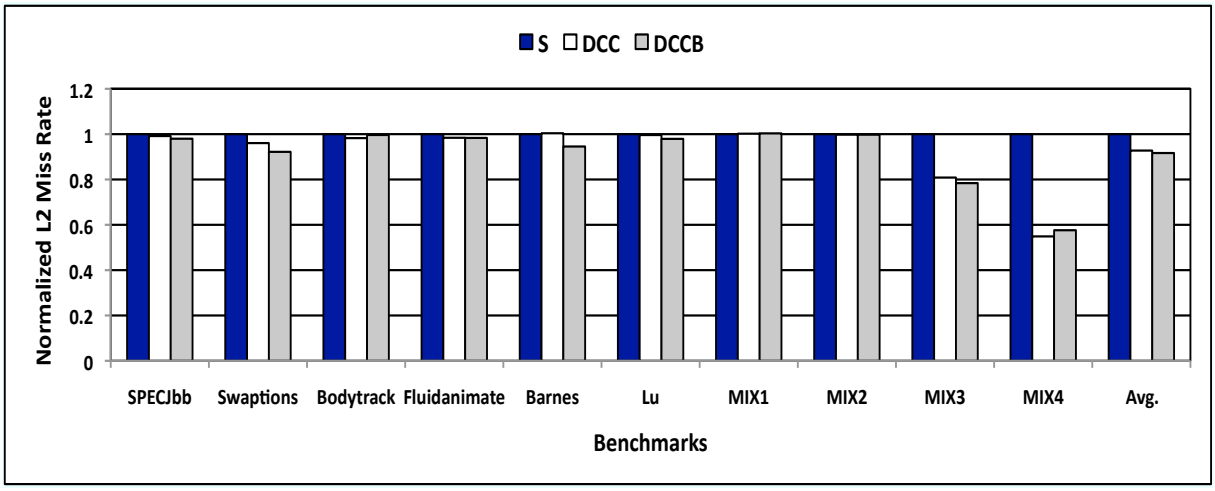


Figure 67: **L2 miss rates of shared (S), DCC, and DCCB schemes (normalized to S).**

Let me now compare the DCCB strategy against the nominal shared (S) and the DCC designs. Fig. 67 shows the L2 miss rates of S, DCC, and DCCB normalized to S. DCC provides an L2 miss reduction over S by an average of 7.2%. On the other hand, DCCB achieves L2 miss rate reductions over S and DCC by averages of 8.3% and 0.9%, respectively. Across all benchmarks, DCCB reduces the L2 miss rate of DCC except for Bodytrack and

MIX4. When DCC contracts or expands clusters it replicates cache blocks at new clusters (specifically at the dynamic home tiles or current DHTs) without invalidating the previously mapped blocks at the previous DHTs. These supplementary blocks are left for the LRU policy to replace in case they are not accessed for a while. Nonetheless, these blocks incur pressure on the available cache capacity and FSB might attempt to retain them when they are evicted. When retaining these blocks, FSB can potentially replace blocks that have no replicas and which may be requested in the future. Clearly, this will cause a little degradation in the L2 miss rate. Lastly, note that for the remaining benchmark programs DCCB provides L2 miss rate reductions over DCC only by little margins. This can be correlated to the fact that DCC already optimizes for the L2 miss rate in its algorithm as discussed in Section 7.3.4 though it entails replication at the L2 cache space.

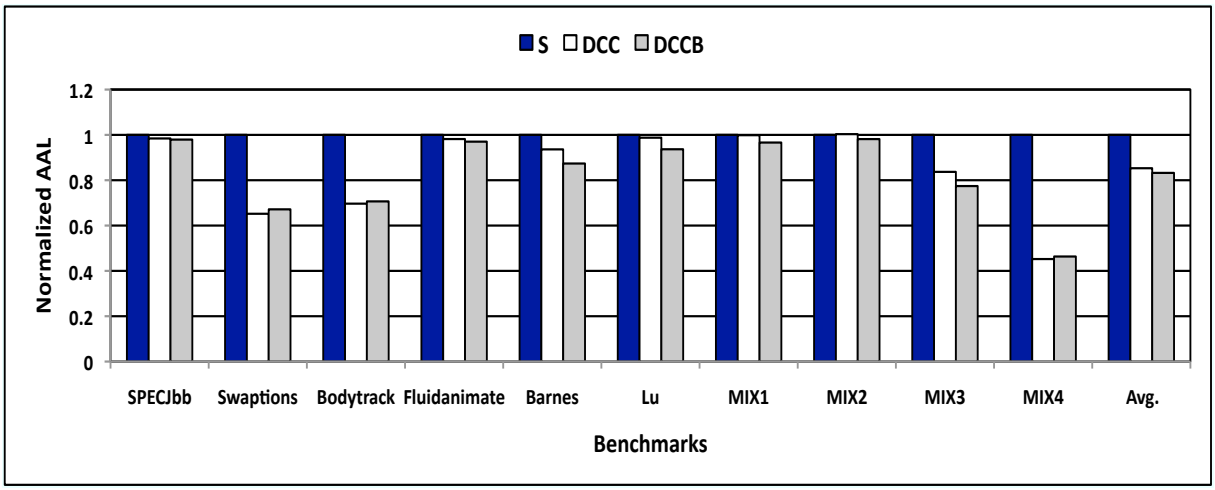


Figure 68: **Average L2 access latencies (AALs) of shared (S), DCC, and DCCB schemes (normalized to S).**

Fig. 68 demonstrates the average L2 access latency (AAL) of S, DCC, and DCCB normalized to S. DCC and DCCB accomplish AAL improvements over S by averages of 14.7% and 16.7%, respectively. For the Swaptions benchmark, although DCCB achieves more miss rate reduction than DCC, DCC provides more AAL improvement. This is due the latency overhead incurred by the FSB lookup mechanism (see Section 5.2.3). To that end, Fig. 69 depicts the execution times of S, DCC, and DCCB normalized to S. DCC and DCCB out-

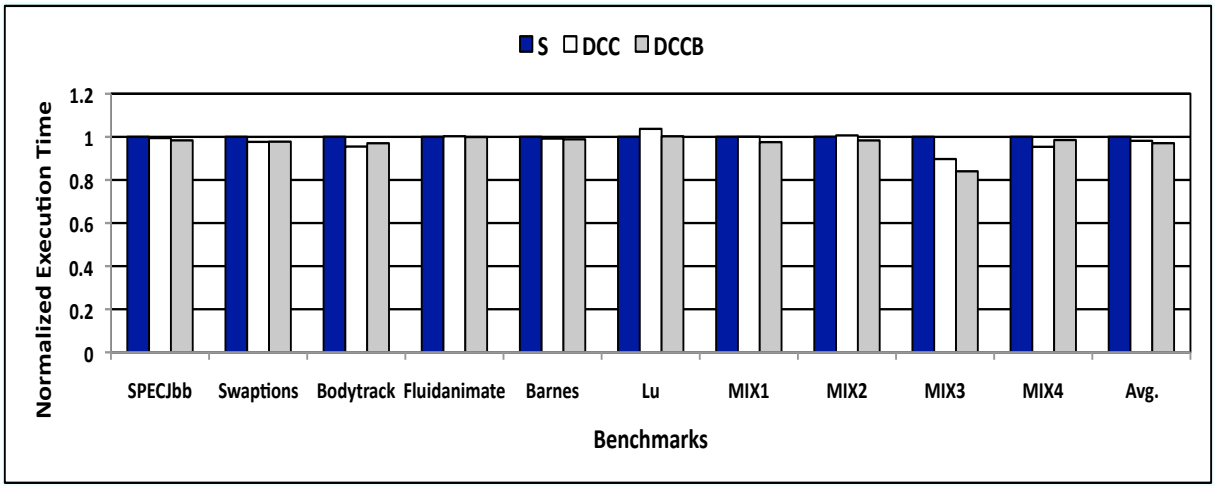


Figure 69: **Execution times of shared (S), DCC, and DCCB schemes (normalized to S).**

perform S by averages of 1.8% and 2.9%, respectively. Note that the objective of DCC (and subsequently DCCB) is not to improve upon S (specifically) but to converge to the best static alternative among the fixed schemes with sharing degrees of 1, 2, 4, 8, and 16 (for a 16-tile CMP) or FS1, FS2, FS4, FS8, and FS16 as denoted in Section 7.2.1. In section 7.4.1 it has been shown that no single fixed design can provide the best performance for all kinds of workloads. Hence, for some workloads FS16 (or S) might perform the best while for some others FS1, FS2, FS4, or FS8 might perform superlatively. DCC dynamically adapts to the irregularities exposed by different working sets and always provides performance comparable to the best static alternative which, for instance, might not be S (FS16) in the case of the Lu program.

Finally, we have seen that FSB demonstrates a high effectiveness under S (see Section 5.3.1) but not under DCC. *Clearly, a main observation is that implementing more components of CC-FR doesn't necessarily correlate to a monotonic improvement in system performance.* Nevertheless, the incurred hardware overhead from DCCB via agumenting DCC and FSB might not even justify the obtained little performance improvement.

8.4 SUMMARY

Deciding upon the placement of a cache block (especially if shared) from the first touch, as employed by PDA, might be sub-optimal. The best location for a cache block can't be known until runtime. ACM synergistically monitors the access patterns of cores and periodically migrates blocks to banks that minimize the access time for the sharing cores. As such, and to make PDA more practical, PDA and ACM are suggested to be combined in one scheme referred to as dynamic pressure and distance aware placement (DPDA). Accordingly, DPDA implements CC-FR's data placement and relocation categories and addresses interference misses and growing non-uniform access latencies challenges.

To employ all CC-FR's approaches and effectively address all the presented CMP caching challenges, DCC and FSB are suggested to be combined in one scheme referred to as the dynamic cache clustering and balancing (DCCB) scheme. With DCCB, system performance has been shown to be intensified but by a little margin. A conclusion has been drawn that implementing more components of CC-FR doesn't necessarily correlate to a monotonic improvement in system performance. Nevertheless, the incurred hardware overhead from DCCB via augmenting DCC and FSB might not even justify the obtained little performance improvement.

9.0 CONCLUSIONS

In this chapter I present a summary and draw my conclusions on work performed in this dissertation.

9.1 SUMMARY AND CONCLUSIONS

In this dissertation I presented a general CMP caching framework (CC-FR) that defines three components: (1) *data placement*, (2) *data retention*, and (3) *data relocation*. I claim that any proposed CMP caching scheme would implement one or more of CC-FR's components. CC-FR categorizes CMP caching schemes and specifies caching objectives that each scheme is attempting to satisfy. For instance, if the target is to alleviate interference misses then a scheme proposed in this direction would apply CC-FR's data placement and/or data retention approaches. On the other hand, if the objective is to reduce non-uniform access latencies then a suggested strategy in this direction would implement CC-FR's data relocation component.

Another framework that can be thought of is to simply define two components: (1) L2 miss rate and (2) average L2 access latency. It can then be claimed that the goal of any CMP cache management scheme is to improve one or both of these components. Such a framework is valid; however, it illustrates only the challenge(s) that a CMP caching scheme is attempting to address. On the other hand, CC-FR demonstrates the adopted strategy of the proposed scheme (i.e., placement, relocation, and/or retention) as well as the challenge(s) that the scheme is targeting. Clearly, this makes CC-FR more informative and instructive.

In Chapter 3, I proposed Constrained Associative-Mapping-of-Tracking-Entries (C-AMTE),

a scalable strategy to facilitate flexible and efficient distributed cache management in large-scale CMPs. C-AMTE enables data placement and relocation designs. It stores in each core tracking data structures to avoid on-chip interconnect traffic outburst or long distance directory lookups. Three schemes (i.e., PDA, ACM, and DPDA) in this thesis make use of C-AMTE to rapidly locate L2 cache blocks and mitigate the average L2 access latency. In fact, C-AMTE can be applied whenever associative mapping is used for cache blocks, either in case of one-to-one (i.e., migration) or one-to-many (i.e., replication). I believe that C-AMTE opens opportunites for architects to propose creative CMP cache management organizations as it precludes the necessity to stick to either the traditional private or shared paradigms (all the proposed CMP caching scheme in literature start from one of these two conventional schemes). I showed in Section 3.3 that C-AMTE is very effective in that it can achieve a cache access latency improvement by up to 34.4%, close to that of a perfect location strategy.

To implement CC-FR's data placement component and motivated by the large non-uniform distribution of memory accesses across cache sets in different L2 banks, Chapter 4 presented the Pressure and Distance Aware (PDA) placement strategy. Unlike conventional CMP caching schemes that involve blok placement that are oblivious to the disparity in the pressure on shared cache sets, PDA innovatively decouples the physical locations of cache blocks from their addresses and makes, consequently, the placement process fully aware of the cache capacity. Spatial pressure at the on-chip last-level cache is continuously collected at a group (comprised of local cache sets) granularity and periodically recorded at the memory controller(s) to guide the placement process. An incoming block is consequently placed at an underutilized cache group that is closest to the requesting core. Without C-AMTE, PDA might not be practical. PDA utilizes C-AMTE to rapidly locate L2 cache blocks on subsequent accesses. Simulation results show that PDA outperforms the nominal shared scheme by an average of 8.9% and by as much as 21.1% for the examined benchmark programs. In summary, PDA reveals the importance and the effectiveness of incorporating flexible (i.e., address independent) placement strategies accross banks in the CMP caching domain.

In Chapter 5, Flexible Set Balancing (FSB) is presented. While PDA exploits the large

asymmetry in cache set's usages across cache sets in different L2 banks, FSB explores a different direction where it leverages the non-uniformity of memory accesses in cache sets within a local L2 bank. FSB implements CC-FR's data retention component and addresses the interference misses challenge. Cache lines evicted from highly pressured sets are retained at underutilized ones (in the very same L2 bank) so as to satisfy far-flung reuses. FSB adapts to phase changes in programs and promotes a very flexible sharing among cache sets. An underutilized set is allowed to share its space by any stressed set during any point in a program's execution, a policy that I refer to as *one-from-many* sharing. Besides, many sets are allowed to share their capacities with a highly utilized set, a policy that I refer to as *many-from-one* sharing. FSB incurs a little storage, area, and energy overheads. FSB achieves an average miss rate reduction of 36.5% for the benchmark programs I examined. This produces an average execution time improvement of 13%. Furthermore, evaluations manifested the outperformance of FSB over some relevant designs including DSBC [55] and V-WAY [53]. In summary, FSB has been shown to be general, extensible, and practical in that it can be applied to single-core as well as multi-core architectures to reduce interference misses.

As the best host L2 bank for a cache block can't be known until runtime and in an attempt to minimize the average L2 access latency, Adaptive Controlled Migration (ACM) has been proposed in Chapter 6. ACM implements CC-FR's data relocation component and addresses mainly the growing non-uniform access latencies challenge. Migration in literature has been shown to be less effective for CMP caches than for uniprocessors [7, 40]. The key problem in the CMP domain is that migration in multiple directions can cause access conflicts between sharing cores and may result in shared blocks ping-ponging between processors. ACM solves this problem very effectively via moving cache blocks to the center of gravity from various sharing cores. Specifically, access patterns of cores are dynamically monitored and blocks are migrated to banks that minimize the access time for the sharing cores. However, by relocating cache blocks from their static home tiles, a location strategy capable of rapidly locating blocks on subsequent accesses is required. ACM utilizes C-AMTE to achieve fast location of L2 cache blocks. In summary, unlike the previously studied migration strategies in CMP literature, ACM revealed the usefulness of data migration in CMP cache management.

Simulation results demonstrate that ACM yields an average L2 access latency that is on average 20.4% better than that of a shared NUCA scheme.

As the traditional private and shared designs are subject to a principal deficiency where they both entail static partitioning of the available cache capacity and don't tolerate the variability among different working sets and phases of a working set, I promoted the Dynamic Cache Clustering (DCC) scheme in Chapter 7. I sheded light on the irregularity of working sets and described DCC as a strategy that can synergistically react to programs' behaviors and judiciously adapt to their different working sets and varying phases. DCC monitors the behavior of a scheduled program, and based upon its runtime cache demand makes related architecture-adaptive descions. The tension between higher or lower cache demands is driven by optimizing the L2 miss rate versus the average L2 access latency. Each core is initially started with a certain cache allocation (in terms of L2 cache banks), referred to as its cache cluster. Subsequently, and after every re-clustering point on a time interval, the cache cluster is contracted, expanded, or kept intact, depending on the cache demand. DCC implements CC-FR's data placement and relocation components. Simulation results show that DCC improves the average L1 miss time by as much as 21.3% (10% execution time) versus previous static designs. In summary, it has been shown that any single fixed scheme (see Section 7.2.1 for definition of fixed schemes) always fails to adapt to the variability across the working sets. DCC, in contrast, can always adapt to such irregularities and provides performance comparable to the best static alternative.

Deciding upon the placement of a cache block from the first touch might be sub-optimal. For instance, though PDA places blocks at underutilized L2 banks that are closest to requesting cores, I showed and discussed in Chapter 6 that multiple other sharing cores can compete for the blocks after placed. As such, and to make PDA a more practical scheme that saves latency not only for the initial requesters but also for all the sharers and to keep at the same time the cache capacity under control, I proposed in Chapter 8 augmenting ACM with PDA in one scheme and referred to that as the Dynamic Pressure and Distance Aware Placement (DPDA) scheme. As a result of such a combination between the two schemes, DPDA combines CC-FR's data placement and relocation categories and addresses both interference misses and growing non-uniform access latencies challenges. Simulation results

demonstrate that DPDA outperforms the nominal shared and PDA schemes.

To explore the case of implementing all CC-FR's components and addressing all the caching challenges presented in this thesis, I also suggested in Chapter 8 combining DCC and FSB together in one scheme and referred to that as the Dynamic Cache Clustering and Balancing (DCCB) scheme. Of course, FSB is an orthogonal design in that it can be applied to any of the proposed schemes in this dissertation; however, the rationale behind such a specific selection is to optimize the L2 miss rate experienced by DCC. DCC increases the aggregate cache footprint via allowing replication of shared cache blocks at multiple clusters. On the other hand, DCC already implements two of CC-FR's components (data placement and relocation) and requires only a retention strategy in order to fully apply all CC-FR's approaches. Nonetheless, it has been shown that FSB doesn't show the same effectiveness in reducing the L2 miss rate of DCC as with S. Under DCCB, system performance has been improved over DCC by an average of only 1.1%. This can be correlated to the fact that DCC already optimizes for the L2 miss rate in its algorithm though it entails replication at the L2 cache space.

As a conclusion, implementing more components of CC-FR doesn't necessarily correlate to a monotonic improvement in system performance. In fact, targeting only one CC-FR approach might provide more justifiable system performance (as is the case with FSB, ACM, or PDA alone) than targeting all or many of the approaches (as is the case with DCCB). In summary, implementing all or many of CC-FR's components might potentially incur a high hardware overhead which can be probably unjustifiable, especially if the resulted performance improvement is very little.

Lastly, I note that the pressure model utilized by PDA and DPDA which collects cache pressure at various group-based granularities can be employed not only by schemes that implement CC-FR's placement component, but further by strategies that apply the data retention approach. For instance, FSB retains lines only within an L2 cache bank (intra-tile pressure aware retention). However, when an L2 bank can't absorb anymore the working set of a scheduled program, the lines selected for replacements are simply discarded. In fact, in the meantime there might be other L2 banks that are underutilized. As such, one can promote retaining lines across L2 banks (inter-tile pressure aware retention), rather

than only within a single L2 bank. Nonetheless, inter-tile pressure aware retention can be implemented not only under FSB but rather under many other caching strategies including, but not limited to, the nominal shared and private designs. For instance, Chang and Sohi [12] proposed cooperative caching (CC) based on the private design that spills (retains) singlet blocks (blocks that have no replicas at the L2 cache space) upon evictions to (at) other *random* L2 banks seeking to reduce the L2 miss rate. Qureshi [50] showed that such a strategy might degrade system performance because it doesn't consider the cache demands of banks. Henceforth, Qureshi proposed dynamic spill-receive (DSR) that uses set dueling [51] to specify of whether an L2 bank should act as a "spiller cache" or a "receiver cache" but not both. Inter-tile pressure aware retention can, in fact, be straightforwardly applied to CC and blocks evicted from highly pressured banks can be retained at underutilized ones.

BIBLIOGRAPHY

[1] A. Agarwal, J. Hennessy, and M. Horowitz. "Cache performance of operating systems and multiprogramming," *In ACM Transactions on Computer Systems, 6*, pp. 393–431, Nov 1988.

[2] A. Agarwal and S. D. Pudar. "Column-associative Caches: A Technique For Reducing The Miss Rate Of Direct-Mapped Caches," *Proc. Int'l Symp. Computer Architecture*, 1993.

[3] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. "Dynamic Hardware-Assisted Software-Controlled Page Placement to Manage Capacity Allocation and Sharing within Large Caches," *Proc. Int'l Symp. High-Perf. Computer Arch*, Feb. 2009.

[4] L. Barroso et al. "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *ISCA*, May 2000.

[5] A. Basu, N. Kirman, M. Chaudhuri, and J. F. Martínez. "Scavenger: A New Last Level Cache Architecture with Global Block Priority," *MICRO*, pp. 421–432, Dec. 2007.

[6] B. M. Beckmann, M. R. Marty, and D. A. Wood. "ASR: Adaptive Selective Replication for CMP Caches," *Proc. Int'l Symp. Microarchitecture*, Dec. 2006.

[7] B. M. Beckmann and D. A. Wood. "Managing Wire Delay in Large Chip-Multiprocessor Caches," *Proc. Int'l Symp. Microarchitecture*, pp. 319–330, Dec. 2004.

[8] C. M. Bienia, S. Kumar, J. P. Singh, and K. Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *PACT*, Oct. 2008.

[9] B. Calder, D. Grunwald, and J. S. Emer. "Predictive sequential associative cache," *HPCA*, pp. 244–253, Feb. 1996.

[10] L. Censier and P. Feautrier. "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Comput. C-27 (12): 1112- 1118*, Dec. 1978.

[11] J. Chang. "Cooperative Caching for Chip Multiprocessors," *PhD thesis, University of Wisconsin-Madison*, 2007.

[12] J. Chang and G. S. Sohi. "Cooperative Caching for Chip Multiprocessors," *Proc. Int'l Symp. Computer Architecture*, June 2006.

[13] M. Chaudhuri. "PageNUCA: Selected Policies for Page-grain Locality Management in Large Shared Chip-multiprocessor Caches," *Proc. Int'l Symp. High-Perf. Computer Arch*, pp. 227-238, Feb. 2009.

[14] M. Chaudhuri. "Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches," *MICRO*, pp. 401-412, Dec. 2009.

[15] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. "Distance associativity for high-performance energy-efficient non-uniform cache architectures," *MICRO*, pp. 55–66, 2003.

[16] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. "Optimizing Replication, Communication, and Capacity Allocation in CMPs," *Proc. Int'l Symp. Computer Architecture*, pp. 357–368, June 2005.

[17] S. Cho and L. Jin "Managing Distributed Shared L2 Caches through OS-Level Page Allocation," *Proc. Int'l Symp. Microarchitecture*, Dec 2006.

[18] A. Chishti, M. D. Powell, and T. N. Vijaykumar. "Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures," *Proc. Int'l Symp. Microarchitecture*, Dec. 2003.

[19] J. D. Collins and D. M. Tullsen. " Runtime Identification of Cache Conflict Misses: The Adaptive Miss Buffer," *ACM Trans. Comput. Syst.*, pp. 413439, 2001.

[20] H. Dybdahl and P. Stenstrom "An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors," *Proc. Int'l Symp. High-Performance Computer Architecture*, Feb. 2007.

[21] Digital Equipment Corporation, Hudson, MA. "Digital Semiconductor 21164 AlphaMicroprocessor Product Brief," *Technical Document EC-QP97D-TE*, Mar. 1997.

[22] Z. Guz, I. Keidar, A. Kolodny, and U. C. Weiser. "Utilizing Shared Data in Chip Multiprocessors with the Nahalal Architecture," *SPAA*, June 2008.

[23] A. Gupta, W. D. Weber, and T. Mowry. "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," *Int'l Conference on Parallel Processing*, August 1990.

[24] E. G. Hallnor and S. K. Reinhardt. "A fully associative software managed cache design," *ISCA*, pp. 107–116, 2000.

[25] M. H. Hammoud, S. Cho, and R. Melhem. "ACM: An Efficient Approach for Managing Shared Caches in Chip Multiprocessors ," *Int'l conf. on High-Performance Embedded Architectures and Compilers*, pp. 319–330, Jan. 2009.

[26] M. H. Hammoud, S. Cho, and R. Melhem. "Dynamic Cache Clustering for Chip Multiprocessors ," *Proceedings of the ACM Int'l Conference on Supercomputing* , June. 2009.

[27] M. H. Hammoud, S. Cho, and R. Melhem. "Cache Equalizer: A Cache Pressure Aware Block Placement Scheme for Large-Scale Chip Multiprocessors ," *Technical Report TR-09-167, Department of Computer Science, University of Pittsburgh*, July. 2009.

[28] M. H. Hammoud, S. Cho, and R. Melhem. "C-AMTE: A Location Mechanism for Flexible Cache Management in Chip Multiprocessors ," *Technical Report TR-09-166, Department of Computer Science, University of Pittsburgh*, July. 2009.

[29] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," *Proc. Int'l Symp. Computer Architecture*, June 2009.

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

[31] J. Held, J Bautista, and S. Koehl. "From a Few Cores to Many: A Tera-scale Computing Research Overview," *White Paper. Research at Intel*, Jan. 2006.

[32] HP Labs. "http://www.hpl.hp.com/research/cacti/"

[33] H. Huang, K. G. Shin, C. Lefurgy, and T. Keller. "Improving Energy Efficiency by Making DRAM Less Randomly Accessed," *ISLPED*, August 2005.

[34] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. "A NUCA Substrate for Flexible CMP Cache Sharing," *Proc. Int'l Conf. Supercomputing*, pp. 31–40, June 2005.

[35] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. Steely Jr., and J. Emer. "Adaptive Insertion Policies for Managing Shared Caches," *PACT* , pp. 208–219, Oct. 2008.

[36] L. Jin and S. Cho. "Taming Single-Thread Program Performance on Many Distributed On-Chip L2 Caches," *Proc. Int'l Conference on Parallel Processing* , pp. 487–494, September 2008.

[37] N. P. Jouppi. " Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. Int'l Symp. Computer Architecture*, 1990.

[38] T. Johnson and U. Nawathe. "An 8-core, 64-thread, 64-bit Power Efficient SPARC SoC," *IEEE ISSCC*, Feb. 2007.

[39] M. Kandemir, F. Li, M. J. Irwin, and S. W. Son. "A Novel Migration-Based NUCA Design for Chip Multiprocessors," *Proc. Conference on High Performance Computing*, Nov. 2008.

[40] C. Kim, D. Burger, and S. W. Keckler. "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," *Proc. Int'l Conf. Architectural Support for Prog. Languages and Operating Systems*, pp. 211–222, Oct. 2002.

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

[42] J. Laudon and D. Lenoski. "The SGI Origin: A ccNUMA Highly Scalable Server," *ISCA*, June 1997.

[43] F. Li, M. Kandemir and M. J. Irwin. "Implementation and Evaluation of a Migration-Based NUCA Design for Chip Multiprocessors," *ACM SIGMETRICS*, June 2008.

[44] M. R. Marty and M. D. Hill. "Virtual Hierarchies to Support Server Consolidation," *Proc. Int'l Symp. Computer Architecture*, pp. 336–345, June 2007.

[45] G. Memik, G. Reinman, andW. H.Mangione-Smith. " Reducing Energy and Delay Using Efficient Victim Caches," *Proc. Int'l Symp. on Low Power Electronics and Design*, 2003.

[46] R. Mullins, A. West, and S. Moore "Low-Latency Virtual-Channel Routers for On-chip Networks," *Proc. Int'l Symp. Computer Architecture*, June 2004.

[47] K. Olukotun, L. Hammond, and J. Laudon. "Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency ," *Synthesis Lectures on Computer Architecture*, 1st Ed., Morgan and Claypool, Dec. 2007.

[48] J. Peir, Y. Lee, and W. Hsu. "Capturing Dynamic Memory Reference Behavior with Adaptive Cache Topology," *ASPLOS*, pp. 240–250, 1998.

[49] W. Qiang, M. Margaret, W. C. Douglas, V. J. Reddi, C. Dan, W. Youfeng, L. Jin, and B. David "A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance," *Proc. Int'l Symp. Microarchitecture*, pp. 271–282, 2005.

[50] M. K. Qureshi. "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," *Proc. Int'l Symp. High-Perf. Computer Arch*, pp. 45–54, Feb. 2009.

[51] M. K. Qureshi, A. Jaleel, Y. N. Patt, and S. C. Steely Jr.. "Adaptive Insertion Policies for High Performance Caching," *ISCA*, pp. 381–391, June 2007.

[52] M. K. Qureshi and Y. N. Patt. "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance," *MICRO*, pp. 423–432, Dec. 2006.

[53] M. K. Qureshi, D. Thompson, and Y. N. Patt. "The V-WAY Cache: Demand-Based Associativity via Global Replacement," *ISCA*, pp. 544–555, June 2005.

[54] Research at Intel. "Introducing the 45nm Next-Generation Intel Core^{TM} Microarchitecture," *White Paper.*,

[55] D. Rolán, B. B. Fraguela, and R. Doallo "Adaptive line placement with the set balancing cache," *MICRO*, pp. 529–540, Dec. 2009.

[56] A. Ros, M. E. Acacio, and J. M. García "Scalable Directory Organization for Tiled CMP Architectures," *Proc. Int'l Conference on Computer Design*, July 2008.

[57] A. Seznec. "A case for two-way skewed-associative caches," *ISCA*, pp. 169–178, May 1993.

[58] T. Sherwood, B. Calder, and J. Emer. "Reducing CacheMisses Using Hardware and Software Page Placement," *Proc. Int'l Conf. Supercomputing*, June 1999.

[59] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. "POWER5 System Microarchitecture," *IBM J. Res. & Dev.*, 49(1):–25, July. 2005.

[60] Y. Solihin. "Fundamentals of Prallel Computer Architecture," *Solihin Books*, 1st Ed., 2008.

[61] S. Srikantaiah, M. Kandemir, and M. J. Irwin. "Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 135-144, March 2008.

[62] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," *Proc. Int'l Symp. High-Perf. Computer Arch*, pp. 63-74, Feb. 2007.

[63] Standard Performance Evaluation Corporation. `http://www.specbench.org`.

[64] D. Tam, R. Azimi, L. Soares, and M. Stumm. "Managing Shared L2 Caches on Multicore Systems in Software," *In Workshop on the Interaction between Operating Systems and Computer Architecture*, 2007.

[65] N. Topham, A. Gonzalez, and J. Gonzalez. " The Design and Performance of a Conflict-Avoiding Cache," *Proc. Int'l Symp. Microarchitecture*, pp. 71–80, 1997.

[66] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. "An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS," *ISSCC*, Feb 2007.

[67] H. Vandierendonck, P. Manet, and J.-D. Legat. " Application-Specific Reconfigurable XOR-Indexing To Eliminate Cache Conflict Misses," *Proc. Conference on Design, Automation and Test*, pp. 357–362, 2006.

[68] Virtutech AB. Simics Full System Simulator "http://www.simics.com/"

[69] D. Weiss, J. J. Wuu, and V. Chin. "The on-chip 3-mb subarray-based third-level cache on an itanium microprocessor," *In IEEE journal of solid state circuits*, pp. 1523–1529, Nov. 2002.

[70] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. Int'l Symp. Computer Architecture*, pp. 24–36, July 1995.

[71] Y. Xie and G. H. Loh. "PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-core Shared Caches," *ISCA*, pp. 174–183, June 2009.

[72] C. Zhang. " Balanced Cache: Reducing Conflict Misses of Direct-Mapped Caches," *Proc. Int'l Symp. Computer Architecture*, June 2006.

[73] M. Zhang and K. Asanović "Victim Migration: Dynamically Adapting Between Private and Shared CMP Caches," *Technical Report TR-2005-064, Computer Science and Artificial Intelligence Labratory. MIT*, pp. 211–222, Oct. 2005.

[74] M. Zhang and K. Asanović. "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," *Proc. Int'l Symp. Computer Architecture*, pp. 336–345, June 2005.