Cache Equalizer: A Placement Mechanism for Chip Multiprocessor Distributed Shared Caches

Mohammad Hammoud, Sangyeun Cho, and Rami G. Melhem
Department of Computer Science, University of Pittsburgh
Pittsburgh, PA, USA
mhh@cs.pitt.edu, cho@cs.pitt.edu, melhem@cs.pitt.edu

ABSTRACT

This paper describes Cache Equalizer (CE), a novel distributed cache management scheme for large-scale chip multiprocessors (CMPs). Our work is motivated by large asymmetry in cache sets' usages. CE decouples the physical locations of cache blocks from their addresses for the sake of reducing misses caused by destructive interferences. Temporal pressure at the on-chip last-level cache is continuously collected at a group (comprised of cache sets) granularity, and periodically recorded at the memory controller to guide the placement process. An incoming block is consequently placed at a cache group that exhibits the minimum pressure. Simulation results using a full-system simulator demonstrate that CE achieves an average L2 miss rate reduction of 13.6% over a shared NUCA scheme and by as much as 46.7% for the benchmark programs we examined. Furthermore, evaluations showed that CE outperforms related cache designs.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*cache memories*

General Terms

Design, Performance, Management

Keywords

Chip Multiprocessors, Private Cache, Shared Cache, Pressure-Aware Placement, Group-Based Placement

1. INTRODUCTION

Crossing the billion-transistor per chip barrier has had a profound influence on the emergence of chip multiprocessors (CMPs) as a mainstream architecture of choice. As CMPs' realm is continuously expanding, they must provide high and scalable performance. One of the key challenges to obtaining high performance from CMPs is the management of the limited on-chip cache resources (typically the L2 cache) shared by multiple executing threads/processes.

Tiled chip multiprocessor architectures have recently been advocated as a scalable processor design approach. They replicate identical building blocks (tiles) and connect them with a switched network on-chip (NoC) [24]. A tile typically incorporates private L1 caches and an L2 cache bank. L2 cache banks are accordingly physically distributed over the processor chip. A conventional practice, referred to as the shared scheme, logically shares these physically distributed cache banks. On-chip access latencies differ depending on the distances between requester cores and target banks creating a Non Uniform Cache Architecture (NUCA) [18]. Alternatively, a traditional organization denoted as the private scheme, assigns each bank to a single core. Private design doesn't provide capacity sharing between cores. Each core attracts cache blocks to its associated L2 bank.

The private scheme offers two main advantages. First, cache blocks are read quickly. Second, performance isolation is inherently provided as an imperfectly behaving application cannot hurt the performance of other co-scheduled applications [22]. However, private caches increase aggregate cache footprint through undesired replication of shared cache lines. Nonetheless, even with low degrees of sharing, the pressure induced on a per-core private L2 bank can significantly increase as a consequence of an increasing working set size. This might lead to expensive off-chip accesses that can tremendously degrade the system performance. Recent proposals explored the deficiencies of the private design and suggested providing capacity sharing for efficient operation [22, 5].

Shared caches, on the other hand, offer increased cache space utilization via storing only a single copy of each cache line at the last level cache. Recent research work on CMP cache management has recognized the importance of the shared scheme [27, 9, 14, 30, 17]. Besides, many of today's CMPs, the Intel Core^{TM}2 Duo processor family [23], Sun Niagara [19], and IBM Power5 [26], have also featured shared caches. Nevertheless, shared caches suffer from an interference problem. A defectively behaving 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.

To establish a key hypothesis that there are significant destructive interferences between concurrently running threads/processes, we present in Fig. 1 the distribution of the L2 cache misses for 9 benchmarks executed on a 16-tile

HiPEAC 2011 Heraklion, Crete, Greece

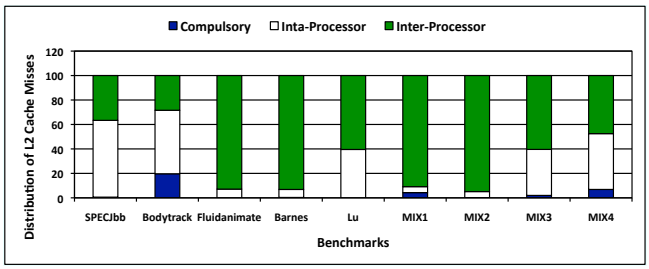

Figure 1: Distribution of L2 cache misses (compulsory, intra-processor, and inter-processor).

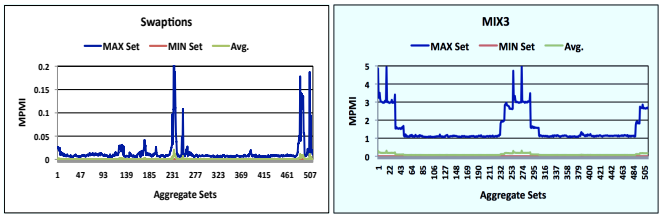

Figure 2: Number of misses per 1 million instructions (MPMI) experienced by two local cache sets (the ones that experience the max and the min misses) at different aggregate sets for two benchmarks, Swaptions and MIX3.

CMP platform employing a shared NUCA design[1]. Misses in a CMP with a shared scheme 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 a different processor) misses [27]. For the simulated applications, on average, 3.7% of misses are compulsory, 28.9% are intra-processor, and 67.2% are inter-processor. Compulsory misses can be reduced by hiding their latencies (i.e., data prefetching [28]). In this work we focus on reducing inter-processor and intra-processor misses in order to provide faster CMP NUCA architectures.

We primarily correlate destructive interferences problem to the *root* of CMP cache management, the cache placement algorithm. Fig. 2 demonstrates the number of misses per 1 million instructions experienced by cache sets across L2 cache banks (or aggregate sets) for two benchmarks, Swaptions (from the PARSEC suite [4]) and MIX3 (see Section 4.1 for details on this benchmark). We define an *aggregate set* with index i as the union of sets with index i across L2 cache banks. More formally, an *aggregate* $set_i = \bigcup_{k=1}^{n} set_{ki}$ where set_{ki} is the set with index i at bank k. We refer to each set_{ki} as a *local* set. Again, we assume a 16-way tiled CMP platform with physically distributed, logically shared L2 banks. We only show results for two local sets that exhibit the maximum and the minimum misses, in addition to the average misses, per each aggregate set. Clearly, we can see that memory accesses across aggregate sets are asymmetric. A placement strategy aware of the current pressures at banks can reduce the workload imbalance among aggregate sets by preventing placing an incoming cache block at an exceedingly pressured local set. This can potentially minimize interference misses and maximize system performance.

We identify two main requirements for enabling pressure-aware block 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 location 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-aware placement algorithm, a location strategy capable of rapidly locating cache blocks is required.

This paper explains the importance of incorporating pressure-aware placement strategies to improve CMP system performance. We propose cache equalizer (CE), a novel mechanism that involves a low hardware overhead framework to monitor the L2 cache at a *group* granularity (comprised of local cache sets) and 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 from the main memory, CE looks up the pressure array at the memory controller, identifies the group with minimum pressure, and places the block at that group.

In this work we make the following contributions:

- We propose a practical pressure-aware group-based placement mechanism that provides robust performance for distributed shared caches.
- We evaluate our proposal using a full system simulator and find that CE successfully reduces cache misses of a shared CMP design by an average of 13.6% and by as much as 46.7%.
- We compare CE to various related schemes. We find that CE outperforms victim caching [16], cooperative caching [5], and victim replication [36] by averages of 8%, 5.8% (5.2% when the cooperation throttling probability is set to 70%), and 8.7%, respectively.

The rest of the paper is organized as follows. A summary of prior work is given in Section 2. We detail the CE mechanism in Section 3. CE and alternative mechanisms are evaluated in Section 4. We conclude in Section 5.

2. RELATED WORK

Much work has been done to effectively manage CMP 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. Besides, previous work examined reducing either miss rate or latency in NUCA caches, or simply miss rate in a uniform cache architecture (UCA) caches. We briefly discuss below some of the prior related work and describe how our proposed Cache Equalizer (CE) mechanism differs from them.

Reducing conflict misses in uniprocessor caches has been a hot topic of research [16, 20, 31, 32, 35]. Summarily, two main directions have been proven to reduce conflict misses effectively: (1) higher set associativity and (2) victim caching (VC) [16, 20]. In Section 4.5 we present a study on reducing misses in shared CMPs through increasing associativity and cache size, and in Section 4.6 we compare against VC.

[1]Section 3.1 describes the adopted CMP platform and Section 4.1 details the experimental parameters and the benchmark programs.

In the context of chip multiprocessors, Hammoud et al. [10] recently introduced the idea of using pressure-aware placement in CMP caches. In this paper, we study a detailed implementation of a scheme that adopts pressure-aware placement, elaborate on aspects of such an implementation, and describe several optimizations to reduce the incurred hardware overhead. Sirkantaiah et al. [27] proposed adaptive set pinning (ASP) to reduce intra-processor and inter-processor misses. They associate processors to cache sets and solely grant them permissions to evict blocks from their sets on cache misses. As such, references that could potentially cause inter-processor misses can't interfere with each other even if they index to the same set. Blocks that would lead to inter-processor misses are redirected to small processor owned private (POP) caches. While ASP reduces misses effectively, it is not directly applicable to large-scale CMPs with multiple cache banks. ASP work is based on a UCA architecture (but claimed to be easily extensible to NUCA architectures). In contrast, our work focuses on large-scale CMP NUCA architectures.

Chang and Sohi [5] proposed *cooperative caching* (CC) based on the private scheme to create a globally managed shared aggregate on-chip cache. CC employs spilling singlet blocks (blocks that have no replicas at the L2 cache space) to other random L2 banks seeking to reduce intra-processor misses. CC is directly applicable to CMPs with multi-banking architectures. CE shares the same objective with CC but in addition to intra-processor misses, CE targets inter-processor ones. We compare CE and CC in Section 4.6. With CC, each private cache can spill as well as receive cache blocks. Hence, the cache requirement of each core is not considered. A recent work by Qureshi [22] proposed *dynamic spill-receive* (DSR) to improve upon CC by allowing private caches to either spill or receive cache blocks, but not both at the same time.

All of the above schemes attempt to reduce cache misses at block granularity. Many other researchers examined reducing cache misses at coarser (page) granularity [25, 15, 8, 1]. Sherwood et al. [25] proposed reducing cache 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 their allocation of physical pages. Cho and Jin [8] proposed an OS-based page allocation algorithm applicable to NUCA architectures. Cache blocks are mapped to the L2 cache space using a simple interleaving on page frame numbers. Cho and Jin color pages only upon first touch. As such, the optimal behaviors of workloads running over many phases might not be effectively reflected. Awasthi et al. [1] addressed this shortcoming and attempted to re-color pages at runtime (via an elegant use of shadow addresses to rename pages) moving them to the center of gravity from all the requesting cores. Their proposed mechanism relies on the OS to spread the working set of a single program across many colors under high capacity demands. In comparison to these schemes, CE performs a block-grain placement without any OS involvement and provides, accordingly, a transparent solution.

Lastly, many researchers have explored CMP cache management designs to reduce cache hit latency in CMP NUCA caches. Zhang and Asanović [36] proposed *victim replication* (VR) based on the nominal shared NUCA scheme. VR seeks to mitigate the average on-chip access latency via keeping replicas of local primary cache victims within the local L2 cache banks. Chishti et al. [7] proposed *CMP-NuRAPID* that controls replication based on usage patterns. Beckmann et al. [2] proposed *adaptive selective replication* (ASR) that dynamically monitors workloads behaviors to control replication on the private cache organization. Beckmann and Wood [3] examined block migration to alleviate access latency in CMPs and suggested *CMP-DNUCA*. Guz et al. [9] presented a new shared cache architecture and diverted only shared data to centered cache banks close to all cores. Chaudhuri [6] also evaluates data migration but at a coarser page granularity. Access patterns of cores are dynamically monitored and pages are migrated to banks that minimize the access time for the sharing cores. Hardavellas et al. [12] proposed *R-NUCA* that relies on the OS to classify cache accesses into either private, shared, or instructions. R-NUCA then places private pages at the local L2 cache banks of the requesting cores, the shared at fixed address-interleaved on-chip locations, and instructions at non-overlapping clusters of L2 cache banks. Huh et al. [14] proposed a spectrum of degrees of sharing to manage NUCA caches.

In summary, while we stand on the shoulders of many, three main things differentiate our work from the above listed proposals. First, we reveal the importance of *pressure-aware block* placement strategies in CMPs. Second, we offer a fully address-independent data placement process for distributed shared caches. Third, we present a simple novel framework to monitor CMP caches at various group-based granularities. Such a framework can, in fact, be generally applied to a variety of CMP cache schemes. For instance, it can be adopted by migration mechanisms (e.g., [11]) to guide promotions/demotions of blocks. Also, it can be utilized by schemes that offer capacity sharing for private caching (e.g., [5]) to guide spillings of blocks.

3. CACHE EQUALIZER (CE)

Cache Equalizer (CE) alleviates destructive interferences in shared NUCA designs by employing a pressure-aware group-based placement strategy. We first provide a brief background on the baseline architecture and then detail CE.

3.1 Baseline Processor Architecture

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 is split into multiple banks and distributed on a chip. Besides, economic, manufacturing, and physical design considerations suggest tiled 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) [12]. A tile typically includes a core, private L1 caches (I/D), and an L2 cache bank. Fig. 3 displays a typical 16-tile CMP architecture with a magnified single tile to demonstrate the incorporated components. In this paper we assume a 16-tile CMP model with a 2D mesh NoC.

The distributed L2 cache banks can be either assigned one bank per one core (private scheme), or one bank per many

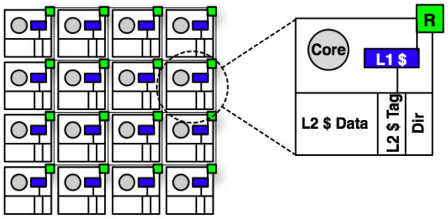

Figure 3: Tiled CMP architecture.

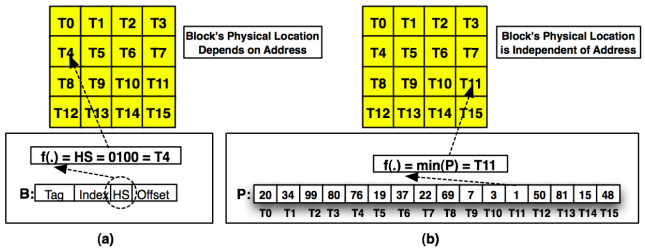

Figure 4: Address-based versus pressure-aware placements. (a) Shared scheme strategy. (b) Pressure-aware strategy. (*f(.)* denotes the placement function, HS is the home select bits of block B, and P is the pressure array)

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. 3. Dir stands for directory). In contrast, the shared scheme requires an engine to maintain coherence only at the L1 level as no replication of shared cache blocks is allowed at the L2 space. 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. The shared scheme, therefore, follows an address-based placement strategy. This work assumes a shared NUCA design and employs a distributed directory protocol for coherence maintenance.

3.2 Pressure-Aware Placement

We propose a pressure-aware placement strategy that maps cache blocks to the L2 cache space depending on the observed pressures at the L2 cache banks (refined later to groups of local cache sets). 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. The bank that corresponds to the slot that exhibits the minimum value (pressure) is selected to host the fetched cache block. Fig. 4 demonstrates a descriptive comparison between the placement strategies of the nominal shared NUCA design and our proposed scheme. As described earlier, 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 SHT. Assuming the HS bits of B are 0100, B is accordingly placed at tile T4. Alternatively, by using our pressure-aware placement strategy, the pressure array at the memory controller is inspected before B is mapped to L2. The pressure array indicates that tile T11 has the minimum pressure, thus selected.

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-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, we don't use misses to collect pressures at L2 banks but rather hits. More specifically, we quantify a pressure value as the number of lines that yield cache hits during a time interval, referred to as an *epoch*, and designate that as *temporal pressure*.

CE 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 CE to adapt to phase changes of applications, at the copy time we 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-aware placement algorithm, a location strategy capable of rapidly locating cache blocks at the L2 cache space is required. In this case, many strategies can be incorporated. First, a broadcast-based policy can easily fulfill the objective but might heavily burden the NoC. Second, a directory (either centralized or distributed) can be maintained and pointers can be kept to point to the current locations of blocks. This incurs, however, 3-way cache-to-cache transfers. A third option resolves the problem without broadcasting and with minimal 3-way communications and is referred to as cache-the-cache-tag (CTCT) [11] location policy.

CE adopts CTCT to achieve fast location of L2 cache blocks. Upon placing a cache block, B, at an L2 bank using CE, CTCT stores two corresponding tracking entries, replicated and principal, in special tracking entries (TR) tables at the requesting and the static home tiles of B, 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. CTCT suggests that a tracking entry encompasses the tag of the related block (typically 22 bits), a bit vector to keep related tracking entries coherent (16 bits for a 16-tile CMP model), and an ID that points to the tile that is currently hosting the block (4 bits for 16 tiles).

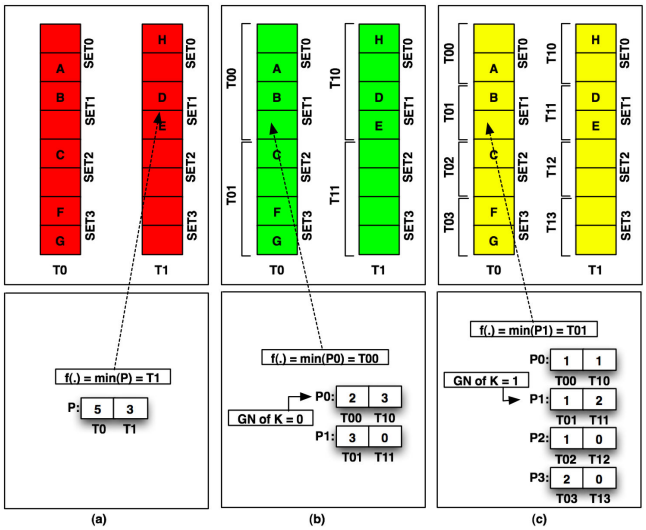

Figure 5: Placing block K (with index = 1) using the proposed pressure-aware group-based placement strategy with various granularities. (a) 1-group. (b) 2-group. (c) 4-group. (GN is the group number)

3.3 Group-Based Placement

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. We 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 the minimum 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. 5 demonstrates our pressure-aware group-based placement strategy using different granularities. For intuitive presentation, we 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. Fig. 5(a) illustrates our placement strategy operating at 1-group granularity. We start with a pressure array of zero values and assume that each 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 least pressured) 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). This explains the rationale behind collecting pressures at finer granularities for the sake of a more precise behavior.

Fig 5(b) demonstrates our proposed placement strategy 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 the minimum pressure and is, accordingly, selected to host K. Compared to a 1-group operating pressure-aware placement strategy (illustrated in Fig. 5(a)), no conflict miss is incurred. In Fig. 5(c) we 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 the minimum pressure 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.

3.4 An Illustrative Example and Three Optimizations

As described earlier, CE adopts the CTCT policy to achieve fast location of L2 cache blocks. We demonstrate through an example how CE combines CTCT and the proposed pressure-aware group-based placement strategy to offer an efficient cache management scheme for distributed shared caches. Furthermore, we offer three optimizations to reduce the area overhead required by CTCT. Fig. 6 shows CE in operation. Fig. 6(a) demonstrates a request made by core 3 to a cache block H. Core 3 looks up its local tracking entries (TR) table. We assume a miss is incurred and the request is subsequently forwarded to H's SHT, T12 (assuming the HS bits of H = 1100). The TR table at T12 is then looked up. We assume no principal tracking entry corresponding to H is found and an L2 miss is reported. Block H is then fetched from the main memory and placed at tile T11 (dictated by our employed placement strategy). Besides, principal and replicated tracking entries are stored at H's SHT, T12, and at the requester tile, T3, respectively. Fig. 6(b) displays the residences of H and each corresponding tracking entry h. Fig. 6(b) further illustrates a scenario where core 3 requests H again. Core 3 looks up its TR table and a hit on h occurs. As such, the request is straightforwardly directed to T11. Lastly, note that if any other core requests H, T12 can be always approached to locate H.

On an L2 request to a tile, probing always the local L2 bank and the TR table in parallel has a number of effects: (1) reducing latency as the requested block might be hosted locally, (2) reducing space because as a consequence we need not keep tracking entries (principal and replicated) for a block that maps to its own SHT. Specifically, if H (see Fig. 6(b)) is mapped to its SHT, T12, we need not keep any corresponding tracking entry, h, at any tile. To explain this, assume, to the contrary, that we do cache H, a corresponding principal tracking entry h, and a replicated copy h at tiles T12 (at the L2 bank), T12 (at the TR table), and T3, respectively. Consequently, a hit on h at the requester

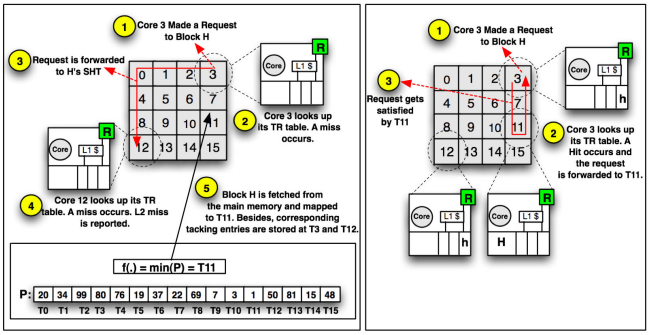

Figure 6: CE in operation. (a) A miss occurs at L2. (b) A hit occurs at L2.

tile T3 would trigger an access to T12, the host of H. On the other hand, a miss would trigger also an access to T12, the SHT of H. Thus, having h at T3 becomes redundant as T12 is anyway accessed. Besides, having the principal tracking entry h at T12 becomes also redundant when H is a resident of T12. In particular, upon accessing T12, if we look up its L2 bank and TR table concurrently we would hit at the L2 bank directly without any need for the principal entry h. **Therefore, a first optimization (O1) for CE would be not to cache any tracking entry (principal or replicated) for a cache block that is mapped to its SHT and to always lookup concurrently the L2 bank and the TR table at the SHT**.

Now assume that H is cached at the requester tile T3 instead of T12, the SHT of H. Assume moreover that a corresponding replicated tracking entry h is stored at T3. Upon requesting H, if we look up T3's local L2 bank concurrently with its TR table we will satisfy the request directly from the L2 bank without any need for the replicated entry h. As such, h becomes superfluous. On the other hand, if H is requested by a tile different than T3, H's SHT (T12) needs to be contacted to locate H. Hence, in this case we still need to maintain a principal tracking entry for H at its SHT. **Therefore, a second optimization (O2) for CE would be not to cache a replicated tracking entry for a block that is mapped to the requester tile and to always lookup concurrently the L2 bank and the TR table at the requester tile**.

Finally, and as a third optimization (O3), a cache block that is placed at a tile different than its SHT can be always promoted upon eviction back to its SHT if the SHT tile has space for an incoming block. By space at SHT we mean the presence of an invalid line or otherwise a ripple effect would occur (i.e., an eviction triggers another eviction). As an example, if we evict H from T11 (see Fig. 6(b)), we investigate first H's SHT, T12, for an invalid block. If we succeed in finding one, we place H at T12. As a result, we can subsequently apply O1. That is, we can invalidate all H's corresponding tracking entries because H is residing now at its SHT. Clearly, the goal of the proposed optimizations (O1, O2, and O3) is to reduce the overall area required by the TR tables.

4. QUANTITATIVE EVALUATION

4.1 Methodology

Component	Parameter
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/16way
L2 Bank Access Penalty	12 cycles
L2 Replacement Policy	LRU
Latency Per NoC Hop	3 cycles
Memory Latency	320 cycles

Table 1: 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)
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 2: Benchmark programs

We present our results based on detailed full-system simulation using Virtutech's Simics 3.0.29 [33]. We use our own CMP cache modules fully developed in-house. We 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. Each tile encompasses a switch, 32KB I/D L1 caches, and a 512KB L2 cache bank. A distributed MESI-based directory protocol is employed. We adopt an epoch length of 20 million instructions for measuring pressures at groups. Table 1 shows our configuration's experimental parameters.

We compare CE to the nominal shared (S) CMP design and three related proposals; victim caching (VC) [16], cooperative caching (CC) [5], and victim replication (VR) [36]. All schemes are studied using a mixture of multithreaded and multiprogramming workloads. For multithreaded workloads we use the commercial benchmark SPECJbb [29], five shared memory programs from the SPLASH2 suite [34] (Ocean, Barnes, Lu, Radix, and FFT), and two applications from the PARSEC suite [4] (Bodytrack and Fluidanimate). Four multiprogramming workloads have been also composed using the five listed SPLASH2 benchmarks and other five applications from SPEC2006 [29] (Hmmer, Sphinx, Milc, Mcf, and Bzip2). Table 2 shows the data sets and other important features of the simulated workloads. Lastly, the programs are fast forwarded to get past of their initialization phases. After various warm-up periods, each SPLASH2 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.

4.2 Comparing with the Shared NUCA Design

Let us first compare CE against the baseline shared (S)

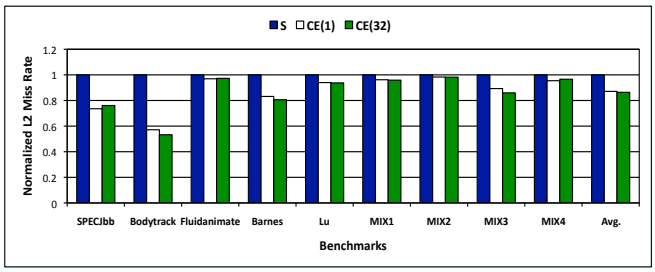

Figure 7: L2 miss rates of CE(1), CE(32), and shared (S) schemes (normalized to S).

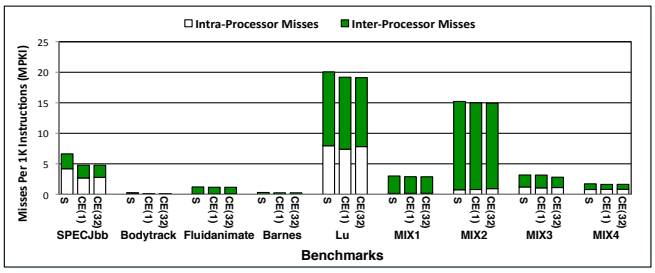

Figure 8: Misses Per 1K Instructions (MPKI) of CE(1), CE(32), and shared (S) schemes.

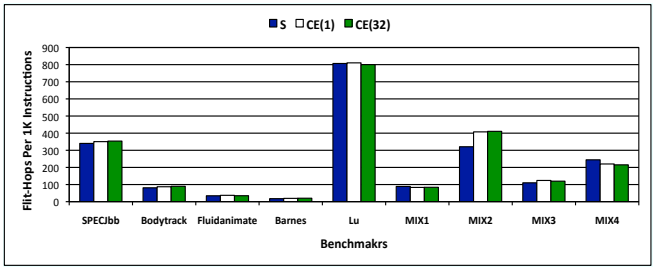

Figure 9: On-chip network traffic.

scheme. In this section we consider a tracking entries (TR) table with 16K entries. Each access to a TR table requires 1.35ns estimated using CACTI v5.3 [13]. Section 4.4 presents a sensitivity study of CE to different TR table sizes. Fig. 7 shows the L2 miss rates of S, CE(1), and CE(32) normalized to S. As discussed in Section 3.3, CE can run with different granularities (varying from 1-group to 512-group given our employed number of sets per L2 bank). CE(1) and CE(32) correspond to CE running with 1-group and 32-group granularities. In Section 4.3 we prove that dividing a bank into only 32 groups (i.e., a counter per each group of 16 sets) provides close benefits to dividing it into 512 groups (i.e., a counter per each set). On average, CE(1) and CE(32) achieve L2 miss rate reductions of 12.8% and 13.6% over S, and by as much as 42.8% and 46.7% for the Bodytrack program, respectively.

Three main factors affect the eligibility of applications for cache miss reductions provided by CE: (1) gravity of destructive interferences, (2) accessibility patterns, and (3) working set sizes. For instance, Bodytrack's shared and thread-private data contend aggressively for a limited amount of cache capacity [4]. The Bodytrack program experiences 51.9% and 28.3% intra-processor and inter-processor misses, respectively. CE resourcefully alleviates caustic contention and equalizes cache sets usages. CE(1) reduces the intra-processor and inter-processor misses of Bodytrack by 57.8% and 49.1% over S, respectively. CE(32), on the other hand, reduces intra-processor and inter-processor misses by 58.7% and 56.5%, respectively. Considering examples of homogenous programs, MIX1 and MIX2 demonstrate uniform pressure patterns over cache physical locations. Besides, MIX1 and MIX2 have large working set sizes. CE(1) accomplishes L2 miss rate reductions for MIX1 and MIX2 over S by only 3.8% and 1.6%, respectively. In contrast, CE(32) offers L2 miss rate reductions of 4.1% and 1.7% for MIX1 and MIX2, respectively.

To demonstrate CE's potential in reducing interference misses, Fig. 8 shows the number of references per 1K instructions that lead to intra-processor and inter-processor misses for all the examined programs. On average, CE(1) accomplishes reductions of 12.7% and 11.3% in intra-processor and inter-processor misses per 1K instructions (MPKI) over S, respectively. On the other hand, CE(32) provides average intra-processor and inter-processor MPKI reductions of 5.3% and 15.8% over S, respectively. We note, however, that for some benchmarks CE increases intra-processor (but decreases related inter-processor) misses (e.g., 5.3% for MIX2 under CE(1)). This occurs due to an increase in the number of references from the same processor to cache groups that are eligible for data eviction.

The L2 miss rate reductions provided by CE come sometimes at a small expense of higher network on-chip (NoC) traffic. Fig. 9 shows the number of flit-hops per 1K instructions experienced by S, CE(1), and CE(32). We define a flit-hop as one flit traveling one hop on a router in the 2D mesh NoC. On average, CE(1) and CE(32) increase the NoC traffic over S by 5.7% and 5.4%, respectively. For some benchmarks CE improves upon S (e.g., MIX4) while for some others CE degrades against S (e.g., MIX2). The NoC traffic increase generated by CE correlates to the use of the cache-the-cache-tag (CTCT) location policy. CTCT introduces more coherence traffic on the NoC for maintaining consistency among principal and replicated tracking entries. To the contrary, NUCA designs suffer from what is called, the NUCA latency problem. Specifically, a requested block might be placed far away from the requester core, thus causing the core significant latency (traffic) to locate the block. CE can sometimes potentially place blocks closer to requester cores thus reducing NoC traffic against S. If the gain from mitigating the NUCA problem offsets the loss from the incurred CTCT interconnect traffic, CE diminishes NoC traffic over S, otherwise, CE degrades versus S.

To that end, Fig. 10 presents the execution times of S, CE(1), and CE(32) normalized to S. Across all benchmarks, CE(1) and CE(32) achieve superiority over S by averages of 5.7% (by as much as 18.8% for SPECJbb) and 6.8% (by as much as 18.2% for SPECJbb), respectively. We make two observations: (1) although CE(32) achieves more miss rate reduction than CE(1) for Barnes, CE(1) outperforms CE(32) and (2) some benchmarks exhibit performance improvements that surpass the obtained miss rate reductions (e.g., Lu). As described earlier, CE can potentially place blocks closer (or further) to requester cores than S, and, accordingly, reduce (or increase) the average L2 access latency (AAL). For instance, Lu reveals a performance improvement of 13.3% with only 6.2% L2 miss rate reduction under CE(32). We found that CE(32) achieves a 7.6% AAL

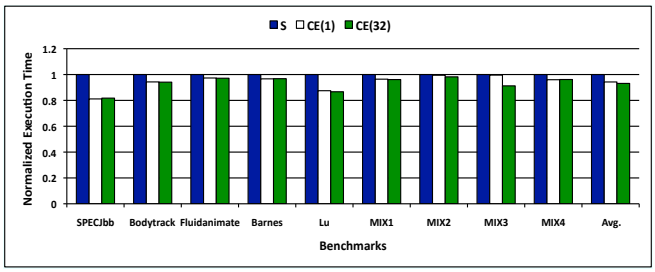

Figure 10: Execution times of CE(1), CE(32), and shared (S) schemes (normalized to S).

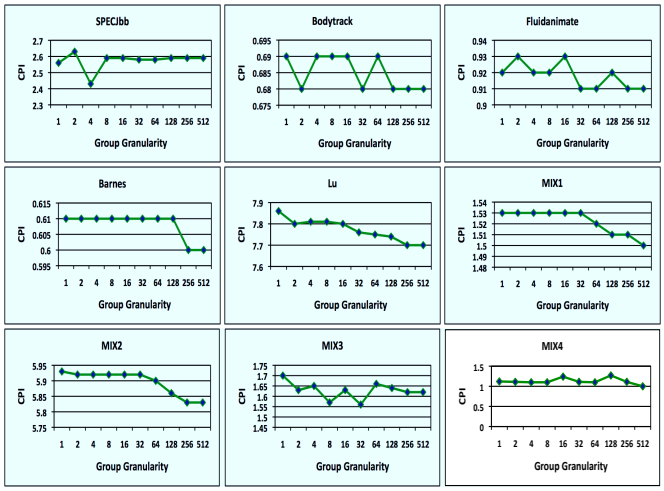

Figure 11: The CE behavior with different granularities (varying from 1-group to 512-group).

improvement over S for Lu.

4.3 Sensitivity of CE to Different Group Granularities

We demonstrate CE's behaviors with all possible group granularities. Fig. 11 plots the outcome. For each program we show cycles per instruction (CPI). As explained in Section 3.3, collecting pressures at a more refined granularity makes CE performing better (e.g., MIX2) but not necessarily until striking the upper bound (e.g., SPECJbb). Besides, we note that some programs show irregularities in performance (e.g., Fluidanimate) 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 we update the array at the memory controller) causing the array to be a little biased in representing actual pressures at cache groups. Lastly, we conclude that dividing a bank into only 32 groups provides close benefits as compared to dividing it into 512 groups.

Additionally, we note that for all the examined programs, CE always provides a robust performance versus S. That is, none of the programs, running under any group granularity, shows performance degradation against S. Fig. 12 demonstrates the *S-Curve*[2] of the CPI improvement provided by

[2]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 [22].

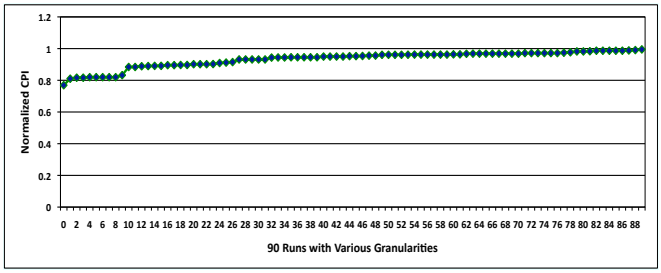

Figure 12: S-Curve for CPI improvement of CE over S.

K Entries	K Bytes Per Tile	% Increase of On-Chip Cache Capacity	Access Time (ns)
16	88	16%	1.35
8	44	8%	1.19
4	22	4%	1.12

Table 3: TR tables storage overhead and access times.

CE for the 90 runs (9 workloads each with 10 group granularities).

4.4 Sensitivity to Different TR Table Sizes

So far, we have been using 16K entries for a TR table per tile. In this section we study CE with two more TR table sizes. Specifically, we consider TR table sizes of 8K and 4K entries. Table 3 illustrates the 3 TR configurations with the incurred area overhead and access times estimated using CACTI v5.3 [13]. Fig. 13 demonstrates the execution times of S, CE(16K), CE(8K), and CE(4K) normalized to S. CE(16K), CE(8K), and CE(4k) denote CE with 16K, 8K, and 4K entries TR table sizes. We ran the 3 CE configurations with 1-group granularity. Across all benchmarks, CE(16K), CE(8K), and CE(4K) outperform S by averages of 5.7% (by as much as 18.8% for SPECJbb), 4% (by as much as 10.9% for SPECJbb), and 0.4% (by as much as 6.5% for MIX2), respectively. As the TR table size is decreased, the performance improvement over S also decreases. This is because with smaller TR table sizes more principal tracking entries are replaced. When a principal tracking entry is replaced, it requires evicting the corresponding replicated tracking entries and the L2 line. Therefore, it becomes a tradeoff between area overhead and performance. We, however, select a 16K entries TR table size as a default configuration for CE and justify the incurred overhead in the next subsection.

4.5 Impact of Increasing Cache Size and Associativity

We can improve cache performance not only through efficient cache management but also via increasing cache size and associativity. To conduct a fair comparison and create as a realistic match as possible, we add to each cache set of S two more ways. Given our 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. We refer to this configuration as S(2W). Moreover, we examine S's performance by doubling the size of

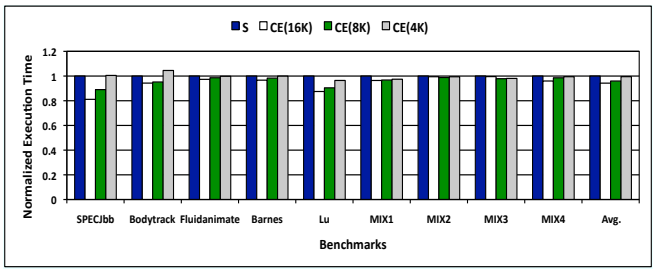

Figure 13: Normalized execution times of shared (S), CE(16K) (CE with 16K entries TR table size), CE(8K) (8K entries), and CE(4K) (4K entries).

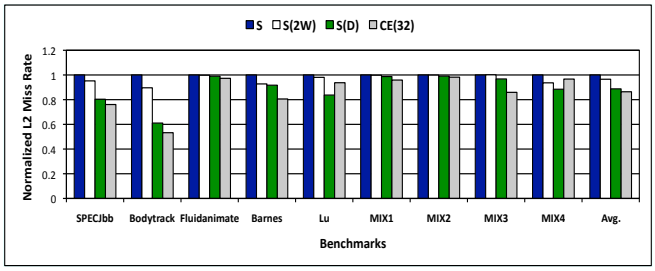

Figure 14: L2 miss rates of shared (S), shared with two more ways added (S(2W)), shared with double sized cache (S(D)), and CE(32) (normalized to S).

each L2 bank (i.e., from 512KB to 1MB). We refer to the latter configuration as S(D). Fig. 14 shows the L2 miss rates of S, S(2W), S(D) and CE(32) normalized to S. In this and the upcoming sections we consider only CE(32) as being an appropriate representation of CE. S(2W), S(D), and CE(32) achieve L2 miss rate reductions over S by averages of 3.4%, 11.2%, and 13.6%, respectively. We conclude that CE is quite attractive as with small design and storage overhead (i.e., 1.4MB increase in aggregate) it provides miss rate reduction benefits over S with twice its cache size (i.e., 8MB increase in aggregate).

Lastly, we observe that although increasing L2 cache associativity and size reduces misses, the contribution of inter-processor misses to the non-compulsory misses (as percentage of non-compulsory misses) changes very little. Fig. 15 shows that for some benchmarks, the contribution of inter-processor misses increases (e.g., SPECJbb) while for some others it either remains the same (e.g., Barnes) or negligibly decreases (e.g., MIX4). We conclude that the motivation for mitigating destructive interferences in shared NUCA designs remains.

4.6 Comparing with Related Designs

In addition to comparing with the nominal shared scheme, S, we compare CE(32) against victim caching (VC) [16], cooperative caching (CC)) [5], and victim replication (VR)) [36]. VC effectively extends the associativity of hot sets in the cache and reduces conflict misses. For fair comparison, we choose the size of an L2 victim cache per tile to approximately match the area increase in CE. Consequently, we 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 [13]. The CC design, on the other hand, attempts to reduce intra-processor misses. The performance of CC

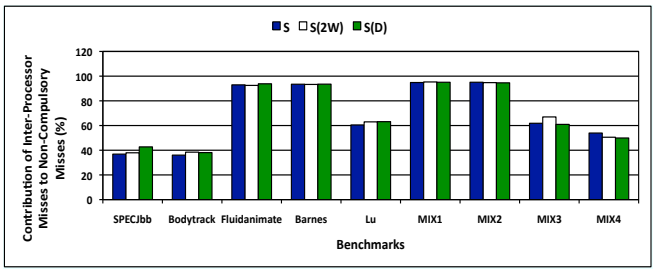

Figure 15: Contribution of inter-processor to non-compulsory misses of shared (S), shared with two more ways added (S(2W)), and shared with double sized cache (S(D)).

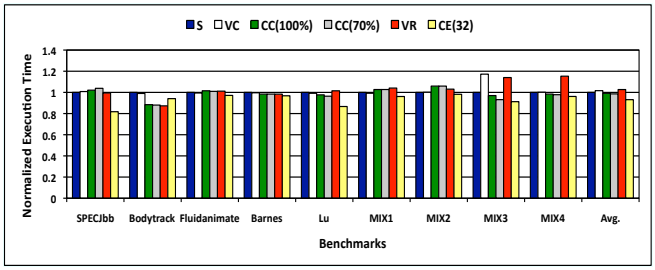

Figure 16: Execution times of shared (S), victim caching (VC), cooperative caching 100% (CC(100%)), cooperative caching 70% ((CC(70%)), victim replication (VR), and CE(32) schemes (normalized to S).

is highly dependent on the cooperation throttling probability [22]. Accordingly, we evaluate two configurations of CC, one with probability of 100% (CC(100%)) and another with probability of 70% (CC(70%)).

Fig. 16 depicts the execution times of all the compared schemes normalized to S. First, when multiple hot sets compete for a victim cache space, the victim cache is flushed quickly and fails subsequently to reduce capacity and conflict misses appreciably (e.g., MIX3). VC shows a performance degradation over S by an average of 1.6%. Second, CC spills cache blocks to neighboring L2 banks without knowing if spilling helps or hurts cache performance [22]. As such, CC sometimes degrades performance (e.g., SPECJbb) while it some other times demonstrates improvement (e.g., Bodytrack). On average, CC(100%) and CC(70%) surpass S by only 0.8% and 1.4%, respectively. Third, VR replicates evicted L1 blocks uncontrollably at local L2 banks and might, accordingly, increase the L2 miss rate significantly [2]. If VR fails to offset the lost latency (caused by the increased L2 miss rate) from the saved latency (gained by replica hits), performance degrades (e.g., MIX4). On average, VR shows a performance degradation over S by 2.6%. As compared to CE, CE(32) outperforms VC, CC(100%), CC(70%), and VR by averages of 8%, 5.8%, 5.2%, and 8.7%, respectively. Finally, we observe that while every related scheme degrades the performance of at least one application, CE(32) improves the performance of all the simulated benchmark programs.

5. CONCLUSIONS AND FUTURE DIRECTIONS

This paper investigates the interference problem inherent in distributed shared CMP caches and proposes cache equal-

izer (CE), a novel strategy that mitigates intra-processor and inter-processor misses. We indicate the significance of applying a pressure-aware group-based placement strategy on a shared CMP organization to achieve high system performance. Temporal pressure information is collected at a group granularity and recorded in an array at the memory controller. On an incoming cache block, CE inspects the pressure array, identifies the group with the minimum pressure, and maps the block to that group. Simulation results using a full system simulator demonstrate that CE reduces the cache misses of a shared NUCA design by an average of 13.6% and by as much as 46.7%. Furthermore, results show that CE outperforms victim caching [16], cooperative caching [5], and victim replication [36] by averages of 8%, 5.8% and 8.7%, respectively.

We set forth two main future directions. First, CE can be studied with further kinds of pressures. For instance, we can employ *spatial* (rather than temporal) pressure (how many *unique* lines yield cache hits during a time interval) and based on that explore CE's behavior. Finally, we will incorporate more parameters (e.g., distance to reduce NUCA latency) to CE's placement algorithm.

6. REFERENCES

[1] M. Awasthi, K. Sudan, R. Balasubramonian, J. Carter. "Dynamic Hardware-Assisted Software-Controlled Page Placement to Manage Capacity Allocation and Sharing within Large Caches," *HPCA*, Feb. 2009.

[2] B. M. Beckmann, M. R. Marty, and D. A. Wood. "ASR: Adaptive Selective Replication for CMP Caches," *MICRO*, Dec. 2006.

[3] B. M. Beckmann and D. A. Wood. "Managing Wire Delay in Large Chip-Multiprocessor Caches," *MICRO*, Dec. 2004.

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

[5] J. Chang and G. S. Sohi. "Cooperative Caching for Chip Multiprocessors," *ISCA*, June 2006.

[6] M. Chaudhuri. "PageNUCA: Selected Policies for Page-grain Locality Management in Large Shared Chip-multiprocessor Caches," *HPCA*, Feb. 2009.

[7] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. "Optimizing Replication, Communication, and Capacity Allocation in CMPs," *ISCA*, June 2005.

[8] S. Cho and L. Jin "Managing Distributed Shared L2 Caches through OS-Level Page Allocation," *MICRO*, Dec 2006.

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

[10] M. Hammoud, S. Cho, and R. Melhem. "A Dynamic Pressure-Aware Associative Placement Strategy for Large Scale Chip Multiprocessors," *Computer Architecture Letters*, May 2010.

[11] M. Hammoud, S. Cho, and R. Melhem. "ACM: An Efficient Approach for Managing Shared Caches in Chip Multiprocessors ," *HiPEAC*, Jan. 2009.

[12] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," *ISCA*, June 2009.

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

[14] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. "A NUCA Substrate for Flexible CMP Cache Sharing," *ICS*, June 2005.

[15] L. Jin and S. Cho. "Taming Single-Thread Program Performance on Many Distributed On-Chip L2 Caches," *ICPP* , September 2008.

[16] N. P. Jouppi. " Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *ISCA*, 1990.

[17] M. Kandemir, F. Li, M. J. Irwin, and S. W. Son. "A Novel Migration-Based NUCA Design for Chip Multiprocessors," *Proc. HiPC*, Nov. 2008.

[18] C. Kim, D. Burger, and S. W. Keckler. "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," *ASPLOS*, Oct. 2002.

[19] P. Kongetira, K. Aingaran, and K. Olukotun. "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, March-April 2005.

[20] G. Memik, G. Reinman, andW. H.Mangione-Smith. " Reducing Energy and Delay Using Efficient Victim Caches," *ISLPED*, 2003.

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

[22] M. K. Qureshi. "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," *HPCA*, Feb. 2009.

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

[24] A. Ros, M. E. Acacio, and J. M. García "Scalable Directory Organization for Tiled CMP Architectures," *ICCAD*, July 2008.

[25] T. Sherwood, B. Calder, and J. Emer. "Reducing CacheMisses Using Hardware and Software Page Placement," *ICS*, June 1999.

[26] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. "POWER5 System Microarchitecture," *IBM J. Res. & Dev.*, July. 2005.

[27] S. Srikantaiah, M. Kandemir, and M. J. Irwin. "Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors," *ASPLOS*, March 2008.

[28] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," *HPCA*, Feb. 2007.

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

[30] D. Tam, R. Azimi, L. Soares, and M. Stumm. "Managing Shared L2 Caches on Multicore Systems in Software," *WIOSCA*, 2007.

[31] N. Topham, A. Gonzalez, and J. Gonzalez. " The Design and Performance of a Conflict-Avoiding Cache," *MICRO*, 1997.

[32] H. Vandierendonck, P. Manet, and J.-D. Legat. " Application-Specific Reconfigurable XOR-Indexing To Eliminate Cache Conflict Misses," *DATE*, 2006.

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

[34] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations," *ISCA*, July 1995.

[35] C. Zhang. " Balanced Cache: Reducing Conflict Misses of Direct-Mapped Caches," *ISCA*, June 2006.

[36] M. Zhang and K. Asanović. "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," *ISCA*, June 2005.