Cache Equalizer: A Cache Pressure Aware Block Placement Scheme for Large-Scale Chip Multiprocessors

Mohammad Hammoud, Sangyeun Cho, and Rami Melhem
Department of Computer Science
University of Pittsburgh

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. CE provides Quality of Service (QoS) by robustly offering better performance than the baseline shared NUCA cache. Simulation results using a full-system simulator demonstrate that CE outperforms shared NUCA caches by an average of 15.5% and by as much as 28.5% for the benchmark programs we examined. Furthermore, evaluations manifested the outperformance of CE versus related CMP cache designs.

1 Introduction

As large uniprocessors are no longer scaling in performance, chip multiprocessors (CMPs) have become the trend in computer architecture. 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 [19]. From amongst the many key challenges to obtaining high performance from CMPs is the management of the limited on-chip cache resources (typically the L2 cache) shared by the multiple executing threads.

Tiled chip multiprocessor (CMP) 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) [22]. A tile typically incorporates a private L1 cache 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) [16]. Alternatively,

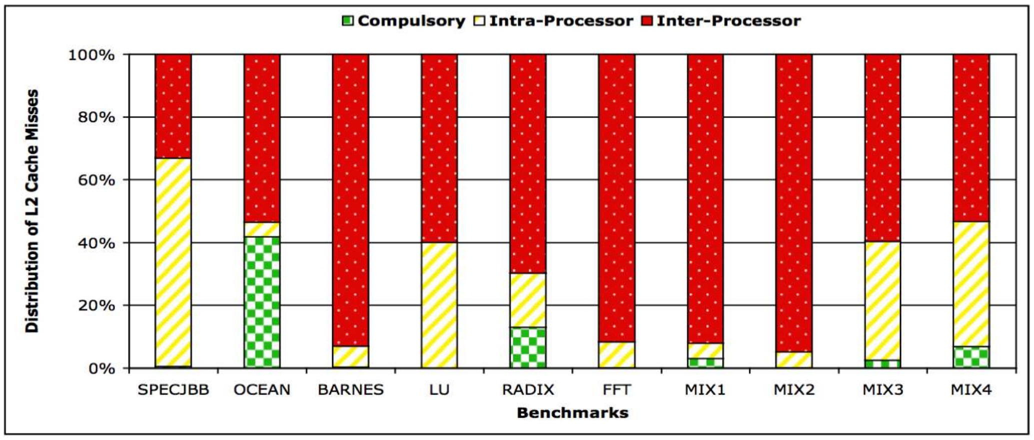


Figure 1. Distribution of L2 cache misses (compulsory, intra-processor, and inter-processor).

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 concurrently executing applications [20]. 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 inherent to the private design and suggested providing capacity sharing for efficient operation [4, 20].

Shared caches, on the other hand, offer increased cache space utilization via storing only a single copy of each cache line. Recent academic works on CMP cache management have recognized the importance of the shared scheme [9, 10, 12, 15, 25, 28]. Many of today's multi-core processors, the Intel Core^{TM}2 Duo processor family [21], Sun Niagara [17], and IBM Power5 [24], have also featured shared caches. Nevertheless, shared caches lack performance isolation. A defectively behaving application can evict useful L2 cache content belonging to other concurrently running programs. Thus, a program that exposes temporal locality can experience high cache misses caused by such destructive interferences between applications.

To establish a key hypothesis that there are significant destructive interferences between concurrently running threads, we present in Fig. 1 the distribution of the L2 cache misses for 10 benchmarks executed on a 16-tile CMP platform employing 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 [25]. For the simulated applications, on average, 6.8% of misses are compulsory, 23% are intra-processor, and 70% are inter-processor. Compulsory misses can be reduced by hiding their latencies (i.e., data prefetching [26]). In this work we focus rather on reducing inter-processor and intra-processor misses in order to provide faster CMP NUCA

[1]Details about the experimental parameters and the benchmarks are described in Section 4.

architectures.

We primarily correlate the destructive interferences phenomenon to the root of the cache management problem, the cache placement algorithm. A placement strategy aware of the current cache pressure can circumvent placing an incoming cache block at a highly *pressured* cache location, thus diminishing undesired contention. Traditionally, cache blocks are stored at cache locations solely based on their physical addresses. As such, the mapping process is unaware of the disparity in the hotness of shared cache locations.

We identify two main requirements for enabling cache 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 mapping 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 map to any of these 128-way entries. Second, by having a pressure-aware placement strategy, a location strategy capable of rapidly locating the cache blocks requested at later times would be 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 combines pressure-aware placement and smart location strategies. A low-hardware overhead framework is involved to monitor the L2 cache at a *group* granularity (comprised of cache sets) and record pressure information at an array embedded within the memory controller. The collected pressure information is utilized to guide the mapping 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, low-overhead pressure-aware placement mechanism that provides robust performance isolation for distributed shared NUCA caches.
- We evaluate our work using a full system simulator and find that CE successfully reduces cache misses of shared NUCA by an average of 21.9% and by as much as 50.1%.
- In addition to shared NUCA, we compare CE to various related schemes. We find that CE outperforms private scheme, victim replication (VR) [34], and cooperative caching (CC) [4] by averages of 15.5%, 15.3%, and 14.7%, respectively.
- Contrary to VR and CC, we find that CE provides Quality of Service (QoS) by robustly offering better performance than the baseline shared scheme.

The rest of the paper is organized as follows. Section 2 presents the baseline processor architecture. The CE mechanism is detailed in Section 3. We evaluate CE and competing alternative mechanisms in Section 4. Section 5 summarizes prior work and we conclude in Section 6.

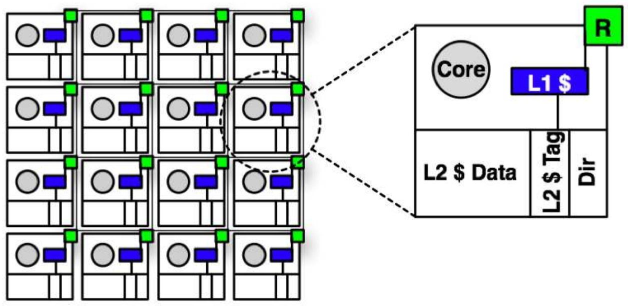


Figure 2. Tiled CMP architecture (Figure not to scale).

2 Baseline Processor Architecture

Exponential increase in cache sizes, 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 single die. Access latencies to the cache banks are functions of proximity between distributed requesting cores and target banks. 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) [11]. A tile typically includes a core, a private L1 cache, and an L2 cache bank. Fig. 2 displays a typical 16-tile CMP architecture with a magnified single tile to demonstrate the incorporated components. In this work we assume a tiled CMP architecture with 16 tiles and a 2D mesh NoC.

The distributed L2 cache banks can be either assigned one bank to one core (private scheme), or one bank to many cores (shared scheme). Private scheme requires an engine to maintain coherence (typically by using a distributed directory protocol) at the L1 and L2 caches because shared blocks are replicated at both levels (see Fig. 2(a). Dir stands for directory). On a local L2 miss, the coherence directory is inspected before reporting an L2 miss. If a hit occurs at the directory, the requested block is transferred (copied) from a hosting bank to the L2 bank of the requesting core. Conversely, the shared scheme maintains the exclusiveness of cache blocks at the L2 level. Thus a coherence engine is required to maintain coherence only at the L1 level. A core maps and locates a cache block, B, to and from a target 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. As such, the shared scheme follows an address-based placement strategy. On an L1 miss, B's SHT is directly approached. If a miss occurs, the main memory is accessed (as no other tile can contain the requested data). This work is based on the shared scheme and employs a distributed directory protocol for coherence maintenance.

3 Cache Equalizer (CE)

Cache Equalizer (CE) seeks performance isolation among concurrently running applications by attempting to alleviate destructive interferences. For that sake, we make the placement process of cache blocks aware of the underlying cache pressure, a technique that we refer to as a pressure-aware placement strategy. However, as described earlier, placing cache blocks independent of their addresses would require a mechanism capable of rapidly locating the blocks when reused. We achieve fast location of cache blocks through the recently proposed cache-the-cache-tag (CTCT) [10] location policy. We briefly describe CTCT next.

3.1 Cache Equalizer: The Cache-The-Cache-Tag (CTCT) Location Strategy

The shared scheme adopts an address-based placement strategy. The placement and location functions are the same (both use the HS bits from an address). When the mapping of a block B is decoupled from its address, B can be hosted by any of the distributed L2 banks. For any cache scheme that does such a relaxation in the mapping process, the cache-the-cache-tag (CTCT) strategy can be used to subsequently locate cache blocks. Assume a block B can map to anywhere on the L2 cache space, CTCT maintains a *tracking entry* corresponding to B at B's static home tile (SHT). This tracking entry is referred to 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. CTCT also supports caching tracking entries for B at requester tiles. These entries are referred to as *replicated* tracking entries. A replicated tracking entry at a requester tile also points to the current host of B and can be rapidly checked by a requester core to straightforwardly locate B (instead of checking with B's SHT).

Based on the above discussion, per each tile T, a principal tracking entry is kept for each cache block B whose static home tile is T but had been mapped to another tile. Besides, replicated tracking entries are retained at T to track the locations of other cache blocks that have been recently accessed by T but whose static home tile is not T. Though both, principal and tracking entries essentially act as pointers to the current hosts of cache blocks, a distinction is made between them for consistency and replacement purposes (more on this shortly). A per-tile data structure, referred to as the tracking entries (TR) table, can be added to host both classes of tracking entries pertaining that a hardware extension (i.e. an indicative bit) is used to distinguish between them.

Whenever a core issues a request to a block B, its TR table is checked first for a matching tracking entry. On a miss, the SHT of B is reached and its TR table is looked up. If a miss occurs (on B's principal tracking entry) at the TR table of B's SHT, B is fetched from the main memory and placed at a tile specified by the underlying placement method (using the pressure-aware placement strategy in our case). Additionally, a principal tracking entry is cached at the TR table of B's SHT. If B, however, exists on chip, a hit occurs at the TR table of B's SHT and B is located at its current host. A replicated tracking entry is further cached at the requester's TR table. Finally, if the requester core hits at its TR table, B is straightforwardly retrieved from the current host that the replicated tracking entry designates. This makes the location process fast due to avoiding 3-way cache-to-cache communication scenarios, specifically between the requester, the SHT, and the host tiles. An illustrative example combining CTCT and the proposed placement strategy is given in Section 3.3.

The principal and replicated tracking entries need to be kept consistent. This is accomplished by embedding a bit vector with each principal tracking entry at the TR tables to indicate which tiles cached related replicated tracking

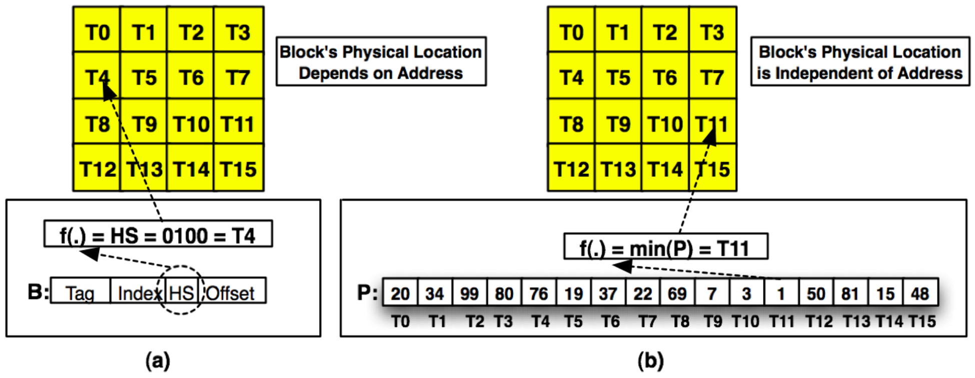


Figure 3. Address-based versus pressure-aware placements. (a) Shared scheme strategy. (b) Pressure-aware strategy. (*f(.)* denotes the placement function, HS is the home select bits of block B, and P is the pressure array)

entries. Finally, as each per-tile TR table can hold principal and replicated tracking entries, it is wise to never evict a principal tracking entry in favor of a replicated one. An eviction of a principal tracking entry requires an eviction of the corresponding cache block and all the corresponding replicated tracking entries. As such, the TR replacement policy should replace the following three classes of entries in descending order: (1) an invalid entry, (2) an LRU replicated tracking entry, (3) and an LRU principal tracking entry. Besides, upon caching a replicated tracking entry, only the first two classes are considered. If no entry belonging to one of these two classes is found, a replicated tracking entry is not cached.

3.2 Cache Equalizer: A Pressure-Aware Placement Strategy

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 cache sets). The pressure (more on this shortly) 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 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. 3 demonstrates a descriptive comparison between the placement strategies of the nominal shared NUCA 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 static home tile (SHT). Assuming the HS bits of B are 0100, B is accordingly placed at tile T4 (see Fig. 3(a)). 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 slot T5 (corresponding to the cache bank at tile T5) has the minimum pressure, thus selected (see Fig. 3(b)).

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). CE mechanism employs the CTCT strategy to locate cache blocks. Thus, the TR tables, either at requesting tiles or the SHT of B, can keep tracking entries for B. However, if B's corresponding principal tracking entry is missed at the TR table on B's SHT, an L2 miss is reported. This L2 miss can't in fact be correlated to any specific L2 bank but rather to the whole L2 cache space. Hence, we don't use misses to represent pressures at L2 banks but rather hits.

Hits can be used to indicate two types of pressures, *spatial* and *temporal*. We define spatial pressure as the number of *unique* lines yielding cache hits during a time interval, and temporal as the number of lines yielding cache hits during a time interval. Spatial pressure pertains to the utilized cache space while temporal pressure effectively reflects the frequency of the successful cache accesses during a time interval. In this work we use temporal pressure. Spatial, and possibly other pressure functions (i.e., a combination between spatial and temporal), can be explored in a future work.

Our pressure-aware placement policy doesn't rely on a prior knowledge of the program but on hardware counters. A saturating counter(s) can be installed at each tile to count the number of successful accesses during a time interval referred to as an *epoch*. At the end of every epoch (20M instructions in this paper) the values of counters are copied from the local tiles to the pressure array at the memory controller. The IDs of the tiles are used to index the array. Besides, in order to allow our mechanism to adapt to undergoing 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.

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 term group size as the number of sets 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 have minimally one set). 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 a pressure array would consist of n rows and p columns. Therefore, a 1-group (bank) granularity indicates a linear pressure array and can be probed straightforwardly. 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 K can be simply determined by dividing the index of K by the group size.

Fig. 4 demonstrates our pressure-aware placement strategy using different granularities. For intuitive presentation, we assume a simplified 2-tile (T0 and T1) CMP version 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. 4(a) illustrates our pressure-aware placement strategy

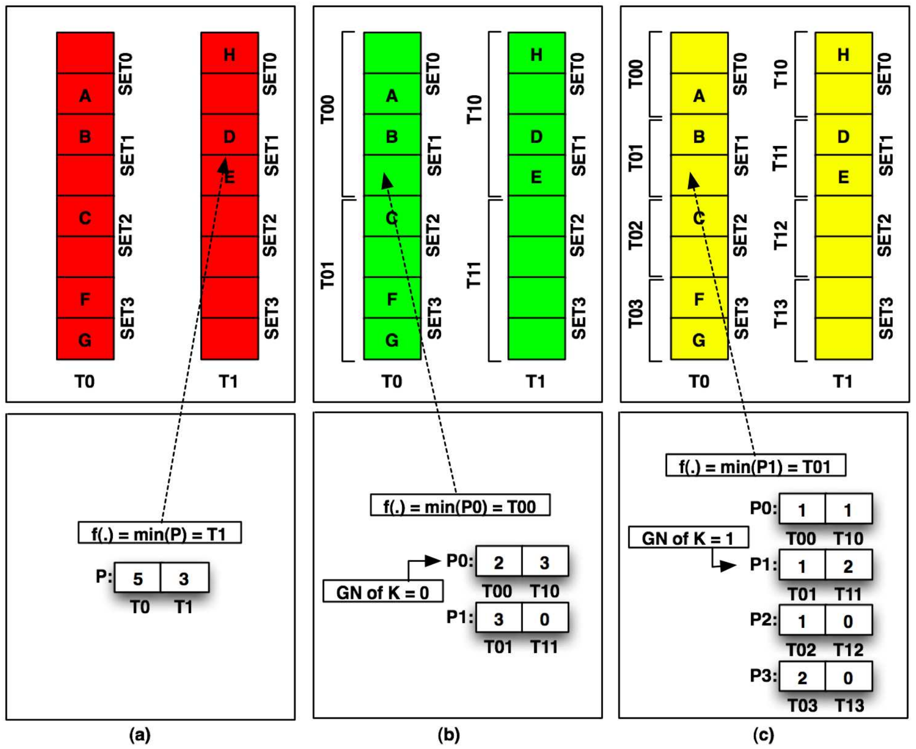


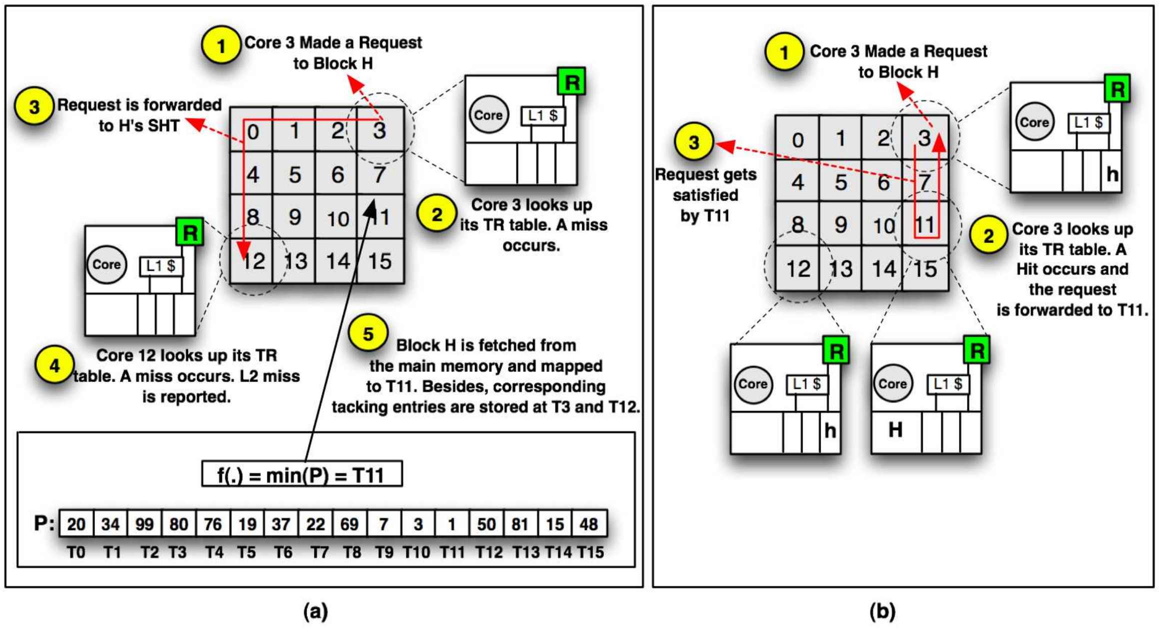
Figure 4. **Placing block K (with index = 1) using the proposed pressure-aware placement strategy with various granularities. (a) 1-group. (b) 2-group. (c) 4-group. (GN is the group number)**

operating at 1-group granularity. We assume that each of the blocks on the banks has been successfully accessed for only one time during the last epoch and that the pressure array had zero values before (this describes the numbers displayed in the arrays). 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 maps accordingly to set1 of bank T1. As a consequence, a conflict miss occurs. Had bank T0 (though exposing higher pressure) been selected, no conflict miss would have been incurred (because set1 of bank T0 has a free space for an incoming block).

Fig 4(b) demonstrates our proposed pressure-aware placement strategy operating at 2-group granularity. Given that the index of the incoming block K is 01, GN of K is accordingly 0 (index/group size = 0/2). Hence, row 0 is investigated. Group T00 at bank T0 exhibits the minimum pressure thus selected to host K. Compared to a 1-group operating pressure-aware placement strategy (illustrated in Fig. 4(a)), no conflict miss is incurred. In Fig. 4(c) we refine the granularity more, specifically to 4-group. GN of K is now 1, and row 1 is therefore explored. Group T00 at bank T0 reveals the minimum pressure thus selected. Note that the placement strategy with 4-group and 2-group granularities demonstrate similar behavior for K. This hints to the fact that refining the granularity might not always provide better performance. This usually occurs when the pressures become uniform across groups.

3.3 Cache Equalizer: Illustrative Example and Optimizations

We demonstrate through an example how the CE mechanism combines the cache-the-cache-tag location and the pressure-aware placement strategies to offer an efficient cache management scheme for distributed NUCA caches.



1 Core 3 Made a Request to Block H
3 Request is forwarded to H's SHT
2 Core 3 looks up its TR table. A miss occurs.
4 Core 12 looks up its TR table. A miss occurs. L2 miss is reported.
5 Block H is fetched from the main memory and mapped to T11. Besides, corresponding tacking entries are stored at T3 and T12.
f(.) = min(P) = T11
P: 20 34 99 80 76 19 37 22 69 7 3 1 50 81 15 48
T0 T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12 T13 T14 T15
(a)
1 Core 3 Made a Request to Block H
3 Request gets satisfied by T11
2 Core 3 looks up its TR table. A Hit occurs and the request is forwarded to T11.
(b)

Figure 5. The CE mechanism in operation. (a) A miss occurs at L2. (b) A hit occurs at L2.

Fig. 5 shows the CE mechanism in operation. Fig. 5(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 static home tile (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 (decided by our employed pressure-aware placement strategy). Fig. 5(b) displays the residences of H and corresponding tracking entries h. As illustrated in Fig. 5(b), when core 3 requests H again, it looks up its TR table and a hit on h occurs. Thus 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 SHT. Specifically, if H (see Fig. 5(b)) is mapped to its SHT, T12, we don't keep any corresponding tracking entry, h, at any tile. To explain this, assume contradictorily that we do really cache H, a corresponding principal tracking entry h, and another replicated one h at tiles T12, T12 (at the TR table), and T3, respectively. Consequently, a hit on h at the requester tile T3 would trigger an access to T12, the host of H. Besides, a miss on h at T3 would trigger an access to T12, the SHT of H. Thus, having h at T3 becomes redundant as T12 is always accessed. Also having the principal tracking entry h at T12 becomes redundant because H resides at the associated L2 bank. Upon accessing T12, if we look up its L2 bank and TR table concurrently we would hit at the L2 bank straightforwardly without any need for the principal h. **As such, an optimization for CE would be not to cache any tracking entry for a cache block that maps to its SHT and to always lookup the L2 bank and the TR table at the SHT in parallel**.

Now assume that H is cached at the requester tile T3 instead of T12, H's SHT (see Fig. 5(b)). Assume moreover

COMPONENT	PARAMETER
Cache Line Size	64 B
L1 I/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 or 8MB aggregate/16way
L2 Bank Access Penalty	12 cycles
L2 Replacement Policy	LRU
Latency Per NoC Hop	3 cycles
Memory Latency	300 cycles

Table 1. System parameters

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 straightforwardly from the L2 bank without any need for the replicated h. However, if H is requested by a tile different than T3, H's SHT (T12) needs to be approached in order 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 for CE would be not to cache a replicated tracking entry for a block that maps to the requester tile and to always lookup the L2 bank and the TR table of the requester tile in parallel**.

Lastly, and as a third optimization, a cache block that maps to a tile different than its SHT can be always promoted upon eviction to its SHT if it has space (an invalid line) for an incoming block (to avoid another eviction and a ripple effect). Specifically, if we evict H from T11 (see Fig. 5(b)), we investigate first H's SHT, T12, for an invalid block. If we successfully find an invalid block at T12, we place H at T12 and invalidate all its corresponding tracking entries. The latter step is a direct application of the first optimization because T12 is H's SHT.

4 Quantitative Evaluation

4.1 Methodology

We present our results based on detailed full-system simulation using Virtutech's Simics 3.0.29 [31]. We fully developed our own NUCA cache module including a 2D mesh NoC model. We implemented the XY-routing algorithm and accurately modeled congestion for both coherence and data. A tiled CMP architecture comprised of 16 UltraSPARC-III Cu processors is simulated running Solaris 10 OS. Each processor uses an in-order core model. The tiles are organized as 4×4 grid connected by the 2D mesh NoC. Each tile encompasses a switch, a 16KB I/D L1 cache, a 512KB L2 cache bank, and a tracking table with 16K entries. The latency to lookup a tracking table is hidden under the delay to enqueue the request in the port scheduler of the local switch [5]. We implemented, and fully verified and tested, a distributed MESI-based directory protocol. Finally, an epoch length of 20 million instructions is employed for measuring pressures at groups. Table 1 shows our configuration's experimental parameters.

We compare CE to the nominal shared (S) baseline architecture, the private scheme (P), and two related proposals; cooperative caching (CC) [4], and victim replication (VR) [34]. All schemes are studied using a mixture of multithreaded and multiprogramming workloads. For multithreaded workloads we use the commercial benchmark

NAME	INPUT
SPECjbb	Java HotSpot (TM) server VM v 1.5, 4 warehouses
Ocean	1026×1026 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, Ocean, Radix, Lu, Milc, Mcf, Bzip2, and Hmmer (2 threads/copies each)
MIX4	Barnes, FFT, Lu, and Radix (4 threads each)

Table 2. Benchmark programs

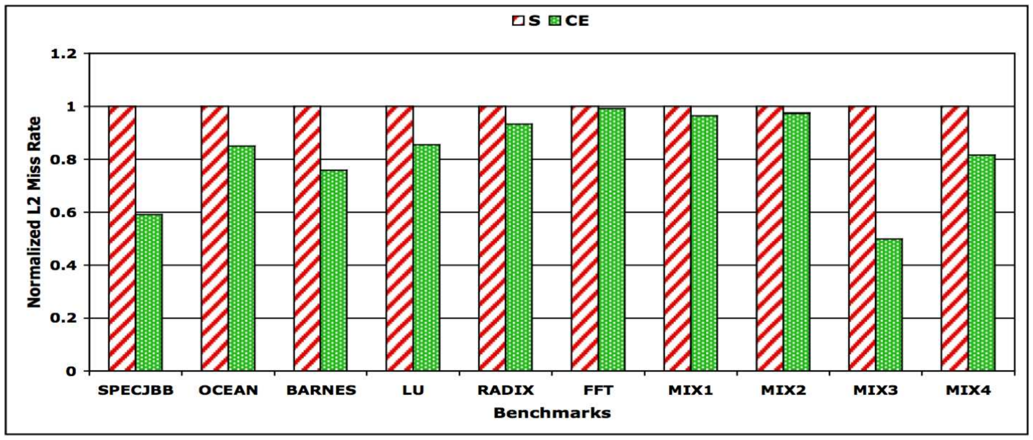


Figure 6. L2 miss rates of Cache Equalizer (CE) and Shared (S) schemes (normalized to S).

SPECJBB in addition to five shared memory ones from the SPLASH2 suite [32] (OCEAN, BARNES, LU, RADIX, and FFT). Four multiprogramming workloads have been also composed from the five SPLASH2 benchmarks and other five applications from SPEC2006 [27] (HMMER, SPHINX, MILC, MCF, and BZIP2). Table 2 shows the data set and other important features of each of the 10 simulated workloads. Lastly, the programs are fast forwarded to get past of their initialization phases. After various warm-up periods, each SPLASH2 benchmark is run until the completion of its main loop, each of MIX1 and MIX2 is run for 20 billion user instructions, and each of 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) scheme. CE offers a systematic solution to reduce misses in shared caches. Fig. 6 shows the L2 miss rates (normalized to nominal shared) for all the simulated benchmarks with both S and CE. We show results for CE operating at 1-group (bank) granularity (we will study shortly CE results with various different granularities). CE offers an L2 miss rate reduction over S by an average of 17.6% across all benchmarks and to an extent of 50.1% for MIX3. Two aspects characterize MIX3. First, MIX3 is a mix application with a non-uniform access pattern. This effectively creates non-uniform pressured cache physical locations. CE efficiently exploits such an opportunity and maps cache blocks to highly biased locations with minimum pressures. Second, MIX3 reveals 37.9% intra-processor and 59.5% inter-processor misses while only exposing 2.5%

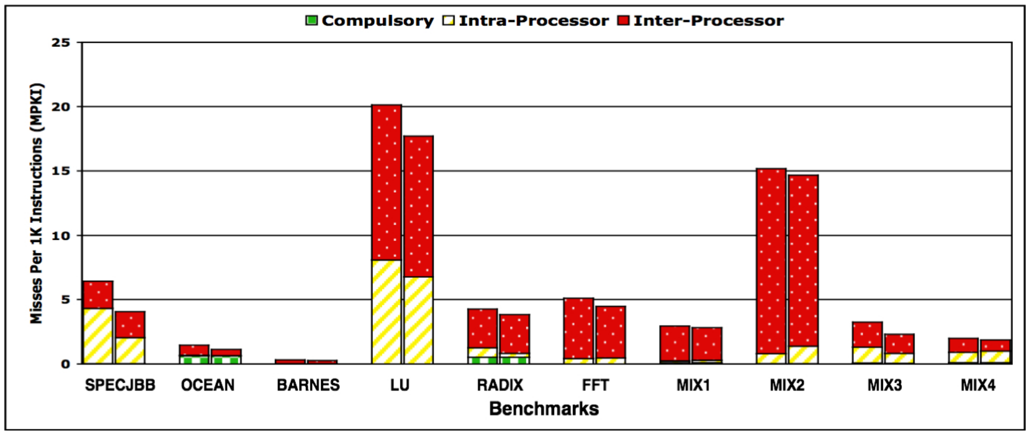


Figure 7. **Misses Per 1K Instructions (MPKI) of Cache Equalizer (CE) and Shared (S) schemes.**

compulsory misses. As explained earlier, CE doesn't influence compulsory misses. CE reduces intra-processor and inter-processor misses. As such, MIX3 becomes very fertile for CE to perk up. CE reduces intra-processor and inter-processor misses for MIX3 by 37.3% and 23.3%, respectively. Intra-processor misses are significantly reduced due to reduced number of references to cache banks (1-group granularity) that are eligible to evict data. Fig. 7 displays the number of references per 1K instructions that lead to compulsory, intra-processor, and inter-processor misses for all the simulated benchmarks. CE offers reductions in intra-processor and inter-processor misses over S by averages of 9.2% and 14.2%, respectively. Overall, CE offers an average reduction of 14.2% in misses per 1K instructions (MPKI) over S.

Besides accessibility patterns and percentages of inter-processor and intra-processor misses, working set sizes of programs affect their eligibility for miss reductions provided by CE. Though OCEAN exposes 41.8% compulsory misses and FFT only 0.12%, OCEAN surpasses FFT and shows 17.6% miss rate reduction compared to only 0.8% reduction shown by FFT. Both programs have homogenous access patterns but expose different working set sizes. FFT and OCEAN exhibit 71.9% and 2% L2 miss rates running over S, respectively. Alongside, MIX1 and MIX2 have absolute homogenous access patterns (very little sharing and uniform cache demands because the very same program is replicated on each core) and high L2 miss rates (76.1% and 94.1%, respectively). CE provides 3.6% and 2.5% miss rate reductions for both over S, respectively. Finally, as shown in Fig. 7, MIX2 manifests additional intra-processor misses over S but demonstrates contrarily fewer inter-processor misses. Intra-processor misses can increase if the number of references from a certain core to a certain bank increases significantly.

The L2 miss rate improvement accomplished by CE comes at the expense of higher interconnect traffic. Fig. 8 depicts a comparison of the number of message-hops per 1K instructions between CE and S. S offers preeminent on-chip network traffic as compared to CE. The increase in the interconnect traffic generated by CE correlates to the use of the cache-the-cache-tag (CTCT) location policy. Upon a request initially made by a certain core, CTCT experiences a 3-way communication scenario (between the requester, the home, and the host tiles). On a block reuse, CE starts mimicing S by directly locating the block at its host tile after a hit at the requester's tracking table. Besides, CTCT introduces more coherence traffic on the NoC for the sake of maintaining consistency between principal and

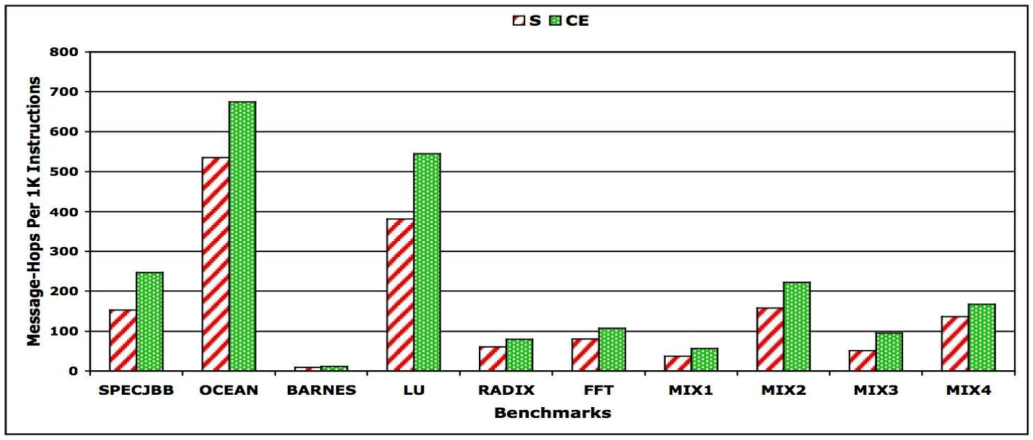


Figure 8. **On-chip network traffic.**

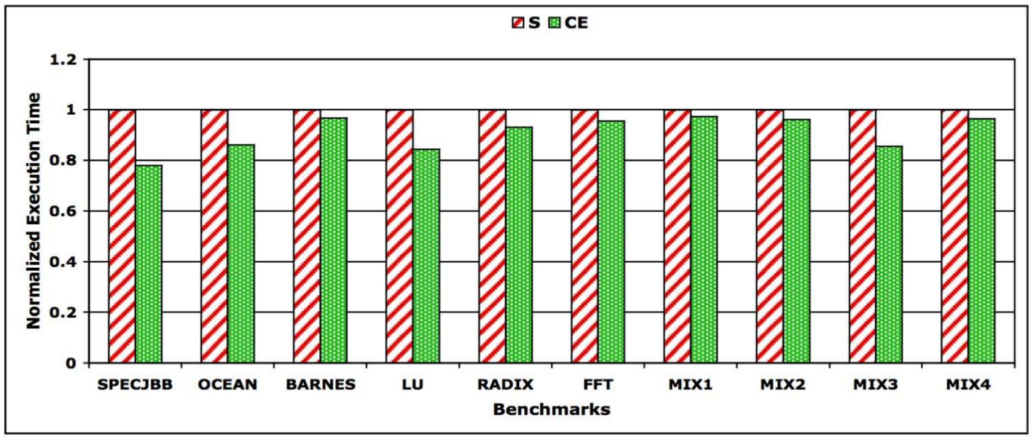


Figure 9. **Execution time (normalized to S).**

replicated tracking entries. Overall, CE increases the NoC traffic (including both data and coherence) by an average of 41.8% over S. This traffic increase, however, doesn't effectively hinder CE from outperforming S. Fig. 9 presents the execution time (normalized to S) of CE and S. Across all benchmarks, CE achieves superiority over S by an average of 9% and by as much as 22% for SPECJBB.

Lastly, Fig. 10 demonstrates the CE behavior running with different granularities (varying from 1-group to 512-group). Given our adopted configuration, 512 is the number of cache sets per bank denoting the upper bound for the number of groups. For each benchmark we show two metrics, the misses per 1K instructions (MPKI) and the cycles per instruction (CPI). For some benchmarks, CE performs best (in terms of CPI) with 1-group (OCEAN, BARNES, MIX3, and MIX4), with 2-group (SPECJBB and LU), with 8-group (FFT), with 16-group (MIX1), and with 512-group (RADIX and MIX2). As explained earlier, in this work we use cache hits to indicate temporal pressures on groups. This sort of pressure is effectively an approximation of real cache demands. Real pressures (successful accesses) on groups at the current epoch might deviate as a consequence of a phase change and accordingly render the pressure array currently utilized in the mapping process a little skewed. As such, increasing granularity might manifest little MPKI irregularities but that typically doesn't translate to high CPI irregularities (e.g., OCEAN). Fur-

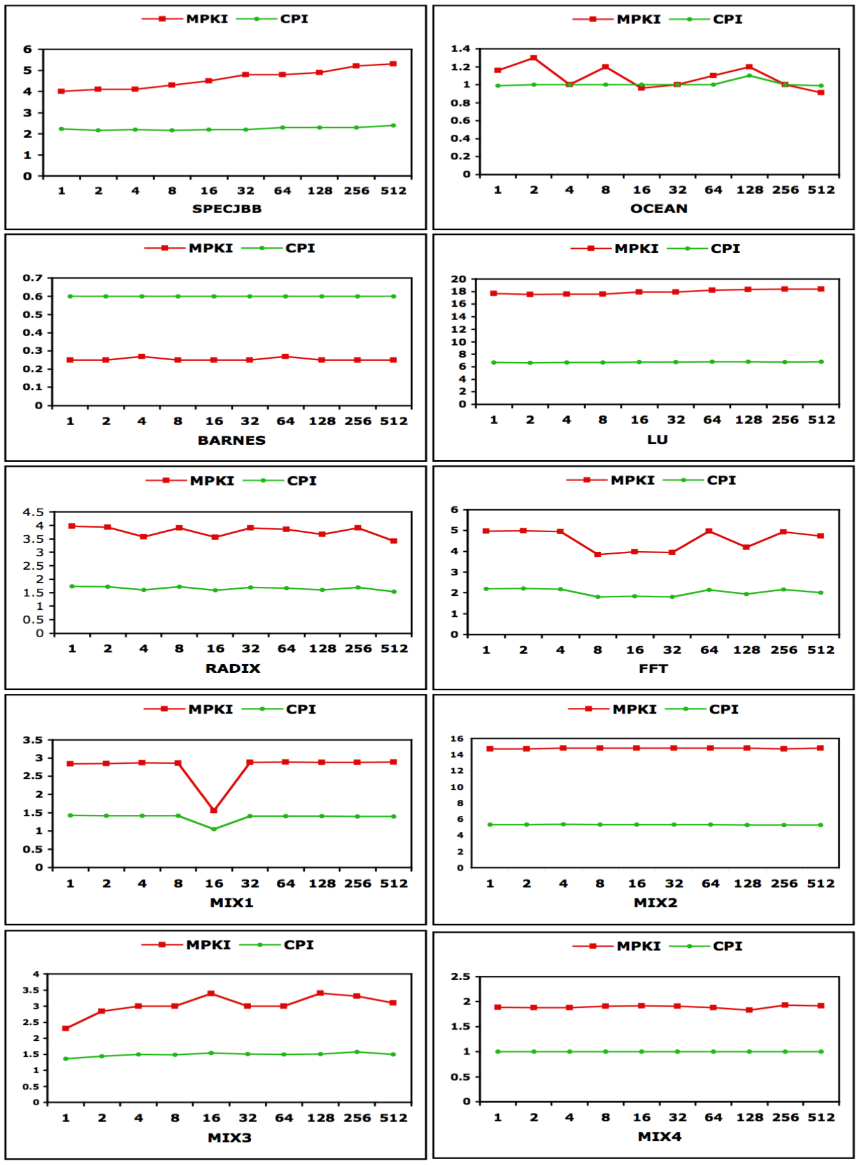


Figure 10. The CE behavior with different granularities (varying from 1-group to 512-group) for all benchmarks.

thermore, uniform pressures might be experienced across groups thus signifying the utmost out of what we can gain from granularity refinements (e.g., MIX4- See also Fig. 4 for a descriptive example). Hence, an increased granularity might not always correlate to better performance; yet even sometimes it might show little MPKI degradation (e.g., SPECJBB). However, under any granularity CE always robustly offers Quality of Service (QoS), which means that the performance of an application remains at least similar, or better than, the baseline scheme [20].

To that end, Fig. 11 shows the *S-Curve*[2] of the CPI improvement provided by CE over the baseline scheme S for a total of 100 simulations (running the 10 workloads each with 10 different granularities). We denote *CE(best)* as the CE that gives the best performance under a certain granularity for a given workload. For example, CE(best) of OCEAN is the 1-group CE. Across all benchmarks, CE(best) achieves better performance than S by an average

[2]An S-Curve is plotted by sorting the data from lowest to highest. Each point on the graph then represents one data-point from this sorted list [20].

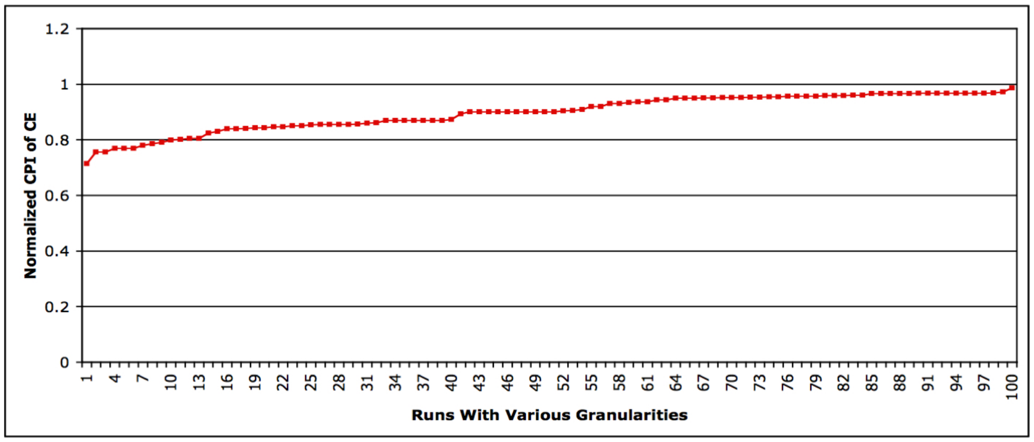


Figure 11. **S-Curve for CPI improvement of CE over S.**

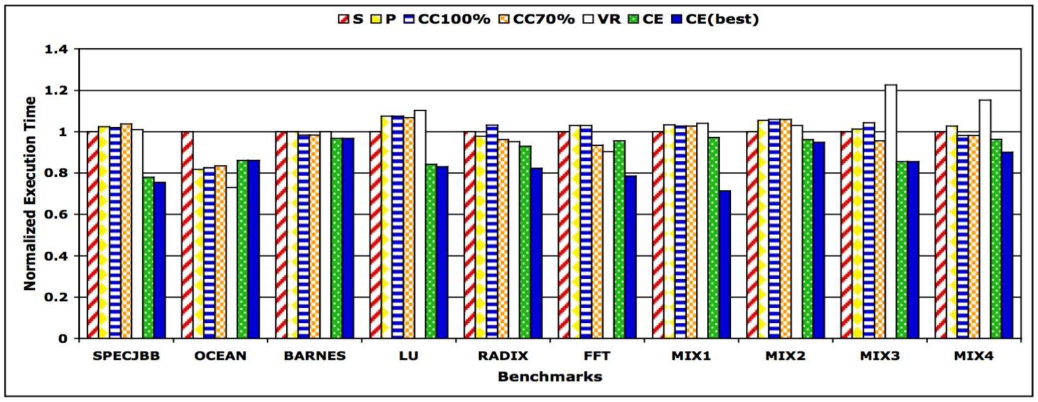


Figure 12. **Execution time of CE and CE(best) as compared to Shared (S), Private (P), Cooperative Caching 100% (CC100%), Cooperative Caching 70% (CC70%), and Victim Replication (VR) schemes.**

of 15.5% and by as much as 28.5%. We can in fact conclude from the results that 1-group granularity might be fair enough for CE (using temporal pressure) to produce good performance and above 16-group might not be really required. Most of the workloads don't perform superlatively with CE running above 16-group granularity. Moreover, MIX2, for instance, that shows superior performance with 512-group granularity (5% improvement over S) exhibits alternatively 3.7% improvement with 1-group granularity.

4.3 Comparing with Related Designs

In addition to comparing with the baseline nominal shared scheme, in this section we compare CE (with 1-group granularity) against the private (P) design, and two related works, cooperative caching (CC) [4], and victim replication (VR) [34]. The performance of CC is highly dependent on the cooperation throttling probability [20]. As such, we evaluate two configurations of CC, one with probability of 100% (CC100%) and another with probability of 70% (CC70%). For a clearer picture, we further show results for CE(best). Similar to CE, VR is based on the nominal shared scheme but approaches the cache management problem reversely via seeking to mitigate the NUCA latency problem inherent in the shared design. CC, on the other hand, is based on the nominal private cache and attempts to reduce intra-processor misses. Fig. 12 depicts the execution time (normalized to S) of all the compared schemes. A main observation is that neither CC nor VR provides QoS (contrary to CE). The basic problem with CC is that it performs spilling without knowing if spilling helps or hurts cache performance [20]. Thus sometimes CC performance surpasses the baseline P (e.g., MIX4 with both CC100% and CC70%) while on some other times it doesn't (e.g., OCEAN with both CC100% and CC70%). CC100% degrades P by an average of 0.2% while CC70% outperforms P by an average of 1.9%. CE and CE(best), on the other hand, outperform P by averages of 9.1% and 15.5%, respectively. CE and CE(best) improve upon CC100% by averages of 9.4% (by as much as 23.6%) and 15.7% (by as much as 26%), respectively. Furthermore, CE and CE(best) improve upon CC70% by averages of 7.2% (by as much as 24.9%) and 13.8% (by as much as 27.2%), respectively. Lastly, the basic problem with VR is that it uncontrollably replicates evicted L1 lines at local L2 banks. Accordingly, if VR fails to offset the lost latency caused by the elevated miss rate (due to replication) from the saved latency (due to replica hits), it can lead to performance degradation. This explains VR's behaviors with LU, MIX1, MIX2, MIX3, and MIX3 benchmarks. For the simulated benchmarks, VR degrades S by an average of 1.5% but improves the performance of some benchmarks by as much as 26.9% (i.e., OCEAN). CE and CE(best) outperform VR by averages of 8.8% and 15.3%, respectively.

5 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 based their work on either the nominal shared or private schemes. Besides, previous work looked at reducing either miss rate or latency in NUCA, or simply miss rate in UCA architectures. We briefly discuss below the proposals that are directly related to ours and describe how our work differs from them.

Srikantaiah et al. [25] is one of the first to present a systematic classification of cache misses in shared CMP caches. Specifically, they classify misses into compulsory, intra-processor, and inter-processor. They then propose adaptive set pinning (ASP), a technique to reduce the latter two classes. Processors are associated to cache sets and solely granted permission to evict blocks from the sets on cache misses. As such, references that could potentially cause inter-processor misses can't interfere between 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 caches. ASP work is based on a UCA architecture. Our work focuses contrarily on large tiled CMP NUCA architectures. A further subtle difference is that ASP modifies the replacement policy such that the cache blocks that would evict other processor's elements are themselves cached on POP caches while we, in contrast, modify the placement policy to avoid misses prior to their occurrences. Finally, another key characteristic of CE is that it actually offers a global equalization method with dependent information being efficiently collected at various coarser (bank) and finer (set) granularities.

Reducing conflict misses in uniprocessor caches has been a hot topic of research [8,14,18,29,30,33]. Summarily, two main directions have been proven to reduce conflict misses effectively: (1) higher set associativity and (2) victim caching (VC) [14, 18]. [25] presents a valuable study on reducing misses in shared CMPs through increasing associativity and cache size, and compares with VC. The bottom line of their study is that with increasing associativity and the size of the L2 cache, the contribution of inter-processor misses to the non-compulsory misses (as percentage of non-compulsory misses) increases and that of intra-processor misses decreases. As such, inter-processor misses become the bottleneck (as far as reducing non-compulsory misses is concerned). The motivation for reducing misses in shared CMP caches thus remains. Finally, a key difference between CE and VC is that VC caches the victims after they are evicted by the replacement policy while we again target the problem from its root (the placement process) and offer a global strategy applicable to distributed CMP caches. We compared in Section 4.3 our scheme with victim replication (VR) [34] that offers essentially a dynamic VC.

Chang and Sohi [4] proposed *cooperative caching* (CC) based on the private scheme to create 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 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 compared CE against CC in Section 4.3. 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 [20] 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 [1, 7, 13, 23]. Sherwood et al. [23] 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. 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 [7] 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 of its requests from various cores. Their proposed mechanism relies on the OS to spread the working set of a single program across many colors when exposing high capacity demands. In comparison to these schemes, CE performs block-grain placement without involving OS and provides a transparent solution.

Lastly, many researchers have explored CMP cache management designs to reduce cache hit latency in CMP NUCA caches. Zhang and Asanović [34] 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 (instead of evicting or writing them back to their SHTs) within the local L2 cache banks. Chishti et al. [6] proposed *CMP-NuRAPID* that controls replication based on usage patterns. Beckmann et al. [2] proposed *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*. Hammoud et al. [10] proposed *ACM* relying on prediction to perform data migration. Guz et al. [9] presented a new shared cache architecture and diverted only shared data to centered cache banks close to all cores. Chaudhuri [5] 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. [11] proposed *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, the shared into fixed address-interleaved on-chip locations, and instructions into non-overlapping clusters of L2 cache banks. Huh et al. [12] proposed a spectrum of degrees of sharing to manage NUCA caches.

In summary, while we clearly 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-relaxed data placement process for NUCA caches. Third, we present a simple novel framework to monitor CMP caches at various finer granularities. Such a framework can be generally applied to a variety of CMP cache schemes. For instance, it can be utilized by migration mechanisms (e.g., [10]) to guide promotions of blocks. Also, it can be used by schemes that offer capacity sharing for private caching (e.g., [4]) to guide spilling of blocks.

6 Conclusions and Future Directions

Cache management in CMP is crucial to fuel its performance growth. This paper investigates the shared NUCA misses problem (caused by destructive interferences) and proposes cache equalizer (CE), a mechanism that provides isolation capabilities to reduce misses. We indicate the significance of applying pressure-aware placement strategies to CMP NUCA caches in order to achieve performance isolation. CE embarks upon such a key factor and suggests mapping cache blocks to the on-chip last level cache based on temporal pressures. Pressure information (how many lines yield cache hits during a time interval) is collected at a group (composing of cache sets) granularity and recorded in an array at the memory controller. On an incoming cache block, CE looks up the pressure array, selects the minimum pressure, and places the block at the corresponding group. Simulation results using a full system simulator demonstrate that CE reduces the shared cache misses by an average of 21.9% (translates to 15.5% execution time average improvement) and by as much as 50.1%. Furthermore, results show that CE outperforms victim replication [34] and cooperative caching [4] by averages of 15.3% and 14.7%, respectively.

As a future direction, CE can be studied with further kinds of pressures. For instance, we can employ spatial pressure (how many unique lines yield cache hits during a time interval) and based on that explore CE's behavior. We can also have a combination of various pressure kinds and scrutinize the consequent performance. Lastly, we can incorporate more parameters to the mapping process. When CE decides upon a group to host an incoming cache block, we can check up the Manhattan distance between the requester tile and the L2 bank of the selected group. We can then choose to map the block to a different group (with a little higher pressure) at a bank closer to the requester tile thus saving latency on reuse. Clearly, this would require measuring the gain (better latency on block reuse) and the loss (potentially higher misses) and deciding on placements accordingly.

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," *Proc. Int'l Symp. High-Perf. Computer Arch*, Feb. 2009.

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

[3] 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.

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

[5] 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.

[6] 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.

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

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

[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. 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

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

[34] 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.