Dynamic Cache Clustering for Chip Multiprocessors

Mohammad Hammoud, Sangyeun Cho, and Rami 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 proposes DCC (Dynamic Cache Clustering), a novel distributed cache management scheme for large-scale chip multiprocessors. Using DCC, a per-core cache cluster is comprised of a number of L2 cache banks and cache clusters are constructed, expanded, and contracted dynamically to match each core's cache demand. The basic trade-offs of varying the on-chip cache clusters are average L2 access latency and L2 miss rate. DCC uniquely and efficiently optimizes both metrics and continuously tracks a near-optimal cache organization from many possible configurations. Simulation results using a full-system simulator demonstrate that DCC outperforms alternative L2 cache designs.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: System architectures

General Terms

Design, Management, Experimentation, Performance

Keywords

Chip Multiprocessor (CMP), Non-Uniform Cache Architecture (NUCA)

1. INTRODUCTION

As the industry continues to shrink the size of transistors, chip multiprocessors (CMPs) are increasingly becoming the trend of computing platforms. IBM recently introduced the Power6 processor with dual high performance cores each supporting 2-way multithreading [18]. Niagara2 has been released by Sun Microsystems with 8 SPARC cores each supporting 8 hardware threads all on a single die [9]. This shift towards CMPs, however, presents new key challenges to computer architects. One of these challenges is the design of an efficient memory hierarchy especially in light of some conflicting requirements: the reduction of the average L2 access latency (AAL) and the L2 miss rate (MR) [1].

Tiled CMP architectures have recently been advocated as a scalable design [17]. They replicate identical building blocks (tiles) connected over a switched network on-chip (NoC). A tile typically incorporates a private L1 cache and an L2 cache bank. Traditional practices of CMP cache organizations are either shared or private. The shared strategy implies that the on-chip cores share the physically distributed L2 banks. On the other hand, the private scheme entails that each core has its own L2 bank. The degree of sharing, or the number of cores that share a given pool of cache banks, could also be set somewhere between the shared and the private designs. The work in [8] explores five static sharing degrees (1, 2, 4, 8, and 16) for caches in a 16-core CMP. For instance, a sharing degree of 2 means that every two CMP cores share their L2 cache banks.

One of the main advantages of the private scheme is the proximity of data to requester cores. Each core maps and locates the requested cache blocks to and from its corresponding L2 cache bank. As such, cache blocks are typically read very fast. However, if a per-core L2 bank is small relative to a working set size, many costly accesses could occur either to the main memory or to some neighboring L2 banks. Besides, shared data reduces the available on-chip cache capacity as each core replicates a copy at its L2 bank. This could increase the L2 miss rate significantly, and if not offset by replica hits, performance could potentially degrade.

Contrary to the private design, the shared scheme resourcefully utilizes the available cache capacity by caching only a single copy of a shared block at a tile, referred to as the *home tile* of the block. However, the shared strategy offers non-uniformity in L2 access latency. The latency to access B at L2 essentially depends on the distance between the requester core and B's home tile. This model is referred to as a Non Uniform Cache Architecture (NUCA) [8]. In NUCA, cache blocks of a small working set running on a core, may map far away from the core thereby deteriorating the average L2 access latency and possibly degrading the system performance.

In reality, computer applications exhibit different cache demands. Furthermore, a single application may demonstrate different phases corresponding to distinct code regions invoked during its execution [15]. A program phase can be characterized by different L2 cache misses and durations.

This work is supported, in part, by NSF awards CCF-0702452 and CCF-0702236 as well as a research gift from Intel.

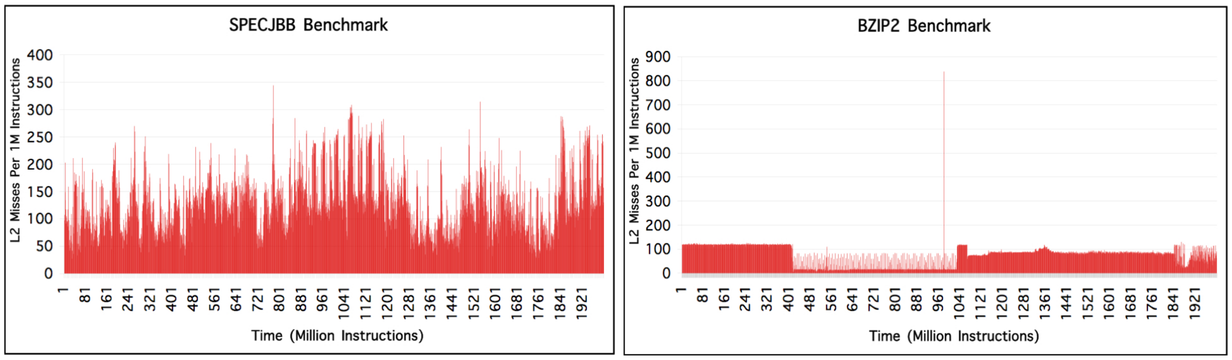


Figure 1: Cache demands are irregular among different applications and within the same application.

Fig. 1 illustrates the L2 misses per 1 million instructions experienced by SPECJBB and BZIP2 from the SPEC2006 benchmark suite [20]. The two workloads were run separately on a 16-tile CMP platform (details about the platform and the utilized experimental parameters are described in Sections 2 and 5). The behaviors of the two programs are clearly different and demonstrate characteristically different working set sizes and irregular execution phases.

The traditional private and shared designs are subject to a principal deficiency. They both entail static partitioning of the available cache capacity and don't tolerate the variability among different working sets and phases of a working set. For instance, a program phase with high cache demand would require enough cache capacity to mitigate the effect of high cache misses. On the other hand, a phase with less cache demand would require smaller capacity to mitigate the NoC communications. Static designs provide either fast accesses or capacity but not both. A crucial step towards designing an efficient memory hierarchy is to offer both fast accesses and capacity.

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

The paper makes the following contributions:

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

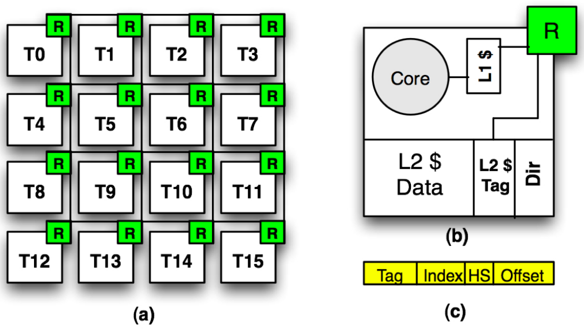


Figure 2: The adopted CMP model (figure not to scale). (a) 16-Core tiled CMP model. (b) The microarchitecture of a single tile. (c) Components of a cache block physical address (HS = Home Select).

The rest of the paper is organized as follows. Section 2 presents the baseline architecture. A brief background on some of the fixed cache designs is given in Section 3. Section 4 delves into the proposed DCC scheme. We evaluate DCC in Section 5. Section 6 recapitulates some related work, and conclusions and future directions are given in Section 7.

2. BASELINE ARCHITECTURE

This paper assumes a 2D 16-tile CMP model as portrayed in Fig. 2(a). There are two main advantages to the tiled CMP architecture: They scale well to larger processor counts and can easily support families of products with varying number of tiles, including the option of connecting multiple separately tested and speed-binned dies within a single package [24]. The CMP model employs a 2D mesh switched network, and replicated tiles are connected to one another via the network and per-tile routers. Each tile includes a core, a private L1 cache, and an L2 cache bank as shown in Fig. 2(b). Besides, a directory table (Dir) is used to maintain the L1 coherence in case of shared L2, and to keep the coherence of both L1 and L2 in case of private L2. The model introduces a NUCA design. An access to an L2 bank on another tile traverses the NoC fabric and experiences varying latencies depending on the NoC congestion and the

Manhattan distance between the requester and the target tiles. The dimension-ordered (XY) routing algorithm [13] is employed where packets are first routed in the X and then the Y directions.

3. BACKGROUND

3.1 Fixed Cache Schemes

The physically distributed L2 cache banks of a tiled CMP can be organized in different ways. At one extreme, each L2 bank can be made private to its associated core. This corresponds to contracting a traditional multi-chip multi-processor onto a single die. At the other extreme, all the L2 banks can be aggregated to form one logically shared L2 cache (shared scheme). Alternatively, the L2 cache banks can be organized at any point in between private and shared. More precisely, [8] defines the concept of *sharing degree* (SD) as the number of processors that share a pool of L2 cache banks. In this terminology, an SD of 1 means that each core maps and locates the requested cache blocks to and from its corresponding L2 bank (private scheme). An SD of 16, on the other hand, means that each of the 16 cores shares with all other cores the 16 L2 banks (shared scheme). Similarly, an SD of 2 means that 2 of the cores share their L2 banks. Fig. 3 demonstrates five sharing schemes with different sharing degrees (SD= 1, 2, 4, 8, and 16) as implied by our 16-tile CMP model. We refer to these sharing schemes as Fixed Schemes (FS) to distinguish them from our proposed dynamic cache clustering (DCC) scheme.

3.2 Fixed Mapping and Location Strategies

At an L2 miss, a cache block, B, is fetched from main memory and mapped to an L2 cache bank. A subset of bits from the physical address of B, denoted as the home select (HS) bits (see Fig. 2(c)), can be utilized and adjusted to map B as required to any of the shared regions of the aforementioned fixed schemes. If B is a shared block, it might be mapped to multiple shared regions. However, as the sharing degree (SD) increases, the likelihood that a shared block maps within the same shared cache region increases. As such, FS16 maps each shared block to only one L2 bank. We identify the tile at which B is mapped to, as a *dynamic home tile* (DHT) of B. For any of the above defined fixed schemes, the utilized HS bits depend on SD. Furthermore, the function that uses the HS bits of B's physical address to designate the DHT of B can be used to subsequently locate B.

3.3 Coherence Maintenance

The fixed scheme FS16 maintains the exclusiveness of shared cache blocks at the L2 level. Thus, FS16 requires maintaining coherence only at the L1 level. However, for the other fixed schemes with lower SDs, each L2 shared region might include a copy of a shared block. This, consequently, requires maintaining coherence at both, the L1 and the L2 levels. To achieve such an objective, two options can be employed: a centralized and a distributed directory protocols. The work in [8] suggests maintaining the L1 cache coherence by augmenting directory status vectors in the L2 tag arrays. A directory status vector associated with a cache block, B, designates the copies of B at the private L1 caches. For the L2 cache coherence, [8] utilizes a centralized engine. A centralized coherence protocol is deemed non-scalable especially with the advent of medium-to-large scale CMPs and the projected industrial plans [17]. A high-bandwidth distributed on-chip directory can be adopted to accomplish the task [17, 25].

By employing a distributed directory protocol, directory information can be decoupled from cache blocks. A cache block B can be mapped to its DHT, specified by the underlying cache organization. On the other hand, directory information that corresponds to B can be mapped independently to a potentially different tile, referred to as the static home tile (SHT) of B. The SHT of B is typically determined by the home select (HS) bits of B's physical address (see Fig. 2(c))[1]. For the adopted 16-tile mesh-based CMP model, a duplicate tag embedded with a 32-bit directory status vector can represent the directory information of B. For each tile, one bit in the status vector indicates a copy of B at its L1, and another bit indicates a copy at its L2 bank. To reduce off-chip accesses, Dir (see Fig. 2(b)) can always be checked by any requester core to locate B at its current DHT, using 3-way cache-to-cache transfers.

4. DYNAMIC CACHE CLUSTERING (DCC)

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

4.1 Average Memory Access Time (AMAT)

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

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

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

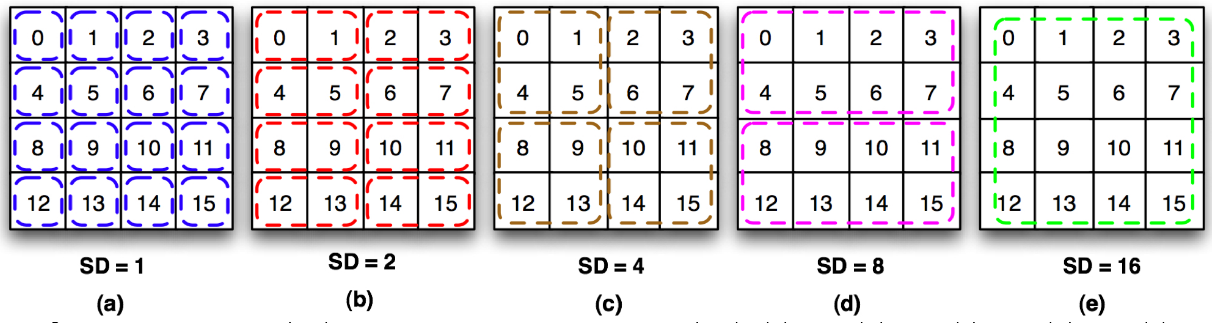
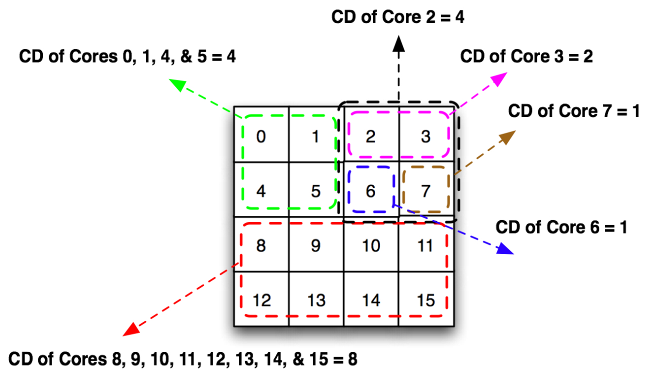


Figure 3: Fixed Schemes (FS) with different sharing degrees (SD). (a) FS1 (b) FS2 (c) FS4 (d) FS8 (e) FS16


CD of Core 2 = 4
CD of Cores 0, 1, 4, & 5 = 4
CD of Core 3 = 2
CD of Core 7 = 1
CD of Core 6 = 1
CD of Cores 8, 9, 10, 11, 12, 13, 14, & 15 = 8
Figure 4: A possible cache clustering configuration that the DCC scheme can select dynamically at runtime.

ric also typically translates into an improvement in system performance.

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

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

4.2 The proposed Scheme

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

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

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

4.3 DCC Mapping Strategy

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

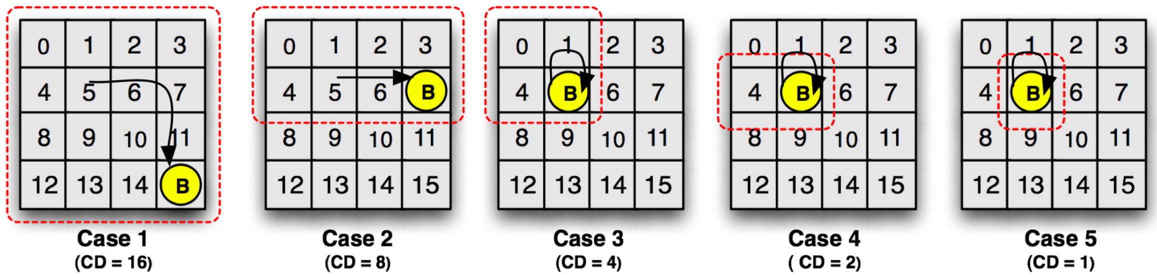


Figure 5: An example of how the DCC mapping strategy works. Each case depicts a possible DHT of the requested cache block B with HS = 1111 upon varying the cache cluster dimension (CD) of the requester core 5 (ID = 0101).

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

Table 1: Masking Bits (MB) for a 16-tile CMP Model.

are used if $CD_i = 1$. More formally, the following function determines the DHT of B:

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

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

4.4 DCC Algorithm

The AMAT metric defined in equation (2) could be utilized to judiciously gauge the benefit of varying the cache cluster dimension of a certain core, i. We suggest a run-time monitoring mechanism that can infer enough about a running process behavior and feed the collected information to an algorithm that can make related architecture-adaptive decisions. In particular, a process P starts running on core i with an initial cache cluster (i.e., $CD_i = 16$). After a period time T, the $AMAT_i$ experienced by P is evaluated and stored, and the cache cluster of core i is contracted (or expanded if chosen so and CD_i has started from a value smaller than 16). This is the initial $AMAT_i$ of P. At every potential *re-clustering* point a new $AMAT_i$ ($AMAT_i$ current) is evaluated and deducted from the previously stored $AMAT_i$ ($AMAT_i$ previous). Suppose, for instance, that a contraction action has been initially taken. Accordingly, a resultant positive value of the difference means that $AMAT_i$ has degraded after contracting the cache cluster of core i. As such, we infer that P didn't actually benefit from the contraction process. On the other hand, a negative outcome means that $AMAT_i$ has improved after contracting the cache cluster of core i and we infer that P benefited in fact from the contraction process. Let $\triangle$ be defined as follows:

The DCC Algorithm Executed on Each Core *i*

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

Figure 6: The dynamic cache clustering algorithm.

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

Therefore, a positive $\triangle_i$ indicates a *loss* while a negative one indicates a *gain*. At every re-clustering point, the value of $\triangle_i$ is fed to the DCC algorithm executing on core i (the DCC algorithm is local to each CMP core). The DCC algorithm makes in return some architecture-adaptive decisions. Specifically, if the gain is less than a certain threshold, T_g, the DCC algorithm decides to keep the cache cluster as it is for the next period time T. However, if the gain is above T_g, the DCC algorithm decides to contract the cache cluster a step further, predicting that P is likely to gain more by the contraction process. On the other hand, if the loss is less than a certain threshold, T_l, the DCC algorithm decides to keep the cache cluster as it is for the next period time T. If the loss is above T_l, the DCC algorithm decides to expand the cache cluster to its previous value (one step backward) assuming that P is currently experiencing a high cache demand. Fig. 6 shows the suggested algorithm.

4.5 DCC Location Strategy

A core i can contract or expand its cache cluster at every re-clustering point. Hence, the generic mapping function defined in equation (3) can't be utilized straightforwardly to locate blocks that have been previously mapped by core i to the L2 cache space. Fig. 7(a) illustrates an example of core 0 (with CD = 8) fetched and mapped a cache block B (with HS=1111) to DHT 7 determined by equation (3). Fig. 7(b) demonstrates a scenario with core 0 contracting its CD from 8 to 4 and subsequently requesting B from L2. With current CD = 4, equation (3) designates tile 5 to be the current

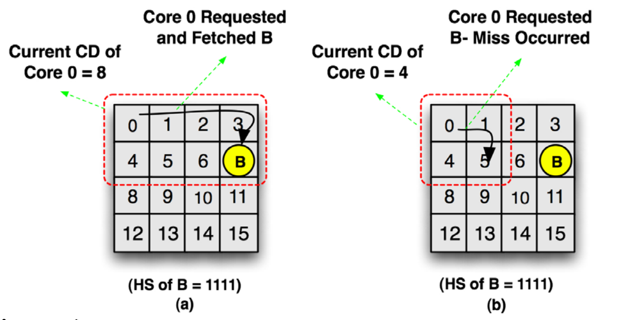


Figure 7: An example of the DCC location strategy using equation (3). (a) Core 0 with current CD = 8 requesting and mapping a block B to DHT 7. (b) Core 0 missed B after contracting its CD from 8 to 4 banks.

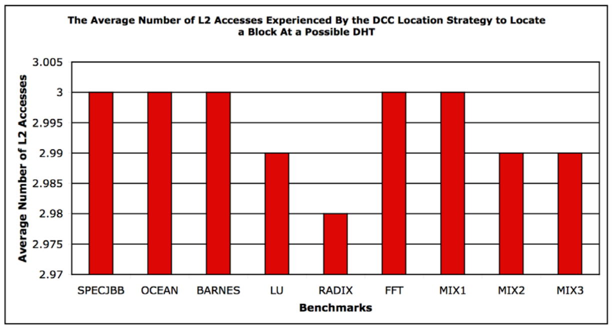


Figure 8: The average behavior of the DCC location strategy.

DHT of B. However, if core 0 simply sends its request to tile 5, a false L2 miss will occur. After a miss to tile 5, B's SHT (tile 15), which keeps B's directory information, can be accessed to locate B at tile 7 (assuming this is the only tile currently hosting B). This is a quite expensive process as it requires multiple inter-tile communications between tiles 0, 5, 15, 7 again, and eventually 0 to fulfill the request. A better solution could be to straightforwardly send the L2 request to B's SHT instead of sending it first to B's current DHT and then possibly to B's SHT. This still might not be acceptable because it entails 3-way cache-to-cache communications between tiles 0, 15, and a prospective host of B. Such a strategy fails to exploit *distance locality*. That is, it incurs significant latency to reach the SHT of B though B resides in close proximity. A third possible solution could be to re-copy all the blocks that correlate to core 0 to its updated cache cluster upon every re-clustering action. Clearly, this is a costly and complex process because it will heavily burden the NoC with superfluous data messages.

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

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

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

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

$$AV(1) = 1 \qquad (6)$$

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

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

For a CMP with n tiles, the number of possible cluster dimensions is $ln(n)$. Hence, the average number of possible DHTs is $1 + \frac{1}{2}ln(n)$. Specifically, for n = 16, the average number of possible DHTs is $1 + \frac{1}{2}ln(16) = 3$. Fig. 8 shows simulation results for the average number of L2 accesses experienced by the DCC location strategy using 9 benchmarks (details about the benchmarks and the utilized experimental parameters are described in Section 5). Clearly, the results confirm our theoretical analysis.

Multiple copies of a cache block B can map to multiple cache clusters of multiple cores. As such, a request from a core C to a block B can hit at multiple possible DHTs. However, if a miss occurs at the DHT of B that corresponds to the current cache cluster dimension of C (current DHT), though a hit occurs at some other possible DHT, a decision is to be made of whether to copy B to B's current DHT or not. If *none* of the possible DHTs that host B resides currently inside the cache cluster of C, we copy B to its current DHT, otherwise we do not. The rationale behind this policy is to minimize the average L2 access latency. Specifically, a possible DHT hosting B and contained inside C's cache cluster is always *closer* to C than is the current DHT. Thus, we don't copy B from that possible DHT to its current DHT. The decision of whether to copy B to its current DHT can be made by B's SHT. The SHT of B retains B's directory information and is always accessed by our location strategy (B's SHT is a possible DHT).

Finally, after inspecting B's SHT, if a copy of B is located on-chip (i.e, mapped by a different core with different CD) and none of the possible DHTs is found to host B, the SHT satisfies the request from the host that is *closest* to C (in case many hosts are located). Fig. 9(a) illustrates a scenario where core 0 with CD = 4 issues a request to cache block B with HS= 1111. Simultaneous L2 requests are sent to all the possible DHTs of B (tiles 0, 1, 5, 7, and 15). Misses occur at all of them. The directory table at B's SHT (tile 15) is inspected. A copy of B is located at tile 3 indicated by the corresponding bit within the directory status vector of

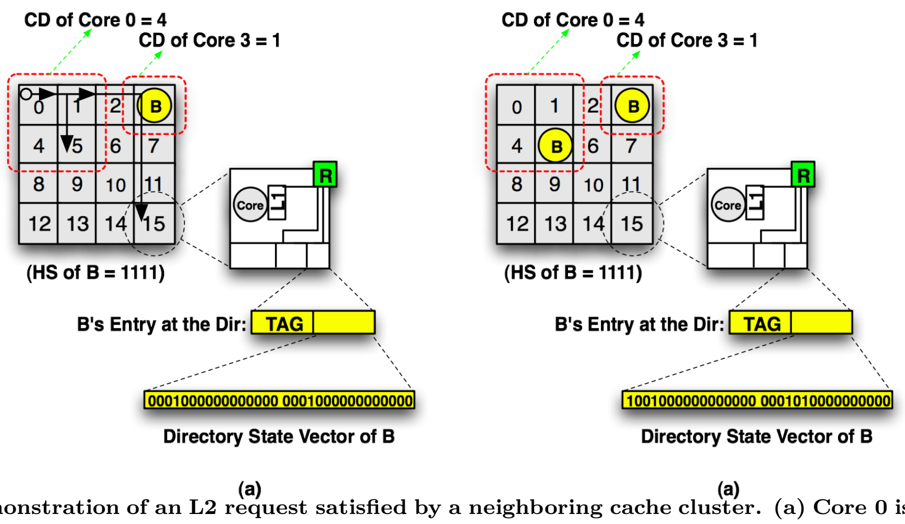


Figure 9: A demonstration of an L2 request satisfied by a neighboring cache cluster. (a) Core 0 issued an L2 request to block B. (b) Core 3 satisfied the L2 request of Core 0 after re-transmitted to it by B's SHT (tile 15).

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

Table 2: System parameters

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

Table 3: Benchmark programs

B. Fig. 9(b) depicts B's directory state and residences after it has been forwarded from tile 3 to its current DHT (tile 5) and to the L1 cache of the requester core 0. The figure depicts only copies at the L2 banks within tiles. However, the shown directory status vector reflects the presence of B at the L1 cache of core 0.

5. QUANTITATIVE EVALUATION

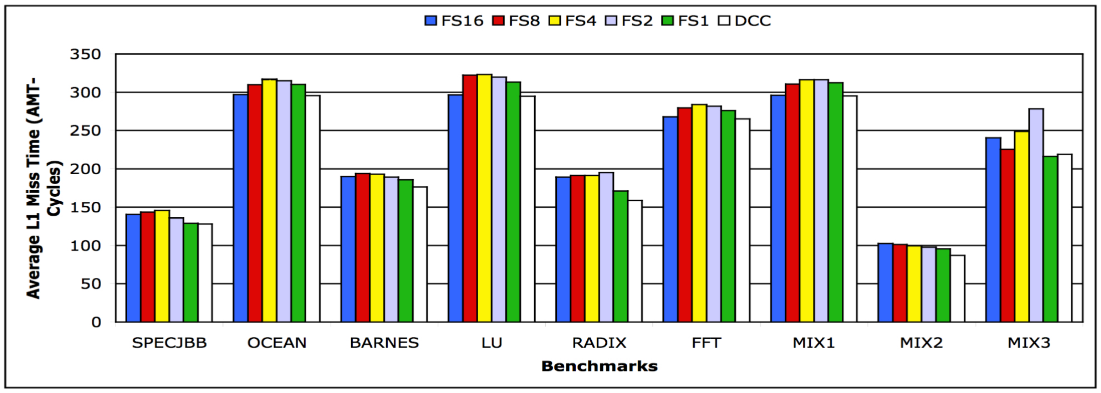
5.1 Methodology

Evaluations presented in this paper are based on detailed full-system simulation using Simics 3.0.29 [22]. We simulate a tiled CMP machine model similar to the one described in Section 2 (see Fig. 2(a)). The platform comprises 16 UltraSPARC-III Cu processors and runs under the Solaris 10 OS. Each processor uses in-order issue, and has a 16KB I/D L1 cache and a 512KB L2 cache bank. Table 2 shows a synopsis of the main architectural parameters. We compare the effectiveness of the DCC scheme against the 5 alternative static designs, FS1, FS2, FS4, FS8, and FS16, detailed in Section 3, and the cooperative caching scheme [3]. Cache modules with a distributed MESI-based directory protocol for all the evaluated schemes have been developed and plugged into Simics. We faithfully verified and tested the employed distributed protocol. Finally, we implemented the XY-routing algorithm and modeled congestion (coherence and data) over the adopted mesh-based NoC.

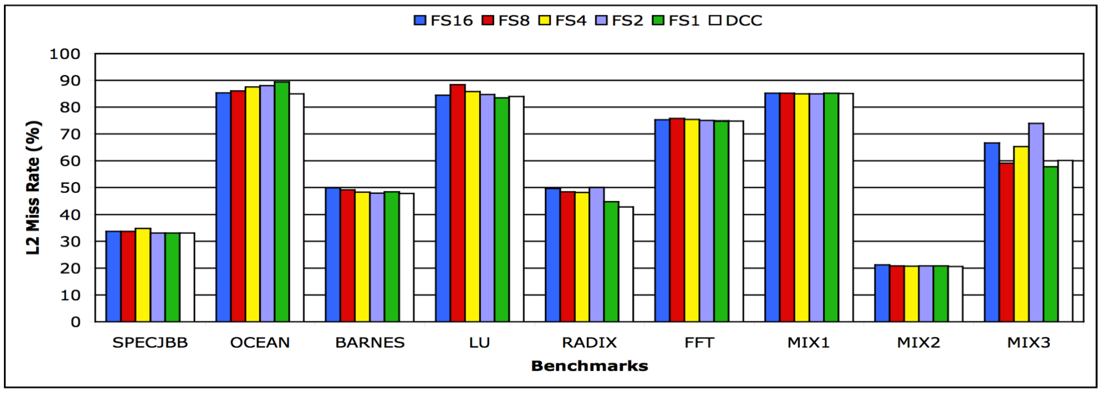
We use a mixture of multithreaded and multiprogramming workloads to study the compared schemes. For multithreaded workloads we use the commercial benchmark SPECjbb, and 5 other shared memory benchmarks from the SPLASH2 suite [23] (Ocean, Barnes, Lu, Radix, and FFT). Three multiprogramming workloads have been composed from 5 representative SPEC2006 [20] applications (Hmmer, Sphinx, Milc, Mcf, and Bzip2). Table 3 shows the data set and other important features of each of the 9 simulated workloads. Lastly, we ran Ocean, Barnes, Radix, and FFT in full and stopped the remaining benchmarks after a detailed simulation of 20 Billion Instructions.

5.2 Comparing With Fixed Schemes

This section presents the experimental evaluation of the DCC scheme against the 5 alternative static designs, FS1, FS2, FS4, FS8, and FS16. The set of parameters, the period time T, the loss and gain thresholds T_l and T_g ($\{T, T_l, T_g\}$) utilized by the DCC algorithm are different for each simulated benchmark and selected from amongst 10 sets presented in the next subsection. The sensitivity analysis in Section 5.3 shows that the results are not much dependent on the value of parameters $\{T, T_l, T_g\}$. First of all, we study the effect of the average L1 miss time (AMT), defined in equation (1), across the compared schemes. Fig. 10(a) portrays the AMTs experienced by the 9 simulated workloads. A main observation is that no single static scheme provides the best AMT for all the benchmarks. For instance, Ocean and MIX1 are best performing under FS16. On the other hand, SPECjbb and Barnes perform superlative under FS1. As such, a single static scheme fails to adapt to the vari-



(a)



(b)

Figure 10: Results for the simulated benchmarks. (a) Average L1 Miss Time (AMT) in cycles. (b) L2 Miss Rate.

eties across the working sets. The DCC scheme, however, dynamically adapts to the irregularities exposed by different working sets and always provides performance comparable to the best static alternative. Besides, the DCC scheme sometimes even surpasses the best static option due to the fine-grained caching solution it offers (see Subsection 4.2). This is clearly exhibited by SPECjbb, Ocean, Barnes, Radix, and MIX2 benchmarks. Fig. 10(a) illustrates the outperformance of DCC over FS16, FS8, FS4, FS2, and FS1 by an average of 6.5%, 8.6%, 10.1%, 10%, and 4.5% respectively across all benchmarks, and to an extent of 21.3% for MIX3 over FS2. In fact, DCC surpasses FS1, FS2, FS4, FS8, and FS16 for all the simulated benchmarks except one. That is, MIX3 running under FS1. The current version of the DCC algorithm doesn't adaptively select an optimal set of thresholds $\{T, T_l, T_g\}$. Thus, we expect that such a diminutive superiority (1.1 %) of FS1 over DCC for MIX3 is simply because of that reason. Nevertheless, DCC always favorably converges to the best static option.

The DCC scheme manages to reduce the L2 miss rate (MR) as it varies cache clusters per cores depending on their L2 demands. Fig. 10(b) illustrates the MR produced by each of the 6 compared schemes for the simulated benchmarks. As described earlier, when the sharing degree (SD) amongst the static designs decreases, MR increases. This is because the likelihood that a shared cache block maps within the same shared cache region decreases. For instance, the L2 miss rate of Ocean increases monotonically as SD decreases. On the other hand, the L2 miss rate of Radix outshines with FS1. This is due to the fact that additional cache resources might not always correlate to better L2 miss rates [16]. A workload might manifest poor locality, and cache accesses could sometimes be ill-distributed over sets. We observed that Radix has a great deal of L2 misses produced by heavy interferences of cores on cache sets (inter-processor misses). The DCC scheme, however, efficiently resolves this problem and resourcefully exploits the available cache capacity. DCC improves the Radix L2 miss rate by 4.2% and generates 7.3% better AMT.

As the sharing degree (SD) of the static designs and the cache cluster dimension (CD) of the DCC scheme change, the hits to local L2 and to remote L2 banks also change. The hits to local L2 banks monotonically increase as SD decreases. This is revealed in Fig. 11 that depicts the data accesses breakdown of all the simulated benchmarks. Increases in hits to local L2 banks improves the average L2 access latency (AAL) as it decreases inter-tile communications, but, on the other hand, it might exacerbate MR thus causing both, AAL and MR to race in conflicting directions. For instance, though FS1 produces the best local L2 hits for Ocean, Fig. 10(b) shows that Ocean has the worst MR. Increasingly mapping cache blocks to local L2 banks can boost capacity misses, and if the gain acquired from higher local hits doesn't offset the loss incurred from higher memory accesses, performance will degrade. This explains the AMT behavior of Ocean under FS1. DCC, however, increases hits to local L2 banks but in a controlled and balanced fashion that it doesn't increase MR to an extent that ruins AMT.

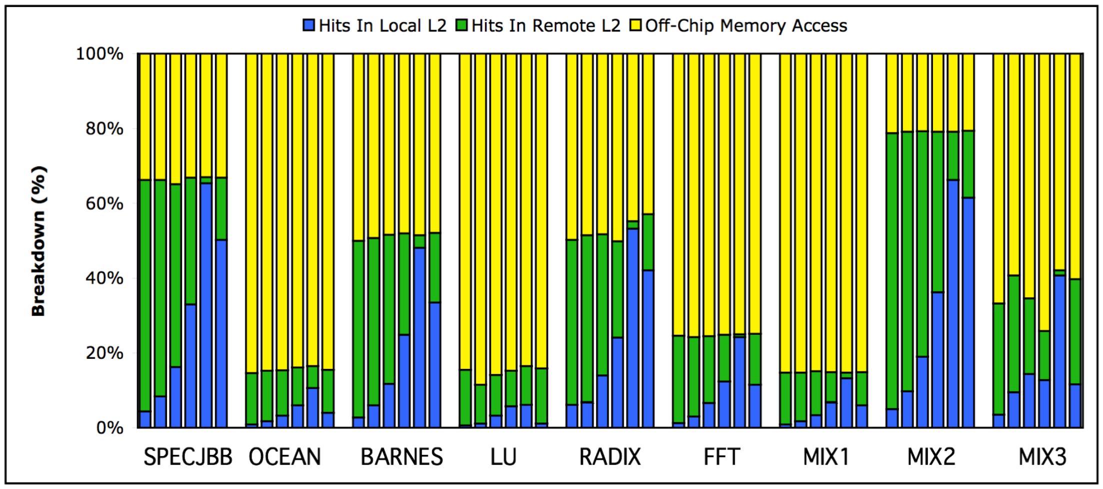


Figure 11: Memory access breakdown. Moving from left to right, the 6 bars for each benchmark are for FS16, FS8, FS4, FS2, FS1, and DCC schemes, respectively.

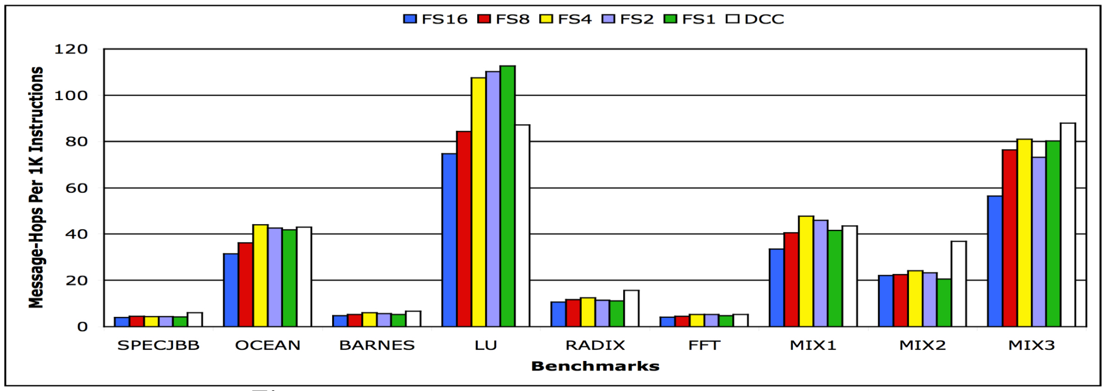


Figure 12: On-Chip network traffic comparison.

Thus, for instance, DCC degrades hits to local L2 banks of Ocean by 62.3% over FS1 but improved in return its MR by 4.9%. As a result, DCC generated 4.7% better AMT for Ocean as compared to FS1. This reveals the robustness of DCC as a mechanism that tunes up AAL and MR so as to obtaining high performance from CMP platforms.

Fig. 12 depicts the number of message-hops (including both data and coherence) per 1k instructions for all the simulated applications with the 6 compared schemes. FS16 offers the preeminent on-chip network traffic savings (except for MIX2) as compared to other schemes. For each L1 miss, FS16 issues always one corresponding L2 request and that is to the static home tile (SHT) of the requested cache block, B. In contrary, the number of L2 requests issued by the remaining static designs depends on the access type. For a write request, B's SHT is accessed (in addition to accessing the shared region of the requester core) in order for the requester core to obtain an exclusive ownership on B. Besides, for a read request which misses at the shared region, B's SHT is also accessed to check if B resides on-chip (on some other shared regions) before an L2 miss is reported. However, for read requests that hit in the shared regions, an L1 miss corresponds always to a single L2 request. As such, if the message-hops gain (G) from read hits surpasses the message-hops loss (L) from read misses and writes, the interconnect traffic outcome of either FS1, FS2, FS4, or FS8 will improve over FS16. This explains the behavior of MIX2 with FS1. On the other hand, if L surpasses G, the interconnect traffic outcome of FS16 will improve over the 4 alternative static schemes. This explains the behavior of the remaining benchmarks. Finally, DCC results in increased traffic due to multicast location requests (on average 3 per request). On average, DCC increases interconnect traffic by 41.1%, 24.7%, 11.7%, 16.6%, and 21.5% over FS16, FS8, FS4, FS2, and FS1, respectively. This increase in message-hops doesn't effectively hinder DCC from outperforming the static designs as demonstrated in Fig. 10(a).

Lastly, Fig. 13 presents the execution time of the compared schemes, all normalized to FS16. For Barnes, Radix, MIX1, MIX2, and MIX3, the superiority of DCC in AMT over the static designs translates to better overall performance. However, diminutive AMT improvements of DCC by 0.6% over FS1, 0.5% over FS16, 0.6% over FS16, and 0.9% over FS16 for SPECjbb, Ocean, Lu, and FFT, respectively didn't translate to an effectively better overall performance. Nonetheless, the main objective of DCC is still successfully met. DCC performs favorably comparable to the best static alternative. DCC outperforms FS16, FS8, FS4, FS2, FS1 by an average of 0.9%, 3.1%, 3.6%, 2.8%, and 1.4%, respectively across all benchmarks, and to an extent

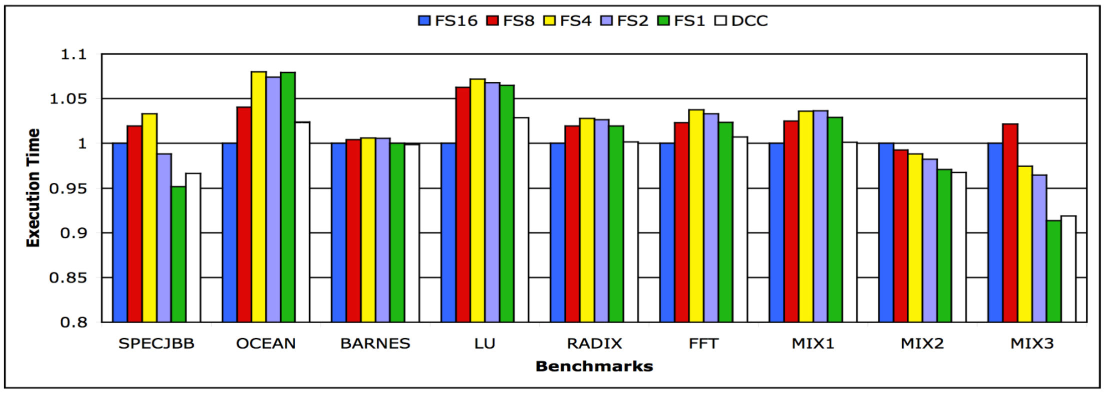


Figure 13: Execution time (Normalized to FS16).

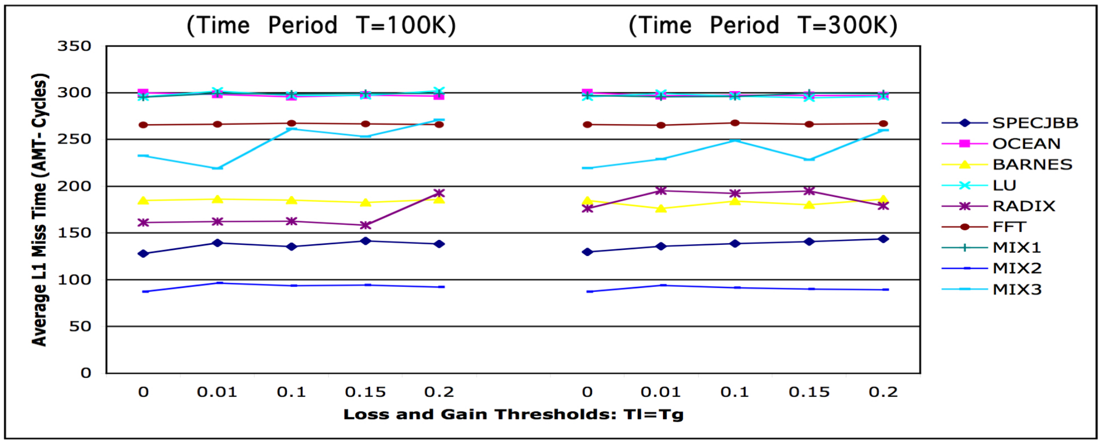


Figure 14: DCC sensitivity to different T, Tl, and Tg values.

of 10% for MIX3 over FS8. DCC is expected to way surpass all static strategies had it adaptively selected $\{T, T_l, T_g\}$ parameters. As such, DCC could provide more accurate estimations regarding expansions and contractions. Having established the effectiveness of DCC as a scheme that can synergistically adapt to irregularities exposed by different working sets and within a single working set, proposing an adaptive mechanism for selecting the $\{T, T_l, T_g\}$ thresholds is an obvious next step.

5.3 Sensitivity Study

The DCC algorithm utilizes the set of parameters $\{T, T_l, T_g\}$ to controllably tune up cache clusters and avoid potential noise that might hurt performance. As the current version of the algorithm assumes a fixed set of these parameters, we offer a study of DCC sensitivity to different $\{T, T_l, T_g\}$ values. Ten sets have been simulated, five with $T = 10,000$ (T1), and another five with $T = 300,000$ (T2) instructions. T_l and T_g were assigned values 0, 0.01, 0.1, 0.15, and 0.2 and ran with both, T1 and T2. Fig. 14 portrays the study outcome. A main conclusion is that no single fixed set of parameters provides superlative AMT for all the simulated benchmarks. For instance, SPECjbb performs best with T1 and $T_l = T_g = 0$. On the other hand, Barnes performs best with T2 and $T_l = T_g = 0.01$.

Overall, the DCC results with T1 are better than those with T2. Essentially, performance deteriorates when the partition period is too short or too long. Short partitions can hurt the accuracy of an estimation regarding a working set phase change. Long partitions, in contrary, can delay a detection of a phase change. The DCC algorithm doesn't expand or contract cache clusters upon every possible re-clustering point. It just checks the eligibility of an expansion or contraction step, and if found beneficial takes the action. Thus, DCC takes re-clustering actions only safely. Fig. 15 demonstrates a time varying graph that shows the activity of Barnes for 100 consecutive re-clustering points run under DCC with T2 and $T_l = T_g = 0.01$. A computation overhead of the DCC scheme at every re-clustering point is mainly that of computing the $\triangle$ metric, defined in equation (4). A performance overhead, on the other hand, can occur only if estimations about re-clustering actions fail. This is assumed, however, to be relatively little because of how the DCC algorithm inherently makes the architecture-adaptive decisions. This essentially explains why T1 yielded overall better DCC results than T2. The T1 moderate period of time attempts safely to capture a potential change in a program phase as soon as it emerges. We expect that with a time period smaller than T1, the information to feed to the DCC algorithm can be potentially skewed. As such, the estimations concerning program phases might possibly fail, and performance might, accordingly, degrade.

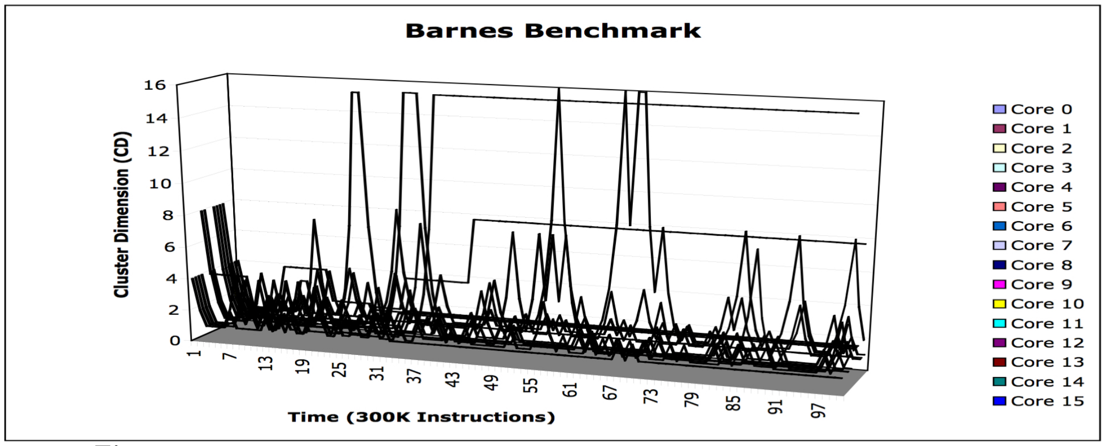


Figure 15: **Time varying graph showing the activity of the DCC algorithm.**

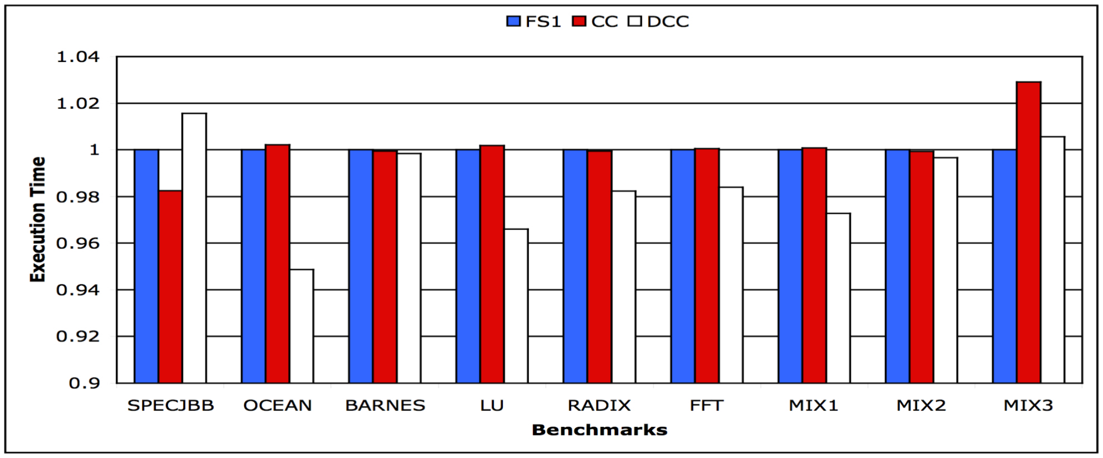


Figure 16: **Execution time of FS1, cooperative caching (CC), and DCC (normalized to FS1).**

5.4 Comparing With Cooperative Caching

This section presents a comparison between DCC and the related work, cooperative caching (CC) [3]. CC dynamically manages the aggregate on-chip L2 cache resources by combining the benefits of the private and shared schemes, AAL and MR, respectively. CC approaches the CMP cache management problem by basing its framework on the nominal private design and seeks to alleviate its implied capacity deficiency. If a block B is the only on-chip copy, CC refers to it as a *singlet*, otherwise as a *replicate* (because replications exist). To improve cache capacity, CC prefers to evict the following three classes of blocks in descending order: (1) an invalid block, (2) a replicate block, (3) and a singlet block. As such, CC refines cache capacity by reducing replicas as much as possible. Furthermore, CC employs spilling a singlet block from an L2 bank into another L2 bank for expected future usage. Fig. 16 demonstrates the execution time results of DCC and CC, both normalized to FS1. The shown CC is the default cooperative caching scheme that uses 100% cooperation probability (allows always the collection of the CC mechanisms to be used to optimize capacity). DCC always performs competitively, if not better, than the best static alternative. Thus DCC performs sometimes equivalently to FS1 and sometimes surpasses it (in case it is not the best caching option). On the other hand, across all the simulated benchmarks, CC outperforms only SPECjbb by 1.7%. Surprisingly, CC degrades FS1 performance by 0.16%, on average. The reason is that CC uses the minimum replication level for each benchmark thus heavily affecting the average L2 access latency (AAL). Replication typically mitigates AAL if done controllably [1, 5].

6. RELATED WORK

As CMP has become the mainstream architecture of choice, many proposals in the literature advocated managing the last level of caches using hardware and software techniques. Data migration and replication have been suggested as techniques to manage CMP caches via tuning either the average L2 access latency (AAL) or the L2 miss rate (MR) metrics. Migration has the advantage of maintaining the uniqueness of cache blocks on-chip offering, thereby, better L2 miss rate. In contrary, replication generally results in reduced average L2 access latency. Many of the proposals base their work either on the shared or the private design with an aim to mitigate the implied deficiency. Zhang and Asanović [25] proposed *victim replication* based on the shared paradigm, and seeks to mitigate AAL via keeping *replicas* of local primary cache victims within the local L2 cache banks. Chang and Sohi [3] proposed *cooperative caching* based on the private scheme, and seeks to create a *globally* managed shared aggregate on-chip cache. Chishti et al. [5] proposed *CMP-NuRAPID* based on the private design, and tries to control replication based on usage patterns. Beckmann and Wood [2] examined block migration in CMPs and suggested the *CMP-DNUCA* mechanism that allows data to migrate towards the requester processors to alleviate AAL. Beckmann et al. [1] proposed a hardware-based mechanism that dynamically monitors workload behaviors to control replication on the private cache organization. Huh et al. [8] proposed a spectrum of degrees of sharing to manage NUCA L2 caches in CMPs. Nayfeh et al. [14] examined the impact static clustering can have in small-scale shared-memory multiprocessors. They assumed a spectrum of degrees of sharing amongst processors (referred to as clusters) and evaluated static clusters composed of 1, 2, 4, or 8 processors (connected together using a shared global bus) sharing L2 caches. As an outcome, they suggested that clustering can reduce the bus traffic.

As all of the above studies essentially use hardware techniques to manage caches in CMPs, some other works have recognized the need for software to approach the CMP cache management problem. Cho and Jin [6] proposed an OS-level page allocation algorithm for shared NUCA caches to mainly reduce AAL. Liu et al. [11] proposed an L2 cache organization called *Shared Processor-Based Split* L2 that depends upon a table-based mechanism maintained by the OS to split the cache capacity amongst processors. Finally we note that DCC is unique and general in the sense that it does not limit itself to any of the two traditional schemes, shared or private. Nonetheless, and as described in Section 4.2, it offers a novel fine-grained caching solution for the CMP cache management problem.

7. CONCLUDING REMARKS

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

As future work, the proposed DCC location strategy can be improved by maintaining a small history about a specific cluster expansions and contractions activity. For instance, with an activity chain of 16-8-4-4-8, we might predict that a requested block can't exist at a DHT corresponding to $CD = 1$ or 2 and has a high probability to exist at a DHT that corresponds to $CD = 4$ and $CD = 8$.

8. REFERENCES

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

[2] B. M. Beckmann and D. A. Wood. "Managing Wire Delay in Large Chip-Multiprocessor Caches," *MICRO*, pp. 319–330, Dec. 2004.

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

[4] A. Chishti, M. D. Powell, and T. N. Vijaykumar. "Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures," *MICRO*, Dec. 2003.

[5] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. "Optimizing Replication, Communication, and Capacity Allocation in CMPs," *ISCA*, pp. 357–368, June 2005.

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

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

[8] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. "A NUCA Substrate for Flexible CMP Cache Sharing," *ICS*, pp. 31–40, June 2005.

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

[10] C. Kim, D. Burger, and S. W. Keckler. "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," *ASPLOS*, pp. 211–222, Oct. 2002.

[11] C. Liu, A. Sivasubramaniam, and M.Kandemir. "Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs," *HPCA*, pp. 176–185, Feb 2004.

[12] H. E. Mizrahi, J. L. Baer, E. D. Lazowska, and J. Zahorjan "Introducing memory into the switch elements of multiprocessor interconnection networks," *ISCA*, pp. 158–166, 1989.

[13] R. Mullins, A. West, and S. Moore "Low-Latency Virtual-Channel Routers for On-chip Networks," *ISCA*, pp. 188–197, June 2004.

[14] B. A. Nayfeh, K. Olukotun, and J. P. Singh. "The Impact of Shared-Cache Clustering In Small-Scale Shared-Memory Multiprocessors," *HPCA*, 1996.

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

[16] M. K. Qureshi and Y. N. Patt "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," *ISCA*, pp. 423–432, 2006.

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

[18] B. Stolt, Y. Mittlefehldt, S. Dubey, G. Mittal, M. Lee, J. Friedrich, and E. Fluhr. "Design and Implementation of the POWER6 Microprocessor," *Solid State Circuits. IEEE Journal.*, pp. 21–28, Jan. 2008.

[19] E. Speight, H. Shafi, L. Zhang, and R. Rajamony. "Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors," *ISCA*, June 2005.

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

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

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

[23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations," *ISCA*, pp. 24–36, July 1995.

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

[25] M. Zhang and K. Asanović. "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," *ISCA*, pp. 336–345, June 2005.