ACM: An Efficient Approach for Managing Shared Caches in Chip Multiprocessors

Mohammad Hammoud, Sangyeun Cho, and Rami Melhem

Department of Computer Science
University of Pittsburgh
{mhh,cho,melhem}@cs.pitt.edu

Abstract. This paper proposes and studies a hardware-based adaptive controlled migration strategy for managing distributed L2 caches in chip multiprocessors. Building on an area-efficient shared cache design, the proposed scheme dynamically migrates cache blocks to cache banks that best minimize the average L2 access latency. Cache blocks are continuously monitored and the locations of the optimal corresponding cache banks are predicted to effectively alleviate the impact of non-uniform cache access latency. By adopting migration alone without replication, the exclusiveness of cache blocks is maintained, thus further optimizing the cache miss rate. Simulation results using a full system simulator demonstrate that the proposed controlled migration scheme outperforms the shared caching strategy and compares favorably with previously proposed replication schemes.

1 Introduction

Advances in process technology have enabled integrating billions of transistors within a single chip. These advances combined with an incessant need to improve computer performance paved the road to the emergence of chip multiprocessors (CMPs). Chips capable of small-to-medium scale multiprocessing are commercially available [15, 21] and platforms having more cores are forthcoming [13].

As the realm of CMP is continuously expanding, it must provide high and scalable performance. The constantly widening processor-memory speed gap will substantially increase the capacity pressure on the on-chip memory hierarchy. The lowest-level on-chip cache not only needs to utilize its limited capacity effectively, but also has to mitigate the increased latencies due to wire delays [8]. Accordingly, a key challenge to obtaining high performance from CMP architectures is to manage the last level cache so that the access latency is reduced effectively and the capacity utilized efficiently.

CMP caches are typically partitioned into multiple banks for reasons of growing wire resistivity, power consumption, thermal cooling, and reliability considerations. Non-Uniform Cache Architecture (NUCA) has been employed to organize the resultant multiple cache banks [4, 10, 14, 24, 25]. One common practice for NUCA is the shared scheme where cache banks are all aggregated to form a logically shared cache [4, 15, 21, 24, 25]. Each memory block is mapped to a unique cache set in a unique cache bank offering thereby high cache capacity utilization. Unfortunately, shared caches have a latency problem. A cache block may reside at a bank far away

A. Seznec et al. (Eds.): HiPEAC 2009, LNCS 5409, pp. 355–372, 2009.

from the requester core. It is a challenge to maintain the advantage (cache capacity) of the shared cache CMP design and preclude the disadvantage (access latency).

Block replication and migration have been suggested as techniques for shared CMP caches to tackle the latency problem by frequently copying or moving accessed blocks to cache banks closer to the requesting processors [3, 6, 7, 8, 15, 16, 17, 21, 25]. Replication in general results in reduced cache hit latencies but may degrade cache hit rate. In fact, blind replication can be detrimental since the capacity occupied by replicas could increase significantly resulting in performance degradation [3]. Migration, on the other hand, maintains the exclusiveness of cache blocks on chip and preserves the high utilization of the caching capacity. Furthermore, it maintains the simplicity of the underlying cache coherence protocol. However, migration has been shown to be less effective for CMP caches than for uniprocessors [4, 16]. The issue in the CMP domain is that migration in multiple directions can cause migration conflicts, with shared blocks ping-ponging between processors [14]. Besides, locating migratory blocks in bank sets may turn out to be very expensive to an extent that it offsets the benefits offered by the migration technique.

We demonstrate through an example the difficulty behind block migration and the inefficiency it may cause with shared L2 cache design in the CMP context. Figure 1(a) illustrates a 16-core tiled CMP. We assume a shared scheme where L2 cache slices are logically shared among all tiles. Upon an L2 miss, a line is fetched from the main memory and placed in a home tile determined by a subset of bits of the line's physical address. The figure shows a case where a block, B, has been originally requested by tile 3 and mapped to tile 6. Later tiles 0 and 8 request the same cache block B. Tile 3 incurs 6 network hops, computed as twice the Manhattan distance between the requester and the target tiles (dimension-ordered routing [20]), to reach the home tile and satisfy its request. Tiles 0 and 8 incur 8 hops each to satisfy their requests.

Figure 1(b) illustrates a naïve first touch migration policy that directly migrates B to the original requester. Employing that, tile 3 will save the 6 network hops when touching B for the second time, assuming that it checks its local L2 tags before accessing the home tile. Block B, however, has been pulled away from the other two sharers incurring additional 6 hops for each one to locate the block on its new host tile. Consequently, even though one tile made a gain, in total there is a loss of 6 network hops. Besides, the on-chip network traffic increases due to the three-way cache-to-cache communications

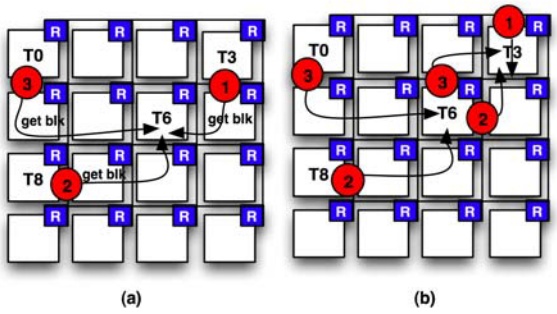


Fig. 1. (a) The Original Shared CMP Scheme. (b) A Simple Migration Example.

to satisfy sharers' requests. Specifically, when tile 0 (or 8) requests B, it has to check with tile 6 first, which is B's home tile, before its request is redirected to tile 3.

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

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

The contributions of the ACM mechanism are as follows:

- It demonstrates that migration, if done in an adaptive controlled fashion, yields an average L2 access latency that is on average 20.4% better than the nominal non-uniform shared L2 cache scheme, and 20.8% better than a conventional replication strategy for the simulated benchmarks.
- The proposed mechanism demonstrates the effectiveness of migration in the CMP domain and opens new research opportunities and directions for computer architects.
- The proposed mechanism avoids replication of cache blocks and reduces the overall L2 cache access latency without degrading the cache miss rate.

The rest of the paper is organized as follows. Section 2 delves more onto the idea of ACM and explains it in detail. Section 3 presents the ACM microarchitecture and hardware cost. Section 4 discusses our evaluation methodology and presents a quantitative assessment of our proposed mechanism. Other related works are recapitulated in Section 5, and Section 6 concludes.

2 Adaptive Controlled Migration

2.1 Baseline Architecture

This study assumes a tiled CMP model similar to the one depicted in Figure 1 and appeared many times in literature [9, 24, 25]. Tiled CMPs scale well to larger processor counts and can easily support families of products with varying numbers of tiles, including the option of connecting multiple separately tested and speed-binned dies within a single package [24]. The CMP model is organized as a 2D array of replicated tiles each with a core, a private L1 cache, an L2 bank, a directory for L1 coherence

maintenance, and a switch that connects the tile to the on-chip network. The L2 banks form a logically shared L2 cache. For the L1 level coherence enforcement, a distributed directory-protocol is modeled. The directory is distributed among all the tiles by mapping each memory block to a *home tile* [16, 18, 25]. On an L2 miss, a block is fetched from the main memory and mapped to its home tile determined by a subset of bits of the physical address called the home select (HS) bits. Consecutively the fetched block is copied to the L1 cache of the requester core. If any other core requests the same cache block at a later time, the home tile is accessed and the cache block is copied to its L1 cache. Cores that maintain copies of a certain cache block in their L1 caches are all denoted as *sharers* of this block.

A cache block may map to a tile that is close to or far away from the requester core. Therefore, the model introduces a NUCA design where accesses to any L2 bank varies depending on the network congestion and number of network hops between the requester and the target tiles. We assume a mesh network topology with dimension-ordered (XY) routing [20] where packets are first routed in the X and then the Y direction. Therefore, the number of network hops can be computed as twice the Manhattan distance between the requester and the target tiles.

2.2 Predicting Optimal Host Location

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

We propose a simple location algorithm that attempts to locate the optimal host of a cache block at runtime and designates it as its new *host tile*. It computes the *total latency cost* for a given cache block on each of the potential hosts and chooses the *minimum*. In order to achieve this, the algorithm keeps some runtime information, particularly, a pattern for the accessibility of the cache block. The pattern is essentially a bit vector to indicate whether the block has been accessed by a specific core or not. It can be built at run time with different *migration frequency levels*. The migration frequency level is the number of times a block is accessed before attempting to migrate it. Whenever a core accesses a block, its corresponding bit is set in the associated bit vector and a *use counter* associated with the block is incremented. This continuously shapes up an accessibility pattern for the given block and provides the aspired runtime information. When the use counter reaches the specified migration frequency level, the location algorithm interferes and selects a new host for the block that minimizes the total L2 access latency for all the sharers identified by the accessibility pattern. The pattern and the use counter are both cleared when the block is migrated so as to initiate a new pattern construction. This is a simple prediction scheme that depends on the past to predict the future. A core that accessed a block in the past is likely to access it again in the future. Because our algorithm makes its decision based on this pattern, we call the located host a *predicted optimal host*.

To exemplify how the ACM mechanism works, Figure 2 portrays 4 different cases for potential hosts that a cache block may migrate to. S stands for a sharer and H for

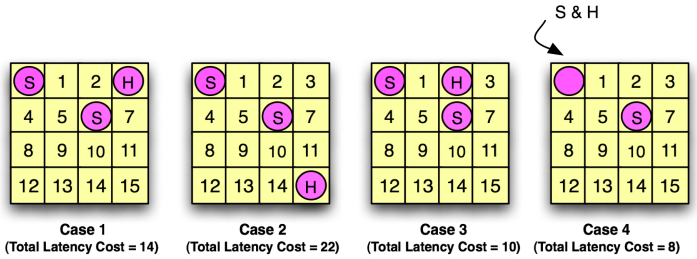


Fig. 2. An Example of How ACM Works (S = Sharer, H = Host)

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

2.3 Locating Migratory Blocks and Cache the Cache-Tags Policy

After locating the new predicted optimal host for a cache block, migration is performed. However, a question that follows is: how can a sharer later locate a cache block that is no more at its original home tile but migrated to a different tile? Zhang and Asanović [24] proposed an extra array of tags per tile to keep track of the locations of migratory blocks. Specifically, when a block, B, is migrated for the first time out of its home tile, an entry for B, serving as a short *fingerprint*, is allocated in this extra tag array at the home tile to point to the new location of B. Later if a sharer S reaches the home tile of B and fails to find a matching tag in the regular L2 tag array but hits in the extra tag array, the current host of B, pointed out by the matched fingerprint, satisfies the request. Specifically, data is forwarded to S from the current host of B using three-way cache to cache communication. If S fails to find a matching tag in both the regular and the extra tag arrays, an L2 miss is reported and data is fetched from the main memory. If S hits in the regular L2 tag array, then the L2 CMP shared protocol is simply followed. Figure 3(a) illustrates how a sharer can locate a cache block, B, that has already been migrated. This migration policy saves latency only for the tile to where B has been migrated. For other sharers, it fails to exploit *distance locality*. That is, the request may incur significant latency to reach the home tile even though the data is located in close proximity. Furthermore, it causes extra on-chip network traffic.

ACM uses a similar data structure found in [24] but with a simple, yet essential extension. We name this extended data structure the Migration Table (MT table). The idea is to *cache the cache-tags* in the MT table. Specifically, the MT table of a tile T can now hold two types of tags: (1) a tag for each block B whose home tile is T and had been migrated to another tile, and (2) tags to keep track of the locations of the migratory blocks that have been recently accessed by T but whose home tile is not T. We refer to the first type as *local MT tags* and to the second as *remote MT tags*.

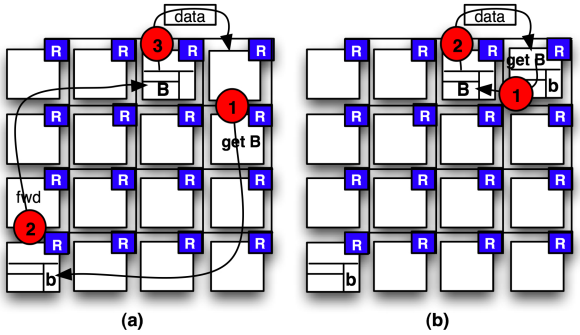


Fig. 3. Locating a Migratory Block B (a) Without *Cache the Cache-Tags* Policy. (b) With *Cache the Cache-Tags* Policy.

Whenever a sharer issues a request to a block, its local MT table is checked first for a matching tag. On a miss, the home tile is reached. If the block at the home tile has already been migrated, it is located in its new host and its tag is further cached in the requester's MT table, so as to reduce latency for subsequent accesses. If the block on the home tile was not migrated yet, the usual L2 shared protocol is followed. If the requester core hits in its local MT table, the corresponding block is directly retrieved from the location that the local tag designates; that is, the tile currently hosting the block. This is done without any need to visit the home tile. Accordingly, the three-way cache to cache communication problem exposed by the proposal in [24] is solved by the cache the cache-tags policy upon hits in the MT table. Figure 3 (b) shows how the cache the cache-tags idea solves the 3-way communication problem. The sharer finds a matching tag in its local MT table and quickly locates the data block on its host. Consequently, the profit gained by the migration technique is maintained, and the on-chip network traffic is improved.

The remote and local MT tags need to be kept consistent. That is, each time the block migrates, the corresponding remote and local MT tags in the MT tables need to be updated to indicate the block's new located host. We accomplish this by embedding a directory state in each of the MT tags. This directory state acts as a bit mask to indicate which tile has cached a remote MT tag for the associated cache block. It is simply a bit vector with a single bit corresponding to each core. Whenever a core caches a tag, its corresponding bit is set in the bit vector augmented with the local MT tag at the home tile. Effectively, each MT tag, or *MT entry* hereafter, is composed of four components: (1) The tag of the migratory cache block, (2) a bit vector that acts as a directory to maintain consistency of multiple copies of MT entries cached in different MT tables, (3) a bit to specify whether the entry is remote or local, and (4) an ID that points to the tile that is currently hosting the cache block. Hence, whenever a cache block migrates, its corresponding MT entry at the home tile is accessed and the ID is updated to point to the new host location. Furthermore, the cached remote MT entries, identified by the augmented bit vector, are all updated to indicate the new block's host. Note that the home tile that is to be reached to update the local MT entry is known via checking

the HS bits of the given physical address of the L2 request that triggered the migration of the L2 block. Accordingly, we need not store any backward pointer to locate the home tile.

As each associative set of blocks in the MT table contains a combination of local and remote MT tags, it is wise to never evict a local MT tag in favor of a remote one. This is because the local MT tag may have many active sharers. It may further have multiple associated remote MT tags stored at sharers' MT tables. Consequently, replacing it calls for eviction of the data block itself and all its associated remote tags. To avoid potential performance degradation, the MT table replacement policy replaces the following two classes of tags in descending order: (1) an invalid tag, (2) the least recently used remote MT tag. For now, we assume that the size and the associativity of the MT table are identical to those of the regular L2 tag array. In Section 4 we study the sensitivity of the ACM mechanism to different MT table sizes.

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

After the location algorithm designates a new host for a block B and the migration is to be performed, a decision must be made about which block to replace in the new host T located for B. If there is no invalid block in the target set at T, a naïve approach would replace the LRU block, say, D. However, because cache accesses might not be well distributed over the cache sets, there could be a capacity pressure at T, and D could be requested again. Hence, we try not to discard block D but to swap it with B so as to maintain the copy on chip. B and D could be migratory or non-migratory blocks. Migratory blocks are those that already migrated out of their home tiles while non-migratory ones are those that have not migrated yet. If B is non-migratory, a local MT entry should be allocated at its home MT table. If no entry is found to be replaced at the MT table, as planned by the MT replacement policy, migration is not performed. If B could be migrated and D is a migratory block, then they are simply swapped and D's associated MT entries are all updated to expose the change of the host location. If D is non-migratory, its local MT table is checked for a valid entry to replace. If a valid entry is found, a local MT entry is allocated and B and D are swapped. If no valid entry is found then D is simply discarded and B migrates to the new located host. Of course, if B also is a migratory block then when migrated, all its associated MT entries are updated to denote its new host.

Such a swapping policy is, in fact, very effective and robust that it makes our scheme applicable even to workloads that don't share cache blocks. We illustrate this effectiveness and robustness via an example. Figure 4 depicts a case for a single thread that exhibits *no sharing* at all and runs on tile 3 (the microarchitecture of the portrayed tiles is discussed in Section 3). We assume that the thread's working set is too large to fit entirely in the L2 cache bank of tile 3. To reduce L2 access latency, and with migration frequency level of 1, our location algorithm will choose to migrate all requested blocks to the L2 bank of tile 3 after accessing each for the second time. Figure 4(a) depicts the placement of block B after it has been requested by tile 3 and mapped to the home tile 5 (HS of B = 0101). Figure 4(b) demonstrates the migration of B to the L2 bank of tile 3 after tile 3 accessed B for the second time. Note that a local MT tag, b, is allocated in

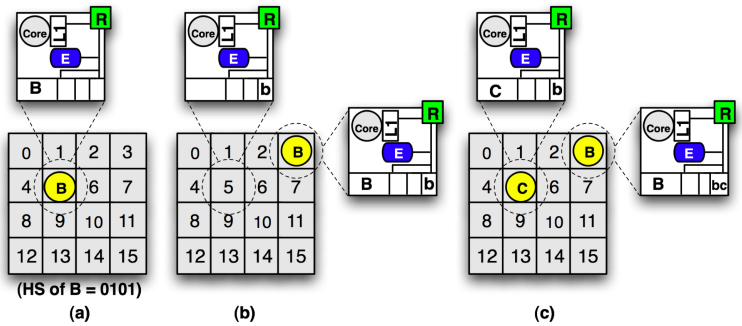


Fig. 4. An Automatic Data Attraction Case offered by ACM

the MT table at the home tile 5 and a remote one allocated at tile 3, as planned by the cache the cache-tags policy.

The case shows the capability of the ACM mechanism to automatically attract data to local tiles, thus allowing cores to access blocks very fast for subsequent requests. However, ACM is robust enough that it doesn't allow such *automatic data attraction* to continue freely and blindly, potentially causing increase in L2 miss rate. The *swapping with the LRU* policy suggests that when no invalid block at the target set of the located host T is found (capacity pressure), an attracted block B is swapped with the LRU block at T, thus avoiding increase in L2 miss rate. Figure 4(c) assumes that when B has been migrated to tile 3 it produces capacity pressure. The LRU block, C, at tile 3 is accordingly swapped with B as planned by the swapping with LRU policy, thus maintaining C on chip and accordingly avoiding any increase in the L2 miss rate. Note that a local MT tag c is allocated in the MT table at tile 3 as planned by the cache the cache-tags policy.

The case in fact demonstrates some resemblance to the victim replication strategy [25]. Victim replication also automatically attracts data to local tiles upon L1 evictions in order for subsequent accesses to save latency. However, it doesn't provide control on the capacity usage and can greatly reduce the available caching space. Section 4 presents a comparison between ACM and the victim replication scheme.

Finally, different cache blocks may experience entirely different degrees of sharing over time and demonstrate diverse access patterns. The ACM algorithm collects those non-uniform access patterns and based on them, finds a suitable location for data blocks that best minimize the L2 access latency. Thus ACM inherently doesn't prefer any specific on-chip tile over the other. However, if at any course of execution, a cache bank receives a capacity pressure more than other banks (similar to the above case), ACM robustly relaxes the pressure via the swapping with the LRU policy.

2.5 Discussion: Cache Blocks in Transit

The ACM mechanism can easily preclude any potential for *false misses* which occur when an L2 request falsely fails to hit a cache block because it is in transit from one bank to another. When migration is to be performed, a copy B' of the cache block B is kept at the current bank so as if an L2 request arrives while B is in transit, the request

is immediately satisfied without incurring any delay. When B reaches the new host, an acknowledgment message is sent back to the old host to discard B'. The old host keeps track of any tile that accesses B', and when receiving the acknowledgment message, sends an update message to the new host to indicate the new sharers that requested B while it was in transit. The directory state entry of B is consecutively updated.

3 Microarchitecture and Hardware Cost

The ACM mechanism is a completely hardware-oriented scheme. Figure 5 depicts its microarchitecture design within a CMP tile. The added structures are shaded. The ACM engine incorporates the hardware implementation of the ACM algorithm. It further updates the use counters and pattern vectors augmented to cache blocks at the time they are accessed. Note that only one adder per tile is needed to increment the use counters. If the number of accesses reaches the specified migration frequency level maintained by the ACM engine, the pattern vector of the block is read and the optimal host is computed. The ACM engine logic is simple as required by the ACM mechanism. The area and power budget are expected to be modest.

The performance improvement of the ACM mechanism comes at small storage overhead to the on-chip cache hierarchy. We assume 41 bit physical address and 64 byte cache block size. Nine bits of the physical address are used by the nominal shared scheme to index the L2 cache bank within a tile beside six bits for the offset. Four more bits are used for the HS to select the home tile. The tag width is accordingly 22 bits. Each MT table stores tags using a format similar to the tags in the corresponding L2 cache bank. However, for each entry in the MT table, additional 16 bits are required for the directory vector, 1 bit to specify whether the tag is remote or local, and 4 bits to indicate the ID of the tile currently hosting the block. Additionally, augmented to each cache block, are 4 bits to store the use counter (assuming that the maximum allowed migration frequency is 16) and 16 bits for the pattern vector. The table in Figure 6 breaks down ACM's storage requirement for our cache configuration. The MT table has been reduced to quarter its size with a tradeoff of giving up some of the performance gain produced by the ACM mechanism as demonstrated in Section 4. Moreover, the peripheral circuitry and interconnects are not taken into consideration when measuring the percentage increase of the on-chip cache capacity thus the result shown in the table in Figure 6 is in reality much smaller.

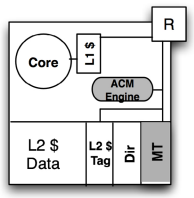


Fig. 5. Michroarchitecture for the ACM mechanism (Figure not to scale)

Component	Bits Per Entry	K Entries	K Bytes Per Tile
TAG	22	2	5.5
Directory Vector	16	2	4
Remote/Local	1	2	0.2
ID	4	2	1
Pattern Vector	16	8	16
Use Counter	4	8	4
Total KBytes			30.7
% Increase of On-Chip Cache Capacity			4.8%

Fig. 6. ACM Storage Overhead

4 Quantitative Evaluation

In this section, we evaluate the ACM mechanism and alternative cache designs. We compare ACM with the non-uniform shared cache scheme (S), as a baseline, and the victim replication (VR) proposal [25] that we consider a traditional design on this direction. The rationale behind such a comparison is to demonstrate that migration, if done in an adaptive-controlled fashion, generates favorable results compared to replication. Replication has been considered useful for tackling the NUCA problem [3, 8]. Migration, on the other hand, hasn't been proved to be highly effective in the context of CMPs [4, 24]. By showing that ACM outperforms the nominal shared cache scheme and one of the conventional replication designs, we thereby demonstrate that migration is in fact an effective technique for managing L2 caches in CMPs. Though there has been more recent work than VR on replication [3, 6, 8], our objective is not to compare against all these, but rather to expose the potential of migration and to elide the fallacy regarding such a technique in the CMP domain.

4.1 Experimental Methodology

Our evaluation employs a detailed full system simulator built on Simics 3.0.29 [2]. We simulate a 16-way tiled CMP architecture organized as a 4×4 2D mesh grid and runs under the Solaris 10 OS. Programs are compiled for an UltraSPARC-III Cu processor. Each core runs at 1.4 GHz, uses in-order issue, and has a 16KB I/D L1 cache and a 512KB L2 cache with the LRU replacement policy. The aggregate L2 cache is consequently 8MB for the 16-tiled CMP model. Each L1 cache is 4-way set associative with 1-cycle access time and 64 byte line. Each L2 cache is 16-way set associative with 6-cycle access time and 64 byte line. A 5-cycle latency per hop, based on a recent processor from Intel [22], is incurred when a datum traverses through the mesh network including both, a 3-cycle switch [9, 24, 25] and a 2-cycle link latencies. The 4-GB off-chip main memory latency is set to 300 cycles.

ACM, S, and VR are studied using a mixture of single-threaded, multithreaded, and multiprogramming workloads. For multithreaded workloads, we use the commercial benchmark SPECjbb, and four other shared memory benchmarks from the SPLASH2

Table 1. Benchmark programs

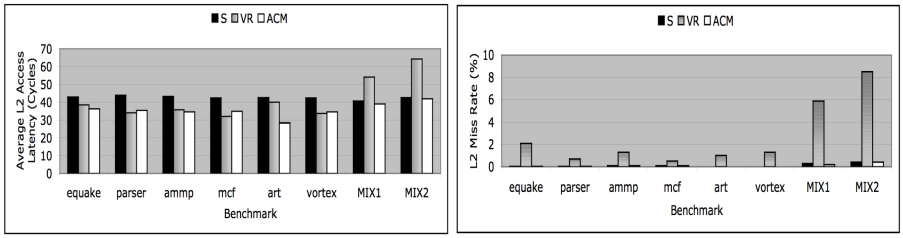
NAME	INPUT
SPECjbb	Java HotSpot (TM) server VM v 1.5, 4 warehouses
lu	1024×1024 matrix (16 threads)
ocean	514×514 grid (16 threads)
radix	2M integers (16 threads)
barnes	16K particles (16 threads)
parser	reference
art	reference
equake	reference
mcf	reference
ammp	reference
vortex	reference
MIX1	reference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

suite [23] (Ocean, Barnes, LU, Radix). For single-threaded workloads we use six programs from SPEC2K [1], three integers (vortex, parser, mcf) and three floating-points (art, equake, ammp) with the reference data sets. These benchmarks were chosen because they demonstrate different access patterns and different working set sizes [1, 10]. Two multiprogramming workloads, MIX1 (vortex, ammp, mcf, equake) and MIX2 (art, equake, parser, mcf), are constructed from the selected 6 SPEC2K programs. Initialization phases of applications are skipped using magic breakpoints from Simics. For the single-threaded and multiprogramming workloads, a detailed simulation is run for each benchmark until at least one core completes 1 billion instructions. Table 1 summarizes all the simulated benchmarks. Last but not least, we fix the migration frequency level to 10 throughout the simulation.

4.2 Simulation Results

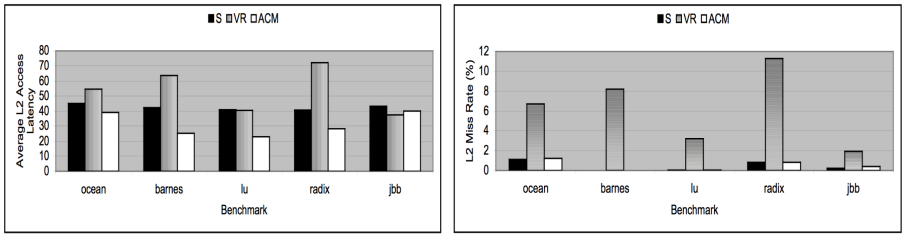
In this subsection we report our simulation results and analyze the effectiveness of ACM over S and VR schemes. The aim is to study the efficiency of the ACM mechanism in the presence of *no-sharing*, *little sharing*, and *sharing* of cache blocks. These cases are essentially offered by the single-threaded, multiprogramming, and multithreading workloads, respectively. The main target of the ACM mechanism is to tackle the non-uniformity in access latency that the shared scheme exposes (NUCA problem). Accordingly, the primary evaluation metric that we adopt is the average L2 access latency (AAL) experienced by an access to the L2 cache from any core on the tiled CMP. Upon an L1 miss, an access to L2 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. Three scenarios may occur thereupon. First, the request may hit in its local L2 bank, and we assume that this incurs only 6 cycles. Second, It may miss locally but hit remotely, thus incurring an access latency that varies depending on the network congestion and the number of network hops. Third, it may miss on chip and consequently reach the main memory to satisfy the request. Thus AAL combines both, the access to L2 (locally or remotely) and the L2 miss rate and penalty. Clearly, an improvement in the AAL metric translates to an improvement in the overall system performance. The average memory access cycles spent in L1, L2, and main memory serving 1K instructions is furthermore shown to give an overall performance picture. The L2 miss rate is also demonstrated to assure the claim that it is maintained by the ACM mechanism and because of its importance to the VR strategy. We also report the message-hops per 1k instructions.

Comparing Schemes, Single-threaded and Multiprogramming Workloads: Multiprogramming workloads tend to have very little sharing among the different threads [6, 24]. Single-threaded benchmarks represent the *no-sharing* case. VR is very appealing in this situation because it can automatically attract data blocks to the only tile running the thread, thus supposedly reducing access latency by decreasing inter-tile accesses from replica hits. However, this may make the tile running the thread experience some high capacity demand. This may result in poor utilization of the on-chip cache capacity. If the scheme fails to offset the increased miss rate then this could lead to performance degradation. This intuition is confirmed by the results shown in Figure 7. The L2 miss rates of all the single-threaded benchmarks shown for VR are all much larger



(a) (b)

Fig. 7. Single-threaded and Multiprogramming Results (S = Shared, VR = Victim Replication)



(a) (b)

Fig. 8. Multithreaded Results (S = Shared, VR = Victim Replication)

than that for S. However, across all the workloads VR successfully offsets the miss rate from fast replica hits. Contrary to that, VR fails to offset the increase in L2 miss rate for the multiprogramming workloads. Clearly, the SPEC2k applications have small working sets that more or less fit in L1 and L2 caches as they expose negligible L2 miss rates as is shown in the figure for the S scheme. The L2 miss rate for the 6 single-threaded benchmarks is on average 0.04%. As the memory footprint of the benchmark decreases, the space made available to replicas increases and accordingly more performance improvement can be achieved. For the multiprogramming workloads, 4 benchmarks are now sharing the L2 cache space. Hence, the memory footprint of the workload has been increased. The L2 miss rate for MIX1 and MIX2 is now 0.3% on average, or 8.7x more than that of the single-threaded workloads. VR failed to offset this increase and produced 41.8% and 46.0% AAL degradation over S and ACM schemes respectively.

Contrary to that, ACM still offers this automatic data attraction functionality suggested by the VR scheme but in a very controlled fashion that it can efficiently customize allocation of on-chip capacity via the swapping with LRU policy as is discussed in Section 2. Consequently, it successfully generated AALs that are on average 20.5% and 3.7% better than S and VR respectively for the single-threaded workloads, and 2.8% and 31.3% better than S and VR respectively for the multiprogramming ones. VR performs better than ACM only for the benchmarks vortex, parser, and mcf. It has been observed that 81%, 50%, and 57% of the cache blocks of the vortex, parser, and mcf benchmarks respectively are accessed for less than 10 times (the specified migration frequency level). As a result, for all these cache blocks, the ACM mechanism didn't

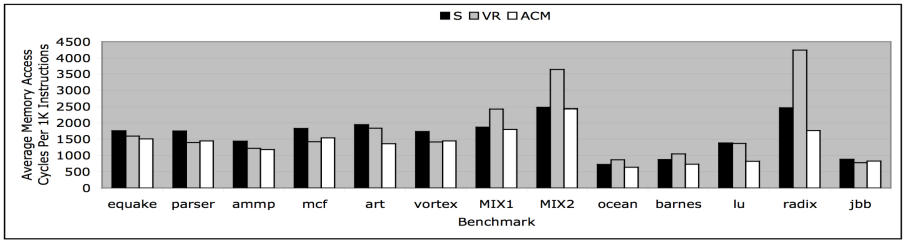


Fig. 9. Average Memory Access Cycles Per 1K Instructions Results (S = Shared, VR = Victim Replication)

even attempt to migrate them to better hosts so as to minimize the L2 access latency. This is because we fixed the migration frequency level throughout simulations. Migration to any cache block is triggered upon being accessed for the number of times that the migration frequency level specifies. For that reason, ACM is not exhibiting its full ability to exploit the optimum performance though it still on average greatly surpasses both of the schemes, S and VR. With an adaptive tunable migration frequency level, the ACM mechanism would hit its optimum and consequently provide larger performance improvements. This is to be explored in future research work.

As clearly shown in Figure 7, ACM maintains the L2 miss rates of S for all the simulated single-threaded and multiprogramming benchmarks. Moreover, it optimizes some of them because of the swapping with the LRU policy and generates on average 2x and 1.5x reductions in L2 miss rates over S for the single-threaded and the multiprogramming benchmarks respectively. Finally, Figure 9 shows the average memory access cycles per 1K instructions experienced by all the simulated benchmarks. VR performs on average 15.1% better than S and 38.4% worse than S for the single-threaded and the multiprogramming benchmarks respectively. ACM, on the other hand, performs on average 18.6% and 2.6% better than S, and 3.4% and 29.4% better than VR for the single-threaded and the multiprogramming benchmarks respectively.

Comparing Schemes, Multithreaded Workloads: The multithreading workloads expose different degrees of sharing among threads and accordingly allow us to study the efficiency of the ACM mechanism with such a case. Figure 8 depicts AALs and the L2 miss rates of the multithreading workloads compared to S and VR. ACM exhibits AALs that are on average 27.0% and 37.1% better than S and VR respectively. VR reveals 26.7% worse AAL than S for all the simulated benchmarks. This is due to the fact that VR has a static replication policy that depends on the blocks' sharing behaviors. An increase in the degree of sharing suggests that the capacity occupied by replicas could increase significantly leading to a decrease in the effective L2 cache size. As such, if replicas displace too much of the L2 cache capacity, the L2 miss rate could increase considerably, degrading thereby the average L2 access latency. This was clearly illuminated by the behaviors of the Ocean, Barnes, and Radix benchmarks where reduction in latencies via replica hits failed to offset the excessive latencies deduced by the increased miss rate. Barnes for instance utilizes a tree data structure that exhibits a sharing degree of 71% [4] and accordingly incurs a significant increase in capacity pressure when VR

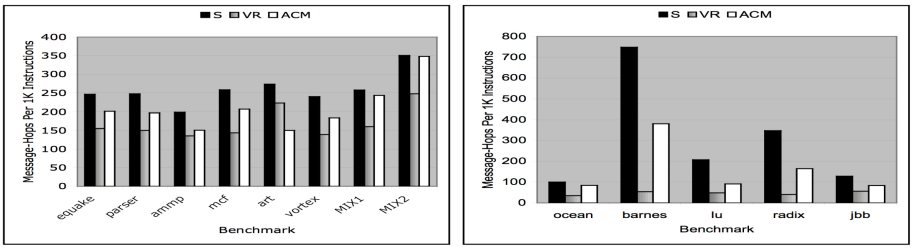


Fig. 10. On-Chip Network Traffic Comparison (a) Single-threaded Workloads (b) Multithreaded Workloads

is used. Note that such an inferred VR behavior is more elucidated in this work than in the original evaluation [25] as the L2 cache has been downsized to half. VR was successful in offsetting the impact of increased offchip accesses for the LU and JBB workloads.

On the other hand, ACM, a pure migration technique, maintains the exclusiveness of cache blocks on chip and consequently preserves the L2 miss rates of S for the three benchmarks Barnes, Lu, and Radix. Only Ocean and JBB reveal a small increase in the L2 miss rate for ACM over S. This is because when some block, B1, is to be migrated to a new host, H, no valid entry for the LRU block, B2, that is to be swapped with B1, is found in the MT table of H, and accordingly discarded as planned by the swapping with the LRU policy. That discarded block, B2, can be requested again by some other threads. VR only performs better than ACM for the JBB benchmark. The reason is the fixed migration frequency level that we assume throughout the simulation process. We ran JBB with doubling and tripling the migration frequency and respectively obtained 3.7% and 6.7% more AAL improvements over the base run with a migration frequency of 10. Lastly, Figure 9 shows the average memory access cycles per 1K instructions. For the multithreading benchmarks, VR performs on average 19.6% worse than S. ACM, in contrary, performs on average 20.7% and 29.7% better than S and VR respectively.

On-Chip Network Traffic: A supplementary advantage of the ACM mechanism is the reduced on-chip network traffic that it offers. Figure 10 depicts the number of message-hops per 1k instructions that the three schemes, S, VR, and ACM exhibit for the single-threaded, multiprogramming, and multithreaded workloads. The ACM scheme offers 25.3%, 3.0%, and 41.6% on-chip network traffic reduction over S for the single-threaded, multiprogramming, and multithreaded workloads respectively. The VR scheme, on the other hand, offers 35.6%, 33.6%, and 75.7% on-chip network traffic reduction over S for the three workloads respectively. Consequently, VR offers more on-chip network reduction over S than what ACM does because it decreases more inter-tile accesses from replica hits. Though the ACM mechanism bears some resemblance to the VR strategy for the single-threaded workloads as discussed in Section 2, but with a fixed migration frequency level of 10, a tile waits for 10 accesses to the block to attract it to its local L2 bank. Therefore, it incurs more inter-tile accesses compared to VR that tries to attract the block to its local L2 bank immediately after being evicted

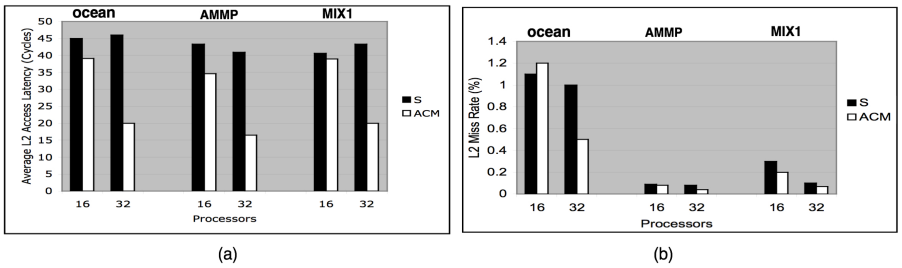


Fig. 11. Results for CMP Systems with 16 and 32 Processors (a) Average L2 Access Latencies (b) L2 Miss Rate

from L1. If VR fails to replicate cache blocks while ACM succeeded to attract blocks to its local L2 bank, ACM will surpass VR. The Art benchmark is the only case that exhibits such a situation. Finally, we studied the increase in network congestion for ACM over S without employing the cache the cache-tags policy. We found on average an increase of 17.6%, 4.1%, and 1% over S for the single-threaded, multiprogramming, and multithreaded workloads respectively. Clearly, this demonstrates the decisiveness and usefulness of such a policy when applying block migration in CMPs.

ACM Adaptability to Futuristic CMP Models: As the number of tiles on a CMP platform increases, the NUCA problem exacerbates. In order for any cache management scheme to be useful in tackling the NUCA problem, it should demonstrate adaptability to upcoming futuristic CMP models. ACM is very expedient in this direction as it always selects a host for a cache block that minimizes the total L2 access latency for all the sharers of that block independent of the underlying CMP size. Thus more exposure to the NUCA problem translates effectively to a larger benefit from the ACM scheme. We show such a pro of ACM via extending our CMP model to 32 tiles and running simulations for three selected benchmarks. Each tile still maintains, as with the 16-tiled CMP model, a 16KB I/D L1 cache and a 512KB L2 cache bank. The three benchmarks that have been chosen to conduct the study are, Ocean, ammp, and MIX1, from the multithreaded, single-threaded, and multiprogramming workloads respectively. These benchmarks revealed the highest L2 miss rates among the others in their sets; hence, selected.

Figure 11 depicts the AALs and the L2 miss rates of the selected benchmarks. For 16 tiles, ACM shows AAL improvements of 13.1%, 20.0%, and 1.7% for Ocean, ammp, and MIX1 over S respectively. However, for 32 tiles, ACM shows AAL improvements of 56.5%, 59.6%, and 53.8% over S respectively. As a result, ACM exhibits on average 11.6% and 56.6% AAL improvements over S for the 16-tiles and 32-tiles models respectively. The 32-tiles CMP model produced latency results that are 4.8x times better than those generated by the 16-tiles one for the simulated benchmarks.

Sensitivity and Stability Studies: So far we have assumed for simplicity that the size and associativity of the MT tables are identical to that of the L2 caches. There is nothing, in fact, that prevents this data structure from being of a smaller size or associativity. To study the sensitivity of the ACM mechanism to this component, we ran simulations

for the three benchmarks, Ocean, ammp, and MIX1 with MT table sizes reduced to *half* (50%) and *quarter* (25%) the size of the *base* cache, and with a 16-way set associativity. With half and quarter configurations, we got AAL increases of 5.9% and 11.3% respectively over the base one, but still improvements of 7.6% and 2.9% respectively over S. The highest contribution for the AAL increases was from the ammp benchmark. The ammp benchmark shows alone 19.9% AAL increase, averaged for both the half and quarter configurations, over the base, though still 6.1% better than S. It was observed that 60.7% of the cache blocks in ammp are accessed at least 10 times (the specified migration frequency level) before getting evicted from L2, consequently triggering migrations. To decrease the pressure on the MT table, we ran simulations with migration frequency of 20 rather than 10. Compared to the 19.9% AAL increase, we obtained only 12.3% increase, averaged for both the half and quarter configurations, over the base one and consequently 10.1% on average better than S.

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

5 Related Work

Migrating data to improve memory access latency has been extensively studied in the context of distributed shared-memory multiprocessors [5, 11, 12, 19]. Kim et al. [16] proposed D-NUCA as a mechanism that allows important data to migrate towards the processor within the same level of cache in the context of uniprocessor. Beckmann and Wood [4] examined block migration in CMPs and suggested CMP-DNUCA similar to the original D-NUCA proposal. They employ a gradual migration policy and move blocks along 6 bankcluster chain (the cache banks are physically separated into 16 different bankclusters). However, high degree of cache block sharing complicates the policy and blocks tend to congregate at the center of the cache away from all the processors.

A recent study [17], undertaken independently, proposed an efficient migration scheme to address the NUCA problem via modeling it in the L2 space as a two-dimensional post office placement problem. The scheme determines a suitable location for each data block at any given time during execution. Though the proposed design bears resemblance to ACM in that it dynamically finds a proper L2 location for each cache line after tracking its cache pattern, it doesn't discuss the mechanism with which migrating cache blocks is efficiently done.

Zhang and Asanović [24] proposed data migration in the context of tiled CMP. They allow replication and employ a migration strategy similar to the first touch policy presented in Section 1, but with certain conditions to move cache blocks to the requester tiles. Lastly, many proposals in the literature advocate data replication to manage the

non-uniformity in latency exposed by the nominal shared L2 cache design [3, 8, 25]. A recent study [3] demonstrated that blind replication is dangerous because the capacity occupied by replicas could increase significantly. The study proposed a hardware-based mechanism that dynamically monitors workload behavior to control replication.

6 Conclusions and Future Work

Managing L2 caches in chip multiprocessors is essential to fuel its performance growth. This paper studied a strategy to manage non-uniform shared caches in CMP by dynamically migrating cache blocks to optimal locations that provide the minimal L2 access latency. The proposed mechanism optimizes the L2 miss rate via maintaining the uniqueness of cache blocks on chip. Besides, Cache the Cache-tags policy has been proposed to effectively simplify the process of locating migratory blocks. Simulation results demonstrated the robustness, scalability, and stability of ACM. Unlike previously studied migration strategies in CMP literature, the proposed mechanism revealed and confirmed the usefulness of data migration in chip multiprocessors.

The strategy that we proposed to locate optimal hosts for cache blocks is simple and assumes that if a block has been accessed by a certain core in the past, then it is likely to be accessed by the same core in the future. Chishti et al. [8] observed that many blocks brought to the cache are not reused in some workloads. Thus, the proposed hardware host predictor can be easily improved by introducing more weights for the cores that accessed a cache block often.

Finally, we fixed throughout our simulation process the migration frequency level to 10. As is shown, this doesn't exhibit the full capability of the ACM mechanism. An adaptive tunable migration frequency level would allow the ACM mechanism to hit its optimum. Having established the effectiveness of the main idea of ACM, improving performance through an adaptive algorithm to dynamically tune the migration frequency level is the obvious next step.

References

1. Standard performance evaluation corporation, `http://www.specbench.org`
2. Virtutech, A.B.: Simics full system simulator, `http://www.simics.com/`
3. Beckmann, B.M., Marty, M.R., Wood, D.A.: Asr: Adaptive selective replication for cmp caches. In: MICRO (December 2006)
4. Beckmann, B.M., Wood, D.A.: Managing wire delay in large chip-multiprocessor caches. In: MICRO (December 2004)
5. Chandra, R., Devine, S., Verghese, B., Gupta, A., Rosenblum, M.: Scheduling and page migration for multiprocessor compute servers. In: ASPLOS (October 1994)
6. Chang, J., Sohi, G.S.: Cooperative caching for chip multiprocessors. In: ISCA (June 2006)
7. Chishti, A., Powell, M.D., Vijaykumar, T.N.: Distance associativity for high-performance energy-efficient non-uniform cache architectures. In: MICRO (December 2003)
8. Chishti, Z., Powell, M.D., Vijaykumar, T.N.: Optimizing replication, communication, and capacity allocation in cmps. In: ISCA (June 2005)
9. Cho, S., Jin, L.: Managing distributed shared l2 caches through os-level page allocation. In: MICRO (December 2006)

10. Dybdahl, H., Stenstrom, P.: An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors. In: HPCA (February 2007)
11. Falsafi, B., Wood, D.A.: Reactive numa: A design for unifying s-coma and cc-numa. In: ISCA (June 1997)
12. Hagersten, E., Landin, A., Haridi, S.: Ddm-a cache-only memory architecture. IEEE Computer (September 1992)
13. Held, J., Bautista, J., Koehl, S.: From a few cores to many: A tera-scale computing research overview. White Paper. Research at Intel. (January 2006)
14. Kim, C., Huh, J., Shafi, H., Zhang, L., Burger, D., Keckler, S.W.: A nuca substrate for flexible cmp cache sharing. In: ICS (June 2005)
15. Johnson, T., Nawathe, U.: An 8-core, 64-thread, 64-bit power efficient sparc soc. In: IEEE ISSCC (February 2007)
16. Kim, C., Burger, D., Keckler, S.W.: An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In: ASPLOS (October 2002)
17. Li, F., Kandemir, M., Irwin, M.J.: Implementation and evaluation of a migration-based nuca design for chip multiprocessors. In: ACM SIGMETRICS (June 2008)
18. Marty, M.R., Hill, M.D.: Virtual hierarchies to support server consolidation. In: ISCA (June 2007)
19. Mizrahi, H.E., Baer, J.L., Lazowska, E.D., Zahorjan, J.: Introducing memory into the switch elements of multiprocessor interconnection networks. In: ISCA (1989)
20. Mullins, R., West, A., Moore, S.: Low-latency virtual-channel routers for on-chip networks. In: ISCA (June 2004)
21. Sinharoy, B., Kalla, R.N., Tendler, J.M., Eickemeyer, R.J., Joyner, J.B.: Power5 system microarchitecture. IBM J. Res. & Dev. (July 2005)
22. Vangal, S., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Iyer, P., Singh, A., Jacob, T., Jain, S., Venkataraman, S., Hoskote, Y., Borkar, N.: An 80-tile 1.28tflops network-on-chip in 65nm cmos. In: ISSCC, New York (February 2007)
23. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The splash-2 programs: Characterization and methodological considerations. In: ISCA (July 1995)
24. Zhang, M., Asanović, K.: Victim migration: Dynamically adapting between private and shared cmp caches. Technical Report TR-2005-064, Computer Science and Artificial Intelligence Labratory. MIT (October 2005)
25. Zhang, M., Asanović, K.: Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In: ISCA, New York (2005)