Evaluating Placement Policies for Managing Capacity Sharing in CMP Architectures with Private Caches

AHMAD SAMIH and YAN SOLIHIN, North Carolina State University
ANIL KRISHNA, IBM Systems and Technology Group

Chip Multiprocessors (CMP) with distributed L2 caches suffer from a cache fragmentation problem; some caches may be overutilized while others may be underutilized. To avoid such fragmentation, researchers have proposed capacity sharing mechanisms where applications that need additional cache space can place their victim blocks in remote caches. However, we found that only allowing victim blocks to be placed on remote caches tends to cause a high number of remote cache hits relative to local cache hits.

In this article, we show that many of the remote cache hits can be converted into local cache hits if we allow newly fetched blocks to be selectively *placed* directly in a remote cache, rather than in the local cache. To demonstrate this, we use future trace information to estimate the near-upperbound performance that can be gained from combined placement and replacement decisions in capacity sharing. Motivated by encouraging experimental results, we design a simple, predictor-based, scheme called *Adaptive Placement Policy* (APP) that learns from past cache behavior to make a better decision on whether to place a newly fetched block in the local or remote cache. We found that across 50 multiprogrammed workload mixes running on a 4-core CMP, APP's capacity sharing mechanism increases aggregate performance by 29% on average. At the same time, APP outperforms the state-of-the-art capacity sharing mechanism that uses only replacement-based decisions by up to 18.2%, with a maximum degradation of only 0.5%, and an average improvement of 3%.

Categories and Subject Descriptors: B.3.2 [Memory Structures]: Design styles; C.4 [Performance of Systems]: Design studies

General Terms: Design, Performance, Measurement

Additional Key Words and Phrases: Memory systems, chip multiprocessor, private caches, capacity sharing, placement policies, stack distance profiling, limit studies, QoS

ACM Reference Format:

Samih, A., Solihin, Y., and Krishna, A. 2011. Evaluating placement policies for managing capacity sharing in CMP architectures with private caches. *ACM Trans. Architect. Code Optim.* 8, 3, Article 15 (October 2011), 23 pages.

DOI = 10.1145/2019608.2019614 <http://doi.acm.org/10.1145/2019608.2019614>

1. INTRODUCTION

Chip Multiprocessor (CMP) design presents interesting design choices in how to organize the on-chip caches. While typically L1 caches are private per core due to their tight timing requirement, whether the L2 (or L3) cache should be private to each core or shared by all cores is an open question. A physically shared L2 cache allows applications to more fluidly divide up the cache space, and maximizes the aggregate cache capacity because no block is replicated in the L2 cache. However, a large cache has a high access latency. On the other hand, private per-core L2 caches provide low

This research is supported in part by National Science Foundation (NSF) grant CCF-0347425.

Contact author's address: A. Samih, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695-7256; email: aasamih@ncsu.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2011 ACM 1544-3566/2011/10-ART15 \$10.00

DOI 10.1145/2019608.2019614 <http://doi.acm.org/10.1145/2019608.2019614>

latency accesses to the corresponding core, provide performance isolation between cores, and allow a more scalable multicore configuration. However, private L2 caches have the major disadvantage of *capacity fragmentation*, that is, when a diverse mix of sequential programs runs on different cores, some programs may overutilize the local L2 cache due to their large working set, while others may underutilize it due to their small working set.

Recognizing the capacity fragmentation problem in CMPs with private caches, prior work, such as Cooperative Caching [Chang and Sohi 2006] (CC) and Dynamic Spill Receive [Qureshi 2009] (DSR), has proposed schemes that allow cores to share their local caches with other cores. Such *capacity sharing* mechanisms are enabled by allowing a core to spill a block evicted from the local cache to a remote cache. CC allows any cache to spill into any other cache regardless of cache usage behavior of the applications running on the corresponding cores. While this provides fluid capacity sharing among cores, sometimes it produces an unwanted effect. An application, which cannot benefit from additional cache capacity, may pollute the cache of a core running an application that really needs all the cache space it has. To avoid this drawback, DSR identifies applications that can benefit from additional cache capacity (called *Acceptors*), and applications that have excess cache capacity that can be donated without impacting performance (called *Donors*). Further, it proposes a dynamic mechanism to identify when an application is allowed to spill, and when an application is allowed to receive a cache block.

Both CC and DSR have a common characteristic: remote caches are treated as the victim cache for the local cache. This article investigates if such a strategy is the best way for providing capacity sharing. In particular, we investigate a previously unexplored strategy that considers placing newly fetched blocks directly in remote caches, in addition to considering placing locally evicted blocks in remote caches. A remote placement strategy is motivated by our observation that the stack/reuse distance of applications that benefit from capacity sharing tends to be high, which causes a lot of remote cache hits compared to local cache hits. By placing incoming blocks in remote caches, we less disturb the local cache population, and many remote hits can be converted into potential local hits. To test the potential performance improvement from remote placement, we developed a hypothetical oracular scheme that, based on a future-access trace, determines whether a newly fetched block should be placed in the local cache or remote cache. In addition, in case it decides to place a newly fetched block in the local cache, it then consults the future-trace again to identify the ideal replacement candidate in the local cache. We call this hypothetical capacity sharing technique Oracular Placement and Replacement Policy (OPRP). Figure 1 shows the speedup of the state-of-the-art DSR [Qureshi 2009] and the oracle scheme (OPRP) compared to the base case where the CMP has private caches without capacity sharing. $A_x D_y$ represents workloads that have x number of Acceptor applications and y number of Donor applications. The figure shows the average behavior within subsets of workloads grouped by a unique Acceptor-to-Donor ratios, as well as the average across all workloads.

The figure shows that although DSR improves the speedup over the base case, there is a significant performance gap between it and OPRP. Considering one of the main differences between these two schemes, in addition to having an oracle replacement policy, it also adds placement decisions to capacity sharing (by allowing newly fetched blocks to be placed at both the local and remote caches), we are motivated to investigate the significance of placement decisions in capacity sharing strategies.

Our investigation confirms that applications that benefit from additional cache capacity (Acceptors) are also ones that tend to experience a high number of remote hits in DSR. The high number of remote cache hits is a significant reason for the performance gap between DSR and OPRP. To narrow the gap, we design a simple, predictor-based

Fig. 1. Speedups for fifty workload mixes showing the performance of DSR and the oracle scheme over the base case of private caches without capacity sharing. Additional details on simulated machine parameters and workloads can be found in Section 5.

scheme called *Adaptive Placement Policy* (APP) that learns from an application's past cache behavior to make a remote placement decision. APP's predictor structure is small (32 bytes in size) and has a simple organization. APP tracks whether a local cache block is accessed by the processor while it resides in the local cache and marks blocks that are never accessed as potential candidates for remote placement. Further, APP uses Set-Dueling [Qureshi 2009] to dynamically identify applications that should be allowed to place their blocks in remote caches. Despite its simplicity, APP performs robustly across a wide range of workloads and scenarios. We evaluate APP on a quad-core CMP system with 1 MB, 8-way, private last level L2 caches. We use 50 multiprogrammed workload mixes each consisting of 4 SPEC2006 applications. The workload mixes contain varying ratios of Acceptor and Donor applications. Our evaluation shows that APP improves the performance of a CMP without capacity sharing by 29% on average. APP also outperforms the state-of-the-art capacity sharing technique, DSR, by up to 18.2%, with a maximum degradation of only 0.5%, and an average improvement of 3%. Further, we study the sensitivity of APP and DSR to increasing remote cache hit latency, and a range of L2 cache sizes, as would be expected in larger-scale CMPs. We find that APP continues to perform robustly under these scenarios.

The remainder of this article explores the various aspects of APP's design and performance, and is organized as follows. Section 2 describes related work in cache management of Uni- and Multiprocessor systems. Section 3 motivates our approach to solving the problem of private L2 management. Section 4 describes the design and working of APP, and OPRP in details. Section 5 describes our evaluation methodology. Section 6 provides the results from our evaluations and analyzes the findings. Section 7 concludes this work.

2. RELATED WORK

Management of shared caches in CMPs. There is a rich body of work that has studied the problem of effectively managing shared on-chip cache resources in CMPs. Solutions include software-controlled schemes [Rafique et al. 2006; Tam et al. 2009; Awasthi et al. 2009], and hardware-only techniques. Hardware techniques differ in their approach: partitioning cache, ways [Qureshi and Patt 2006; Suh et al. 2004], partitioning cache sets [Rolan 2009; Srikantaiah et al. 2008], partitioning groups of lines [Kim et al. 2004], thread-aware replacement policy [Liu et al. 2008; Chaudhuri 2009], thread-aware insertion policy [Qureshi et al. 2007; Jaleel et al. 2008], and thread-aware insertion and promotion policies [Xie and Loh 2009].

Dynamic Insertion Policy (DIP) [Qureshi et al. 2007] and Thread-Aware Dynamic Insertion Policy (TADIP) [Jaleel et al. 2008] motivates the need for careful insertion of a block into a cache set. Another recent proposal, Pseudo-LIFO [Chaudhuri 2009] effectively impacts the insertion position of a block by utilizing both the recency stack and the fill stack, which ranks lines based on the order in which they were inserted into a cache set. This body of work explores the impact insertion (and replacement) decisions can make in shared caches.

Private caches present a new set of challenges and opportunities. Placement across caches, not insertion in the local cache at a particular stack position, should be considered. Placement decisions, even on a hit, can significantly impact the block's access latency. In addition, there is typically no global recency or fill stack information across corresponding sets in the private caches.

Management of private/distributed caches in CMPs. Victim Replication (VR) [Zhang and Asanovic 2005] and Adaptive Selective Replication (ASR) [Beckmann et al. 2006] studied techniques to allow remotely-homed blocks to be replicated in the local cache. These schemes primarily apply to multi-threaded workloads, where blocks identified to be more important are replicated locally (or victimized remotely). With multiprogrammed workloads, which is the focus of this study, there is no significant data sharing and therefore no reason to replicate data.

The work that comes closest to ours in terms of the scope of the problem is work that studies how capacity can be shared among distributed caches. We refer to such a technique as *capacity sharing* [Samih et al. 2009]. Capacity sharing techniques in prior studies include Cooperative Caching (CC) [Chang and Sohi 2006] and Dynamic Spill Receive (DSR) [Qureshi 2009]. CC was the first work to bring the notion of capacity sharing across private caches. Prior to this, CMP NU-RAPID [Chishti et al. 2005] proposed Capacity Stealing (CS) in order to manage the capacity of a shared cache's banks.

CS and CC allow each core to spill an evicted block to a remote private cache. Such capacity sharing allows any core to spill into other caches regardless of the temporal locality of the application running on that core. DSR improves upon CC, and identifies the existence of Acceptors and Donors. Further, it presents a dynamic mechanism to classify which application can spill, and which application can receive at any given time. Cooperative Cache Partitioning (CCP) [Chang and Sohi 2007] also builds upon CC by prioritizing cooperative caching opportunities, across applications in a coschedule that are competing for cache space, in a time-sliced manner. CS, CC, CCP, and DSR explore remote block placement on eviction of a local block. In contrast, in this article we study remote block placement when a block is first brought on chip.

3. MOTIVATING THE NEED TO CONSIDER PLACEMENT DECISIONS

In Section 1, we showed results that show a gap between the performance of DSR, a state-of-the-art capacity sharing technique, and the performance of an oracle scheme (OPRP). Our hypothesis is that this gap primarily exists because of the oracle scheme's ability to selectively place newly fetched/accessed blocks in remote caches. In this section, we investigate this hypothesis in detail as a first contribution of this article.

If it were true that OPRP outperforms DSR due to its selective placement of incoming blocks in a remote L2 cache, then it must be the case that the incoming blocks are going to be reused by the processor later than blocks that are already stored in the local cache. This implies an anti-LRU reuse pattern, in that blocks that were less recently used (those already in the local L2 cache) are more likely to be accessed in the near future compared to blocks that were more recently used (the most recent local L2 miss). For OPRP to significantly outperform DSR, there has to be a relatively large number of blocks that exhibit this anti-LRU temporal reuse pattern.

Fig. 2. Stack Distance Profiles for 10 SPEC CPU2006 applications: x-axis is the cache way number, bars represent the percentage of hits to the cache blocks from the MRU position on the left to the LRU position on the right; 32 ways make up a 4MB cache; right-most bar displays the % of misses.

Table I. System Configuration

Cores	2GHz 4cores, in-order single issue
Private L1 I/D caches	32KB, 2-way, 2-cycle access latency 64B block size
Private L2 caches	1MB, 8-way, 10-cycle local hit (or 512KB, 8-way, 7-cycle local hit) (or 2MB, 8-way, 12-cycle local hit) 40-cycle remote L2 cache hit (varied in sensitivity study[40-100]) 64B block size LRU replacement policy
Memory	300-cycle access latency

In order to investigate this anti-LRU temporal reuse pattern behavior, Figure 2 shows the *stack distance profile* [Mattson et al. 1970] for 10 SPEC CPU2006 benchmark applications. The L1 cache configuration (Table I) is 32KB in size and 2-way associative, across all experiments. The x-axis in the figure shows the cache ways sorted based on their recency of accesses assuming a 32-way 4MB total cache capacity (hence, each way corresponds to $4\text{MB}/32 = 128\text{KB}$). The y-axis shows the percentage of total cache accesses that hit in a given stack position. The leftmost bar represents the most-recently used (MRU) position, and the rightmost bar represents fraction of accesses that miss on all 32 ways. Under LRU replacement, a hit rate for a given cache size is represented by the area under the curve from the first until the last stack position the cache can hold. For example, the hit rate of a 1MB cache is the sum of bars way0, way1, ..., way7.

The figure shows two rows of applications, where the top row contains ones that show significant miss rate reduction when the cache capacity is increased from 1MB to 4MB (Acceptors), as evident in the large area under the curve between way 8–31. The second row shows applications that do not benefit from increasing the cache capacity from 1MB to 4MB (Donors). Some of the applications in the second row have very small miss rates (namd, povray, and sjeng), indicating small working sets, while others have very large miss rates (libquantum and milc), indicating either very large working sets or streaming memory access patterns with little temporal reuse. These donor applications not only cannot benefit from larger cache capacity, but also have substantial cache capacity that can be donated without affecting their miss rates.

If an application has a *perfect LRU* temporal reuse behavior, that is, blocks that were accessed more recently have a higher likelihood of being accessed again, the stack distance profile will show monotonically decreasing bars. The figure shows that all Donor applications exhibit perfect LRU behavior. However, Acceptors show more interesting behavior. Four out of five Acceptor applications (xalancbmk, omnetpp, soplex, and hmmer) show imperfect (or anti-) LRU temporal reuse behavior as evidenced by significant bumps at way8–32 in the stack distance profile. Only one Acceptor application (bzip2) has a nearly perfect LRU profile. The extent of anti-LRU temporal reuse behavior varies across applications. For example, omnetpp has less than 3.60% hits in the first 8 stack positions (1MB of cache) but 95.90% hits in the next 24 ways (1 to 4MB of cache). The high correlation between Acceptors and anti-LRU behavior makes sense, because with LRU's monotonic decrease, once one bar has a small fraction of access, accesses to higher stack positions must be lower than it. Therefore, overall we can conclude that Acceptors tend to exhibit some anti-LRU temporal locality behavior, while Donors tend to exhibit perfect LRU temporal locality behavior.

Note that if a capacity sharing technique only places blocks evicted from the local cache in remote caches, then necessarily the local cache stores more recently used blocks than remote caches. Since all existing capacity sharing techniques (CC and DSR) use remote caches as the victim cache for the local cache, they guarantee that remote caches store less recently used blocks. However, our earlier observation concludes that applications that benefit from additional cache capacity (i.e., Acceptors) are also the ones with anti-LRU behavior. Therefore, by treating remote caches as a victim cache for the local cache, CC and DSR guarantee the high occurrence of remote cache hits relative to local cache hits.

While remote hits are cheaper than misses, local cache hits are much preferred over remote hits. Remote L2 cache hits are costly from both a performance and a power point of view. A remote hit has a significantly longer latency compared to a local hit due to the coherence action needed for the remote cache to source the block into the local cache. In addition, it also consumes more power due to the coherence request posted on the bus, snooping and tag checking activities in all remote caches, and data transfer across the bus. Furthermore, future technology scaling allows more cores to be implemented on a chip, which increases the relative distance of remote caches relative to the local cache, for instance, due to additional on-chip network hops and routing delay. In contrast, the clock frequency growth of the processor may be slowing; hence, the off-chip memory access latency is only growing slowly relative to the access latency of on-chip cache. Thus, remote cache hit latency will become a larger problem in the future. It is very important, therefore, to maximize the local hits by converting remote hits into local hits.

The key to converting remote hits into local hits is to *selectively* decide to place blocks locally versus remotely when they are brought into the L2 cache from the lower level memory hierarchy. Remote placement allows blocks that have the highest chance to be accessed by the processor, regardless of their stack positions, to be placed in the local cache.

4. DESIGNING PLACEMENT POLICIES FOR CAPACITY SHARING

This section discusses how to dynamically identify which and when applications should be allowed to place blocks in remote caches (Section 4.1), our oracular placement policy (Section 4.2), and a hardware-implementable placement policy (Section 4.3).

4.1. Dynamic Identification of Spillers and Receivers

Not all applications should be allowed to place their blocks in remote caches, because they may not benefit from increased cache capacity. Evaluating which and when

applications should be allowed to place blocks in remote caches should be performed dynamically and continually, because temporal reuse behavior is not only application specific, but may also be phase-specific. We refer to a cache (or the application using it) that is allowed to place blocks in remote caches as a *Spiller*, and a cache that can only receive blocks from others as a *Receiver*.

Ideally, we want Acceptor applications to be identified as Spillers and Donor applications as Receivers. However, the identification is not that simple for several reasons. First, an application may switch behavior between Acceptor and Donor dynamically based on its execution phase. Second, an application's classification as a Spiller or a Receiver cannot be determined in isolation. For example, in a workload with many Acceptors and few or no Donors, it may be beneficial for system performance to classify some Acceptors as Receivers so that some of their cache capacity can be donated to other Acceptors which respond much more to additional cache capacity. Finally, applications or threads may migrate from core to core, changing the behavior cores exhibit. Therefore, each application must be classified dynamically and continually as Spiller or Receiver.

To achieve that, we rely on *set dueling* [Qureshi 2009], where each cache dedicates a small subset of sets (e.g., 32 randomly chosen sets) across all caches to always spill, and another subset to always receive. The miss rates of spiller subset and receiver subset are continuously compared to determine which subset achieves a lower all-core miss rate. The policy applied to the rest of the cache sets follows one from the subset that achieves the lowest miss rate. In order to choose a winning policy (spill vs. receive), each cache is augmented with a 10-bit saturating PSEL counter [Qureshi 2009].

4.2. Design of Oracular Placement Schemes

In order to study the upper-bound performance improvements from remote placement policies, we developed a hypothetical Oracular Placement and Replacement Policy (OPRP). When a Spiller suffers a cache miss, OPRP determines if the incoming block should be placed in the local L2 or in a remote L2 by looking at a trace of *future* accesses of the Spiller application. This trace is generated prior to simulation run time. Similar to Belady's optimal replacement algorithm [Belady 1966], all blocks in the local cache set of the missed block, and the miss block itself, are compared against the trace. If the missed block is accessed farther in the future compared to currently cached local blocks, the block is placed on a remote cache. Otherwise, a victim block is selected from the local cache to be spilled into a remote cache. In any case, the missed block is supplied to the processor and its L1 cache.

Placement decision must be made not only when there is a global miss to a new block. It must also be made when there is a remote cache hit, where we must decide whether to bring the block into the local cache. For OPRP, when there is a remote L2 cache hit, the future access trace is consulted again. If the remotely hit block is accessed farther in the future compared to currently cached local blocks, the block is left in the remote cache. Otherwise, a victim block is selected from the local cache to be swapped with the remotely hit block. In any case, the original or swapped remote block is marked as the new most recently used block in the remote cache. The block is, as always, supplied to the processor and its L1 cache.

The optimality of the placement or replacement decision can only be guaranteed for a given application and not globally across a mix of applications. This is because there is no way to know a priori how traces of accesses from different applications will be interleaved in the actual multiprogrammed execution. Therefore, OPRP only makes oracular decisions from a Spiller's perspective, but not a Receiver's. For Receivers, we simply incorporate performance guard bands so that their performance is not impacted by much (discussed in Section 4.5).

Note that OPRP consults the future trace to guide *both* placement and replacement. The decision of whether to place an incoming cache block in the local or remote cache is a placement decision, but what block is victimized from the local cache is a replacement decision. In order to bound the performance improvement attainable from the placement decision alone, we design a new Oracular Placement Policy (OPP). OPP uses the same policy as OPRP in determining whether a block should be placed in the local or remote cache. However, when it is decided that a block should be placed locally, the LRU block is selected as a victim, instead of the block used the farthest in the future. Thus, future trace information is only used for making placement decisions.

4.3. Design of APP

Obviously, OPRP and OPP are not implementable as they rely on future trace information. In order to design a placement policy that has a practical hardware implementation, we can approximate *future* information with *past* information. We refer to this scheme as Adaptive Placement Policy (APP). Such a scheme would require a predictor table and logic that determines whether an incoming or remotely hit block should be in the local or remote cache.

4.3.1. APP Predictor Design. To be practical, APP predictor must be low cost and efficient. Each of private L2 cache is augmented with the predictor, but only when a core runs an application that is determined to be Spiller, the predictor is used.

Without future trace information, APP cannot determine exactly when an incoming block will be accessed again in the future. However, it can record the past history of the block and extrapolate its behavior into the future. APP records whether a block was accessed while it was last resident in the cache. A block that was not accessed during its residence in the local L2 cache either indicates that it will never be accessed again, or more likely its stack/reuse distance is larger than the L2 cache associativity. Thus, if the behavior repeats, such a block should still be fetched on chip, but placed in a remote L2 cache so that it does not pollute the local L2 cache while it is waiting to be accessed again. APP predictor attempts to identify such blocks.

We found the behavior of blocks not being accessed during their residence in the L2 cache to be quite common among Acceptors. One situation that leads to such behavior is when the block's temporal and spatial reuse can be captured completely in the L1 cache. Thus, the block is brought into the L2 cache, but is only accessed at the L1 cache, until the block is evicted from the L2 cache, and a new miss prefetches the block.

In order to provide the block usage history during the block's residence in the local L2 cache, each block in the L2 cache is tagged with a single "Accessed" bit that is reset when the block is initially installed in the L2 cache, and is set when the block is accessed. When the block is evicted, the block's information, along with its access bit, is given to the APP predictor to be recorded. The local APP predictor is looked up when placement decision needs to be made (essentially at every local L2 cache miss), and is updated on each local block eviction. Therefore, the lookup and update are not on the critical path and should not affect cache access latency.

Next, we will discuss the APP predictor structures.

Instruction-Based Predictor. The role of APP predictor is to record the usage behavior of evicted blocks when they were last resident in the local L2 cache. An important question is whether the information should be recorded per instruction, per address, or a combination of them. The first option is to record the usage information per program counter (PC) of the memory instruction that accesses the block. With this option, we keep a per-PC access history in a prediction table. Any blocks that are brought into the cache by a PC will be tagged by this PC, and when any of them is accessed during its residence and is evicted, we set the accessed bit for this PC to record the reuse.

With any limited-size predictor structures, we have to deal with the possibility that multiple PCs map to the same entry in the table. Conflicts between PCs that map to the same entry can be handled by discarding older entries from the table, allowing entries to spill to the main memory, or by allowing them to share a common entry. After experimentation with various designs, we find that a small (4096) and simple (direct-mapped) table works well enough to deliver accurate predictions. The reason for why a small predictor table works well is that many cache misses are caused by a small number of load/store instructions, hence keeping a small number of PCs in the table gives good performance.

The predictor table is indexed by 12 least significant bits of the PC, whereas the remaining PC bits are stored in each entry as tag bits. In addition to tag bits, each entry includes a valid bit, which is used to validate a given PC entry, and a 2-bit saturating counter, which is used to approve a remote versus local prediction. The use of these bits will be discussed in greater detail later.

Address-Based Predictor. Another possibility for APP predictor table organization is to keep per-address information. In contrast to PCs, where keeping a few PCs that produce most misses is sufficient, we can expect that the number of unique memory block addresses within a program to be very large. Attempting to learn individual behavior of each block is impractical, as we need a huge prediction table structure to keep all of them, even when entries can spill to main memory. Allowing new entries to discard older entries still will not allow effective recording when the number of active blocks is much higher than number of table entries. This leaves us with the only one choice left: allowing multiple block addresses to share a common entry. Such a sharing policy works if many blocks share the same behavior, but not when they produce different behavior. Fortunately, in most cases, when an application has a high reuse distance, it affects most of the blocks it touches. Hence, allowing multiple block addresses to share a common entry helps accelerate the learning, rather than hurts performance.

To address the prediction table, a hashed index is generated by *folding* down the cache block address tag into an n -bit index, which addresses a table with 2^n entries. By folding we mean dividing up the address tag into n -bit entities and XORing them (an entity is zero-padded before XORing, if it is not a multiple of n -bits). Such hashing does not guarantee that addresses with similar behavior would share a common predictor entry, but because of global access behavior common to many blocks, the choice of hashing function only acts to ensure that entire table is utilized. The global reuse behavior across blocks is corroborated by our experiments, where even a small table with only 256 entries gives the most cost-effective design (Section 6.3).

Instruction-and-Address-Based Predictor. The final predictor structure we try combines PC and block address behavior history recording. To index the prediction table, both the PC and block address are folded into n -bit entities, which are then XORed together. The structure of this predictor is similar to the address-based one in terms of table size. The difference is only the input used for indexing the table.

Predictor Analysis. Let us compare the complexity of PC-based, address-based, and hybrid predictor structures. First, in most memory architectures, the PC of a load/store instruction is not propagated down to the last-level cache (LLC). Requiring a predictor associated with a LLC to have PC information requires relatively major changes to the memory hierarchy architecture. Second, in order to update the predictor table when a block gets evicted from the cache, the PC of the instruction that *brought* the block on chip should be stored along with each cache line. This enlarges the tag array considerably and increases its access latency. Third, if sequential/stride hardware prefetcher is used at the LLC, the prefetcher issues only addresses. If our predictor

Fig. 3. Predicting with (a) and Updating (b) the APP predictor.

Fig. 4. Handling cache-misses with APP (assumes placement prediction is *Remote*).

requires PC information to work, it will not be able to decide where to place prefetched blocks.

As a result to PC-based predictor’s serious drawbacks, we adopt the address-based predictor as a primary design, but we evaluate all three predictor designs in detail: we quantify the hardware costs, predictor accuracy and performance of the three predictor designs in Section 6.3.

Predictor Consult and Update Mechanism. Figure 3(a) illustrates how the APP predictor table makes placement decisions, and Figure 3(b) shows how each cache block is tagged with an “Accessed” bit. As mentioned earlier, the table is indexed by folding the cache block address tag into an 8-bit index. The table is tagless.

Each entry in the prediction table contains a saturating counter, which is decremented when an evicted block records that it was accessed during its recent residence in the local L2 cache (its “Accessed” bit is 1), and incremented when the evicted block was never accessed during its recent residence (its “Accessed” bit is 0). When later the local L2 cache suffers a miss to the same block, the prediction table is looked up. If the value in the saturating counter is 1 or higher, the block is placed in a remote L2 cache. Otherwise, the block is placed in the local L2 cache. The initial values of the saturating counters in the predictor table are 0, hence blocks are initially always placed locally.

In our design, we choose 2-bit saturating counters. We tried 1-bit, 2-bit, and 3-bit saturating counters, and found that there is a slight performance advantage in using 2-bit over 1-bit counters, but there is no performance advantage in using 3-bit over 2-bit counters.

The storage overhead for APP predictor table with 256 entries is 512 bits (or 32 bytes). Compared to a 1MB L2 cache with a 64B block size, the total overheads, including the Accessed bits, is $512 \text{ bits} + 16384 \times 1 \text{ bits} = 2.06 \text{ KBytes}$, which is a negligible 0.2% area overhead compared to a 1MB cache.

Handling a Miss and Remote Cache Hit. Figure 4 illustrates how APP handles a global cache miss, i.e., miss in the local and remote caches. While the Spiller’s L2 miss

Fig. 5. Handling remote-hits with APP (assumes placement prediction is *Local*).

is outstanding, the APP predictor is looked up to determine whether the incoming block should be placed in the local L2 or a remote cache (Step 1 in the figure).

If it is determined that the incoming block should be placed in the local L2, the LRU block is victimized for the incoming block (Step 2a). The victim block is moved to a randomly-selected remote Receiver L2 cache. The random selection of a Receiver cache is appropriate in a bus-based CMP we assume, because the cache latency is uniform across all remote caches. For a NUCA CMP, however, a distance-aware selection is warranted. We leave this as future work.

If the APP predictor determines that the incoming block should be placed in a remote cache, a random remote Receiver cache is selected, and the LRU block there is replaced by the incoming block (Step 2b). The incoming block is marked as the MRU block at the remote cache (Step 3), while the remote victim block is discarded or written back if dirty (Step 4). The incoming block is always supplied to the requesting processor and its L1 cache (Step 5).

Figure 5 illustrates how APP handles a remote cache hit (upon a local L2 cache miss). The APP predictor is looked up to determine whether the remotely hit block should be left where it was or be brought into the local L2 cache (Step 1). If the predictor determines the block should be brought into the local L2, the remotely hit block is swapped with the LRU block in the local L2 cache set (Step 2) and marked as the MRU block. Otherwise, the remotely hit block remains in the remote cache, but becomes the MRU block there. The remotely hit block is supplied to the processor and its L1 cache (Step 3).

4.4. Coherence Protocol Modifications

We assume that L2 caches are interconnected with a shared bus, kept coherent with a MESI broadcast/snoopy coherence protocol. Upon a local cache miss, the APP predictor is consulted, and the miss is broadcast on the bus.

If the block exists in any remote cache, and APP predictor determined local placement, then a swap transaction is initiated. A swap transaction places the local victim block in a special buffer to create a placeholder for the remote block. Then, the remote block is placed on the bus using a regular cache-to-cache transfer, and is picked up by the requesting cache. Finally, the local victim is pushed to the remote cache. The first two steps are already supported in current MESI protocols. The last step, pushing a block to a remote cache, is already supported in some systems, such as the IBM WireSpeed CMP [Franke et al. 2010]. These steps do not involve a load or store instruction, so there are no critical path or memory consistency concerns. The only modifications to the cache coherence protocol are: (1) a new swap transaction to distinguish it from regular read miss, (2) a transient state in the coherence protocol to indicate the swap is pending completion, and (3) potentially additional buffers to hold swapped blocks temporarily.

APP can also be adapted (with some modifications) for tile-based CMPs where caches are interconnected with point-to-point links and kept coherent using a directory-based

protocol. Such a system has a nonuniform cache architecture (NUCA), and is beyond the scope of this paper. However, we will comment on what changes are needed to adapt APP to such a system. One modification is that since cache latencies depend on the physical distance from a given core, APP must be modified to take remote cache distance into account, for instance, by restricting remote placement to near or nearest neighbor caches. In addition, the outcome of APP predictor (local or remote placement) must be sent along the miss request to the directory at the home node. Local placement is handled in a traditional way. Remote placement requires the directory to identify which cache the block should be sent to, considering the physical distance. Finally, the determination for a remote cache hit is more difficult, because the directory information may contain stale information, for example when a clean block in a remote cache is evicted silently. To avoid delaying global miss resolution, a possible solution is for the directory to simultaneously inquire the remote cache and fetch the block from main memory. This avoids delay in fetching from main memory in the case that the remote cache no longer has the block, at the expense of additional power consumption. An alternative solution is to distinguish remotely placed blocks from local blocks, and require the directory to be notified when a remotely placed block is evicted (even when it is clean). Finally, co-ordination with the directory is needed when blocks in the local and remote cache are swapped.

4.5. Ensuring Receiver QoS

While capacity sharing is appealing due to significant performance improvement Spiller applications may achieve, it may reduce the performance of Receiver applications sufficiently to violate some Quality-of-Service (QoS) requirements. Such a situation must be avoided.

We augment APP with a *Miss-Rate Monitoring System* (MRMS) for the Receivers. We leverage the fact that each Receiver cache has 32 sets dedicated to spilling. MRMS monitors the miss rates of the spill sets and other sets. The spill sets record the miss rate if the cache does not accept victim blocks from other caches. If the miss rate in other sets is higher by more than 5% compared to the miss rate of the spill sets, the Receiver disengages from participating in APP and no longer accepts victim blocks from Spiller caches. Receiver accepts victim blocks on the premise that it has excess cache capacity that it does not need. Suffering from >5% increase in miss rate indicates that the donated cache capacity is needed for performance. The miss rate comparison is performed at every 3 million cycle boundaries. With our MRMS safeguard, across all cases in our experiments, the execution time perturbation of Receiver applications is negligible. It is worth mentioning that the 5% and the 3 million cycle thresholds are tunable parameters, and the designer can specify the thresholds which best fit the workload environment.

5. METHODOLOGY

Simulation Model. To evaluate our capacity sharing schemes with oracular placement/replacement, oracular placement, and realizable placement, we build a cycle-accurate multicore machine model on top of Simics [Magnusson et al. 2002], a full system simulation platform. We model a 4-core CMP system, where each core has private L1 caches and a private L2 last level cache. Each of the L2 cache is has 1MB size, 8-way associativity, and access latencies derived from Cacti 6.0 [Muralimanohar et al. 2007]. We also assume that the L2 caches are interconnected with a shared bus, and are kept coherent with a MESI broadcast/snoopy coherence protocol. MESI protocol already has a support for facilitating cache to cache transfer. Table I lists all relevant configuration parameters used in our experiments.

Table II. Acceptor and Donor Applications

Acceptors (anti-LRU)	omnetpp, xalancbmk, soplex, hmmer
Acceptors (perf-LRU)	bzip2
Donors (small WS)	namd, povray, sjeng
Donors (streaming)	milc, libquantum

Multiprogrammed Workloads. Before constructing workloads for this study, we first separate potential Donors from potential Acceptors among 19 C/C++ SPEC CPU2006 [Standard Performance Evaluation Corporation 2006] applications. If they benefit from $> 10\%$ increase in L2 cache hit rates when the L2 cache capacity is increased from 1MB to 4MB, they are categorized as Acceptors, otherwise they are categorized as Donors. Acceptors display two distinct temporal locality patterns. Some Acceptors show a *perfect-LRU* temporal locality behavior, where more recently used blocks have a greater reuse probability than less recently used blocks. Other Acceptors have some *anti-LRU* behavior, where recency of use mostly does not correlate with reuse potential. Figure 2 shows applications with each of these two temporal locality behaviors. Donors, on the other hand, cannot benefit from an increase in the cache space from 1MB to 4MB. This is either because they have small working sets that fit in a 1MB L2 Cache, or they have a streaming data access pattern which suffers a high miss rates regardless of any additional cache capacity. Consequently, we divide Donors further into two groups: small-working-set and streaming.

The applications and the category they fall into are shown in Table II. The first two rows contain all applications we can identify as Acceptors in SPEC CPU2006. The next two rows contain a subset of applications we can identify as Donors in SPEC CPU2006. We chose only a subset of all Donors because we observed that Donors behave similarly from the point of view of the insights we hope to gain from this study. All Donors with a small working set share the same trend (L2 cache miss rate of less than 2%); therefore, we use a small representative set for such Donors. Similarly, all Donors with a streaming data access pattern behave similarly (less than 2% reduction in miss rate upon increasing the cache from 1M to 4MB).

In order to cover all distinct workload scenarios, we choose fifty 4-benchmark workloads that cover different ratios of Acceptors and Donors, different types of Acceptors (anti- and perfect-LRU), and different types of Donors (small-working-set and streaming). The workloads are shown in Table III. A workload denoted as $A_x D_y$ represents a workload with x Acceptors and y Donors. The first ten workload mixes have one Acceptor and three Donors. The next fifteen workload mixes have two Acceptors and two Donors. The next fifteen workload mixes have three Acceptors and one Donor. The last ten workload mixes have four Acceptors and no Donors. Acceptor applications are distributed nearly uniformly across all workloads (i.e., each Acceptor appears in almost the same number of workloads). Donors are selected in a similar fashion. However, since all Donors behave the same from the point of view of this study, they might as well be selected randomly.

To test each workload, each of the four applications in the workload are fast-forwarded for 10 billion instructions in order to skip their initialization phase. After that but prior to statistics collection, the cache models are warmed for 1 billion cycles. Finally, timing simulation is started and each workload is run until the slowest application runs for 250M instructions.

6. RESULTS AND ANALYSIS

In this section we evaluate the performance of APP against a base case of private caches without capacity sharing (Base), Cooperative Caching (CC) [Chang and Sohi 2006],

Table III. Workloads (Grouped by the Acceptor to Donor Ratio)

Group	Workloads in the group
1 Acceptor 3 Donors (A1D3)	$M1$ {omnetpp, namd, povray, milc}, $M2$ {xalancbmk, libquantum, namd, namd}, $M3$ {soplex, milc, povray, namd}, $M4$ {bzip2, sjeng, namd, namd}, $M5$ {omnetpp, milc, milc, namd}, $M6$ {xalancbmk, povray, namd, sjeng}, $M7$ {soplex, libquantum, libquantum, milc}, $M8$ {bzip2, milc, sjeng, libquantum}, $M9$ {hmmmer, namd, namd sjeng}, $M10$ {omnetpp, milc, povray, lib}
2 Acceptors 2 Donors (A2D2)	$M11$ {omnetpp, xalancbmk, namd, povray}, $M12$ {omnetpp, hmmmer, milc, namd}, $M13$ {xalancbmk, bzip2, libquantum, sjeng}, $M14$ {omnetpp, bzip2, namd, sjeng}, $M15$ {xalancbmk, hmmmer, povray, libquantum}, $M16$ {soplex, bzip2, milc, povray}, $M17$ {soplex, hmmmer, sjeng, sjeng}, $M18$ {omnetpp, omnetpp, povray, povray}, $M19$ {xalancbmk, xalancbmk, namd, milc}, $M20$ {bzip2, bzip2, milc, libquantum}, $M21$ {omnetpp, hmmmer, libquantum, milc}, $M22$ {xalancbmk, omnetpp, milc, namd}, $M23$ {soplex, xalancbmk, libquantum, povray}, $M24$ {bzip2, hmmmer, namd, sjeng}, $M25$ {hmmmer, hmmmer, sjeng, sjeng}
3 Acceptors 1 Donor (A3D1)	$M26$ {soplex, xalan, hmmmer, povray}, $M27$ {omnetpp, hmmmer, bzip2, povray}, $M28$ {soplex, xalancbmk, hmmmer, milc}, $M29$ {xalancbmk, bzip2, milc, hmmmer}, $M30$ {omnetpp, xalancbmk, xalancbmk, namd}, $M31$ {xalancbmk, xalancbmk, xalancbmk, namd}, $M32$ {xalancbmk, hmmmer, hmmmer, povray}, $M33$ {soplex, bzip2, hmmmer, milc}, $M34$ {soplex, bzip2, xalancbmk, libquantum}, $M35$ {soplex, sjeng, xalancbmk, libquantum}, $M36$ {xalancbmk, bzip2, bzip2, milc}, $M37$ {omnetpp, xalancbmk, bzip2, libquantum}, $M38$ {soplex, soplex, xalancbmk, povray}, $M39$ {bzip2, hmmmer, soplex, sjeng}, $M40$ {omnetpp, libquantum, soplex, soplex}
4 Acceptors 0 Donors (A4D0)	$M41$ {omnetpp, xalancbmk, xalancbmk, hmmmer}, $M42$ {bzip2, hmmmer, hmmmer, xalancbmk}, $M43$ {omnetpp, soplex, soplex, soplex}, $M44$ {omnetpp, hmmmer, omnetpp, omnetpp}, $M45$ {bzip2, omnetpp, bzip2, xalancbmk}, $M46$ {hmmmer, bzip2, omnetpp, hmmmer}, $M47$ {bzip2, soplex, soplex, hmmmer}, $M48$ {hmmmer, hmmmer, hmmmer, hmmmer}, $M49$ {bzip2, bzip2, bzip2, omnetpp}, $M50$ {xalancbmk, xalancbmk, xalancbmk, xalancbmk}

Table IV.

Performance metrics used and their definitions. $IPC_{i,base}$ represents the IPC of an application running alone on core i without capacity sharing enabled, while IPC_i represents the IPC of an application running on core i with capacity sharing enabled

Metric	Formula
Weighted Speedup	$\sum_{i=1}^n (IPC_i / IPC_{i,base})$
Harmonic Mean	$n / \sum_{i=1}^n (IPC_{i,base} / IPC_i)$
Throughput	$\sum_{i=1}^n IPC_i$

Dynamic Spill Receive (DSR) [Qureshi 2009], Oracular Placement Policy (OPP), and Oracular Placement-Replacement Policy (OPRP).

We use the following metrics to evaluate performance and fairness of the 50 multi-programmed workloads: Weighted Speedup, Harmonic Mean, and Throughput. These three metrics and their significance have been described in detail in prior work [Eyerma and Eeckhout 2008]. Table IV defines these metrics.

Of these metrics, Weighted Speedup is the preferred metric [Eyerma and Eeckhout 2008; Snavey and Tullsen 2000]. It corresponds to a physical, system-level measure of performance: the number of instructions executed across all applications in a multi-program mix per unit of time. This metric equalizes the contribution of each program in the mix by normalizing its performance in the mix to its performance when run in isolation. Weighted Speedup does not bias the measured performance by favoring high-IPC applications.

Fig. 6. Normalized Weighted Speedup for all workloads compared to the base case of private caches without capacity sharing, with average for all workloads shown as the last set of bars in part (d).

6.1. Performance of APP, OPP, and OPRP

Figure 6 shows the Weighted Speedup results for Cooperative Caching (CC), Dynamic Spill Receive (DSR), our scheme APP, and our two oracular schemes: oracular placement (OPP) and oracular placement-replacement (OPRP). Recall that CC provides capacity sharing where all applications can spill their victim blocks to other caches, while DSR improves upon it by detecting Acceptor applications and only allowing them to spill to other caches. The DSR implementation is based on the code provided by the authors of DSR [Qureshi 2009], integrated into our simulation infrastructure. The results are arranged into four charts (a,b,c,d) based on the Acceptor to Donor ratio in each group of workloads. The weighted speedups are normalized to the base case of a CMP with private caches without capacity sharing. The average weighted speedups over all 50 workloads are shown in the last set of bars.

There are two main observations that can be made from the figure. First, all schemes improve the Weighted Speedup for all workloads compared to the base case of private caches without capacity sharing. There is, however, a varying gap between the Weighted Speedup improvements across the various schemes. On average, over fifty workloads, CC and DSR improve the Weighted Speedup by 19% and 25% respectively, while the hypothetical OPP and OPRP improve the Weighted Speedup by 32% and 36% respectively. The Weighted Speedup improvement from the hardware-implementable APP falls right between DSR and OPP, averaging 29%.

Second, across different workload groups, the performance improvement of APP compared to DSR varies. The performance improvement of APP over DSR for A1D3 is the smallest (27.3% versus 26.5%) because only one Acceptor application benefits from APP's remote block placement, whereas the Weighted Speedup reflects the average improvement for all applications running together. Thus, the more acceptor applications

Fig. 7. Local L2 hit rate comparison for all groups of workloads for the base case of private caches without capacity sharing (BASE), CC, DSR, APP, OPP, and OPRP.

in a workload, the more the Weighted Speedup is improved with APP compared to DSR. APP's average Weighted Speedup improvement for A2D2 is 33.6% versus DSR's 29%. For A3D1, APP achieves an average Weighted Speedup of 22% compared to DSR's 18.2%. Finally, for A4D0, APP achieves 33.7% improvement in Weighted Speedup while DSR achieves 28.2%.

At first glance, the average improvement of Weighted Speedups achieved by APP over DSR may seem modest. However, there are three important points to consider. First, improving the performance over the state-of-the-art technique is not a minor feat. While APP's improvement over DSR may seem modest, DSR's improvement upon CC, which was the state-of-the-art of the time, may similarly seem modest (25% vs. 19%).

Second, recall that our workloads are designed to cover a wide range of workload mixes and behavior. For five workloads, APP outperforms DSR by more than 5%, and sometimes by significantly more (18%). Such a difference is significant considering that over DSR, APP relies on reducing cache hit latencies by only 30 cycles (converting a modest remote cache hit latency of 40 cycles to local cache hits latency of 10 cycles). As the number of cores on a chip grows, we expect that the ratio of remote cache hit latency to local cache hit latency will grow substantially, causing APP's benefit to increase as well. We quantify this in a later section when we vary the remote cache hit latencies (Section 6.5).

Finally, in all the workloads, APP is almost never outperformed by DSR, suggesting that APP rarely makes mistakes that result in the decrease of local hit rates versus DSR.

6.2. In-Depth Analysis of APP's Performance

To understand why APP outperforms CC and DSR but is outperformed by OPP and OPRP, Figure 7 shows the average *local cache hit rate* for each group of workloads. The hit rate for Acceptors in each group of workloads is calculated by averaging the hit rates of all Acceptor applications.

We can make the following observations from Figure 7. First, we can see that CC experiences the lowest local L2 hit rates (30.5%). Recall that in CC any core is allowed to spill victim blocks to any core regardless of their cache demand. Hence, the hit rates of Acceptors decrease due to the cache pollution from Donors, compared to the base case of private caches.

Second, DSR and the Base case experience similar local L2 hit rates (32%), with slightly lower hit rates in DSR for all groups of workloads. The similarity in local hit rates is due to the fact that in both policies, a block requested by the Spiller application is always brought into the local cache, and the only difference between them is in how to handle blocks evicted from the local cache. Hence hit rates between them should be identical if everything else is equal. However, since DSR dedicates 32 sets in each cache to always receive (this is necessary to dynamically identify Spillers/Receivers), DSR's local hit rates are always slightly lower than the Base case.

Third, compared to DSR, APP's local L2 hit rates for Acceptors are about 5% higher for all workload groups (37% vs. 32%). This result demonstrates that by selectively placing incoming blocks in remote caches, APP is successful in retaining more useful blocks in the local L2 cache, thus improving the local L2 hit rates. The increase in local L2 cache hit rates is responsible for the performance improvement APP achieves over DSR.

Finally, while APP improves the hit rate significantly over the base case, there remains a gap between Base and OPP hit rates (37% vs. 46.5%). The gap is caused by OPP making placement decisions based on future accesses, whereas APP makes decisions based on past access patterns. In addition, APP uses a relatively simple predictor design. More sophisticated predictor designs can narrow the gap with OPP significantly, at the expense of increased complexities (Section 6.3).

Comparing the average local L2 cache hit rates achieved by OPP versus DSR (46.5% vs. 32%) and versus OPRP (46.5% vs. 56%), we can infer that the majority (roughly 60%) of the performance gap between DSR and OPRP is due to the intelligent placement of blocks, while the remaining is due to perfect versus LRU replacement policies. This corroborates our initial hypothesis that selective placement of cache blocks plays a major role in boosting the performance of capacity sharing schemes, and that our priority should be in providing intelligent placement of blocks, ahead of improving the cache replacement policies.

Workloads that enjoy significant performance improvement in APP (e.g., M11, M31, and M44) are also ones that are dominated by Acceptor applications with anti-LRU temporal reuse patterns. The average local hit rate of the Acceptors in each workload for APP versus DSR are 15% vs. 7% (M11), 20.3% vs. 9.3% (M31), and 22% vs. 5% (M44), respectively. Since remote placement of blocks is the only difference between APP and DSR, the results clearly point to the significant role placement decisions play in improving local cache hit rates of Acceptor applications. The increase in local hit rates also demonstrates that by allowing anti-LRU incoming blocks to be placed in remote caches, existing blocks in the local L2 cache enjoy better temporal locality from less cache perturbation.

A small number of workloads show almost negligible performance improvement, or a slight degradation over DSR (e.g., M25, and M04). Acceptors in these workloads (example bzip2 and hmmer) show an almost perfect-LRU behavior and very little anti-LRU behavior (Figure 2). These workloads already experience high local hit rates (66% for M25, and 82% for M04), hence most of the performance improvement can only come from additional cache capacity rather than converting remote cache hits into local cache hits.

Chip-Wide Hit Rate. Earlier, we have shown that APP increases the local L2 cache hit rate for Acceptor applications by about 5% compared to DSR. Such an increase in the local cache hit rate is sufficient to make APP's performance better than DSR's. However, this assumption holds true if and only if the increase in the local cache hit rate *does not* come at the expense of an inordinate drop in remote cache hits.

Figure 8 shows the average *chip-wide L2 hit rates* (the total number of L2 hits that are satisfied by the local or remote caches, divided by the total number of L2 cache

Fig. 8. Chip-Wide L2 hit rate comparison for all groups of workloads for CC, DSR, APP, OPP, and OPRP.

accesses) for all Acceptors in a given group of workloads. This metric considers both local and remote hits.

CC manages to bridge most of the gap by improving the chip-wide hit rates significantly over the base: from 33% (base) to 66% (CC). DSR improves upon CC and achieves an average of 73%. Not only is APP able to maintain this chip-wide hit rate, but it also manages to increase it slightly. It is interesting to note that APP continues to outperform DSR even in this metric. The reason for this is that by matching the temporal locality of each cache block with the its placement position (local vs. remote), APP is able to reduce perturbation to both the local and the remote cache space, resulting in a slight improvement in chip-wide hit rates.

Worst Case Performance of APP. We discussed in Section 6.1 that applications with an almost perfect-LRU behavior are not expected to benefit from APP since all incoming lines in such an application should be placed locally. However, if APP cannot improve the performance of some applications compared to DSR, it should not degrade them either. Our results confirm that APP’s maximum performance degradation compared to DSR, across all workloads, is a negligible 0.5% (M20). This is in contrast to APP outperforming DSR by 18.2% in the best case (M44), and by 3% on average.

Finally, applications in a given workload mix are continually adapting to act as either Spillers or Receivers. Therefore, it is critically important that the performance improvement for Spiller caches does not come at the expense of degrading any Receiver cache’s performance. We find that the threshold of APP’s Miss-Rate Monitoring System (Section 4.5) is rarely triggered for the QoS level we chose (5% hit-rate decline), confirming the resilience of most Receiver caches to losing some of their cache space. Further, only 4% of all applications (8 out of all the 200 applications in the 50 workload mixes) suffered an IPC degradation of more than 4% compared to the base case of no capacity sharing. Even for these applications, the worst-case slowdown for a Receiver is only 14%, which is more than completely offset by the speedup improvement in Spiller applications. In comparison, the worst-case slowdown for a Receiver in DSR is 13%.

Overall, the experimental results stress that APP outperforms local block placement schemes because of the improvement in local miss rate (more blocks are found in the local L2 cache), as well as the slight improvement in the chip-wide miss rate (more blocks are found in the L2 caches on chip). Further, the performance gain in anti-LRU applications does not come at the expense of degrading perfect-LRU applications. Finally, APP safe-guards Receiver applications from being adversely affected as they donate excess cache capacity to Spiller applications.

Fig. 9. Average weighted speedup over base (a), breakdown of speedup (b), average, minimum, and maximum accuracy for various predictors (c), and hardware cost for various predictors (d).

6.3. Impact of Predictor Design

In Section 4, we described three possible implementations for APP predictor. Figure 9 shows the performance of various designs with other parameters fixed according to Table I. Each predictor is denoted as $X.Y$ where X is the predictor type (PC for PC-based predictor, tag for address-based predictor, and PC.tag for a hybrid PC and address-based predictor); while Y is the number of index bits and the size of the table (2^Y entries).

Figure 9(a) shows the average weighted speedup across all 50 workloads normalized to the base case of private caches. It shows that PC-based predictors outperform address-based and hybrid predictors. To analyze the source of performance improvements, Figure 9(b) breaks down the contributions to speedups from various factors. The lowest component of bars represents a dummy predictor that always makes the opposite placement decision (local vs. remote) vs. OPP. The second component represents the additional speedup from DSR (always placing blocks locally). The third component represents the additional speedup from APP with various predictor designs, whereas the last component represents the additional speedup from OPP. The figure shows that the headroom for improving speedup from the default APP predictor (TAG.8) is relatively small. The PC.12 predictor almost completely bridges the gap of APP and OPP, while PC.8 predictor shows a close performance as well.

The reason why PC-based predictors perform better than address-based and hybrid predictors can be seen in the predictor accuracies in Figure 9(c). PC.12's average accuracy is 82% (93% maximum and 62% minimum), while TAG.8's average accuracy is 73% (80% maximum and 60% minimum). The superiority of PC-based design can be attributed to the small number of PCs in a program phase contributing to most of cache misses, and the uniform temporal reuse behavior for all blocks accessed by the PCs. The address-based predictors, on the other hand, map many more block addresses to a

Fig. 10. Weighted Speedup improvements for APP vs. DSR for different cache sizes compared to the base case of private caches.

small number of prediction table entries, hence accuracy relies on the blocks showing identical reuse distance behavior.

However, PC-based predictor’s performance superiority must be evaluated against its cost and complexity. Figure 9(d) shows the total hardware overheads of the prediction table and the required additional tag information for cached blocks. PC.12 requires 11.5KB prediction table, 33-bit additional tag information per cache block, for a total of 77.5 KB, representing a 7.5% area overhead for a 1 MB cache. TAG.8 is much simpler, only requiring 32-byte prediction table, and 1-bit additional tag information for the “Accessed” bit per cache block, for a total of 2KB, representing a 0.2% area overhead for a 1MB cache. Considering the tiny hardware overhead for address-based predictor, and that it still performs within 3 percentage points in weighted speedup compared to PC-based predictor, we view address-based predictors as more cost effective. Furthermore, a PC-based predictor requires a relatively major change to the entire memory hierarchy, as PC is not normally available at the last level cache (Section 4.3.1).

6.4. APP’s Sensitivity to Cache Size

In this section, we evaluate APP performance when the L2 cache size is varied from 512KB, 1MB (default in other sections), and 2MB. Latencies for various cache sizes are shown in Table I. Figure 10 shows the weighted speedup results across various cache sizes, normalized to the base case of no capacity sharing.

The figure shows that as the cache size increases, the average performance improvement from APP (and DSR) increases. The reason is that with larger caches, there is more excess cache capacity that can be donated by Receivers, and there are more benchmarks which change from Spillers to Receivers as their working sets now fit in the cache. The increase in excess cache capacity in the system increase the number of Spillers and improves the performance of each Spiller.

The figure also shows that across all cache sizes, APP consistently outperforms DSR. However, the relative gap between them narrows with larger cache sizes. The reason is that larger caches also convert many remote cache hits in DSR into local cache hits as the local cache can hold more of the working set of the Acceptor applications. Thus, in a way, larger caches compete with APP in tackling the same problem, which is, improving the local hit rates. However, averages in this case can be misleading, because with larger caches, some Acceptor applications become Donor applications as their working sets now fit in the local cache. Since APP only benefits Acceptor

Fig. 11. Average Weighted Speedup across fifty workloads for APP and DSR with various remote hit latencies (40, 60, 80, and 100 cycles). Other latencies are unchanged.

Fig. 12. IPC throughput (a) and harmonic mean (b) for various schemes, normalized to private caches with no capacity sharing.

applications, the average performance improvement reflects fewer benchmarks that enjoy significant performance improvement.

6.5. APP's Sensitivity to Remote Cache Hit Latency

Thus far, all experiments assume a fixed remote cache hit latency of 40 cycles. As more cores can be integrated on a single CMP, the average remote cache hit latency can be expected to grow relative to the local cache hit latency. For example, consider a 64-core system connected by an 8×8 mesh. Round-trip communication between two diagonal cores requires 28 hops, which translates to a total remote hit latency of 140 cycles, if we assume a 5-cycle hop latency.

Figure 11 shows the average weighted speedup for APP and DSR, with various remote cache hit latencies: 40 (default), 60, 80, and 100 cycles. The figure shows that while the absolute spread between APP and DSR remains constant, the relative improvement of APP over DSR grows as the remote hit latency increases. The reason for this is that with higher remote hit latencies, converting remote hits into local hits becomes a relatively more important factor for performance improvement, whereas converting global misses to remote cache hits becomes a relatively less important factor for performance improvement.

6.6. Other Performance Metrics: Throughput and Fairness

Figure 12 shows IPC throughput (a) and harmonic mean of speedups (b) of DSR, APP, OPP and OPRP, normalized to no capacity sharing. The figures show results that are consistent with the earlier figure showing weighted speedups.

7. SUMMARY

Capacity sharing is a technique for reducing cache fragmentation in a CMP system with private last-level caches; it allows applications that need additional cache space to place their blocks in remote caches. Current capacity sharing mechanisms treat remote caches as the victim cache for the local cache. We have shown that such a strategy guarantees a high number of remote cache hits relative to local cache hits for applications that exhibit anti-LRU behavior, which are usually the same applications that benefit from additional cache capacity.

We have investigated strategies that consider placing not only locally evicted blocks in remote caches, but also *newly fetched blocks* in remote caches. We investigate the upperbound performance that can be gained from combined placement and replacement decisions in capacity sharing, by using future trace information to make the decisions. Based on our findings, we propose a scheme that is implementable in hardware with little hardware overhead. We show that this scheme, APP, improves performance by 29% on average compared to a baseline with no capacity sharing, across 50 multiprogrammed workloads consisting of four SPEC2006 applications. APP outperforms DSR, the state-of-the-art capacity sharing mechanism that only places local victim blocks in remote caches, by up to 18.2%, with an average improvement of 3%. APP dynamically identifies which applications should be allowed to place their blocks in remote caches, and which applications should not. In addition to improving aggregate performance significantly, APP also has safeguards to ensure that applications whose caches are accepting blocks from other cores are not slowed down by much.

REFERENCES

- AWASTHI, M., SUDAN, K., BALASUBRAMONIAN, R., AND CARTER, J. 2009. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *Proceedings of the 15th IEEE International Symposium on High Performance Computer Architecture (HPCA'09)*.
- BECKMANN, B. M., MARTY, M. R., AND WOOD, D. A. 2006. ASR: Adaptive selective replication for CMP caches. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society.
- BELADY, L. A. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.* 5, 2.
- CHANG, J. AND SOHI, G. S. 2006. Cooperative caching for chip multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA'06)*.
- CHANG, J. AND SOHI, G. S. 2007. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the International Conference on Supercomputing*. ACM, 242–252.
- CHAUDHURI, M. 2009. Pseudo-LIFO: The foundation of a new family of replacement policies for last-level caches. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*.
- CHISHTI, Z., POWELL, M. D., AND VIJAYKUMAR., T. N. June 2005. Optimizing replication, communication and capacity allocation in CMPs. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)*.
- EYERMAN, S. AND ECKHOUT, L. 2008. System-level performance metrics for multiprogram workloads. *IEEE Micro* 28, 3.
- FRANKE, H., XENIDIS, J., BASSO, C., BASS, B. M., WOODWARD, S. S., BROWN, J. D., AND JOHNSON, C. L. 2010. Introduction to the wire-speed processor and architecture. *IBM J. Resear. Devel.* 54, 1, 3:1–3:11.
- JALEEL, A., HASENPLAUGH, W., QURESHI, M., SEBOT, J., STEELY, JR., S., AND EMER, J. 2008. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*.
- KIM, S., CHANDRA, D., AND SOLIHIN, Y. 2004. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*.
- LIU, H., FERDMAN, M., HUH, J., AND BURGER, D. 2008. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE.
- MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HALLBERG, G., HOGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. 2002. Simics: A full system simulation platform. *Computer* 35, 2.

- MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J.* 9, 78–117.
- MURALIMANOHAR, N., BALASUBRAMONIAN, R., AND JOUPPI, N. 2007. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. 3–14.
- QURESHI, M. 2009. Adaptive spill-receive for robust high-performance caching in CMPs. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture (HPCA'09)*. IEEE.
- QURESHI, M. K. AND PATT, Y. N. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE.
- QURESHI, M. K., JALEEL, A., PATT, Y. N., STEELY, S. C., AND EMER, J. 2007. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. ACM.
- RAFIQUE, N., LIM, W.-T., AND THOTTETHODI, M. 2006. Architectural support for operating system-driven CMP cache management. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT'06)*. ACM.
- ROLÁN, D., BASILIO, B., AND FRAGUELA, R. D. 2009. Adaptive line placement with the set balancing cache. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*.
- SAMIH, A., KRISHNA, A., AND SOLIHIN, Y. 2009. Understanding the limits of capacity sharing in cmp private caches. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.161.7890>.
- SNAVELY, A. AND TULLSEN, D. M. 2000. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 234–244.
- SRIKANTIAH, S., KANDEMIR, M., AND IRWIN, M. J. 2008. Adaptive set pinning: Managing shared caches in chip multiprocessors. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM.
- STANDARD PERFORMANCE EVALUATION CORPORATION. 2006. <http://www.specbench.org>.
- SUH, G. E., RUDOLPH, L., AND DEVADAS, S. 2004. Dynamic partitioning of shared cache memory. *J. Supercomput.* 28, 1, 7–26.
- TAM, D. K., AZIMI, R., SOARES, L. B., AND STUMM, M. 2009. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. *SIGPLAN NOT.* 44, 3.
- XIE, Y. AND LOH, G. H. 2009. PIPP: Promotion/Insertion Pseudo-Partitioning of multi-core shared caches. *SIGARCH Comput. Archit. News* 37, 3.
- ZHANG, M. AND ASANOVIC, K. 2005. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*. IEEE.

Received October 2010; revised April 2011; accepted July 2011