Understanding the Limits of Capacity Sharing in CMP Private Caches

Ahmad Samih†, Anil Krishna‡, Yan Solihin†

†Dept. of Electrical and Computer Engineering
North Carolina State University
{aasamih, solihin}@ece.ncsu.edu

‡IBM
Systems and Technology Group
{krishnaa}@us.ibm.com

***Abstract*—Chip Multi Processor (CMP) systems present interesting design challenges at the lower levels of the cache hierarchy. Private L2 caches allow easier processor-cache design reuse, thus scaling better than a system with a shared L2 cache, while offering better performance isolation and lower access latency. While some private cache management schemes that utilize space in peer private L2 caches have been recently proposed, we find that there is significant potential for improving their performance. We propose and study an oracular scheme, OPT, which identifies the performance limits of schemes to manage private caches. OPT uses offline-generated traces of cache accesses to uncover applications' reuse patterns. OPT uses this perfect knowledge of each application's future memory accesses to optimally place cache blocks brought on-chip in either the local or a remote private L2 cache. We discover that in order to optimally manage private caches, peer private caches must be utilized not only at a local-cache-replacement time, as has been previously proposed, but also at cache-*placement* time - it may be better to place a missed block directly into a peer L2 rather than the traditional approach of first bringing it into the local L2. We implement OPT on a 4-core CMP with 512KB, 8-way, private caches, across 10 carefully chosen, relevant, multiprogram workload mixes. We find that compared to a baseline system that does not employ capacity sharing across private caches, OPT improves weighted-speedup (performance metric) by 13.4% on average. Further, compared to the state of the art technique for private cache management, OPT improves weighted-speedup by 11.2%. This shows the significant potential that exists for improving previously proposed private cache management schemes.**

I. Introduction

The design of Chip Multi Processor (CMP) memory hierarchies presents interesting design challenges in organizing and managing the on-chip cache resources. Individual L1 caches are typically tightly coupled to individual processor cores due to the tight timing requirement for L1 cache accesses. However, there are interesting design trade-offs with regard to whether the L2 (or L3) cache should be private to each core or shared by all cores.

A shared L2 cache allows applications to more fluidly divide up the cache space, and the aggregate cache capacity is maximized because no block is replicated in the L2 cache. However, keeping a shared L2 cache becomes less attractive in a large CMP due to the increasing access latency of a large cache, and the increasing complexity of the interconnect [1]. Private L2 caches lend themselves naturally to a tiled multicore configuration, which allows it to scale to a larger number of cores simply by replicating tiles. In addition, private L2 caches also enjoy better distance locality, as all data needed by a core is always brought into the local L2 cache of the core. Finally, it is easier to achieve performance isolation in private L2 caches than in a shared L2 cache, due to the natural boundaries between L2 cache tiles.

Unfortunately, a private L2 cache configuration has its own disadvantages as well, and can lead to uneven utilization of the overall cache space. This is especially true when a diverse mix of sequential programs run on different cores. Some programs may over-utilize the local L2 cache (due to their large working set), while others may under-utilize the local L2 cache (due to their small working set).

Such uneven cache utilization can result in a significant loss in the overall performance. Programs that require a large cache capacity are unable to utilize the excess cache capacity that may exist at the neighboring tiles. For example, with 512KB private L2 caches, we found that several SPEC CPU2006 benchmarks (e.g. namd, povray, milc, libquantum) suffer almost no decrease in performance when half their private L2 cache capacity is taken away. Unfortunately, in current private caches, such excess cache capacity cannot be donated to other cores which need more cache capacity.

This paper studies the problem of how we can enable private cache configurations to retain the benefit of small access latency while simultaneously allowing capacity sharing across peer L2 caches. We seek to uncover the *best* performance that can be expected of any collaborative private cache management scheme over the baseline system that does not employ any cache sharing. We also seek to quantify the potential that exists for the current, state of the art, private cache management schemes to extract more performance, and identify insights to guide such research in a promising direction.

Recognizing the need to provide capacity sharing in private caches, prior work, such as, Cooperative Caching [2] (CC) and Dynamic Spill Receive [3] (DSR), has proposed capacity sharing mechanisms. CC allows each core to spill an evicted block to any other private cache on-chip. This allows capacity sharing. Allowing any cache to spill into any other cache regardless of the temporal locality of the application running in the other core may, however, produce an unwanted effect

of polluting that other cache. This may degrade performance of the application running on the other core. Further, this may not even provide much performance improvement to the first application if it has little temporal locality. DSR removes this drawback by distinguishing between applications that benefit from extra cache capacity beyond that provided by their local L2, versus applications that can tolerate a reduction in L2 cache capacity without much performance loss. The former applications are referred to as Spillers and the latter, as Receivers. In DSR, only cache blocks evicted by a Spiller are allowed to spill into caches belonging to Receivers, while Receivers, which cannot benefit from additional cache capacity, are not allowed to spill their blocks. In a sense, Receiver-caches have a new role of being the victim caches of Spiller-caches. In this paper we investigate if treating peer caches as victims at replacement is the best way of utilizing their cache space.

The goal of this paper is to study what is the best way to provide capacity sharing in a CMP with private L2 caches. To accomplish that we study OPT, a hypothetical, oracle, cache co-operation policy that establishes a limit on of how well cache co-operation can work. OPT has perfect knowledge of future cache accesses for Spiller applications. It uses this knowledge to make block placement and replacement decisions.

We implemented OPT on a full-system simulator modeling a 4-core CMP with 512 KB, 8-way, private L2 caches, and running 10 different multi-program mixes with a varying ratio of Spillers and Receivers. OPT outperforms the baseline scheme with no collaboration among private caches by 13.4% on average (by as much as 30%) for the workload mixes we studied. Though DSR improves performance over the baseline, OPT has a 11.2% advantage over DSR on average, for the same workloads, indicating the *significant* opportunity that exists even for the the state of the art private cache management schemes.

OPT has perfect knowledge of an application's future accesses; however, we find that its superior performance is attributable as much to how OPT *uses* such knowledge, as it is to OPT's possessing such knowledge. We find a fundamental limitation of DSR, CC and several capacity sharing solutions proposed in literature. All relevant prior art proposes bringing in all incoming blocks into the local L2 cache first, and *only consider placing a block in a remote L2 on replacement*. In contrast, OPT evaluates the potential of both replacement *and placement* of a block in the local versus remote L2 cache space. We find that a significant portion of OPT's impact is due only to OPT's selective placement of blocks on a cache miss and a cache hit in a remote cache. This work motivates a potentially new direction for research in the area of private cache management in CMPs.

The remainder of this paper is organized as follows. Section II describes related work. Section III motivates our approach to solving the problem of private L2 management. Section IV describes the design of OPT. Section V describes our evaluation methodology. Section VI provides the results from our evaluations and analyzes the findings. Section VIII concludes.

II. Related Work

There is a rich body of work that has studied the problem of effectively sharing on-chip cache resources in CMPs; however, the problem of private cache management for multi-programmed workload mixes remains comparatively less studied.

The work that comes closest to ours in terms of the scope of the problem is Cooperative Caching (CC) [2] and Dynamic Spill Receive (DSR) [3]. CC was the first work, to our knowledge, which brought the notion of hardware-implementable cache sharing across private caches. CC allows 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 Spillers and Receivers. Only Spillers can take advantage of extra cache space; hence, only Spillers are allowed to spill victim blocks from the local cache to a remote, Receiver, cache. Receivers are applications that can give up a part of their cache capacity without suffering any significant performance degradation. Receivers can receive spilled blocks from Spillers, but themselves are not allowed to spill blocks to remote caches.

Managing banked shared caches is similar in concept to managing private caches. CMP NU-RAPID [4] proposes Capacity Stealing (CS) to address the issue of capacity sharing in a banked shared cache. CS is similar to CC and DSR in its placement aspect; incoming blocks are first placed in the group of banks closest to the requesting processor. These blocks are demoted (spilled) to remote banks, which belong to other processors, upon replacement.

CS, CC and DSR explore remote block placement on eviction of a local block. OPT explores remote block placement even when the block is first brought on-chip.

A rich body of work has been proposed recently to alleviate the problem of locality-agnostic, traditional, insertion and replacement policies in a cache. Dynamic Insertion Policy (DIP) [5] and Thread-Aware Dynamic Insertion Policy (TADIP) [6] affect the insertion position of an incoming block in a cache set. They indirectly identify dead blocks by inserting incoming blocks at the counter-intuitive, Least Recently Used, position in the cache set's recency stack. Another recent proposal, Pseudo-LIFO [7] 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. By getting rid of dead blocks faster more of the cache is used for blocks with a loose temporal reuse pattern. 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; insertion decisions are no longer guided by the recency stack position within a cache set. Instead, placement of a block can span across private caches. Further, 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 which maintains recency information across corresponding sets in the private caches.

Prior work, such as Victim Replication (VR) [8] and Adaptive Selective Replication (ASR) [9], has studied the problem of balancing effective on-chip cache space with cache access latency, by allowing remotely-homed blocks to be replicated in the local cache. These schemes work in a somewhat opposite fashion compared to OPT. While OPT may place blocks that are expected to be local, remotely, these schemes may place blocks expected to be homes remotely, locally. These schemes primarily apply to multi-threaded workloads, where blocks identified to be more important are replicated locally (or victimized remotely). With multi-programmed workloads, which is the focus of this study, there is no significant data sharing and therefore no reason to replicate data.

III. Motivating Placement

In Section I, we have discussed results that show that DSR, a current state of the art capacity sharing technique, is far from achieving the performance of an optimum placement of blocks in local versus remote L2 caches. Our hypothesis is that this is because DSR always places all incoming local L2 miss data in the local L2, whereas OPT may place such data in the remote L2s. In this section, we investigate this hypothesis.

If it were true that OPT 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 OPT to significantly outperform DSR there has to be a relatively a large number of blocks that exhibit this anti-LRU temporal reuse pattern.

In order to investigate this anti-LRU temporal reuse pattern behavior, Figure 1 shows the *stack distance profile* [10] for 8 SPEC CPU2006 benchmark applications. The x-axis shows the cache ways sorted based on their recency of accesses, in a 2MB 32-way L2 cache, and the y-axis shows the percentage of cache accesses that hit in a given way. Thus, the leftmost point represents the percentage of accesses to the most recently used blocks. As we go to the right along the x axes, the figure shows the percentage of accesses to less and less recently used blocks. Finally, the right-most point on the x-axis accounts for all the misses which the 32-way, 2MB cache was not able to avoid. Note that if an application has a perfect LRU temporal reuse behavior, i.e. blocks that were accessed more recently have a higher likelihood of being accessed again, the stack distance profile will show monotonically decreasing bars.

The last four profiles in the figure show applications with perfect LRU temporal reuse behavior (monotonically decreasing bars), while the first four profiles show applications with imperfect (or anti-) LRU temporal reuse behavior (bars may rise after they decline). The extent of anti-LRU temporal reuse behavior varies across applications. For example, omnetpp has less than 1% hits in the first 8 ways, but 57% hits in the next 24 ways. Other applications in the first four profiles show a mixture of LRU and anti-LRU behavior, generally with LRU behavior in the first few ways, but anti-LRU in latter ways.

In addition, the first five applications in the figure (xalancbmk, omnetpp, soplex, hmmer and bzip2) have a significant number of hits beyond the 8th (and even the 16th) way. These applications can benefit from extra cache space beyond a 512KB (and even beyond a 1MB) private L2 cache. Hence, we can categorize such applications as Spillers. On the other hand, as shown, other applications do not depict this behavior - namd, milc and povray indicate negligible hits beyond the 3rd, 2nd and 5th ways, respectively. These applications do not gain performance with more cache space, and may even donate some large portions of their cache space with almost no performance degradation. We can categorize such applications as Receivers. Thus, in general, *Spillers usually exhibit some anti-LRU temporal locality behavior, while Receivers usually exhibit perfect LRU temporal locality behavior*.

Note that if each core has a 512KB private L2 cache that is used to store all new incoming blocks (as in DSR), then the first 8 most recently used ways in the stack distance profile will correspond to the local 512KB cache. Because DSR uses remote caches to store victim blocks evicted from the local L2 cache, remote L2 caches strictly hold blocks that are farther away in the recency order than the local L2 cache. This implies that the 24 ways beyond the first 8 in the figure correspond to the total size of three remote L2 caches used to store victim blocks of the local L2 cache (assuming a 4-core CMP). Since DSR treats remote caches as a victim cache, it is guaranteed that blocks in the last 24 ways are going to be located in the remote L2 caches. This creates a situation in which, for Spillers, there is going to be a large number of remote hits.

A large number of remote L2 cache hits is costly from both a performance and an energy 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 energy due to the coherence request posted on the bus, snooping and tag checking activities in all remote caches, and data transfer across the bus. Thus, it is very important for performance and energy consumption reasons to maximize the local hits by converting remote hits to local hits. Remote hits, though worse in performance and energy consumption compared to local hits, are still better than misses. Thus to see improvement in performance and power over the current designs, it is important to increase the number of on-chip hits, and convert a larger fraction of such on-chip hits to local hits.

Unfortunately, with DSR, since all incoming blocks are always placed in the local L2 cache, anti-LRU temporal locality behavior common in Spillers cause these blocks not to be accessed during their lifetime in the local L2. The lines would have moved out to a remote L2 before they are accessed the second time. Therefore, *deciding the placement of blocks*

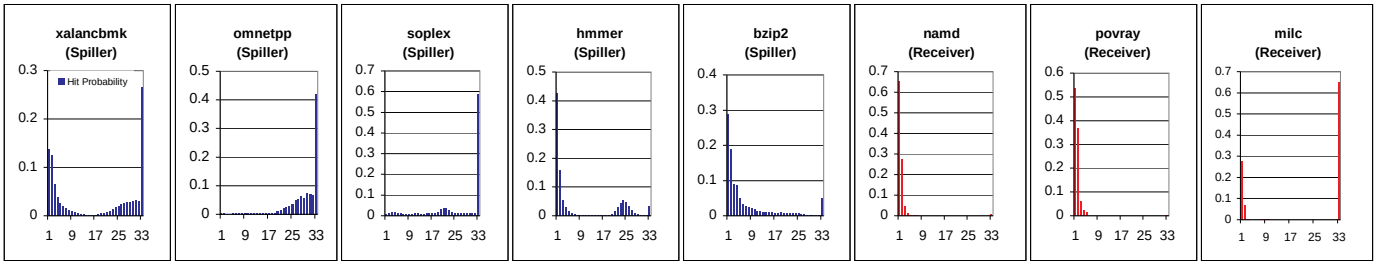

Fig. 1. Stack Distance Profiles for 8 SPEC CPU2006 applications: x-axis is the cache way number, y-axis is the percentage of hits seen by that way; 32 ways make up a 2MB cache; right-most bar displays the percentage of misses.

in local versus remote caches only when a block is evicted from the local L2 cache is inherently unable to increase the fraction of local hits on chip, especially for Spiller applications. In order to improve the number of local hits, we must *selectively* decide to place blocks locally versus remotely when they are brought into the L2 cache from the lower level memory hierarchy.

These observations motivate us to evaluate the potential benefit that may be expected from an optimal local versus remote block placement scheme. To this end, we study an oracle scheme, OPT. OPT uses knowledge of the future access made by the Spiller application to optimally decide whether to place an incoming line in the local or remote L2. Further, if the incoming line is to be placed locally, OPT optimally selects a replacement candidate in the local L2.

By studying OPT's behavior we found that, as expected, by placing blocks with a loose temporal locality in a remote L2 cache, the hit rates of the local L2 cache improves, thus leading to performance improvement.

Of the two orthogonal aspects that OPT optimizes - placement and replacement - replacement has been widely studied. The aspect of placement has not, to our knowledge, been studied in significant depth. Replacement schemes cannot identify that an incoming line should not be placed in the local L2. This work identifies placement as an equally important decision that must be considered by private cache controllers in order to approach the limits established by OPT.

IV. Design of OPT

We developed a hypothetical placement policy, OPT, which approaches the optimal performance that can be expected from the family of collaborative cache management policies for private, last-level, caches. OPT helps develop insights into the potential of placement policies in general, and also helps calibrate the performance of existing and future capacity sharing schemes.

The key insight that went into developing OPT is that for some Spiller applications the temporal locality patterns are such that it may be better not to disturb the existing population of a local L2 at placement time. The block is supplied to the L1 to exploit any immediate spatial and temporal locality in the application's request pattern. The L2 seeks to exploit locality that the L1 cannot exploit, but the L2 can. For some Spiller applications (such as those represented by omnetpp and soplex) temporal locality is sometimes spread out beyond what can be exploited by the local L2 cache. Placing such an incoming block into the local L2 only serves to remove existing, potentially more useful, cache blocks; by the time the installed cache block is requested again by the L1, it would have spilled over to a remote L2 anyway. Placing the block in the remote L2 directly can help improve the hit rate of the local L2, thus leading to performance improvement.

When a Spiller application suffers a cache miss, the OPT policy determines if the incoming block should be placed in the local L2 or in a remote (Receiver) L2 by looking at a trace of future accesses made by the Spiller application. This trace is generated prior to simulation run time; this technique is similar to Belady's optimal replacement algorithm [11]. All the blocks in the cache set involved in the access and the missed block itself are compared against the trace to find the one that is touched farthest in the future; this block is selected to be the victim and is placed remotely in a Receiver L2. Note that this implies that the cache block corresponding to the incoming miss may itself be relegated to a remote L2 without ever placing the block in the local L2.

We assume that the local L2 does not enforce the inclusion property with the L1 cache. Not enforcing the inclusion property in general has the benefit of increasing the aggregate cache capacity of multiple levels of cache, at the expense of possibly having to duplicate the L1 cache tag in order to avoid contention-related stalls when the L1 cache tag is accessed by both the core and the L2 cache snooper. On a writeback from the L1 cache, we assume that the data is merged with the block if it exists on-chip in an L2 cache (remember that the block may exist in a remote cache). If the block is not found on-chip, a new block is allocated in the local L2 cache.

Irrespective of which cache block is placed in the remote L2, the placement location at the remote L2 must be considered. Choosing a remote placement location is done in two steps. First, the LRU candidate in a randomly selected Receiver L2 is evaluated. If it belongs to a Receiver application, it is considered optimal from the Spiller application's point of view. It is evicted (or simply invalidated, as appropriate), and its location selected as the remote placement location. If the selected LRU cache block belongs to any Spiller application, the future access trace for *that* Spiller is consulted. Of all the cache blocks belonging to that Spiller application in *all* the

Receiver caches, the one that is accessed farthest in the future is selected as the eviction candidate.

The rationale behind this two-step scheme is to discover an optimal replacement candidate from the Spiller's point of view; the optimality of OPT is spiller-centric. If the LRU candidate in a selected Receiver L2 belongs to the Receiver application, that is ideal from the Spiller application's perspective since none of its on-chip blocks will be lost by that selection. If the LRU block belongs to this or another Spiller application, we need to identify the block least likely to be used by that Spiller application in order to least hurt it. Remember that OPT is a hypothetical scheme, and is not intended to be implemented by real hardware.

A cache hit in a remote L2 also requires a placement decision. When a Spiller application has a hit in a remote L2, the future access trace for that Spiller application is consulted. The block to be used the farthest in the future among the blocks in the Spiller's local L2's cache set and the remote hit cache block is identified. The identified cache block is swapped with the cache block that hits in the remote cache. Note that if the block that hits remotely also happens to be the block to be used the farthest in the future, the block is left unmoved in the remote cache. The block is, as always, supplied to the processor making the request and its L1 cache.

V. Methodology

System Configuration. We use a cycle-accurate multicore simulator based on Simics [12], a full system simulation infrastructure. Table I lists all relevant configuration parameters used in our experiments. Throughout the evaluations, the default configuration is a 4-core CMP with four 512KB, 8-way, private L2 caches. We use in-order processors attached with the 'g-cache in-order' Simics module. We assume the cores' private L2 caches are interconnected with a shared bus and kept coherent using a MESI broadcast/snoopy coherence protocol.

TABLE I
System Configuration

Cores	3GHz, 4cores, in-order single issue
Block Size	64B
L1 I/D caches	32KB, 2-way, 2-cycle access latency
L2 caches	512KB, 8-way, 7-cycle local hit 50-cycle remote hit LRU replacement policy
Memory	250-cycle access latency

Multi-Programmed Workloads. We characterized all C/C++ SPEC CPU2006 applications based on whether they can benefit from a significant (at least 15%) increase in hit rates if the L2 cache capacity is increased from 512KB to 2MB. If they can, we categorize them as Spillers, otherwise we categorize them as Receivers. Spillers display three distinct temporal locality patterns. Some Spillers show a perfect-LRU temporal locality behavior where more recently used blocks have a greater reuse probability than less recently used blocks. Some Spillers have an anti-LRU behavior, where recency of use does not correlate with reuse potential. Some Spillers show both characteristics - they have perfect-LRU temporal locality for all blocks for smaller cache sizes; with larger caches however, a significant proportion of cache blocks display anti-LRU temporal locality tendencies. Figure 1 in section III shows applications with each of these three temporal locality behaviors. Receivers, on the other hand, cannot benefit from an increase in the cache space from 512KB to 2MB. This is either because they have small working sets that fit in a 512KB 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 Receivers further into two groups.

The applications and the category they fall into are shown in Table II. The first three rows contain all applications we characterized as Spillers in SPEC CPU2006. The next two rows contain a subset of the Receivers in SPEC CPU2006. We chose only a subset of all Receivers because we observed that Receivers behave similarly from the point of view of the insights we hope to gain from this study. All Receivers with a small working set share the same trend (miss ratio is less than 3%); therefore, we use a small representative set for such Receivers. Similarly, Receivers with streaming accesses behave similarly. We used a single application to represent Receivers which have streaming data access patterns, since such applications are a minority among Receiver in SPEC CPU2006.

TABLE II
Spiller and Receiver applications

Spillers (anti-LRU)	omnetpp, soplex
Spillers (perf-anti-LRU)	xalancbmk, hmmer
Spillers (perf LRU)	bzip2
Receivers (Small WS)	namd, povray
Receivers (Streaming)	milc

Spillers and Receivers can be identified at run time using existing hardware techniques, such as, Set Dueling [5] and Shadow Tags [13]. In particular, *Receivers* are easy to identify because they are bimodal: either little/no temporal reuse (streaming), or tight temporal reuse. *Spillers*, on the other hand, can be identified by keeping shadow tags that estimate miss rates in larger cache scenarios. Shadow tags can be kept small through sampling only a subset of cache sets [14]. Such dynamic *Spiller/Receiver* identification is orthogonal to the aspect of cache management we study; OPT can still be applied on top of such mechanisms.

We construct ten multiprogrammed workloads, each consisting of four SPEC CPU2006 benchmarks. The application mixes have been carefully chosen to study a wide range of multi-programmed workload scenarios possible where both Spillers and Receivers exist, and are present in varying proportions. Table III shows the workloads; Spillers are shown in bold font. The first five application mixes have 1 Spiller and 3 Receivers. The next three application mixes have 2 Spillers and 2 Receivers. The last two application mixes have 3 Spillers and

1 Receiver. The workloads are numbered as well as denoted with S*x*R*y* to express x Spillers and y Receivers.

TABLE III
WORKLOADS (SPILLERS SHOWN IN BOLD TEXT)

Mix	Applications in the mix
M01_S1.R3	**omnetpp**-namd-povray-milc
M02_S1.R3	**xalancbmk**-namd-povray-milc
M03_S1.R3	**soplex**-namd-povray-milc
M04_S1.R3	**hmmer**-namd-povray-milc
M05_S1.R3	**bzip2**-namd-povray-milc
M06_S2.R2	**omnetpp-xalanckbmk**-namd-milc
M07_S2.R2	**omnetpp-bzip2**-namd-milc
M08_S2.R2	**xalanckbmk-bzip2**-namd-milc
M09_S3.R1	**omnetpp-xalancbmk-bzip2**-namd
M10_S3.R1	**omnetpp-xalancbmk-bzip2**-milc

For the workloads with one Spiller application, M01 to M05, we pick a different Spiller for each workloads, while keeping the Receivers unchanged. This helps highlight the differences between the various Spillers with respect to our proposed placement scheme. By increasing the number of Spillers to 2(M06,M07 and M08) and 3(M09 and M10), we can assess the sensitivity of our proposed placement scheme to the increasing competition among the Spillers for the reducing amount of excess cache capacity in the Receivers' caches. For these mixes, we pick one application from each spiller sub-category, namely omnetpp (anti-LRU), xalancbmk (perf-anti-LRU), and bzip2 (perf-LRU). We selected workloads which help exemplify the essential characteristic of each of the Spiller sub-categories.

We formed all possible combinations of these three Spillers. As can be seen, bzip2 appears in almost all combinations. Since bzip2 is a Spiller which does not exhibit an anti-LRU behavior at all, it is therefore not expected to benefit as much from OPT. Such mixes help assess if the overall performance suffers when Spillers participating in OPT cannot benefit from it.

To test each workload, all four applications in the workload are fast forwarded for 10 billion cycles (which is equal to 10 billion instructions in the fast mode of Simics). The cache models are then attached to the running simulation and detailed timing simulation is enabled. Each cache is warmed for 1 billion cycles. Finally, the workloads are run for 1 billion more cycles during which the simulation statistics are collected. All the applications use the reference input set.

VI. RESULTS AND ANALYSIS

In this section, we evaluate the performance of OPT, and compare it to the base case of private caches with no capacity sharing (Base), and, Dynamic Spill Receive (DSR).

We use Weighted Speedup to evaluate performance and fairness of the 10 multi-programmed workloads. This metric and its significance has been described in detail in prior work [15]. Here is the definition of weighted speedup of n applications:

$$\sum_{i=0}^{n-1}(IPC\_i_{new}/IPC\_i_{base})$$

Weighted speedup is a preferred metric because (1) it contains a notion of fairness, in which speeding up one application at the expense of others has an offsetting effect, and (2) weighted speedup corresponds to a real system-level meaning of the number of jobs completed per unit time [15].

OPT is spiller-centric, as we saw in Section IV. We find, however, that Receivers do not suffer significantly due to this; the worst case increase in Receiver miss-rate was ¡5% across all the workload mixes we studied. Receiver applications are resilient in retaining their requisite working set in the cache, by keeping those blocks warm. This is attributable to the tight temporal locality displayed by most Receivers. Receivers with streaming behavior, do not suffer a decrease in performance because they do not have much of a temporal locality to begin with.

A. Performance of OPT

Figure 2 shows the weighted speed up for DSR and OPT for various workloads, normalized to the base case of private caches without capacity sharing.

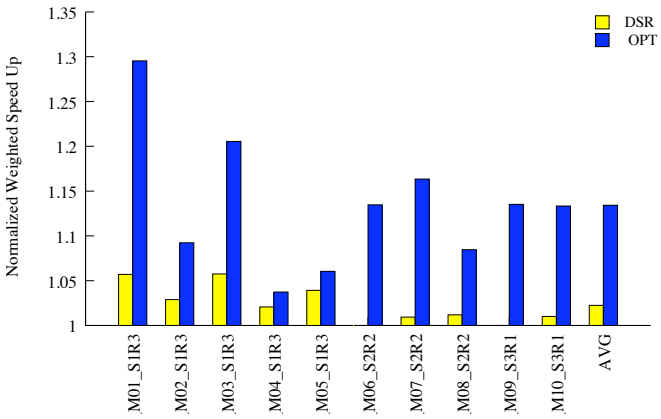

Fig. 2. Weighted speedup improvement over private caches without capacity sharing.

There are several observations that can be made from the figure. First, both schemes improve the weighted speedup of all workloads compared to the base case. There is, however, a wide gap between the weighted speedup improvement of the two schemes. On average, DSR improves the weighted speedup by 2.2%, while OPT improves the weighted speedup by 13.4%. Note that the averages reported in this work are not intended to be average SPEC behavior; rather, these are averages across the workload mixes we study, with varying ratios of Spiller and Receiver applications.

The magnitude of weighted speedup improvements vary across workloads. In all ten workloads OPT outperforms DSR, significantly so in eight of the ten (all except M04 and M05) workloads. Three of the five Spiller applications show an anti-LRU reuse behavior for a significant fraction of the cache blocks. Applying OPT allows selectively placing anti-LRU incoming blocks in remote caches, leaving existing blocks in the local L2 cache, with potentially tighter reuse patterns,

undisturbed. This allows OPT to improve the local L2 hit rates, and therefore, performance, compared to DSR. As the number of Spillers increase in the double-spiller (M06,M07 and M08) and triple-spiller (M09, M10) workloads, the excess capacity in Receivers' L2 caches available for Spillers decreases. Hence, in general, there is less performance benefit that can be provided by capacity sharing, as shown in general by OPT's declining improvements as we go to the right along the x-axis on the figure. It is even more important to utilize the excess cache capacity more efficiently, however, because such capacity is rarer.

There are two workloads in which OPT does not perform significantly better than DSR. In M04 workload, the Spiller application is hmmer, which has a mixture of LRU and anti-LRU temporal reuse behavior (as shown in Figure 1). With 3 Receiver caches, most of the hmmer's working set fits on-chip. However, because of the strong LRU behavior of blocks in the first few most recently used ways (Figure 1), most of the performance improvement comes from additional cache capacity rather than from an increase in local L2 cache hit rate.

In M05 workload, the Spiller application is bzip2, which has an almost perfect LRU behavior (Figure 1). Therefore, the best policy is to always install an incoming block locally, which is already achieved by DSR. The improvement OPT shows over M04 and M05 is due to the optimal *replacement* that OPT achieves.

B. Performance Analysis of Spillers in OPT

Figure 3 shows the *local cache miss rate* (number of misses to the local L2 cache divided by the total number of L2 cache accesses) experienced by the Spiller applications in each mix. The first 5 workloads are single-spiller, and therefore each workload has one cluster of bars corresponding to the Spiller. For two- and three-spiller workloads, there are two and three clusters of bars for each workload, corresponding to the Spillers.

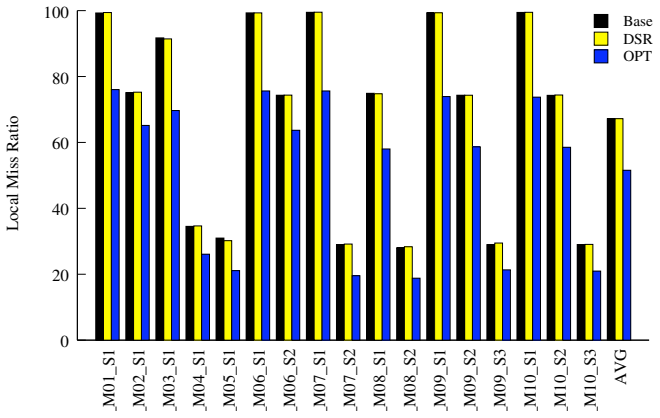

Fig. 3. Local L2 miss rate comparison.

We can make several observations from the figure. First, it confirms that DSR and the Base case experience the same local L2 miss rates, which is expected because both these policies always bring a block requested by the Spiller application into the local cache. The only difference between the base case and DSR is that in DSR the block may be found in a remote cache, whereas without capacity sharing, such a remote hit in the base case will simply be a cache miss. Thus, Spiller performance improvement in DSR solely comes from exploiting excess cache capacity to change some cache misses into remote cache hits. The average local miss rate suffered by the baseline scheme and DSR is 67%; OPT is able to reduce that to average of 52%.

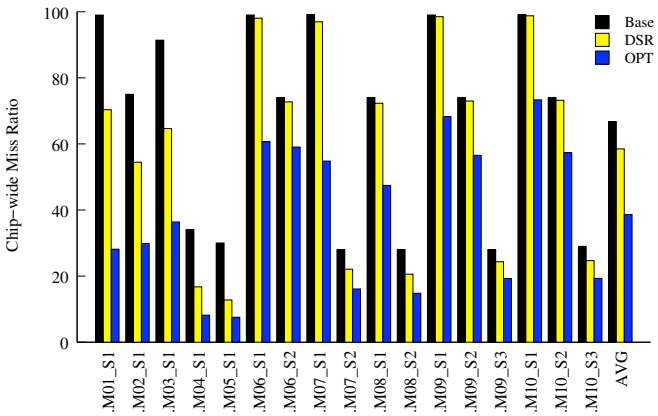

Fig. 4. Chip-wide L2 miss rate comparison.

Interestingly, OPT outperforms DSR not just because of the improvement in local L2 hit rates. Figure 4 shows the *chip-wide L2 miss rate* (the total number of L2 misses that cannot be satisfied by the local and remote caches, divided by all L2 cache accesses) for all Spillers in various workloads. The figure shows that due to capacity sharing, DSR reduces the chip-wide miss rates significantly over the Base case, from 67% to 58% on average. However, OPT reduces the chip-wide miss rates even more across all the workloads studied, to an average of 38%. This may be counter intuitive at first sight, as all OPT does is better placement of blocks in local versus remote caches without increasing the excess capacity available for Spillers to take advantage of. There is a subtle, yet powerful, effect of OPT's placement decisions. Blocks with perfect-LRU locality behavior are placed in the local cache. Blocks with anti-LRU locality behavior are placed in the remote cache. Without a scheme like OPT, anti-LRU blocks are also placed locally, thereby disrupting the perfect-LRU blocks in the local L2. The perfect-LRU blocks get evicted and get placed remotely. This hurts the local hit rate. Moreover, when placed remotely, perfect-LRU blocks disrupt the anti-LRU blocks in the remote cache, potentially replacing them out of the system and hurting remote hit rate as well. By correctly predicting locality behavior of cache blocks, OPT successfully places a block at the best position and in doing so avoids movement of blocks to locations sub-optimal given their locality pattern. Avoiding spills of perfect-LRU lines from the local cache, preserves the anti-LRU lines longer in

the remote caches, thus improving both local and remote hit rates. This leads to a chip-wide reduction in miss rate.

Overall, OPT outperforms schemes, which consider remote placement only at replacement time, because it improves the local miss rate (more blocks are found in the local L2 cache) as well as in the chip-wide miss rate (more blocks are found in the L2 caches on-chip).

C. Sensitivity to Cache Size

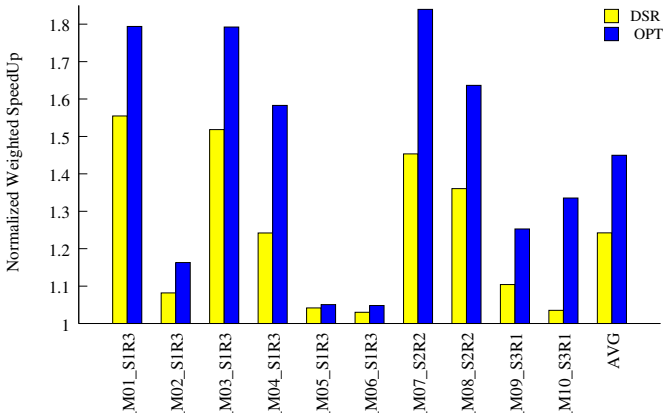

Fig. 5. Weighted speedup improvements for 1MB L2 Cache private caches, for DSR and OPT, both compared to the base case of no private caches.

All results presented so far assume that each private L2 cache has a 512KB size. In order to study how the benefit of OPT changes with cache size, we study cache management for a 4-core system with 1MB, 8-way, private L2 caches.

Figure 5 shows the Weighted Speedup improvements for DSR and OPT, for 1MB L2 private caches. It is clear from the figure that although DSR can acheive an average of 24% improvement in Weighted Speedup, it is still far from reaching OPT's 48%. That indicates that a significant potential for improving private cache management schemes exists even with larger cache sizes.

VII. Future Work

We plan to develop a hardware implementable, low-cost, and hardware efficient approximation to OPT which captures the benefits of selective placement of data to match a block's on-chip location with its temporal locality. We hope to substantially fill the existing gap between current private cache management approaches and the potential shown by this work.

VIII. Conclusions

In this work we have sought to answer the following questions - (1) what is the best performance that can be expected from a private L2 cache management scheme, and (2) what is the impact of the local vs remote placement decision on the overall performance of cache management schemes. In this work we have developed an optimal cache management scheme, OPT, for private L2 caches in a CMP running a multi-programmed mix of sequential workloads. We find that there is still significant scope for performance improvement of cache management schemes implemented for private caches; OPT improves weighted-speedup by 13.4% over the baseline of non-collaborative private caches and by 11.2% over the state of the art private cache management scheme. We identify that cache placement decisions play a significant role in ensuring that private cache space is shared effectively by applications.

References

[1] R. Kumar, V. Zyuban, and D. M. Tullsen, "Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling," *Computer Architecture, International Symposium on*, vol. 0, pp. 408–419, 2005.

[2] J. Chang and G. S. Sohi, "Cooperative caching for chip multiprocessors," in *Computer Architecture, 2006. ISCA '06. 33rd International Symposium on*, 2006, pp. 264–276. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2006.17

[3] M. Qureshi, "Adaptive spill-receive for robust high-performance caching in cmps," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, Feb. 2009, pp. 45–54.

[4] Z. Chishti, M. D. Powell, and T. N. Vijaykumar., "Optimizing replication, communication and capacity allocation in cmps,," in *In the 32th ISCA*, June 2005, pp. 357–368.

[5] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2007, pp. 381–391.

[6] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer, "Adaptive insertion policies for managing shared caches," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2008, pp. 208–219.

[7] M. Chaudhuri, "Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches," in *MICRO*, 2009.

[8] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 336–345.

[9] B. M. Beckmann, M. R. Marty, and D. A. Wood, "Asr: Adaptive selective replication for cmp caches," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 443–454.

[10] J. Lee, Y. Solihin, and J. Torrellas, "Automatically mapping code on an intelligent memory architecture," in *In 7th Intl. Symp. on High Performance Computer Architecture*, 2001.

[11] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5(2), pp. 78–101, 1966.

[12] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[13] J. Jeong and M. Dubois, "Cost-sensitive cache replacement algorithms," in *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 327+.

[14] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell, "Cachescouts: Fine-grain monitoring of shared caches in cmp platforms," *Parallel Architectures and Compilation Techniques, International Conference on*, vol. 0, pp. 339–352, 2007.

[15] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.