Cooling the Hot Sets: Improved Space Utilization in Large Caches via Dynamic Set Balancing

Mainak Chaudhuri
Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur 208016
INDIA
mainakc@cse.iitk.ac.in

30th November, 2008

Abstract

Multi-megabyte on-chip last-level caches are commonplace in high-end computing platforms. Even though these caches are often designed to have very high associativity, they suffer from non-uniform utilization of the sets leading to a high volume of conflict misses. Clustering of physical addresses to a few hot sets happens partly due to poor locality in the access stream and partly due to a mismatch in the access pattern and the virtual address to physical address translation algorithm. In this paper, we propose the first fully dynamic mechanism to improve the utilization of sets by adaptively migrating data blocks from hot sets to the relatively cold sets. We present robust and scalable algorithms for identifying the hot sets and suitable cold sets for holding the migrated data blocks that flow from the hot regions. We discuss a number of optimizations on the basic design to address different aspects of such a mechanism, thereby progressively improving the performance. Our detailed execution-driven simulation results show that with just 5.5% extra book-keeping overhead in a 2 MB 16-way set-associative L2 cache, the dynamic block migration mechanism reduces execution time by 12% on average (geometric mean) for nine memory-intensive applications selected from the SPEC 2000 and SPEC 2006 benchmark suites. Further, when applied to an eight-core chip-multiprocessor with a shared 4 MB 16-way set-associative L2 cache, our technique reduces execution time by 18.1% on average (geometric mean) for a set of multi-threaded kernels and applications. We also present a thorough energy analysis of the proposed cache architecture and a quantitative evaluation of how it interacts with an aggressive multi-stream stride prefetcher.

1. Introduction

The recent high-end computing chips have housed highly associative multi-megabyte last-level caches with the hope of bridging the latency gap between the on-chip caches and the off-chip DRAM modules. However, poor locality in the access stream of applications often leads to clustering of physical addresses to a few sets in the last-level cache leading to a high volume of conflict misses. The situation is often worsened by the fact that the physical addresses are assigned to virtual pages on a demand basis when the page is accessed for the first time without having much knowledge about the set of pages that will be accessed together during the subsequent phases of execution. The end-result is an uneven distribution of addresses to the cache sets and severe under-utilization of the silicon estate devoted to the last level of the on-chip cache hierarchy.

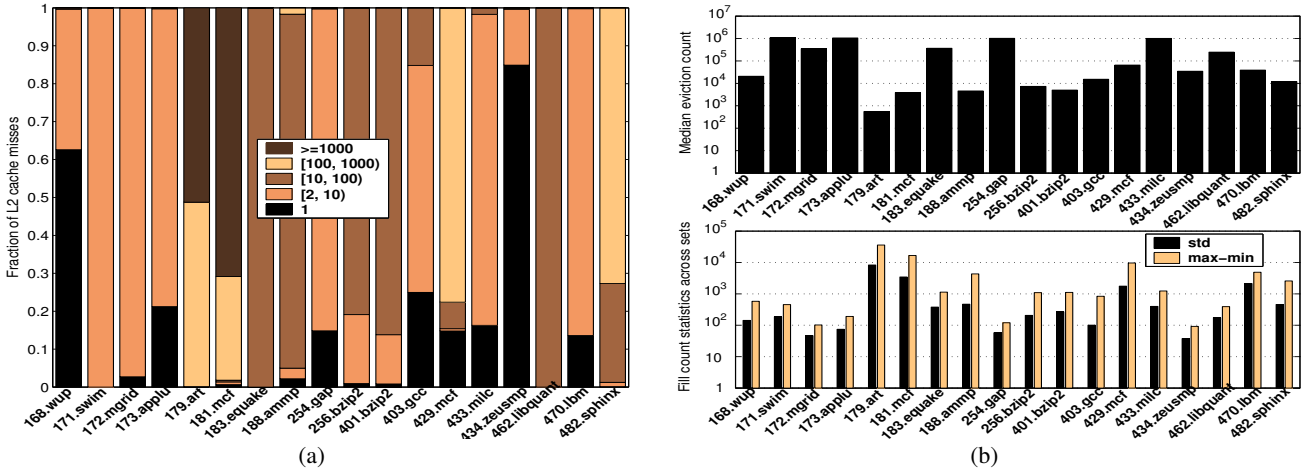

Figure 1. (a) Distribution of cache block addresses in the L2 cache miss stream, (b) Top panel: Median of the number of L2 cache evictions between the eviction and the reuse of a particular L2 cache block; Bottom panel: Distribution statistics of the number of refills (same as the number of misses) across the L2 cache sets. We have taken the liberty of shortening “wupwise” to “wup”, “libquantum” to “libquant”, and “sphinx3” to “sphinx”.

To re-assert the severity of conflict misses, in Figure 1(a), we present the distribution of cache block addresses in the L2 cache miss stream for 18 applications with at least 0.5 misses per kilo instructions (MPKI) chosen from the SPEC 2000 and SPEC 2006 suites running on a 2 MB 16-way set-associative L2 cache having 128-byte blocks and exercising LRU replacement policy. Each application is run for a representative sample of one billion dynamic instructions. Virtual to physical address translations are carried out by implementing the demand-based bin-hopping algorithm, which is known to be one of the cache-friendly schemes [12].[1] In this figure, we classify each missing L2 cache block address into one of the five categories depending on how many times that address has

[1] More details of the simulation environment will be discussed in Section 3.

appeared in the miss stream. For example, in 179.art, block addresses repeated hundred to thousand times in the miss stream constitute about 49% of all the L2 cache misses while those repeated thousand or more times account for the remaining 51% of misses. In 181.mcf, the situation is even worse. In this case about 71% of the L2 cache misses come from addresses repeated thousand or more times. These two applications suffer most due to conflict misses. On the other hand, 434.zeusmp has about 85% of L2 cache misses covered by blocks appearing only once in the miss stream. Similar behavior, at a slightly smaller scale, is exhibited by 168.wupwise. Such applications are unlikely to observe improvements due to techniques that attempt to resolve conflict misses.

While Figure 1(a) helps us identify the probable candidate applications that may benefit from conflict resolution techniques (179.art, 181.mcf, 429.mcf, and 482.sphinx3 among the top few), it is also important to know how far apart the reappearance of a block address in the miss stream from its last eviction is. The upper panel of Figure 1(b) shows the median of the number of evictions between the eviction and the next miss of an L2 cache block address (the y-axis is in log scale).[2] This number ranges from 551 in 179.art to 1067142 in 171.swim. For all applications, except 179.art, this number is bigger than thousand, and for 13 out of the 18 selected applications, this is bigger than ten thousand. This data clearly brings out the difficulty of any technique that tries to retain the repeatedly missing addresses. A conventional fully associative victim cache of reasonable size is bound to fail. We will return to this particular aspect in Section 2.2.3 when presenting our technique.

Finally, the lower panel of Figure 1(b) presents the root cause of the conflict misses. For each application, we show two pieces of statistics (the y-axis is in log scale). The left bar shows the standard deviation of the number of L2 cache block refills (same as the number of misses) across the cache sets. The right bar shows the difference in the number of refills between the hottest (having the most number of refills) and the coldest (having the least number of refills) sets. This data clearly explains the behavior depicted in Figure 1(a). For example, 179.art, 181.mcf, 188.ammp, 429.mcf, 470.lbm, and 482.sphinx3 are among the top few applications that suffer most due to uneven distribution of addresses to cache sets. While 470.lbm does not have as much of block address repetition as 179.art or 181.mcf has in the L2 cache miss stream, it has significant non-uniformity in its refill distribution across the cache sets.

There is a significant body of research that has tried to improve the conflict behavior of caches. We discuss some of these in Section 1.1. In this paper, we present the first fully dynamic design that adaptively migrates L2 cache blocks from hot sets to the cold ones so that the overall retention of the cache improves significantly. We progressively present refinements on the basic design so that a designer has several options to choose from. The

[2] Blocks appearing in the miss stream only once are not considered in this statistic.

basic design involves an algorithm that identifies the cold sets (through a meaningful metric of "coldness") in the L2 cache and every replaced block is migrated to a suitably chosen cold set. However, a block replaced due to a migration is not migrated again. A migration tag cache (MTC), separate from the main cache, keeps track of the migrated blocks. We find that having only half the tags in the MTC compared to the main cache is enough to realize the full potential of conflict resolution via block migration. The most attractive aspect of this design is that the size of the MTC offers an important tuning parameter and helps the designer trade performance for area accurately. We build upon this basic design and propose three important optimizations. First, we propose an algorithm to track the hot sets and selectively migrate the blocks replaced from the hot sets only. Second, we implement an algorithm to impose a dynamic limit on the number of blocks that can be migrated from a single set. This limit is increased dynamically in steps only when our algorithm senses that the opportunity lost has crossed a threshold. Third, we propose a new replacement policy for our cache architecture so that the retention of the migrated blocks is optimized. This is important because just migrating the hot blocks is not enough; they have to be retained in the cache also. We discuss the basic design and the optimizations in Section 2.

The highlights of our simulation results (Sections 3 and 4) include an average 12% reduction in execution time for nine memory-intensive applications (with more than four MPKI) chosen from the SPEC 2000 and SPEC 2006 suites running on a 2 MB 16-way set-associative L2 cache. The improvement for multi-threaded workloads is even more encouraging. On an eight-core chip-multiprocessor (CMP) with a 4 MB 16-way set-associative shared L2 cache, a set of multi-threaded kernels and applications selected from the SPLASH-2, SPEC OMP, and FFTW suites exhibits an average 18.1% reduction in execution time. Further, our technique convincingly outperforms the related proposals such as a reasonably sized fully associative victim cache [11], XOR indexing [8], prime displacement and prime modulus indexing [13], V-way cache [18], dynamic insertion policy [19], and shepherd cache [20]. This excellent performance is achieved at the cost of only 5.5% extra storage overhead for book-keeping. We also present a thorough dynamic as well as leakage energy analysis of the proposed cache architecture and quantitatively evaluate how our technique interacts with an aggressive multi-stream stride prefetcher.

1.1. Related Work

There is an impressive body of research attempting to improve the cache performance in all types of processors. Since it is impossible to do justice to this large body of research in a brief note like this, in the following we discuss some of the studies most relevant to our proposal. We divide the discussion into three parts respectively addressing cache indexing schemes, new cache organizations, and cache replacement policies proposed to reduce the volume

of conflict misses.

Among the cache indexing schemes, the hash-rehash algorithm [1] and the column-associative algorithm [2] were originally designed for direct-mapped caches. The hash-rehash scheme uses two indexing functions to access the cache. If the first access misses, the alternate function is used to access a different set. The column-associative cache amends a serious performance pathology of the hash-rehash scheme which could cause significant amount of thrashing. Overall, under ideal conditions, the column-associative cache can offer performance equivalent to a two-way set-associative cache.

A generalization of the idea of using multiple indexing functions is presented in [23] in the form of skew-associative caches. The skew-associative cache partitions the entire cache into banks and accesses each bank with a different indexing function. The indexing functions are designed in such a way that the addresses that map to the same set in one bank are most likely to map to different sets in other banks. The proposed indexing functions are XOR-based hash functions involving a circularly shifted version of the tag and the normal index. The original proposal was for a two-way banked cache, but the central idea can be extended to more banks by properly designing the index functions.

An excellent treatment of a range of XOR-based indexing schemes can be found in [8]. We choose to compare our proposal against a simple scheme described in [8]. If a block address is $A[n-1:0]$ (obtained after removing the block offset bits), the traditional cache index is $A[s-1:0]$ where the number of sets is 2^s. The XOR scheme that we compare our proposal with uses $A[2s-1:s]$ XOR $A[s-1:0]$ as the cache index.

The possibility of using a prime modulus (as opposed to a power of two) for computing the cache index has been explored in [13]. In the simplest form, the prime number p closest to the number of sets in the cache is chosen and the cache index for block address A is computed as $A \bmod p$. Note that in this case the tag may have to be extended by a few bits depending on the choice of p and the availability of a divider for computing the exact tag. The authors also propose a prime displacement scheme, which computes the cache index as $(c.A[n-1:s] + A[s-1:0]) \bmod p$, where c is an appropriately chosen multiplication factor applied to the tag. We compare our proposal against both simple prime modulus and prime displacement schemes.

Perhaps the most notable contribution in the category of new cache organizations comes from the proposal of fully associative small victim caches [11]. However, such caches work best with direct-mapped or small-associativity caches. We have already indicated in Figure 1 that such a cache may not be very useful for the large last-level caches with high associativity. Nonetheless, we will compare our proposal with a reasonably large fully associative victim cache. A recent proposal [3] shows how to design very large victim caches that are organized

as direct-mapped arrays but are functionally equivalent to fully associative caches. That proposal also shows how to identify the most frequently missing blocks dynamically so that they can be retained in this large victim cache.

The indirect index cache organization [9] involves decoupled tag and data stores and a generational global replacement policy. The tags are organized into a primary 4-way set-associative hash table and a secondary direct-mapped table. On a primary table miss, the secondary table is used to walk the hash collision chain. The generational replacement policy maintains multiple queues of tags. If a tag is not accessed for a long period of time, it is gradually demoted to the lowest priority queue. A replacement tag is always selected from the lowest priority queue.

The V-way cache [18] doubles the tag store compared to a conventional set associative cache while keeping the decoupled data store size unchanged. As a result, the tag store gains one extra index bit allowing the conflicting tags to get distributed over two sets. Each tag maintains a pointer to the corresponding data block. Since the number of tags is twice that of the data blocks, within a set with a very high probability an invalid tag can be found. When searching for an invalid tag, the V-way cache implements a reuse-based global data block replacement policy where the block with the minimum reuse count is found by sequentially examining the reuse counters. The authors impose an upper bound of five cycles for this search. We compare our proposal with the V-way cache.

To resolve conflict misses in large direct-mapped caches, OS-assisted dynamic page remapping is explored in [4]. The pages suffering from a high volume of conflicts are detected dynamically and remapped subsequently by the OS. The group-associative cache presented in [17] shares some similarities with our proposal. The authors propose to keep track of a few MRU blocks in a direct-mapped L1 cache. When one such block is replaced, it is moved to an alternate set so that the block can be retained for some more time. However, locating a suitable alternate set requires a localized search. Our proposal has important differences with this design. First, MRU blocks may not be good migration candidates in a highly associative L2 cache. Second, our proposal does not require any search operation for locating the "cold" sets. Finally and most importantly, we show that retaining a migrated block for a sufficiently long time is absolutely necessary to enjoy the full benefit of block migration in a large highly associative L2 cache.

Among the recent replacement policy proposals, the adaptive cache [25] proposes to dynamically pick the better of the two replacement policies (e.g., LRU and LFU) on a per-set basis. However, the design requires two auxiliary partial tag arrays to keep track of the performance of the two competing policies. Dynamic insertion policy (DIP) [19] improves cache performance by deciding where to insert a newly allocated block. In traditional LRU algorithms, a new block is always made the MRU within a set. DIP explores the potential of dynamically

choosing between the LRU and MRU positions for a newly incoming block. A block inserted in the LRU position is promoted to the MRU position only after it is accessed once more. If the working set is much larger than the cache and exhibits a cyclic access pattern, such a policy succeeds in retaining some part of the working set in the cache. Finally, the shepherd cache [20] dedicates a few of the main cache ways to buffer newly incoming blocks and observe the "future" references with respect to the misses that brought these blocks. These dedicated ways are referred to as the shepherd ways. When a block must be replaced from the shepherd ways (uses a FIFO replacement policy), it is moved to a non-shepherd way by replacing the block with the least imminent access within the shepherding window of the shepherd block that is being moved. Thus the shepherd ways help emulate the optimal replacement within the non-shepherd ways. We compare our proposal with DIP and the shepherd cache.

2. Conflict Resolution via Block Migration

This section presents the proposed cache architecture. First we discuss the basic design and then present a few performance optimizations.

2.1. The Basic Design

The basic design concept was introduced in Section 1: every data block replaced from the L2 cache is migrated to a set "colder" than the parent set (the original set holding the replaced block will be referred to as the parent set of that block). Such a design involves two major components. First, on every replacement a simple hardware should help identify a suitable set where the replaced block can be migrated to. This will be referred to as the destination of migration. Second, auxiliary storage and logic should be in place to locate a block after it is migrated. We discuss these two aspects in the following.

2.1.1 Finding the Destination of Migration

The number of blocks filled into the sets in the L2 cache is used as a metric to rank them according to their congestion. A saturating counter is associated with each set. This counter, referred to as the occupancy counter, is incremented whenever a block is filled into the set either due to an external refill or due to an internal migration. As soon as the counter reaches a value equal to the associativity of the cache, a global counter is incremented. Note that the width of the occupancy counter is chosen such that it can count up to four times the associativity

minus one. When the global counter reaches a value equal to the number of sets in the cache, the entire occupancy counter RAM is reset.

The sets in the cache are divided into smaller clusters of 16 consecutive sets each. For each cluster we maintain a coarse occupancy counter which is incremented whenever any of the occupancy counters of the cluster is incremented. The coarse occupancy counter RAM is reset when the occupancy counter RAM is reset. We assume that the entire L2 cache is divided into smaller banks. In this paper, the bank size is assumed to be 1 MB. Whenever an occupancy counter is incremented, a comparator tree within that bank first computes the new minimum among the coarse occupancy counters. In our design, this comparator tree has 31 comparators organized in a depth-five tree. In a 2 MB cache, there are two banks and a combining comparator computes the minimum among each bank's minimum. In Section 2.3, we will discuss how to scale this design to larger caches. Finally, a comparator tree of depth four computes the minimum among the occupancy counters belonging to the set cluster with minimum coarse occupancy count. It is important to note that the entire latency of these comparator trees can be comfortably hidden under the tag array and data array access required for filling the block (it is common to have serial tag and data access in large L2 caches for saving dynamic energy). The end-result of this hardware is that at the time of every replacement, the set with minimum occupancy count is known. The ties can be broken arbitrarily.

On a replacement from the L2 cache, the minimum occupancy count (as determined above) is compared against the occupancy count of the parent set. If the parent set has a higher occupancy count, the replaced block is flagged as a candidate for migration to the set with minimum occupancy count. Within this target set, the baseline replacement policy (LRU in this paper) is applied to determine the way where the migrated block will be filled. The secondary replacement originating from the target set (to make room for the migrated block) is not migrated recursively and is handled by the baseline replacement protocol.

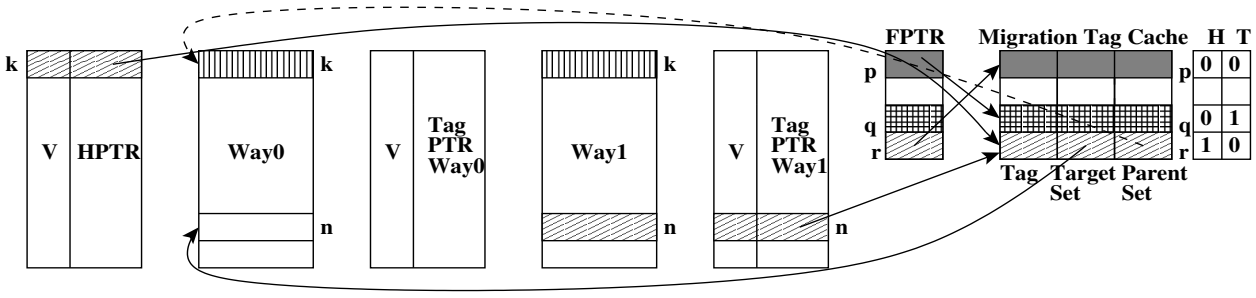

Figure 2. A logical description of the basic design.

2.1.2 Locating the Migrated Blocks

Figure 2 shows the basic storage and logic required for locating a block in the proposed cache architecture. For brevity, in this figure, we show the necessary storage and logic for a baseline cache with two ways (marked Way0 and Way1). The first migrated block from a set initiates a tag list in the migration tag cache (MTC), which is organized as a direct-mapped array. Each set maintains a head pointer (HPTR) and a head pointer valid bit. This pointer indicates the position of the first migrated tag in the MTC for that set. For example, in the figure, the tag of the first migrated block from set k is located at entry r of the MTC. The position r is recorded in the HPTR entry of set k. Each migration tag cache entry contains six logical fields, namely, the tag itself, the parent set index, the target set index where the migrated data block can be found, a forward pointer (FPTR) to the next entry in the migration tag list, a head bit (H), and a tail bit (T). The FPTR and the HT RAMs are shown physically separate from the MTC in the figure. The target set field of an MTC entry is slightly bigger than the parent set field because the former also includes the target way where the block can be found. More specifically, in an A-way L2 cache, the target set field is extended by $\lceil \log_2(A) \rceil$ bits and its least significant $\lceil \log_2(A) \rceil$ bits hold the target way.

Continuing with the example, the target set field of entry r of the MTC points to set n where the migrated data block resides. A direct lookup of the target way data array at that set will return the block (in this case the block is in Way1). The parent set field of entry r of the MTC points to set k of main cache. The H bit of entry r of the MTC is set, since this is the head of a migrated tag list for some set (in this case, set k). Subsequent migrations originating from the set k are linked up in the MTC to form a tag list. In the figure, two more tags are shown at entries p and q of the MTC. The FPTR field of entry r of the MTC points to entry p and the FPTR field of entry p points to entry q, which is the tail of the list (signified by the T bit of that entry).

The above description clearly indicates a straightforward protocol for locating a block in the proposed cache architecture. The lookup begins with a conventional associative tag comparison at the cache index derived from the requested address (in the above example, set k). At the same time a lookup is initiated in the MTC at the entry indicated by the HPTR entry of that set, provided the HPTR entry is valid (actually the MTC lookup begins slightly later due to the access latency of the HPTR RAM). The tag list in the MTC is followed up to the tail until a hit is encountered either in the MTC or in the main cache; if the tag is found neither in the main cache nor in the MTC, a miss is flagged. In case of a hit in the MTC, the target set in the main cache is looked up and the requested block is retrieved. To avoid false hits, each main cache tag is extended by an extra state bit to mark the migrated blocks. For example, a migrated block residing in set k of the main cache should not return a hit even if its tag

matches the tag of a primary lookup. Thus, a primary lookup in the main tag array now has an additional condition for hit, namely, the migrated state must be reset. On the other hand, a secondary lookup carried out due to a hit in the MTC need not check that the migrated state is turned on before flagging a hit because the MTC entry provides the target set as well as the target way. However, to take care of false primary hits, it is absolutely necessary to turn on the migrated state for a migrated block.[3] On an MTC hit, the migrated block is swapped with the LRU block in the parent set. This policy helps improve the average hit latency if the same block is accessed in the near future. In summary, a hit in the primary lookup leaves the hit latency unaltered. But an MTC hit lengthens the hit latency by the latency of a second lookup in the main cache at the migrated set index in addition to any exposed latency of the tag list walk. Note that the second lookup directly accesses the data array of the target way at the target index. It also needs to access the state/tag array of the target set to update the LRU age bits.

While the logic involved in the aforementioned protocol is straightforward, the performance improvement may not be encouraging if proper care is not taken. We address a few important performance and implementation concerns in the following.

Handling Replacement of Migrated Blocks: If a replaced block (due to primary or secondary replacement) from the main cache turns out to be an already migrated one, the involved protocol is slightly more complex. If the block is decided to be replaced from the cache, its entry must be removed from the MTC. To speed up the process, we associate a tag pointer with each block in the main cache. In the figure, these are shown as the Tag PTR RAMs for Way0 and Way1. The tag pointer of a migrated block holds the index of the corresponding MTC entry. A replaced block reads its tag pointer entry provided the migrated state is set and removes itself from the MTC. For this purpose, a backward pointer (henceforth referred to as BPTR) is added to each MTC entry, thereby making each tag list a doubly-linked one (in the interest of brevity, this detail is not shown in the figure). Note that depending on the situation the H/T bits may require modifications leading to changes in the HPTR entry of the parent set. For example, the removal of the tail entry of a list requires updating the T bit of the previous entry and the removal of the head entry of a list requires updating the H bit of the next entry and the HPTR of the parent set.

If a replaced migrated block is selected for re-migration, only the target set/way field of its MTC entry requires a modification.

Managing the Migration Tag Cache Entries: At reset, the MTC entries are organized in the form of a free list. A register holds the index of the current free list head entry. An empty bit indicates if the free list is empty.

[3] It is important to note that the migrated state is not enough to ascertain a secondary lookup hit because there may be multiple migrated blocks residing in the same set with the same tag.

When a tag is allocated in the MTC, the head of the free list is claimed and the next entry in the free list is made the new head. When a tag is freed from the MTC, its entry is returned to the head of the free list and this entry becomes the new free list head. If the free list is found to be empty at the time of allocating a newly migrated tag in the MTC, a randomly chosen entry is first reclaimed before beginning the migration process. While this is an admittedly ad-hoc choice, we have found that a better reclamation protocol does not have a significant impact on performance.

Hit/Miss Critical Path: The critical path of the hit/miss signal plays an important role in any cache architecture. Our preliminary experiments showed that the straightforward lookup protocol depicted above leads to a significant loss of potential performance improvement even though the gains in terms of cache hit rates are high. We implement two notable optimizations to reduce the hit/miss critical path. First, we observe that making the MTC equipped with two read ports doesn't affect the access latency much. We exploit this fact along with the doubly-linked nature of the tag lists to walk the lists in both the directions, namely, from the head and from the tail. This clearly halves the worst case list traversal latency. Such a solution requires maintaining a tail pointer (along with the already existing head pointer) with each main cache set. The head and the tail pointers are read out simultaneously and the MTC lookups are initiated from both ends of the list. We note that replacement of the tail entry of a list in the MTC now requires modifications of the tail pointer in the parent set. For each set, the head and tail pointers are combined into a single entry and will be referred to as the list terminal pointer (LTPTR).

The second optimization for hit/miss critical path aims at shortening the miss path only. With each main cache set, we incorporate a 60-bit filter. The goal of the filter attached to a set is to maintain an approximate summary of the tags that have been migrated from that set. While the filter can return false positives indicating the presence of a tag in the list when actually it is not there, it never returns false negatives. As a result, a negative indication from the filter obviates the need for looking up the MTC. We discuss the functioning of the filter in the following. The 60 bits devoted to the filter are divided into nine segments. Each of the lower eight segments is seven bits in size. The most significant segment is four bits in size. When a tag T is presented to the filter as a query, the lower three bits of the tag ($T[2:0]$) are used to access one of the lower eight segments of the filter. Let the contents of the corresponding segment be $F_0[6:0]$. Further, let the contents of the most significant segment of the filter be $F_1[3:0]$. The filter returns a positive answer if $(F_0[6:0]$ AND $T[9:3])$ equals $T[9:3]$ and $(F_1[3:0]$ AND $T[13:10])$ equals $T[13:10]$. A newly migrated tag T is hashed into the filter in a similar fashion. First, $T[2:0]$ is used to access one of the lower eight segments of the filter and then $T[9:3]$ is ORed with the contents of that segment. At the same time $T[13:10]$ is ORed with the contents of the most significant segment

of the filter. It is important to note that when a migrated tag is replaced from the MTC or moved elsewhere, the filter associated with its parent set is not updated. The filter of a set is updated only when it returns a false positive answer. On a false positive, all the tags in the list have to be visited anyway and the filter contents can be accurately rebuilt. Given the simplicity of the filter, we are adequately satisfied to find that it delivers reasonably low false positive rates for the applications that seldom hit in the MTC. These results are discussed in Section 4.

2.2. Performance Optimizations

A thorough understanding of the performance results of the basic design motivated us to explore three important optimizations. We discuss them in the following.

2.2.1 Selective Migration

All replaced blocks are not of equal importance. Ideally, we would like to migrate the blocks that miss frequently. While it is possible to adopt a Bloom filter in conjunction with a pipelined priority queue proposed in [3] to identify the top frequently missing block addresses, in this paper we propose a much simpler solution. Our solution relies on the observation that the blocks that often miss map to a few hot sets. As a result, it is enough to identify the hot sets and migrate the blocks that are replaced from the hot sets. A set is said to be hot if a sufficiently large number of misses originates from that set.

Each cache set is associated with a counter to count the number of external refills into that set caused by the cache misses (note that refills caused by internal migrations are not counted here). These counters will be referred to as the refill counters. Each refill counter can count up to four times the associativity of the cache minus one (i.e., 63 for a 16-way set-associative cache). Whenever a refill counter reaches its maximum value, all the refill counters in that cache bank (we assume 1 MB 16-way set-associative banks having 512 sets with 128-byte block size) are reset. Each cache bank has two additional registers maintaining the total number of external refills to that bank and the maximum number of refills to a set in that bank (i.e., the number of refills to the hottest set of the bank). Both these registers are reset when the refill counters are reset. A set k is identified as hot if $RefillCount(k) > AvgRefillCount + (MaxRefillCount - AvgRefillCount) >> \delta$, where $RefillCount(k)$ is the refill count of set k, $AvgRefillCount$ is the refill count of the bank containing set k averaged over all the sets of that bank (obtained by shifting the total refill count of the bank by a constant amount to the right), $MaxRefillCount$ is the refill count of the hottest set within the bank, and δ is a dynamically adjusted parameter maintained per bank. Here, the primary intuition is that a set is hot if its refill count exceeds the average refill count by a fraction

of the difference between the maximum and the average. The parameter δ is initialized to three at reset or at context switch. Whenever the number of replaced blocks within a bank that are rejected for migration exceeds a pre-defined threshold T, an attempt is made to increase δ, if required. At this point, if the number of successful migrations originating from that bank is found to be less than T, the value of δ is increased by one; otherwise δ is left unchanged. We set a reasonably large upper limit on δ, which, in this paper, is 16. The threshold T is set to 1024 in this work.

In summary, selective migration is achieved when a block is replaced from a set which satisfies the aforementioned inequality involving the refill counts. This block is migrated to a suitable set identified by the algorithm discussed in Section 2.1.1.

2.2.2 Per-set Migration Limit

Given a limited space in the MTC, it is important to give a fair share to all the hot sets. One possible way of implementing this constraint is to impose a limit on the length of the migration tag list belonging to one parent set. However, it is important to adjust this length according to the demand during the execution. Each main cache set has a saturating counter to hold the current length of the migration tag list belonging to that set. At reset or at context switch the limit on this length is set to 16. However, this limit is monitored periodically based on the demand and increased if needed according to the following algorithm. If the total number of replaced blocks within a bank that are rejected for migration (either due to small list length limit or due to small value of δ discussed in the last section) exceeds a pre-defined threshold T, an attempt is made to increase the list length limit or δ, as required. If the number of successful migrations originating from that bank is found to be less than T, the list length limit for that bank is doubled provided the number of rejected migrations due to inadequate value of list length limit exceeds that due to small value of δ; otherwise the value of δ is increased by one. The threshold T is set to 1024 in this paper.

2.2.3 Retaining the Migrated Blocks

Once the blocks replaced from the hot sets are migrated, it is important to retain them in the cache for a sufficiently long time. As already shown in Figure 1(b), the number of replacements (and hence refills) between the replacement of a block and its reuse tends to be large. Therefore, to experience the full benefit of block migration, it is necessary to modify the replacement policy of the cache so that the hot blocks which are migrated are retained long enough to be reused.

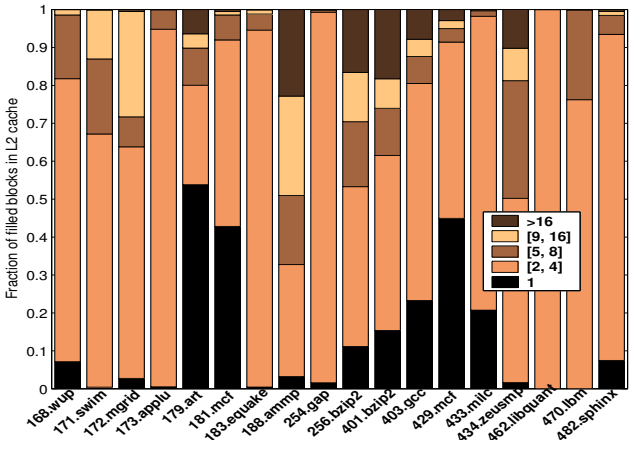

Figure 3. Use distribution of L2 cache blocks.

A recent capacity retention proposal, namely, dynamic insertion policy (DIP) [19], makes the observation that most of the L2 cache blocks are used only once before getting evicted. Based on this observation the authors propose a new insertion policy such that the single-use blocks get filled and immediately evicted on the next miss to the same set leaving the rest of the set undisturbed. Initially, we explored the possibility of using DIP in conjunction with block migration to improve the capacity retention of the cache. However, we find that the number of single-use blocks is small when the ratio between the L2 cache block size and the L1 data cache block size is more than one, which should be the natural choice for exploiting spatial locality in an inclusive cache hierarchy. In Figure 3, we show the use distribution of the blocks filled into the L2 cache during the entire execution. The L2 cache is 2 MB 16-way set-associative with 128-byte block size and the L1 data cache is 32 KB 4-way set-associative with 32-byte block size. It is clear that the dominant use count ranges from two to four. For example, 470.lbm has more than 75% of blocks with use count in this range. The reason for this bias is that the ratio of the L2 cache block size to the L1 data cache block size is four and this defines the "natural use count" of the L2 cache blocks. Nonetheless, we note that 179.art, 181.mcf, and 429.mcf still have significantly high percentages of single-use blocks. However, a robust capacity retention algorithm that does not rely on any specific ratio of the L2 cache block size to the L1 data cache block size must cater to the larger fraction of the applications where the dominant use count is more than one.

Our solution for retaining the migrated blocks classifies the cache sets into two categories, namely, the sets that enjoy a sufficiently high volume of hits and those that do not. For the sets that enjoy a high volume of hits, the baseline LRU replacement policy is left unchanged. However, for the sets that experience a low volume of hits, the non-migrated blocks are given priority in the replacement policy. A saturating counter for maintaining the

number of hits is associated with each main cache set. Note that a hit in the migrated block is counted as a hit for the parent set. The hit counters are reset when the refill counters are reset.

A set is classified as a low-hit set if the number of hits experienced by the set is at most a constant h times the number of external refills to that set. Also, it is necessary for the refill count of the set to be bigger than a constant r before a reliable classification can be determined. If the refill count of a set is less than or equal to r, it is automatically classified as a high-hit set. We fix h at four so that high-hit sets have more than 80% hit rate. Note that this is the minimum "natural hit rate" of an application with good spatial locality, where the notion of "natural use count" of an L2 cache block was introduced above. We set r to one-eighth of the associativity. However, a simulation study revealed that some of the sets take longer to reach a stable behavior and requires a slightly higher target hit rate to avoid misclassification. So we settle with two values of the pair (h, r), namely, $(h_1, r_1) = (5, associativity/2)$ and $(h_2, r_2) = (4, associativity/8)$. To obtain a unified solution, we employ a set sampling technique [19]. Every 64^{th} set is dedicated to measure the effectiveness of the first value pair (h_1, r_1). Every 32^{nd} set which is not an even multiple of 32 is dedicated to measure the effectiveness of the second value pair (h_2, r_2). The rest of the sets follow the current winning value pair. There is a global saturating counter H, which is incremented if one of the dedicated sets in the first category experiences a hit. It is decremented if one of the dedicated sets in the second category experiences a hit. The rest of the sets use the pair (h_1, r_1) if the value in H is greater than or equal to a threshold T_H; otherwise they use the pair (h_2, r_2). In this paper, T_H is set to the mid-point of the value range of H. This is also the value with which the counter is initialized. The hit counter associated with each set is sized such that it can count up to $2 \max(h_1, h_2)$ times the maximum range of the per-set refill counter. H is chosen to be a 32-bit counter.

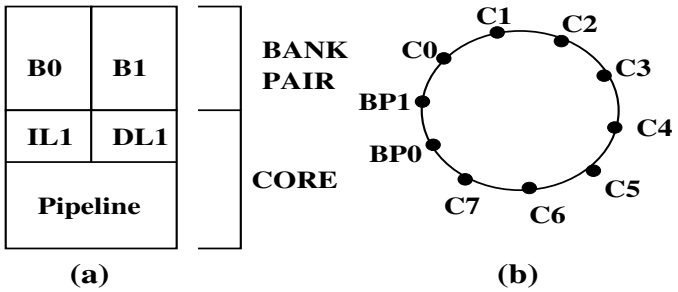

Figure 4. (a) A two-bank single-core architecture. (b) An architecture with more than two banks and/or more than one core. In this particular case, we show eight cores (C0 to C7) and four L2 cache banks (organized as two bank pairs: BP0 and BP1) connected over a bidirectional ring.

2.3. Scaling to Larger Caches and CMPs

Large caches are usually divided into smaller banks. In this paper, we assume that each L2 cache bank is 1 MB 16-way set-associative with 128-byte blocks. Larger L2 caches are built by aggregating multiple such banks. For example, an 8 MB 16-way set-associative L2 cache would have eight banks. The associativity of each bank is same as the aggregate cache. Only the sets are distributed across the banks in a cyclic fashion. Each bank gets 512 sets. In Figure 4, we show two architectures. The left one is an example of a single-core architecture with two L2 cache banks. For architectures with more than two L2 cache banks and/or more than one core, we resort to an organization shown on the right. Here we group the L2 cache banks into pairs and connect each pair to a switch on a bidirectional ring. The cores with their private L1 caches also connect to the ring. For chip-multiprocessors simulated in this paper, we assume that the entire L2 cache is shared among the cores.

To simplify the block migration protocol, we allow block migration among a pair of banks that are connected to the switch. Of course, if an architecture has only two L2 cache banks, this policy boils down to allowing migration between any two sets. For larger caches such as those shown in Figure 4(b), we find that the performance improvement achieved by allowing cross-switch migration is not justified when the increase in coherence protocol complexity is considered. While a full description of the coherence protocol in the presence of cross-switch migration is beyond the scope of this paper, we would like to mention that the primary difficulty arises from the fact that a request now may involve a series of interconnect transactions. This introduces a significant number of deadlock cases involving the virtual channel buffer management in the switches. One possible solution could be to inform the cores about the new location of each migrated L2 cache block. This is equivalent to sending the information stored in the MTC to the cores so that an L1 cache miss request can be directly routed to the target L2 cache bank. However, this solution requires replicating the full MTC information at each core, thereby increasing the storage overhead significantly. So we do not consider this solution.

Allowing migration only between two adjacent L2 cache banks makes our solution scale seamlessly to arbitrarily large caches. Each bank is equipped with its own MTC keeping track of the blocks migrated from that bank. The target set can belong to either the same bank or the adjacent bank. Therefore, if each bank has 512 sets, the target set can be encoded in 14 bits for a 16-way cache and the parent set in 9 bits. In addition to the MTC, each bank maintains the other auxiliary structures discussed in the previous sections. For example, Figure 2 can be considered as showing just one bank of a 2-way set-associative L2 cache. Most of the storage and logic can be designed on a per-bank basis. There are only two pieces of hardware that span two adjacent banks. First, we need

a comparator to connect the per-bank local comparator trees discussed in Section 2.1.1. Second, the global counter H discussed in Section 2.2.3 is maintained per-bank pair.

2.3.1 Synergy with Proximity Management

Non-uniform cache access (NUCA) architectures as shown in Figure 4(b) lead to different access latencies for different L2 cache banks from the viewpoint of the same core. The primary reason for this is the distributed nature of the L2 cache resulting in different interconnect latencies to different banks from the same core. A few studies have been done to improve the proximity of data by migrating them to L2 cache banks closer to the core requiring the data. The choice of disallowing cross-switch migration for conflict resolution makes it orthogonal to the migration for proximity enhancement. Proximity enhancing migration will be necessarily cross-switch because proximity of a particular piece of data does not improve unless the number of interconnect hops between the core and the bank holding the data improves. Once a piece of data is migrated over the interconnect for improving the proximity (the grain of this migration can be one cache block or a suitable number of consecutive cache blocks if the book-keeping overhead is a concern), the exact location within the target pair of banks can be chosen to reduce the chance of conflict misses by using the technique proposed in this paper. We leave the quantitative evaluation of this synergy to future work and focus only on conflict resolution via block migration in this paper.

2.4. Implications on Memory System

Before closing this section, we discuss a rather unexpected impact of block migration on the DRAM performance. Consider a string of L2 cache miss addresses $M_1, M_2, \ldots, M_k$ and the corresponding string of evicted addresses $E_1, E_2, \ldots, E_k$. The addresses in the miss string cause DRAM reads, while the dirty blocks belonging to the eviction string cause DRAM writes. Often the DRAM bank number is derived from the upper few bits of the last-level cache index. For example, according to the DRAM bank computation scheme suggested in [22], we derive the bank number of a 4 GB DRAM module (four 1 GB DIMMs) with each DRAM chip having four internal banks as follows. The lowest three bits of the physical address are the column offset within a DRAM column[4], the next 11 bits form the column, the next two bits are the bank number within a DIMM, the next two bits are the DIMM number, and the upper 14 bits form the row number. Thus, the bits [17:14] of the physical address are used to compute the DIMM number and bank number. On the other hand, in a 2 MB 16-way L2 cache with 128-byte blocks, the set index is derived from the bits [16:7] of the physical address.

[4] The 64-bit columns are distributed among 16 x4 DRAM chips in a DIMM [15].

Now consider an application that sequentially streams through a large amount of data updating a significant fraction of that leading to a large number of dirty evictions. Since the L2 cache set index of M_i is exactly same as that of E_i, we can conclude that the set index of E_i differs from that of M_j only in the lower few bits if $|i - j|$ is small (i.e. within a time window). As a result, the eviction addresses increase the chance of DRAM bank conflicts significantly because the DRAM bank number is derived from the upper few bits of the L2 cache set index. To reduce the volume of such bank conflicts, the bank number can be calculated by XORing the actual bank number bits and the lower few bits of the L2 cache tag [26]. Since the address M_i necessarily has a different tag compared to E_i, such a scheme reduces the chance of bank conflicts.

In the presence of block migration, such bank conflicts are reduced dramatically. This is because the replaced block E_i may get migrated to some other cache set s and the final evicted address will originate from the set s. Since set s may belong to a completely different region of the cache compared to the sets that generate the misses M_i, the chances of the evicted blocks conflicting with the miss addresses in DRAM reduce significantly.

Table 1. Simulated single-core system

Parameter	Value	Parameter	Value
Process/V_{dd}/V_t	65 nm/1.1 V/0.18 V	FP div. latency	12 (SP)/19 (DP) cycles
Frequency	4 GHz	ITLB, DTLB	64/fully assoc./Non-MRU
Pipe stages	18	Page size	4 KB
Front-end width	4	L1 Icache	32 KB/64B/4-way/LRU
Commit width	8	L1 Dcache	32 KB/32B/4-way/LRU
BTB	256 sets, 4-way	Store buffer	32
Branch predictor	Tournament (Alpha 21264)	L1 MSHR	16+1 for retiring stores
RAS	32 entries	L1 cache hit latency	3 cycles
Br. mispred. penalty	14 cycles (minimum)	L2 cache	2 MB/128B/16-way/LRU
ROB	128 entries	L2 MSHR	16 per bank×2 banks
Branch stack	32 entries	L2 bank tag latency	9 cycles
Integer/FP registers	160/160	L2 bank data latency	4 cycles (one way)
Integer/FP/LS queue	32/32/64 entries	Memory cntr. freq.	2 GHz
ALU/FPU	8 (two for addr. calc.)/3	System bus width/freq.	64 bits/2 GHz
Int. mult./div. latency	6/35 cycles	SDRAM bandwidth	6.4 GB/s
FP mult. latency	2 cycles	SDRAM access time	80 ns+20 ns transfer

3. Simulation Environment

We simulate two types of systems, namely, one with a single core and the other with eight cores. Table 1 presents the relevant details of our MIPS ISA-based single-core baseline system. The cache latencies are determined using CACTI [10] assuming a 65 nm process. For L2 cache access, we assume serial tag/data access where only one way of the data bank is accessed if the lookup hits in the tag array. The baseline L2 cache has two banks each of size 1 MB with 16-way set-associativity and 128-byte block size. In addition to the L2 cache configuration shown

in Table 1, we explore a 32-way set-associative configuration of the same size (i.e., 2 MB) where tag access takes 10 cycles.

The eight-core CMP uses the core depicted in Table 1 as the basic building block. For the CMP, we simulate a 4 MB 16-way set-associative shared L2 cache by connecting two pairs of 1 MB 16-way set-associative banks to two switches on a bidirectional ring. Thus, a number of basic 2 MB 16-way set-associative L2 cache modules can be glued to the interconnect to build bigger caches. Each core with its own L1 caches (same configuration as in Table 1) connects to the ring via a switch. This leads to an architecture shown in Figure 4(b). Each switch on the ring takes one cycle for port scheduling i.e., the hop time is one cycle. The L1 caches are kept coherent via a distributed directory-based coherence protocol. Each L2 cache block maintains a directory entry with an eight-bit sharer vector and four states, namely, M, S, pending, and dirty in L1 (the E state is merged into the M state as in [14]). In both single-core and eight-core configurations, the cache hierarchy maintains inclusion.

Our dynamic energy model is based on an improved version of Wattch [5]. Our subthreshold and gate leakage models are developed using the techniques proposed in [6] and [21]. The model is improved at several places by validating against HSPICE simulations. Our L2 cache data RAM banks implement the drowsy cells [7], which switch a 128 KB subbank to a low voltage supply while retaining the data, if the subbank is not accessed for 1000 clock cycles. Our DRAM energy model is based on published data of highly loaded DDR2-400 512 Mb x4 chip available from the Micron technical notes [15, 16]. We scale up the frequency-dependent power components by a factor of two to match our 400 MHz frequency. The DIMM id and DRAM bank id are extracted from a requested physical address as discussed in Section 2.4. A virtual page is mapped to a physical page frame by the bin-hopping algorithm [12] when the page suffers from a page fault.

Table 2. MPKI of single-threaded programs

168.wupwise	171.swim	172.mgrid	173.applu	179.art	181.mcf	183.equake	188.ammp	254.gap
0.5	5.2	1.5	2.3	16.8	50.8	7.0	0.5	1.8
256.bzip2	401.bzip2	403.gcc	429.mcf	433.milc	434.zeusmp	462.libquantum	470.lbm	482.sphinx3
0.9	1.0	1.0	22.2	8.8	1.1	5.7	8.44	4.42

The single-threaded applications used in this study are drawn from a subset of 33 applications of the SPEC 2000 and SPEC 2006 suites. We select 18 applications that have at least 0.5 MPKI on the baseline system. The selection set of 33 applications includes all the SPEC 2000 applications except the C++ and the Fortran 90 applications and a subset of the SPEC 2006 suite.[5] The selected 18 applications were shown in Figures 1 and 3. For each of the

[5] Specifically, the applications considered from SPEC 2000 include 164, 168, 171, 172, 173, 175, 176, 177, 179, 181, 183, 186, 188, 197, 253, 254, 255, 256, 300, and 301. The applications considered from SPEC 2006 include 401, 403, 429, 433, 434, 435, 445, 456, 458,

applications, we execute a representative sample of one billion dynamic instructions chosen with the help of the SimPoint toolset [24]. All the applications use the `ref` input set. The MPKI for each of the applications on the baseline 2 MB 16-way L2 cache is shown in Table 2.

Table 3. Simulated multi-threaded programs

Name	Input	Source	MPKI
Art	MinneSPEC	SPEC OMP	25.4
Equake	MinneSPEC, ARCHduration 0.5	SPEC OMP	1.1
FFT	256K complex doubles	SPLASH-2	1.0
FFTW	2048×16×16 complex doubles	FFTW	5.9
LU	768×768 matrix, 16×16 tiles	SPLASH-2	0.1
Radix	2M keys, radix 32	SPLASH-2	2.8

We use six multi-threaded programs in this paper for evaluating our proposal on an eight-core CMP. The details of these applications are presented in Table 3 along with their MPKI on the baseline 4 MB 16-way set-associative shared L2 cache. These applications use carefully hand-optimized array locks and tree barriers. All these applications are run to completion.

Before closing this section, we present the details of the additional storage structures needed by our proposal in Table 4. Since the auxiliary data and control can be designed separately for each independent 1 MB 16-way set-associative L2 cache bank, we present the overhead for one bank only. The number of tags the migration tag cache (MTC) RAM can hold is half the number of tags in the main cache. Therefore, for a 1 MB L2 cache bank with 128-byte block size, the number of entries in the MTC is 4096. The MTC requires two read ports and one write port. All other RAMs require one read port and one write port. We have verified with CACTI that each RAM can be accessed comfortably within two cycles. We note that in a baseline 2 MB 16-way set-associative L2 cache, one 1 MB 16-way set-associative bank requires 1048576 bytes for data and 29696 bytes of tag and state (23 bits of tag, 2 bits of block state, and 4 bits of LRU state). We assume a single-core node with a 40-bit physical address.[6] Thus, the single-core baseline system requires 1078272 bytes for each bank. As shown in Table 4, our proposal adds 58856 bytes on top of this leading to an overall storage overhead of 5.5%.

4. Simulation Results

This section evaluates our proposal in terms of performance as well as energy. We first discuss the single-core results and then show how our proposal scales to an eight-core CMP. We conclude this section with an evaluation

462, 464, 470, and 482. These are the applications that we can compile and execute with ref inputs on our simulation infra-structure.

[6] In our simulator, each node simulates a 4 GB DRAM and the upper 8 bits of the physical address are used for node id and ASID in multi-node simulations.

Table 4. Storage overhead of conflict resolution via block migration per 1 MB bank

Structure	Cross Ref.	Configuration	Size
Migration Tag Cache RAM	Section 2.1.2	Width: Tag+target set+parent set=23+14+9 =46 bits, 4096 entries, 4-way banked	4096 entries×46 bits/entry = 23552 bytes
Head/Tail RAM	Section 2.1.2	Width: Head bit+Tail bit=2 bits, 4096 entries, 4-way banked	4096 entries×2 bits/entry = 1024 bytes
FPTR RAM	Section 2.1.2	Width: $\log_2(4096)$=12 bits, 4096 entries 4-way banked	4096 entries×12 bits/entry = 6144 bytes
BPTR RAM	Section 2.1.2	Width: $\log_2(4096)$=12 bits, 4096 entries 4-way banked	4096 entries×12 bits/entry = 6144 bytes
LTPTR RAM	Section 2.1.2	Width: valid+head ptr.+tail ptr.=1+12+12=25 bits, one entry per cache set	512 entries×25 bits/entry = 1600 bytes
Tag PTR RAM	Section 2.1.2, Figure 2	Width: valid+MTC bank (within a pair)+MTC index = 1+1+12 = 14 bits, one entry per cache block, 4-way banked	8192 entries×14 bits/entry = 14336 bytes
Auxiliary per-set RAM	Sections 2.1.1, 2.1.2, 2.2.1, 2.2.2, 2.2.3	Fields: replacement count (6 bits) [Section 2.1.1], filter (60 bits) [2.1.2], refill count (6 bits) [2.2.1], list length (12 bits) [2.2.2], hit count (10 bits) [2.2.3]; width: 98 bits, one entry per set	512 entries×94 bits/entry = 6016 bytes
Coarse replacement count	Section 2.1.1	Width: 10 bits	32 entries×10 bits/entry = 40 bytes
TOTAL	—	—	**58856 bytes**

of our proposal in the presence of a hardware prefetcher.

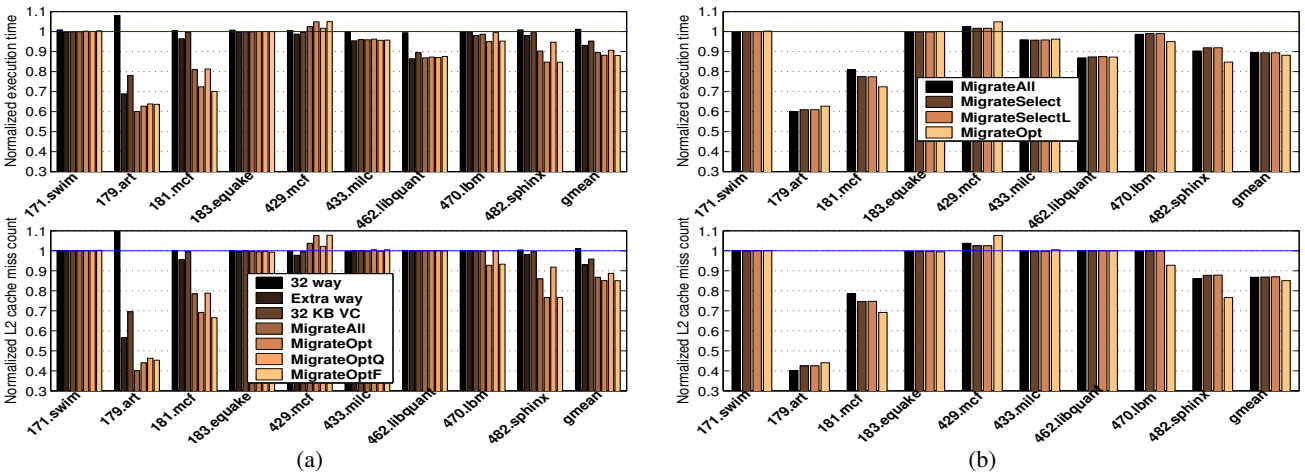

Figure 5. (a) Normalized execution time (upper panel) and L2 cache miss count (lower panel) for applications with more than 4 MPKI. (b) Evaluation of the optimizations presented in Section 2.2.

4.1. Performance and Energy Comparison

Based on the data presented in Table 2, we cluster the single-threaded applications into two categories. One category includes all the applications with more than 4 MPKI. The other category contains the rest, all of which have less than 2.5 MPKI. Figure 5 presents the performance results for the applications with more than 4 MPKI.

In Figure 5(a), we compare the execution time (upper panel) and L2 miss count (lower panel) of seven different configurations. All the results are normalized with respect to the baseline 2 MB 16-way set-associative L2 cache. The left three bars respectively present the results for a 2 MB 32-way set-associative L2 cache, an L2 cache having the same number of sets as the baseline cache with an additional way, and the baseline cache with a 16 KB fully associative victim cache (128 entries$\times$128-byte block) per 1 MB bank giving a total of 32 KB victim caching space. Note that the "Extra way" configuration is a simple way of "giving back" the extra storage needed by our proposal. This configuration has slightly higher storage requirement compared to our proposal. The hit latency of this configuration is conservatively assumed to be the same as the baseline. For the configuration with the victim cache, we implement a not-most-recently-filled random replacement policy and assume that the victim cache lookup is done in parallel with the main cache and its latency is completely hidden under the main cache lookup latency.

The remaining four bars present the results for our proposal. The "MigrateAll" bar represents the design that migrates every replaced block. The "MigrateOpt" bar represents the optimized design after incorporating the optimizations discussed in Section 2.2. The "MigrateOptQ" bar shows the impact of making the number of tags in the MTC one quarter of the tags in the main cache. The "MigrateOptF" bar shows the effect if the MTC can hold the same number of tags as the main cache. Recall that the "MigrateOpt" design sizes the MTC so that it can hold half the number of tags in the main cache. Figure 5(b) further quantifies the transition from "MigrateAll" to "MigrateOpt" as three optimizations are applied progressively: selective migration ("MigrateSelect"), list length limit ("MigrateSelectL"), retention of migrated blocks ("MigrateOpt"). In this figure also the results are normalized to the baseline configuration.

From the upper panel of Figure 5(a) we observe that optimized block migration reduces execution time by 3.7% (433.milc) to 37.3% (179.art) with an average reduction of 12%. 429.mcf is the only application that suffers from a nominal slowdown of 1.7%. The lower panel of Figure 5(a) shows that optimized block migration helps reduce the L2 cache misses by 7.2% (470.lbm) to 56% (179.art) with an average reduction of 15%. A comparison of these two charts shows that 433.milc and 462.libquantum enjoy respectively 3.7% and 12.8% reduction in execution time with optimized block migration, but do not observe any reduction in the L2 cache miss count. We find that these two applications have very high number of DRAM bank conflicts in the baseline configuration caused by a large volume of dirty evictions. With block migration, the percentage of bank conflicts reduces dramatically, as explained in Section 2.4. Compared to baseline, percent DRAM bank conflict reduces by 24.9% and 91% in 433.milc and 462.libquantum, respectively. 462.libquantum has the largest volume of dirty evictions

among these applications. Other applications observe an at most 5% drop in percent DRAM bank conflicts due to block migration. We observe that the "Extra way" configuration and the configuration with a victim cache also enjoy similar benefits for these two applications. The reason why a victim cache helps reduce DRAM bank conflicts is similar to block migration because the miss address and the corresponding replaced address from the VC are likely to be unrelated. Understanding this phenomenon for the "Extra way" configuration requires closer examinations and we find that a prime associativity (17 in this case) staggers the contents of a set in such a way that the miss address and the replaced address become less correlated. A detailed explanation of this phenomenon is beyond the scope of this paper.

Figure 5(b) helps us understand the impact of the three optimizations discussed in Section 2.2. We find that selective migration ("MigrateSelect") does not hurt performance of any of the applications, but improves that of 181.mcf. It is encouraging to note that putting a dynamic limit on the length of the migrated tag list does not hurt the performance at all ("MigrateSelectL"). Finally, our algorithm for retaining the migrated blocks has positive impact on 181.mcf, 470.lbm, and 482.sphinx3 ("MigrateOpt"). In fact, without this algorithm in place, 470.lbm would not experience any improvement at all. However, we note that this algorithm is the sole cause of a 1.7% performance loss in 429.mcf. Referring back to Figure 5(a), we find that "MigrateOptQ" hurts the performance of 181.mcf, 470.lbm, and 482.sphinx3 significantly when compared to "MigrateOpt". On the other hand, doubling the tag store, as in "MigrateOptF", does not improve performance beyond what "MigrateOpt" already achieves. Therefore, we conclude that for this set of applications, the number of migrated tags need not be more than half of the number of main cache tags.

Finally, we note that the 32-way configuration fails to improve performance and, in fact, it increases the execution time of 179.art by 8% due to bigger hit latency. The "Extra way" configuration and the 32 KB VC significantly improve the L2 cache miss count of 179.art, but fail to match the performance of block migration. Referring back to the upper panel of Figure 1(b), we note that 179.art has the lowest median eviction count between the eviction and reuse of an L2 cache block. So it is not surprising that this application would benefit most from a victim cache.

Figure 6(a) presents the normalized execution time (upper panel) and L2 cache miss count (lower panel) for the remaining applications. As expected, none of the applications show any significant improvement. However, 188.ammp and 401.bzip2 exhibit notable reduction in L2 cache misses for "MigrateOpt" (31.9% and 13%, respectively). But they fail to convert this reduction into performance improvement of more than 4%. It is important to note that these two applications have only 0.5 and 1.0 MPKI, respectively.

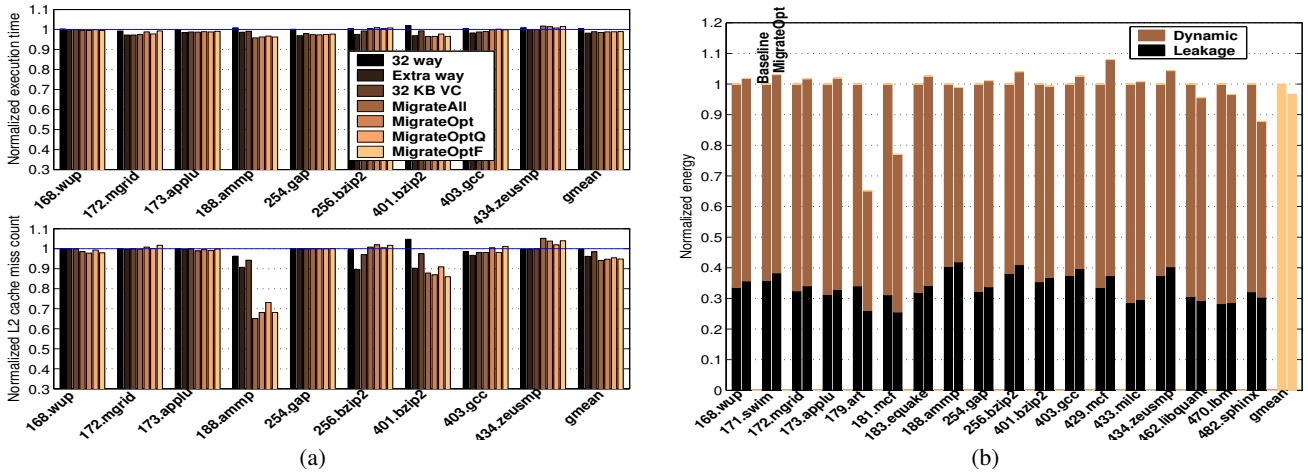

Figure 6. (a) Normalized execution time (upper panel) and L2 cache miss count (lower panel) for applications with at most 4 MPKI. (b) Comparison of block migration and baseline in terms of energy consumption.

Figure 6(b) presents the normalized energy consumption for all the 18 applications. For each application, we show the normalized energy for baseline and optimized block migration broken into leakage and dynamic components. The last group of bars shows the normalized average total energy. The applications that fail to exhibit improvement in execution time with block migration suffer from a slight increase in total energy. This increase mostly comes from an increased leakage in the additional storage structures. The filter turns out to be quite effective in restricting the dynamic energy lost in MTC lookup for these applications (explained further below). On the other hand, the applications that enjoy improvement in execution time also enjoy reduced energy consumption. These are 179.art, 181.mcf, 462.libquantum, 470.lbm, and 482.sphinx3. We note that the reduction in energy in these applications comes mostly from reduced dynamic energy while the leakage component improves only slightly. The major components of this reduction in dynamic energy come from reduced DRAM activities due to less number of L2 cache misses and reduced core pipeline activities due to shortened execution time. Overall, optimized block migration consumes 3.4% less energy on average compared to baseline.

Table 5. Normalized CoV of refill distribution in "MigrateOpt"

168.wupwise	171.swim	172.mgrid	173.applu	179.art	181.mcf	183.equake	188.ammp	254.gap
0.50	1.34	1.50	1.65	0.11	0.57	0.64	0.30	2.98
256.bzip2	401.bzip2	403.gcc	429.mcf	433.milc	434.zeusmp	462.libquantum	470.lbm	482.sphinx3
0.39	0.48	0.64	0.52	1.07	2.12	3.18	0.78	0.45

To further understand the effectiveness of block migration, in Table 5, we present the coefficient of variation (CoV) of the number of refills across the sets in "MigrateOpt" normalized to the baseline CoV. In "Migra-

teOpt", a newly incoming block is filled in its parent set by replacing a block as usual, but the replaced block may get migrated to a target set. In such as situation, the refill is accounted to the target set. The goal of this CoV analysis is to see how well the cache contents get distributed across the sets in the presence of block migration. It is encouraging to note that the applications that enjoy reduction in L2 cache miss count due to block migration exhibit significant improvement in CoV. These applications are 179.art, 181.mcf, 188.ammp, 401.bzip2, 470.lbm, and 482.sphinx3. We also note that there are applications, such as 171.swim, 172.mgrid, 173.applu, 254.gap, 433.milc, 434.zeusmp, and 462.libquantum, for which the CoV degrades in the presence of block migration. However, this does not lead to extra L2 cache misses in these applications. Finally, there are applications, such as 168.wupwise, 183.equake, 256.bzip2, 403.gcc, and 429.mcf, for which the CoV improves, but these applications fail to exhibit any improvement in L2 cache miss count. Our algorithm for capacity retention fails to keep the migrated blocks in cache for sufficiently long time to enjoy any reuse for these applications.

Table 6. MTC characteristics for "MigrateOpt"

Application	Avg. lookup	Percent migrated	Percent hits	Percent false positives
168.wupwise	10.9	12.2	0.8	12.0
171.swim	10.4	45.7	0.1	21.0
172.mgrid	8.6	42.0	0.4	15.9
173.applu	8.7	26.9	0.3	12.8
179.art	3.2	64.1	40.5	62.0
181.mcf	5.3	25.4	23.5	40.8
183.equake	6.6	14.6	0.6	16.9
188.ammp	4.7	24.1	1.3	38.8
254.gap	9.4	64.7	$\sim$0	15.2
256.bzip2	3.9	49.0	4.0	32.7
401.bzip2	4.2	53.5	4.3	44.5
403.gcc	5.1	26.5	1.3	27.7
429.mcf	4.9	12.6	2.8	43.0
433.milc	10.5	30.3	0.2	15.4
434.zeusmp	7.6	47.7	0.2	29.3
462.libquantum	8.3	67.4	$\sim$0	16.9
470.lbm	10.4	7.9	2.8	31.5
482.sphinx3	5.0	19.1	8.1	47.5

Before concluding this discussion, we analyze some of the characteristics of the MTC in Table 6. The second column shows the average number of MTC lookups before a hit/miss can be confirmed in the cases where the lookup misses in the main cache and the filter indicates the presence of the requested address in the MTC. As a result, this data summarizes the critical path when the MTC's outcome is important. Recall that two concurrent lookups to the MTC can be completed in two cycles. Therefore, for 168.wupwise, which has the maximum number of average MTC lookups, the MTC critical path is 5.5 cycles on average. It is important to note that this is not the overall average critical path. The overall average will be much smaller if the negative responses from the filter

and the main cache hits are taken into consideration. The applications that benefit significantly enjoy less number of MTC lookups on average because they are likely to hit before traversing the entire tag list. The third column presents the number of migrated cache blocks as a percentage of all replaced blocks. This result clearly shows the success of selective migration. It is encouraging to note that by migrating only a quarter of the replaced blocks, 181.mcf enjoys significant performance benefit. For 482.sphinx3, only 19.1% replaced blocks are migrated. There are only four applications that migrate more than half of the replaced blocks. The fourth column shows the number of hits in the MTC as a percentage of all hits. As expected, 179.art enjoys 40.5% hits in the MTC, followed by 181.mcf with 23.5% and 482.sphinx3 with 8.1% hits. The last column lists the false positive rate of the MTC access filter as a percentage of the total number filter accesses. Recall that a false positive occurs when a queried address is not present in the MTC, but the filter indicates otherwise. It is encouraging to note that the applications that do not benefit much from block migration have reasonably low false positive rates. The filter plays a critical role for these applications by eliminating most of the futile accesses to the MTC. 179.art has an extremely high false positive rate. This is because the filter is not updated when a hit is encountered in the MTC and the block is swapped with a block in the parent set. It is important to note that 179.art experiences frequent hits in the MTC.

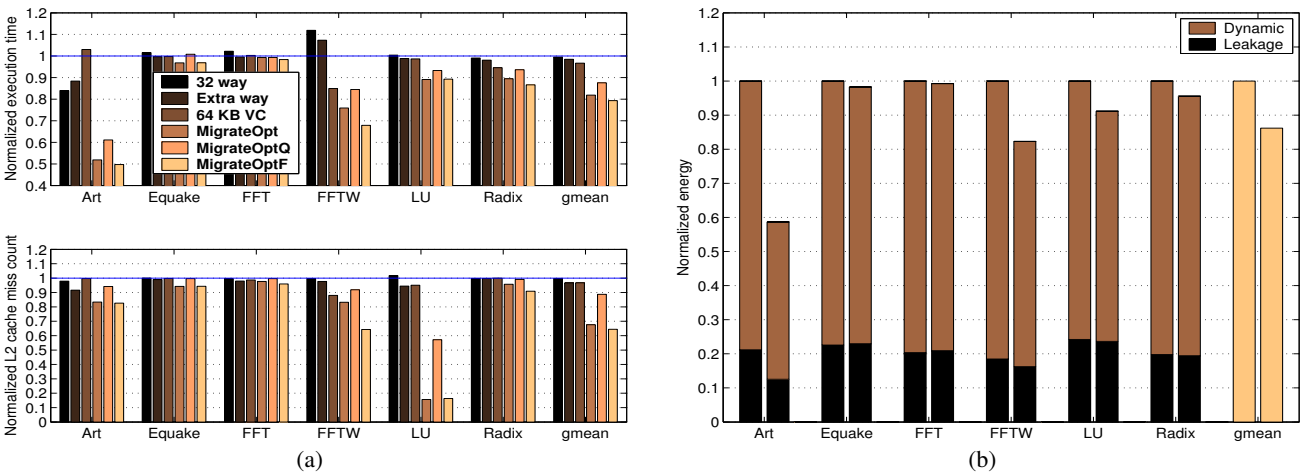

Figure 7. (a) Normalized execution time (upper panel) and L2 cache miss count (lower panel) for multi-threaded applications. (b) Comparison of block migration and baseline in terms of energy consumption on an eight-core CMP.

4.2. Scaling to CMPs

This section evaluates how our proposal scales to an eight-core CMP having a 4 MB 16-way set-associative shared L2 cache. Figure 7(a) presents the normalized execution time (upper panel) and L2 cache miss count (lower panel) for the multi-threaded applications running on the CMP. The configurations are same as those shown in

Figures 5(a) and 6(a), except for the fact that we have dropped the "MigrateAll" configuration. We also note that in the configuration with a 16 KB victim cache per 1 MB bank, the total victim caching space has increased to 64 KB, since there are four 1 MB banks. Block migration ("MigrateOpt") helps reduce the execution time by 3.2% (Equake) to 48.2% (Art) with an average reduction of 18.1%. The corresponding reduction in the L2 cache miss is 32.4% on average. LU enjoys a dramatic 85% reduction in L2 cache miss count. However, this translates into an 11% reduction in execution time because LU does not have too many L2 cache misses (MPKI of 0.1 only). On the other hand, it is surprising to note that for Art and Radix, 16.8% and 4.3% reductions in the L2 cache miss count respectively result in 48.2% and 10.6% reductions in the execution time. For these two applications, we found that the number L2 cache load misses decreases dramatically. But block migration converts some of the store hits of the baseline configuration into misses. In the baseline configuration, these store hits were happening mostly in the cold sets. But block migration disturbs the cold sets and evicts some of these blocks prematurely. In Art, the L2 cache load miss count decreases by 31.5% and in Radix, this reduction is 6.7%. In Radix, an 11.9% reduction in percent DRAM bank conflicts also plays a significant role. As in the case of the single-threaded applications, "MigrateOptQ" delivers significantly worse performance compared to "MigrateOpt" while "MigrateOptF" does not offer better performance compared to "MigrateOpt". Turning to the other configurations, we find that a 32-way set-associative cache is helpful only for Art signifying that this application suffers from a very high volume of conflict misses. However, this configuration increases the execution time of FFTW by more than 10%. The "Extra way" configuration also helps Art while the 64 KB victim cache affects only Radix in a positive way (that too due to a reduction in DRAM bank conflicts).

Finally, Figure 7(b) compares block migration against baseline in terms of energy consumption in the eight-core CMP. The results are very encouraging and the trends closely follow the gains in execution time. Overall, optimized block migration saves 13.8% energy on average compared to baseline.

4.3. Comparison with Related Proposals

In this section, we compare the performance of optimized block migration with a number of relevant proposals such as XOR indexing [8], prime displacement indexing and prime modulo indexing [13], V-way cache [18], dynamic insertion policy (DIP) [19], and shepherd cache [20]. These techniques were briefly discussed in Section 1.1. For prime displacement and prime modulo indexing, we use 511 as the prime modulus and 9 as the tag multiplication factor. Note that 511 is the prime closest to 512, which is the number of sets in a 1 MB bank. We apply these techniques to each bank independently. The V-way cache doubles the number of tags in the base-

line cache. For DIP, we use 32 dedicated sets per 1 MB bank, 1/32 as the BIP epsilon, and an 11-bit saturating counter for DIP insertion policy chooser. Finally, the shepherd cache dedicates four ways (out of 16) in a set for shepherding.

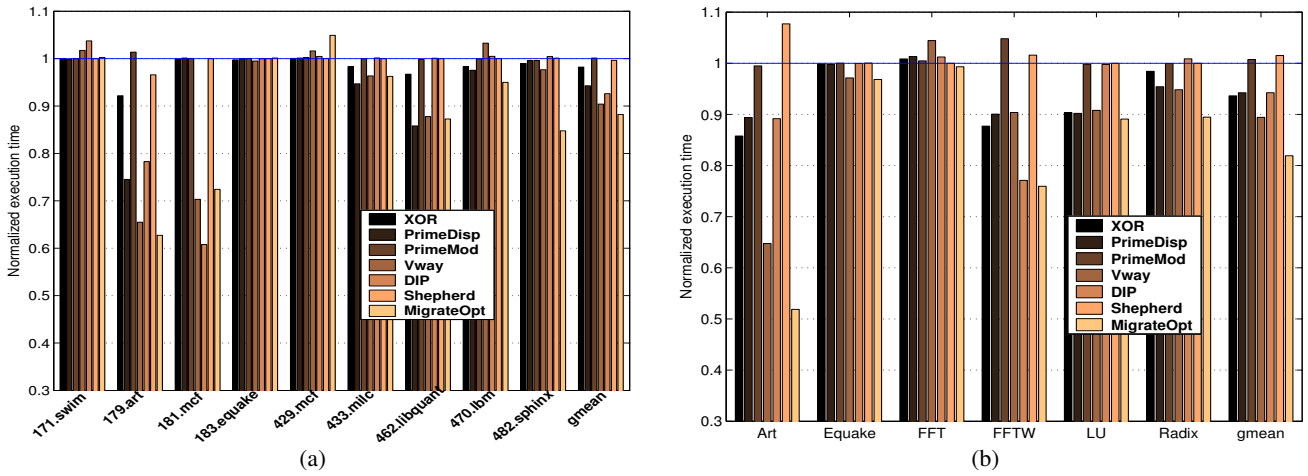

Figure 8. Comparison of optimized block migration with the related proposals: (a) Single-threaded applications with more than 4 MPKI. (b) Multi-threaded applications running on an eight-core CMP.

Figure 8(a) presents the execution time comparison for the single-threaded applications with more than 4 MPKI. All the bars are normalized to the baseline results. Across the board, "MigrateOpt" turns out to be the best technique. Only in 181.mcf, DIP performs better than "MigrateOpt" and in 433.milc, "PrimeDisp" is slightly better than "MigrateOpt". As was evident from Figure 3, DIP improves performance only in 179.art and 181.mcf. The V-way cache turns out to be more robust than DIP and improves the performance of a number of applications, buts fails to perform as well as "MigrateOpt". It is encouraging to note that "MigrateOpt" with only 50% extra tags for keeping track of the migrated blocks outperforms the V-way cache which uses 100% more tags. The shepherd cache succeeds in improving the performance of 179.art only by a small amount. Since most of the non-shepherd ways in a set remain untouched within the shepherding window, this technique frequently falls back to the baseline LRU policy. Among the indexing techniques, "PrimeDisp" turns out to be the best on average. Overall, optimized block migration reduces the execution time by 12%. The V-way cache follows closely with a 9.6% reduction in execution time. The DIP and prime displacement indexing techniques reduce the execution time by 7.4% and 5.8%, respectively.

Figure 8(b) presents similar comparison for the multi-threaded applications running on an eight-core CMP. Here also optimized block migration emerges the best. On average, it reduces the execution time by 18.1% followed by the V-way cache delivering a 10.6% reduction in execution time. The XOR indexing, prime displacement

indexing, and DIP deliver similar performance achieving an average reduction of around 6% in execution time.

4.4. Comparison with Fully Associative L2 Caches

Unconstrained dynamic block migration between any two sets in any two banks allows the L2 cache controller to place a block in any set of the cache. This is equivalent to the functionality of a fully associative L2 cache. While this particular observation alone makes unconstrained dynamic block migration important, the replacement policy of such a cache now becomes the key determinant of end-performance. We have proposed a new replacement policy in this paper to improve the retention of the migrated blocks in a constrained migration environment. In this section, we compare our design against a fully associative L2 cache exercising a global LRU replacement policy (FA-LRU). Since it may not be feasible to implement one such design within reasonable latency budget, we do not compare the execution time and focus solely on the number of L2 cache misses. For the experiments involving the single-threaded applications, we simulate a single bank of fully associative 2 MB L2 cache. For the experiments on the CMP platform, we simulate two banks of fully associative caches, each of size 2 MB. These two banks are connected to the two switches that used to connect the bank pairs in the baseline design.

Table 7. L2 cache misses normalized to baseline

Configuration	179.art	188.ammp	401.bzip2	Art	FFTW	LU
FA-LRU	0.31	0.64	1.06	1.21	0.26	0.32
MigrateOpt	0.44	0.68	0.87	0.83	0.83	0.15

Table 7 summarizes the number of L2 cache misses normalized to baseline only for those applications that get noticeably affected by the FA-LRU configuration. For these applications, we also show the normalized L2 cache miss count achieved by "MigrateOpt". "MigrateOpt" turns out to be inferior to a global LRU policy in 179.art, 188.ammp, and multi-threaded FFTW. However, in 401.bzip2 and multi-threaded Art, the FA-LRU configuration performs worse than the baseline. Finally, there are several applications (that do not appear in this table) where FA-LRU fails to affect the L2 cache miss count, while our policy exhibits significant improvement. In general, we observe that the applications with relatively small working sets benefit from FA-LRU because once the working set mostly fits in the fully associative cache, a global LRU should be close to the best. On the other hand, if the working set cannot fit in the cache, it becomes more important to carefully separate the most frequently used blocks from the rest and retain them. An approximate way to measure the frequency of use of cache blocks over a large span of time is to track the frequency of misses originating from different sets of the cache. This is precisely what our replacement policy tries to achieve.

4.5. Interaction with Prefetching

All the results presented up to this point do not take into account the effects of hardware prefetchers. In this section, we integrate a multi-stream stride prefetcher into the system and evaluate our proposal. Since we are exploring optimizations for the L2 cache, the ideal place to integrate the prefetcher is the L2 cache controller so that the prefetcher can monitor the accesses to the L2 cache and learn the stride patterns. However, our preliminary results revealed that such a design performs poorly even for applications with fairly regular stride patterns. The primary reason for this is that the accesses seen by the L2 cache are often a re-ordered version of the original stream. This re-ordering mostly happens due to out-of-order issue in the load/store pipeline, re-ordering of L1 cache miss requests in the virtual channel buffers of the on-chip interconnect, and non-deterministic behavior of the L1 cache replacement policy. Therefore, to make the evaluation fair, we integrate a multi-stream stride prefetcher with each core's L1 cache controller. Each prefetcher keeps track of sixteen simultaneous load and store streams. The stream size is chosen to be 4 KB to match the page size. The strides are calculated from the committed loads and stores so that speculative wrong path execution does not pollute the prefetcher. The prefetcher prefetches six strides ahead in each stream. It is important to note that since the prefetchers prefetch into the L1 caches, their performance is likely to be better than L2 cache optimization techniques. However, we are interested to see how our proposal performs in the presence of an aggressive hardware prefetcher.

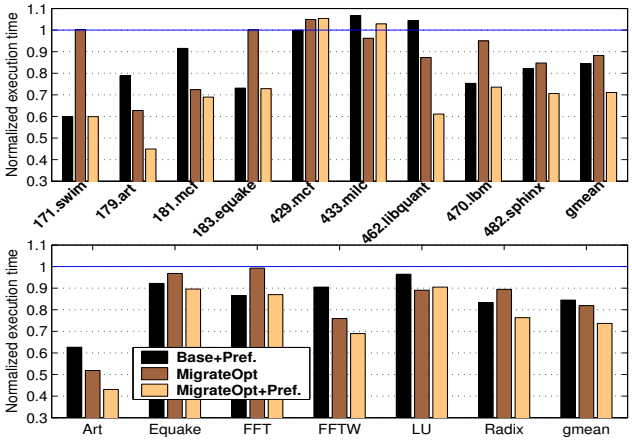

Figure 9. Evaluation of block migration in the presence of a hardware prefetcher. Upper panel: single-threaded applications with more than 4 MPKI. Lower panel: multi-threaded applications running on an eight-core CMP.

Figure 9 presents the execution time for three configurations, namely, baseline with a prefetcher, optimized block migration without a prefetcher, and optimized block migration with a prefetcher. All the execution times are normalized with respect to the baseline. The upper panel shows the results for the single-threaded applications with

more than 4 MPKI and the lower panel shows the results for the multi-threaded applications running on an eight-core CMP. While the "Base+Pref." bar shows the efficiency of the L1 cache prefetcher, we are more interested in comparing this bar with the "MigrateOpt+Pref." bar. It is encouraging to note that even in the presence of hardware prefetching our proposal continues to improve performance for 179.art, 181.mcf, 462.libquantum, and 482.sphinx3. In fact, block migration without a prefetcher turns out to be better than the baseline with a prefetcher for 179.art, 181.mcf, 433.milc, and 462.libquantum. On average, the hardware prefetcher reduces the execution time of the baseline configuration by 15.6%. When block migration is enabled on top of this, the execution time is reduced by 28.9% compared to the baseline. The trends are similar for the multi-threaded applications. In these applications, on average, the hardware prefetcher helps reduce the execution time of the baseline configuration by 15.6%, while block migration in the presence of the prefetcher achieves a reduction of 26.4% in execution time.

5. Summary

We have explored the application of dynamic block migration to resolve conflicts in the last-level cache (L2 in this case) and improve the space utilization of the cache. The central idea is to migrate replaced blocks from hot sets to relatively "colder" regions of the cache. While the idea is conceptually simple, we explore some of the key aspects that need to be addressed to convert the concept into an implementable and scalable design. We present a modular design where each cache bank can be architected independently and block migrations can be confined to a pair of adjacent banks in the interest of scalability. Our design offers the migration tag cache size as a key design parameter that can be chosen appropriately to trade area for performance. Our execution-driven simulation results show that for a 2 MB 16-way set-associative L2 cache, our proposal reduces the execution time by 12% and energy by 3.4% on a set of nine memory-intensive single-threaded applications drawn from the SPEC 2000 and SPEC 2006 suites. On an eight-core chip-multiprocessor with a 4 MB 16-way set-associative shared L2 cache, our proposal reduces the execution time by 18.1% and energy by 13.8% on a set of multi-threaded kernels and applications. This excellent performance comes at the cost of only 5.5% of extra storage.

One of the major contributions of our modular design is that it makes block migration for conflict resolution independent of data migration for proximity management in the emerging non-uniform cache architectures. As the next step toward making such large caches most effective, we will explore the synergy of conflict resolution via block migration and data migration (perhaps at a coarse grain) to enhance on-chip data proximity.

Acknowledgments

The author thanks the Intel Research Council for funding this effort. Gautam Doshi pointed out the importance of comparing our proposal against a fully associative L2 cache.

References

[1] A. Agarwal, J. L. Hennessy, and M. Horowitz. Cache Performance of Operating System and Multiprogramming Workloads. In *ACM Transactions on Computer Systems*, **6**(4): 393–431, November 1988.

[2] A. Agarwal and S. D. Pudar. Column-associative Caches: A Technique for Reducing the Miss Rate of Direct-mapped Caches. In *Proceedings of the 20th International Symposium on Computer Architecture* , pages 179–190, May 1993.

[3] A. Basu et al. Scavenger: A New Last Level Cache Architecture with Global Block Priority. In *Proceedings of the 40th International Symposium on Microarchitecture*, pages 421–432, December 2007.

[4] B. N. Bershad et al. Avoiding Conflict Misses Dynamically in Large Direct-mapped Caches. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, October 1994.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-level Power Analysis and Optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 83–94, June 2000.

[6] X. Chen and L-S. Peh. Leakage Power Modeling and Optimization in Interconnection Networks. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 90–95, August 2003.

[7] K. Flautner et al. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 148–157, May 2002.

[8] A. González et al. Eliminating Cache Conflict Misses through XOR-based Placement Functions. In *Proceedings of the International Conference on Supercomputing*, pages 76–83, July 1997.

[9] E. G. Hallnor and S. K. Reinhardt. A Fully Associative Software-managed Cache Design. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 107–116, June 2000.

[10] HP Labs. CACTI 4.2. Available at http://www.hpl.hp.com/personal/Norman_Jouppi/cacti4.html.

[11] N. P. Jouppi. Improving Direct-mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, June 1990.

[12] R. E. Kessler and M. D. Hill. Page Placement Algorithms for Large Real-indexed Caches. In *ACM Transactions on Computer Systems*, **10**(4): 338–359, November 1992.

[13] M. Kharbutli et al. Using Prime Numbers for Cache Indexing to Eliminate Conflict Misses. In *Proceedings of the 10th International Conference on High-Performance Computer Architecture*, pages 288-299, February 2004.

[14] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[15] Micron Technology Inc. DDR2 Offers New Features and Functionality. *Micron Technical Note TN-47-02*.

[16] Micron Technology Inc. Calculating Memory System Power for DDR2. *Micron Technical Note TN-47-04*.

[17] J-K Peir, Y. Lee, and W. W. Hsu. Capturing Dynamic Memory Reference Behavior with Adaptive Cache Topology. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 240–250, October 1998.

[18] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-way Cache: Demand-based Associativity via Global Replacement. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 544–555, June 2005.

[19] M. K. Qureshi et al. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 381–391, June 2007.

[20] K. Rajan and R. Govindarajan. Emulating Optimal Replacement with a Shepherd Cache. In *Proceedings of the 40th International Symposium on Microarchitecture*, pages 445–454, December 2007.

[21] R. M. Rao et al. Efficient Techniques for Gate Leakage Estimation. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 100–103, August 2003.

[22] S. Rixner. Memory Controller Optimizations for Web Servers. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 355–366, December 2004.

[23] A. Seznec. A Case for Two-way Skewed-associative Caches. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 169–178, May 1993.

[24] T. Sherwood et al. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support on Programming Languages and Operating Systems*, pages 45–57, October 2002.

[25] R. Subramanian, Y. Smaragdakis, G. H. Loh. Adaptive Caches: Effective Shaping of Cache Behavior to Workloads. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 385–396, December 2006.

[26] Z. Zhang, Z. Zhu, and X. Zhang. A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 32–41, December 2000.