Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches

Mainak Chaudhuri
Indian Institute of Technology
Kanpur 208016
INDIA
mainakc@iitk.ac.in

ABSTRACT

Cache blocks often exhibit a small number of uses during their life time in the last-level cache. Past research has exploited this property in two different ways. First, replacement policies have been designed to evict dead blocks early and retain the potentially live blocks. Second, dynamic insertion policies attempt to victimize single-use blocks (dead on fill) as early as possible, thereby leaving most of the working set undisturbed in the cache. However, we observe that as the last-level cache grows in capacity and associativity, the traditional dead block prediction-based replacement policy loses effectiveness because often the LRU block itself is dead leading to an LRU replacement decision. The benefit of dynamic insertion policies is also small in a large class of applications that exhibit a significant number of cache blocks with small, yet more than one, uses.

To address these drawbacks, we introduce pseudo-last-in-first-out (pseudo-LIFO), a fundamentally new family of replacement heuristics that manages each cache set as a fill stack (as opposed to the traditional access recency stack). We specify three members of this family, namely, dead block prediction LIFO, probabilistic escape LIFO, and probabilistic counter LIFO. The probabilistic escape LIFO (peLIFO) policy is the central contribution of this paper. It dynamically learns the use probabilities of cache blocks beyond each fill stack position to implement a new replacement policy. Our detailed simulation results show that peLIFO, while having less than 1% storage overhead, reduces the execution time by 10% on average compared to a baseline LRU replacement policy for a set of fourteen single-threaded applications on a 2 MB 16-way set associative L2 cache. It reduces the average CPI by 19% on average for a set of twelve multiprogrammed workloads while satisfying a strong fairness requirement on a four-core chip-multiprocessor with an 8 MB 16-way set associative shared L2 cache. Further, it reduces the parallel execution time by 17% on average for a set of six multi-threaded programs on an eight-core chip-multiprocessor with a 4 MB 16-way set associative shared L2 cache. For the architectures considered in this paper, the storage overhead of the peLIFO policy is one-fifth to half of that of a state-of-the-art dead block prediction-based replacement policy. However, the peLIFO policy delivers better average performance for the selected single-threaded and multiprogrammed workloads and similar average performance for the multi-threaded workloads compared to the dead block prediction-based replacement policy.

MICRO'09, December 12–16, 2009, New York, NY, USA.

Categories and Subject Descriptors

B.3 [**Memory Structures**]: Design Styles

General Terms

Algorithms, design, measurement, performance

Keywords

Last-level cache, replacement policy, chip-multiprocessor

1. INTRODUCTION

Replacement policies play an important role in determining the performance delivered by a cache. As the on-chip last-level caches grow in capacity, the replacement policies become even more important for good utilization of the silicon estate devoted to the caches. Past research on cache replacement policies has observed that a block often becomes dead after a few uses [7]. This property has been exploited in dead block predictors and dynamic insertion policies.

By identifying the dead blocks and evicting them early, one can correct the mistakes made by the LRU replacement in the L1 caches [3, 7] or in the L2 caches [9]. Specifically, such an approach helps reduce conflict misses by allowing the LRU block, which may not be dead yet, to stay longer in the cache. These proposals select the dead block that is closest to the LRU block in the access recency stack for replacement. However, since the last-level caches are usually large and highly associative, near-term conflicts are few in many applications. As a result, in many cases, the LRU block itself is dead. In such a situation, a traditional dead block predictor would victimize the LRU block and end up following an LRU replacement policy. The primary reason for this phenomenon is that the definition of "death" is myopic and designed specifically to defend against near-term conflict misses. Usually, during the execution of an application, most of the cache blocks undergo periodic use times and dead times. As a result, a cache block that is dead now will probably be used many times in future, although may not be in near-future. This fact is substantiated in Figure 1.

The fourteen single-threaded applications shown in Figure 1 are drawn from SPEC 2000 and SPEC 2006 suites and each of them executes a representative set of one billion instructions on the `ref` input set. The L2 cache (last-level cache in our case) is 2 MB in size with 16-way set associativity and LRU replacement.[1]

[1] More information on simulation environment is available in Section 3.

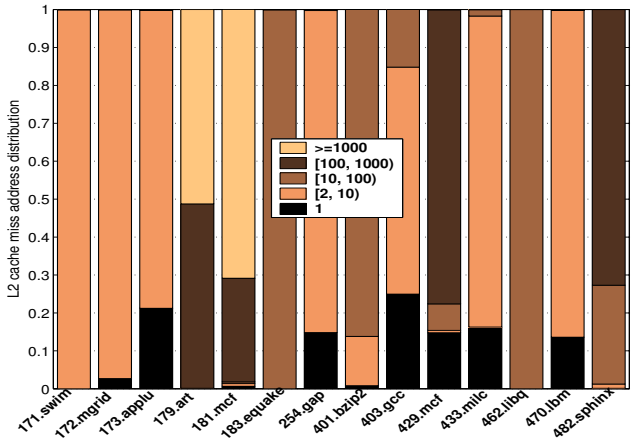

Figure 1: Distribution of L2 cache block addresses in the miss stream for single-threaded applications. We have shortened 462.libquantum to 462.libq and 482.sphinx3 to 482.sphinx.

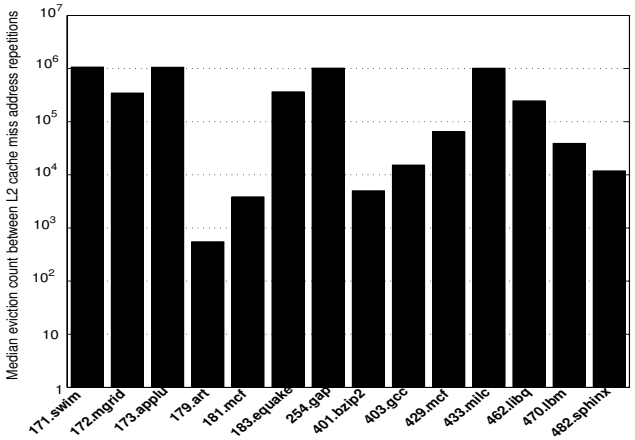

Figure 2: The median count of evictions between eviction of an L2 cache block and re-appearance of the same cache block address in the L2 cache miss stream for single-threaded applications.

Figure 1 classifies each L2 cache block address appearing in the miss stream into one of five categories. A category $[x, y)$ includes all cache blocks that appear in the L2 cache miss stream at least x times and at most $y-1$ times. We have put the cache blocks that appear exactly once in the miss stream in a separate category. The most important take-away point from this chart is that a large number of blocks repeat a significantly high amount of times in the L2 cache miss stream. For example, in 181.mcf, almost 70% of the L2 cache block addresses appear at least thousand times in the miss stream. In 179.art, almost half of the L2 cache block addresses appear hundred to thousand times each, while the remaining half appear at least thousand times each. Figure 2 shows, however, that the median number of L2 cache evictions that take place between eviction of an address and re-appearance of the same address in the L2 cache miss stream is extremely high indicating a large interval between the LRU eviction and the reuse of the same address. The median L2 cache eviction count is more than ten thousand for eleven out of the selected fourteen applications. As a result, many of these highly repeated blocks would be predicted dead and victimized by a traditional dead block predictor. A good replacement policy would, however, retain as many blocks as possible, even if they are dead.

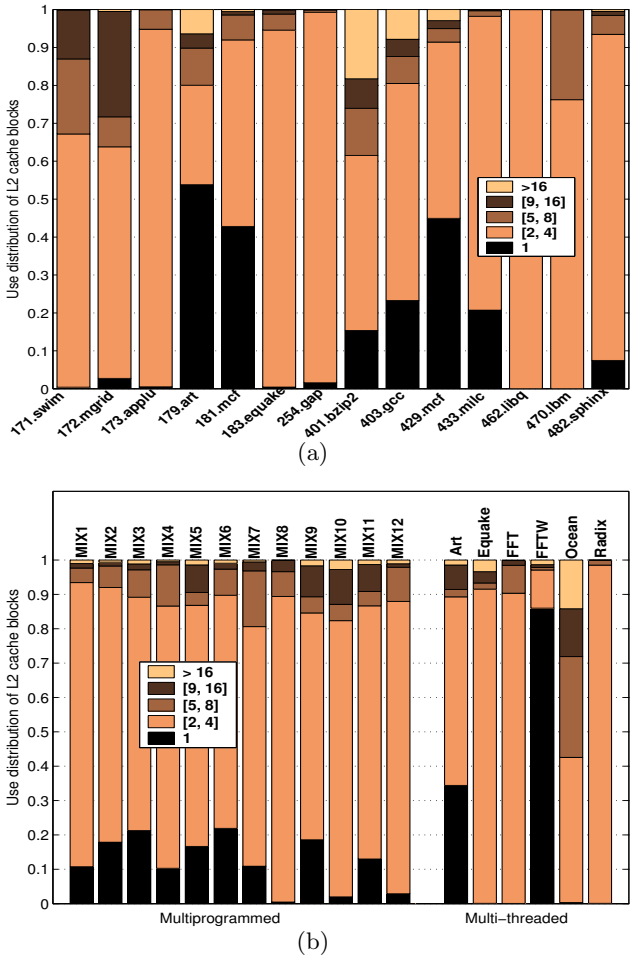

Figure 3: Distribution of L2 cache block uses for (a) single-threaded applications, and (b) multiprogrammed and multi-threaded workloads.

Dynamic insertion policies [11] address this problem to some extent by changing the cache block insertion policy so that the single-use blocks (i.e., dead on fill) are evicted as early as possible. Thus, one can leave a large number of blocks in the cache undisturbed. Such a policy may help improve cache block reuse in workloads that exhibit temporal locality on a large time-scale, as depicted in Figure 1. However, we observe that there is a large class of workloads that do not have many single-use L2 cache blocks. Figure 3 shows the distribution of uses to the cache blocks in an L2 cache. The single-threaded applications are shown in Figure 3(a), while the multiprogrammed and multi-threaded ones are shown in Figure 3(b). The twelve multiprogrammed workloads are prepared by mixing four single-threaded applications and run on a quad-core chip-multiprocessor (CMP) simulator with an 8 MB 16-way set associative shared L2 cache. The multi-threaded workloads execute on an eight-core CMP with a 4 MB 16-way set associative shared L2 cache. The cores have private 32 KB 4-way set associative L1 caches. The L2 cache block size is 128 bytes and the L1 data cache block size is 32 bytes. All the caches exercise LRU replacement. For each workload, we divide all the L2 cache blocks into five categories depending on the number of uses seen by these blocks. A category $[x, y]$ includes all blocks with number of uses at least x and at most y. We observe that for most of the workloads the majority of the cache blocks have number of uses between two and four (note that the ratio of L2 cache block size to L1 cache block size is four). For example, in 462.libquantum, all blocks fall in this

range (see Figure 3(a)). On the other hand, there are workloads with a reasonably high percentage of single-use blocks, e.g., 179.art, 181.mcf, 429.mcf among the single-threaded applications, and FFTW among the multi-threaded ones.

The dynamic insertion policies work well for applications with a large volume of single-use blocks because these applications require only one way in most of the sets to satisfy the short-term reuses. Therefore, the remaining ways can be used to retain a fraction of the working set to satisfy the far-flung reuses. The key observation underlying the idea presented in this paper is that even in the applications where the number of uses per cache block is more than one, this number, on average, is still much smaller than the associativity of the cache. Therefore, a good replacement policy would dynamically learn that all the ways are not needed to satisfy the short-term reuses and would utilize the "spare" ways for working set retention to satisfy some of the far-flung reuses. Based on this observation, we introduce a family of replacement policies called pseudo-LIFO. All the members of this family make use of the fill stack position (in addition to the traditional access recency stack position) of each block within a set to arrive at a replacement decision. The top of the fill stack (position zero) holds the most recently filled block in the set and the bottom of the stack (position $\mathcal{A}-1$ where $\mathcal{A}$ is the associativity of the cache) holds the least recently filled block in the set. The basic idea is to confine the replacement activities within a set to the upper part of the fill stack as much as possible and leave the lower part undisturbed to satisfy the far-flung reuses. In this paper, we specify three members of this family, namely, dead block prediction LIFO (dbpLIFO), probabilistic escape LIFO (peLIFO), and probabilistic counter LIFO (pcounterLIFO). The peLIFO policy is the central contribution of this paper. This replacement policy dynamically learns the probabilities of experiencing hits beyond each of the fill stack positions. Based on this probability function, the peLIFO policy deduces a set of highly preferred eviction positions in the upper part of the fill stack. Complete designs of dbpLIFO, peLIFO, and pcounterLIFO are presented in Section 2.

Our detailed simulation results (Sections 3 and 4) compare dbpLIFO, peLIFO, and pcounterLIFO with an array of proposals discussed in the next section for single-threaded, multiprogrammed, and multi-threaded workloads. The peLIFO policy, while having less than 1% storage overhead, reduces the execution time by 10% on average compared to a baseline LRU replacement policy for the fourteen single-threaded applications shown in Figure 3. It reduces the average CPI by 19% on average for the twelve four-way multiprogrammed workloads and it saves 17% of the parallel execution time on average for the six eight-way multi-threaded programs.

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 this short article, in the following we discuss some of the studies most relevant to our proposal. The first dead block prediction proposal was for L1 data caches [7]. This proposal used a trace-based address-indexed dead block predictor to enhance the quality of correlation prefetching into the L1 data cache. A subsequent proposal used the regularity in live times of the L1 data cache blocks to predict death [3]. A more recent proposal shows how to apply dead block prediction mechanisms to improve replacement quality in the last-level caches [9]. At the time of a replacement in a set, this proposal selects the dead block closest to the least recently used (LRU) position in the access recency stack. Based on the history of reference counts, this proposal predicts death of a block only when a block makes a transition out of the most recently used (MRU) position of the access recency stack (note that this is very different from a transition out of the most recently filled position of the fill stack). This proposal also uses hysteresis counters to improve the confidence of prediction in the case of fluctuating reference counts. Our dbpLIFO policy uses this dead block prediction mechanism.

Perhaps the oldest technique for conflict resolution and working set retention is victim caching [5], where the last few evicted cache blocks are maintained in a small fully associative buffer hoping to enjoy reuses. Selective victim caching based on reload intervals of cache blocks has also been proposed [3]. A more recent proposal shows how to design a large, yet fast, victim cache for the last level of caches with selective insertion based on the frequency of misses to different cache blocks [1].

Dynamic insertion policy (DIP) is a scheme for retaining a large portion of the working set based on the observation that a large number of cache blocks become dead on fill [11]. In this scheme, the insertion policy of a cache block is dynamically learned based on the outcome of a set dueling technique. A newly filled block is inserted into the LRU position, but not promoted to the MRU position until it observes an access. Thus the blocks that are dead on fill get evicted as early as possible without disturbing the remaining blocks in the set. However, for workloads with a small number of single-use blocks, DIP and LRU policies exhibit similar performance. A subsequent proposal extends DIP to make it thread-aware and this policy is referred to as thread-aware DIP with feedback (TADIP-F) [4]. We will refer to it as TADIP.

Sharing the last-level cache among multiple threads usually leads to destructive interference. In such situations, avoiding near-term conflicts is sometimes more important than retaining working sets. Utility-based cache partitioning (UCP) proposes a scheme to partition the ways among the threads based on the marginal utility of the partitions [10]. A recent proposal aims at achieving the best of DIP and UCP by inserting blocks from a thread at a position in the access recency stack that is exactly k positions away from the LRU position where k is the number of ways assigned to this thread by the partitioning algorithm [15]. This proposal combines this insertion policy with a probabilistic promotion policy by associating a static probability to the event of promoting a block up the access recency stack on a hit. This scheme is referred to as promotion/insertion pseudo-partitioning (PIPP).

Adaptive set pinning (ASP) is another scheme for reducing the interference among the threads in the last-level cache [13]. In this scheme, each set is dynamically owned by a certain thread and only that thread is eligible to allocate/evict a block to/from that set. An allocation request from another thread mapping to this set must allocate the block in a small processor owned private (POP) cache partition of the last-level cache. However, an adaptive set-ownership hand-over policy makes sure that a thread does not unnecessarily hold a set for too long when other threads are continuously missing on this set.

A recent proposal explores software-directed cache partitioning algorithms by computing the L2 cache miss rate curves online [14]. This proposal makes use of the hardware performance monitoring units to prepare a log of the L2 cache access trace and subsequently analyzes this log in software to infer the cache demand of the executing application or the mix of applications.

2. PSEUDO-LIFO EVICTION POLICIES

In this section, we present the design and implementation of the pseudo-LIFO family of replacement policies. Section 2.1 motivates the need for maintaining the fill stack position of each block within a set. Sections 2.2 and 2.3 present the dbpLIFO, peLIFO, and pcounterLIFO policies. Section 2.4 discusses the hardware implementation issues.

2.1 Fill Stack Representation

While a block can move both upward and downward in an access recency stack depending on the activities within the set, a block can only move monotonically downward in the fill stack. A block's life starts at the top of the fill stack (position zero), and it makes downward movements until it becomes the least recently filled block and stays at the bottom of the stack. This unidirectional dynamics simplifies arguing about block movement within a set. The fill stack representation is orthogonal to the replacement policy, however. If on a fill, the block at position k of the fill stack is replaced, all blocks in positions zero (top of the stack) to $k-1$ are moved downward by one slot. The newly filled block occupies the position zero.

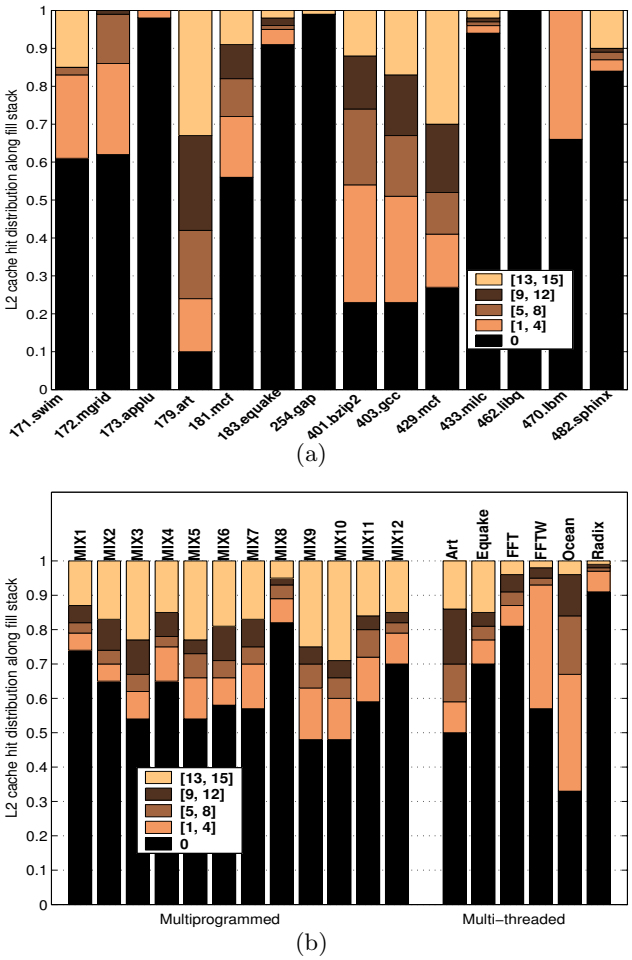

Figure 4: Hit distribution of L2 cache blocks in the fill stack for (a) the single-threaded applications, and (b) the multiprogrammed and multi-threaded workloads.

To establish the connection between the pseudo-LIFO family and the fill stack representation, Figure 4 quantifies the L2 cache hit distribution within the fill stack. The L2 cache configurations are same as those used for Figure 3. We classify the hits into five categories depending on the fill stack position a hit is experienced at. A category $[x, y]$ includes all hits observed in fill stack positions in the range $[x, y]$.[2] The hits at fill stack position zero are grouped in a single category. For most of the workloads, the majority of the hits are observed at fill stack position zero. In fact, for 462.libquantum, all hits take place at the top of the stack. This essentially means that most of the hits experienced by a cache block take place soon after it is filled. However, there are important exceptions. For example, the bottom half of the fill stack experiences a fairly large number of hits in 179.art, 429.mcf, and some of the multiprogrammed and multi-threaded workloads.

We note that this observation does translate to a high fraction of hits in the upper part of the access recency stack in the workloads where most of the L2 cache blocks in a set are not accessed in an interleaved fashion. But, in general, exploiting this property is much more difficult in the access recency stack than in the fill stack due to bidirectional walks of the blocks within the former. In summary, the statistics presented in Figure 4 imply that the cache blocks occupying the lower parts of the fill stack are mostly dead. The pseudo-LIFO family members, instead of evicting them, actually retain them, thereby confining most of the replacement activities in a set to the upper part of the fill stack. We formulate this family below.

Pseudo-LIFO. *Pseudo-LIFO is a family of replacement policies, members of which attach higher eviction priorities to the blocks residing closer to the top of the fill stack. Different members of the family use additional criteria to further refine this ranking so that the volume of premature evictions from the upper part of the fill stack is minimized and capacity retention in the lower part of the fill stack is maximized.*

2.2 Dead Block Prediction LIFO

The dead block prediction LIFO (dbpLIFO) policy is perhaps the most intuitive one among the possible members of the pseudo-LIFO family. Traditional replacement policies based on dead block predictors evict the dead block closest to the LRU position in the access recency stack. The dbpLIFO policy victimizes the dead block that is closest to the top of the fill stack. We use the dead block predictor proposed in [9]. As a result, our design has storage overhead similar to that proposal. This drawback of dbpLIFO motivated us to explore other members of the pseudo-LIFO family with much less book-keeping overhead.

2.3 Probabilistic Escape LIFO

The design of the probabilistic escape LIFO (peLIFO) policy is based on the following definition of escape probability.

Escape Probability. *The escape probability, $P_e(k)$, for fill stack position $k \in [0, \mathcal{A}-1]$ in an $\mathcal{A}$-way set associative L2 cache is defined as the probability that cache blocks experience hits at fill stack positions bigger than k. This can be computed as the number of cache blocks experiencing at least one hit beyond fill stack position k divided by the total number of blocks filled into the cache. $P_e(\mathcal{A}-1)$ is zero by this definition.*

The peLIFO policy attaches higher eviction priority to the blocks residing closer to the top of the fill stack based on the escape probabilities of their fill stack positions. The peLIFO policy is composed of three components. The first component learns the escape probability of each fill stack position over a certain period of time. The second component identifies the top few fill stack positions where the escape probabilities undergo a sharp drop. These fill stack positions will be referred to as the escape points. Based on these escape points, the peLIFO policy takes a replacement decision. The escape points need to be recomputed only on phase changes. The third component of the peLIFO policy is responsible for detecting phase changes.

2.3.1 Learning Escape Probabilities

The escape probabilities can be maintained either for the entire cache or for a set (possibly singleton) of banks of the

[2] The single-use blocks do not participate in this statistic, since they do not experience any hit.

cache depending on the ease of implementation. We will assume that these probabilities are learned for a pair of adjacent banks as this is one of the modular ways of incorporating peLIFO in a multi-bank large last-level cache. An array of $\mathcal{A}$ saturating counters named *epCounter* ($\mathcal{A}$ is the associativity of the cache) per cache bank-pair maintains the hit counts. More specifically, *epCounter*[k] holds the number of blocks in the bank-pair experiencing at least one hit beyond fill stack position k. The *epCounter* array is updated as follows. Each block (or way) maintains its fill stack position when it experienced the last hit. On a hit to a block B currently at fill stack position k, all the *epCounter* locations starting from the last hit position of B up to $k-1$ are incremented. At this point the last hit position of B is updated to k. Note that when a block is filled, no counter is incremented, but the last hit position of the filled block is initialized to zero.

The escape probabilities are computed periodically on every N^{th} refill to the cache bank-pair based on the values in *epCounter*. The time period between two consecutive computations of the *epCounter* values will be referred to as an *epCounter* epoch or, simply, epoch. N is always chosen to be a power of two. We assume it to be 2^n in this discussion. The escape probabilities are computed in three steps as follows. First, each *epCounter* value is rounded to the next power of two, if it is not zero or already not a power of two. Then each positive value is replaced by its logarithm to the base two, leaving the zero values unchanged. Thus, a value in $(2^p, 2^{p+1}]$ will get replaced by $p+1$. Finally, these values are subtracted from n and the subtraction results are stored in the *epCounter* array. We note that these three steps are invoked only on every N^{th} refill to a cache bank-pair and that each of these three steps is fairly easy to realize in hardware. At the end of the third step, *epCounter*[k] holds $\log_2(1/P_e^*(k))$ for all positive $P_e^*(k)$ and n otherwise, where $P_e^*(k)$ is a rounded up over-estimate of $P_e(k)$. Notice that the rounding up step is important for avoiding expensive division operations when computing $P_e(k)$. These values are also copied to another array named *epCounterLastEpoch*. This array will be used for detecting phase changes. Henceforth, when we refer to the content of *epCounter*[k], we will mean $\log_2(1/P_e^*(k))$.

2.3.2 Computing Escape Points

By definition, $P_e(k)$ is a monotonically decreasing function of k. Therefore, the *epCounter* array holds monotonically increasing values, as discussed in the last section. For example, a sample of *epCounter* computed at the end of some *epCounter* epoch for 181.mcf looks like (9, 13, 14, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 16) for a 16-way set associative cache. Here the value of n is also 16. Thus, the probability that a block experiences a hit beyond fill stack position zero within the corresponding epoch is only $\frac{1}{2^9}$ implying that the majority of the blocks become dead at fill stack position zero. On the other hand, a sample of *epCounter* for 429.mcf looks like (1, 1, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 4, 5, 16). In this case, half of the blocks experience hits beyond positions zero and one, but only one-quarter experience hits beyond positions two to eight.

The knees of these *epCounter* values when plotted against k are the most interesting points. For example, the knee at position two of 429.mcf where the *epCounter* value changes from one to two implies that the escape probability observes a noticeable drop at fill stack position two. The next noticeable drop comes at fill stack position nine where the *epCounter* value changes from two to three. These knee positions can be seen as the most preferred eviction or escape points for cache blocks. For example, beyond fill stack position two in 429.mcf, only one-quarter blocks should continue down the stack and the rest can be evicted. We say that a trivial knee is located at position zero if the *epCounter* value at this position is positive. For example, in 429.mcf, half of the blocks can be evicted at fill stack position zero itself. Therefore, the idea would be to identify these escape points and evict an appropriate number of blocks at each escape point as the blocks travel down the fill stack. This is illustrated in Figure 5, which plots the *epCounter* sample of 429.mcf against the fill stack position. On each horizontal staircase of the graph, we show the expected fraction of blocks that should continue down the stack. The expected fraction of blocks that can leave the set at a knee is exactly equal to the difference of the fractions on two consecutive staircases around the knee.

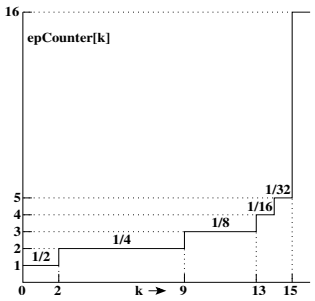

Figure 5: Shape of *epCounter* when plotted against fill stack position k. There are discontinuities at the knees.

We observed that three distinct escape points closest to the top of the fill stack are usually enough to fully capture the dynamics of the upper part of the fill stack for a 16-way set associative cache. However, instead of positioning the escape points at the knees, we position them at the middle of an *epCounter* cluster. We define an *epCounter* cluster as a set of consecutive fill stack positions holding a constant *epCounter* value. For example, in 429.mcf, there are six *epCounter* clusters: $[0,1]$, $[2,8]$, $[9,12]$, $[13,13]$, $[14,14]$, $[15,15]$. Here we express the clusters in terms of the starting and ending fill stack positions. The center of a cluster is defined as $\lfloor (startpos + endpos + 1)/2 \rfloor$, where *startpos* and *endpos* are the starting and ending fill stack positions of the cluster. Thus, the first escape point for 429.mcf is at position zero because *epCounter*[0] is non-zero. The second escape point is at position one, which is the center of the first cluster. The third escape point is at position five, which is the center of the second cluster.

Two corner cases need to be dealt with carefully. First, if the first few entries in the *epCounter* array are zero, the first escape point is positioned at the first knee, as opposed to at the center of this cluster. The center of this cluster cannot be a preferred exit point for the blocks because they will continue to enjoy hits beyond the center until they reach the first knee. For example, in 401.bzip2, one sample *epCounter* is (0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 4, 16). In this case, the first escape point is at position four, where half of the blocks become dead and can be evicted. The second corner case arises when two escape points coincide. Consider the sample *epCounter* contents of 181.mcf discussed earlier. The first escape point is at zero due to positive *epCounter*[0]. The second escape point is also at zero because this is the center of the first cluster. In such situations, where the currently computed escape point coincides with the last escape point, the currently computed one is incremented by one. Thus, for 181.mcf, the first three escape points are at fill stack positions zero, one, and two. Note that all the sets in the entire cache bank-pair will uniformly use these three escape points for deciding replacement candidates.

Once the escape points are computed, they can be used for cache block eviction. The idea would be to evict an

appropriate number of blocks at the most preferred escape point (among the three computed ones) as the blocks travel down the fill stack. To dynamically learn the most preferred escape point among the chosen three within a cache bank-pair, we use a set sampling (or set dueling) technique as proposed in [11]. We have four competing policies $\mathcal{P}_1, \ldots, \mathcal{P}_4$ with $\mathcal{P}_4$ being the LRU replacement policy. The policy $\mathcal{P}_i$, for $i \in \{1, 2, 3\}$, victimizes the block that is closest to the top of the fill stack and satisfies the following two criteria: 1) has not experienced a hit in its current fill stack position, and 2) its current fill stack position is bigger than or equal to the i^{th} escape point. If no block is found to satisfy these two criteria, the LRU block is chosen for eviction. Note that the first criterion guards against premature evictions, which may happen due to the probabilistic nature of the escape points. The set dueling technique for picking one of $\mathcal{P}_1, \ldots, \mathcal{P}_4$ is implemented as follows. Within each cache bank-pair we dedicate a small number of sample sets to each of these policies and each bank-pair can independently elect its most preferred policy. All the non-sampled sets follow the elected policy. To dynamically compute the most preferred policy, we maintain six saturating counters, each initialized to the mid-point, M, of its range. Let us refer to these counters as C_{ij} for $i < j$ and $i, j \in \{1, 2, 3, 4\}$. A particular counter C_{ij} decides the winner among $\mathcal{P}_i$ and $\mathcal{P}_j$. A miss in a set dedicated to $\mathcal{P}_i$ increments all counters C_{ij} and decrements all counters C_{ji}. For example, a miss in a sample belonging to $\mathcal{P}_2$ would decrement C_{12} and increment C_{23} and C_{24}. For $i < j$, the policy $\mathcal{P}_j$ is better than the policy $\mathcal{P}_i$ if $C_{ij} \geq M$; otherwise policy $\mathcal{P}_i$ is better. Therefore, it follows that for $i > j$, the policy $\mathcal{P}_j$ is better than the policy $\mathcal{P}_i$ if $C_{ji} < M$. A particular policy $\mathcal{P}_i$ is the overall winner if all the three counters that it participates in indicate so. For example, the policy $\mathcal{P}_2$ is the overall winner if $C_{12} \geq M$, $C_{23} < M$, and $C_{24} < M$. It is easy to prove that there will always be a total order among the policies.

2.3.3 Detecting Phase Changes

Preliminary experiments revealed that the escape points need recomputation only when the application undergoes a phase change. Interestingly, we find that the escape probabilities serve as excellent features for learning phase changes. To detect a phase change, we compare the contents of the *epCounter* array generated in this *epCounter* epoch (N refills) with the contents of the *epCounter* array saved from the last epoch. If any array location less than the third escape point has a difference of at least D, we declare a phase change. At this point, we revert the entire cache bank-pair to execute LRU policy for the next N refills and based on the hit counts collected during this period, we recompute the escape points and then transition to the peLIFO mode with all the counters C_{ij} reset to the mid-point of their range. The reason for reverting the entire cache bank-pair to execute LRU policy on a phase change is to give the lower part of the fill stack in each set a chance to get flushed out due to LRU replacement so that the new cache blocks accessed in the new phase can be brought in. While the escape points are computed only on phase changes, the escape probabilities have to be computed at the end of every N^{th} refill so that the phase changes can be detected correctly.

2.3.4 Putting It All Together

The entire cache bank-pair starts off in LRU mode and remains in the LRU mode for $N = 2^n$ refills until the escape probabilities and the escape points are computed. At this point, the cache bank-pair switches to the peLIFO mode and remains in this mode until a phase change is detected. In the peLIFO mode, the *epCounter* array is populated with hit counts only from the sampled sets dedicated to the LRU policy to make sure that the probabilities are computed from the prevailing baseline behavior only. Since the number of sets dedicated to each policy is small, the *epCounter* epoch in the peLIFO mode is set to $N' = 2^{n'} < 2^n$. Thus, the escape probabilities are computed only after the sets dedicated to LRU receive N' refills. At this point, the *epCounter* is compared with the *epCounter* from the last epoch provided the last epoch was also executed in the peLIFO mode. A phase change is declared if there exists k less than the third escape point such that $|epCounter[k] - epCounterLastEpoch[k]| \geq D$. Note that this comparison is done only after $epCounter[k]$ is updated to hold $\log_2(1/P_e^*(k))$. If a phase change is detected, the entire cache bank-pair is switched to the LRU mode and the entire cycle starts afresh. It is important to note that in the peLIFO as well as the LRU mode, when looking for a replacement candidate in a set, invalid ways are always filled first before invoking any replacement policy.

Adapting the peLIFO policy to CMPs does not require any change or addition to the design discussed here. The policy remains oblivious to the presence of multiple threads and continues to gather the escape probabilities from the behavior of each bank-pair of the shared L2 cache. The set dueling algorithm also remains oblivious to the presence of multiple threads and computes the preferred policy according to the discussion presented in Section 2.3.2. Understanding the necessity of making the peLIFO policy thread-aware is left to future research.

2.3.5 A Cousin of peLIFO

We have designed a simplified version of peLIFO that does not use set dueling (the LRU set samples are still used to compute the *epCounter* contents). The idea of this simplified policy is to fit the escape probability distribution in the long run. To understand this policy, let us refer to the *epCounter* knees of 429.mcf located at fill stack positions (2, 9, 13, 14, 15), excluding the trivial knee at position zero. According to the *epCounter* values of 429.mcf, once a cache set is completely filled (i.e., no invalid ways), on average half of the new population of blocks should continue past the first trivial knee at position zero, only one-quarter should continue past the second knee at position two, only one-eighth should continue past the third knee at fill stack position nine, etc. The simplified replacement policy tries to enforce this expected flow of blocks within each set. Let us consider a general *epCounter* array with χ non-trivial knees at positions $(K_0, \ldots, K_{\chi-1})$, where $K_i > 0$ for all i. Let the *epCounter* values at positions $(K_0 - 1, \ldots, K_{\chi-1} - 1)$ be $(e_0, \ldots, e_{\chi-1})$. These are the values just before the knee transitions take place. For example, these values for 429.mcf would be $(1, 2, 3, 4, 5)$. We maintain χ counters Q_i for $0 \leq i < \chi$ per cache bank-pair. All the counters Q_i belonging to a bank-pair are incremented modulo 2^{e_i} whenever a new block B is filled into the bank-pair. Next, K_i corresponding to the highest index i such that Q_i is zero is declared the earliest dead position of the filled block B. If no counter is having a zero content, the block is declared dead at position zero itself. Notice that this scheme, on average, enforces the probability distribution of the block flow as computed in the *epCounter* array. For example, in 429.mcf, on average, half of the filled blocks will have zero as their earliest dead position, a quarter will have K_0 (i.e., two) as their earliest dead position, etc. The replacement policy victimizes the block closest to the top of the fill stack satisfying the following two criteria: 1) has not experienced a hit in its current fill stack position (guards against premature eviction), and 2) its current fill stack position is bigger than or equal to its earliest dead position. We will refer to this policy as probabilistic counter LIFO (pcounterLIFO). The counters Q_i are initialized to $i \bmod 2^{e_i}$. The design choice of using simple modulo counters in place of good pseudo-random number generators makes pcounterLIFO less robust and it degrades the performance of a few workloads.

2.4 Implementation Overhead

A new replacement policy usually comes with two types of overhead, namely, extra storage needed for book-keeping and the impact on the critical path delay. All the members of the pseudo-LIFO family need to maintain the fill stack position of each block in a set requiring $\mathcal{A}\log_2(\mathcal{A})$ bits of storage per set (in addition to the bits needed by the baseline for maintaining the access recency stack position). The dbpLIFO policy requires the storage for maintaining the dead block predictor and some more information per block (such as the PC of the instruction filling the block, reference count, and dead bit) as discussed in [9]. The peLIFO policy needs to maintain an $\mathcal{A}$-entry *epCounter* array, an $\mathcal{A}$-entry *epCounterLastEpoch* array, three escape points, and six set dueling counters for each cache bank-pair. These have negligible storage overhead compared to the storage of the bank-pair (in our simulation one cache bank is 1 MB in size). In addition to these, the peLIFO policy needs to maintain the fill stack position of the last hit for each block within a set (requires $\mathcal{A}\log_2(\mathcal{A})$ bits per set) and one bit per block to mark if the block has seen a hit in the current fill stack position. The pcounterLIFO policy needs additional $\mathcal{A}\log_2(\mathcal{A})$ bits per set to remember the earliest dead position of a block assigned to it at the time of fill.

The fill stack position of a block in a set needs to be updated only on a cache fill to the set. Thus, the circuitry for updating the fill stack can be designed completely off the critical path. The computation of the replacement candidate can also be made off the critical path because these circuitries have to be invoked only on a cache fill. The requested L1 sector of the L2 cache block being filled can be forwarded to the L1 cache while the replacement decision in the L2 cache is being computed. Thus, the critical delay on the fill path remains unaltered. On an L2 cache hit, a range of consecutive locations in the *epCounter* array of the corresponding bank-pair will have to be incremented. But this can be comfortably overlapped with the data array access after the hit is detected in the tag array (we assume serial tag/data access in each L2 cache bank).

3. SIMULATION ENVIRONMENT

We simulate two types of systems, namely, one with a single core and the other with multiple cores. Table 1 presents the relevant details of our MIPS ISA-based single-core out-of-order issue baseline system. The cache latencies are determined using CACTI [2] assuming a 65 nm process. We assume serial tag/data access in the L2 cache. The baseline L2 cache has two banks each of size 1 MB with 16-way set-associativity and 128-byte block size. This configuration is shown in Figure 6(a). This single-core simulation environment is used to evaluate our replacement policies on the single-threaded applications. However, the same memory-side configuration is used for both single-core and multi-core environments. We simulate four integrated memory controllers clocked at 2 GHz (half the core frequency) and each connected to a four-way banked memory module via a 64-bit channel. The <controller, bank> id of a physical address is determined with the help of the XOR scheme proposed in [16]. The memory module can transfer 64 bits to the memory controller on both edges of a 400 MHz clock. The DRAM access time to the first 64 bits within a requested 128-byte cache block is 80 ns and on each subsequent 400 MHz clock edge, one eight-byte packet arrives at the memory controller. The memory controller switches each packet on the front-side bus.

We evaluate our policies on four-way multiprogrammed workloads by simulating a quad-core CMP with an 8 MB 16-way set associative shared L2 cache. Similarly, the policies are evaluated on eight-way multi-threaded applications by simulating an eight-core CMP with a 4 MB 16-way set associative shared L2 cache. Both these configurations use

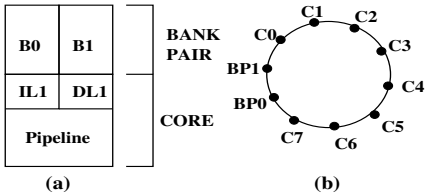

Figure 6: (a) A two-bank single-core architecture. (b) An architecture with more than two banks and/or more than one core. In this particular case, we show eight cores (C0 to C7) and four L2 cache banks (organized as two bank-pairs: BP0 and BP1) on a bidirectional ring.

Table 1: Simulated single-core baseline system

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

the core depicted in Table 1 as the basic building block. The shared L2 cache is composed by connecting a pair of 1 MB 16-way set-associative banks to a switch on a bidirectional ring as shown in Figure 6(b). Each core with its own L1 caches (same configuration as in Table 1) connects to the ring via a switch. 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 directory-based MESI coherence protocol [8]. Each L2 cache block maintains a directory entry with a sharer vector and the necessary states. In both single-core and multi-core configurations, the cache hierarchy maintains inclusion. In all the simulations, the virtual pages are mapped to the physical pages using a demand-based bin-hopping algorithm [6].

We carry out our preliminary evaluations presented in this article on a small set of single-threaded, multiprogrammed,

Table 2: MPKI of single-threaded applications

171.swim	5.2	401.bzip2	1.0
172.mgrid	1.5	403.gcc	1.0
173.applu	2.3	429.mcf	22.2
179.art	16.8	433.milc	8.8
181.mcf	50.8	462.libq	5.7
183.equake	7.0	470.lbm	8.4
254.gap	1.8	482.sphinx	4.4

Table 3: Multiprogrammed and multi-threaded workloads

Multiprogrammed	
183, 429, 462, 482 (MIX1)	171, 181, 470, 482 (MIX7)
181, 183, 433, 482 (MIX2)	171, 183, 254, 462 (MIX8)
181, 183, 429, 482 (MIX3)	172, 254, 403, 429 (MIX9)
181, 183, 470, 482 (MIX4)	172, 254, 401, 462 (MIX10)
172, 403, 429, 462 (MIX5)	172, 254, 403, 433 (MIX11)
181, 183, 429, 462 (MIX6)	173, 403, 462, 470 (MIX12)

Multi-threaded	
Application	Problem size
Art	MinneSPEC
Equake	MinneSPEC, ARCHduration 0.5
FFT	256K points
FFTW	4096×16×16 points
Ocean	258×258 grid
Radix	2M keys, radix 32

and multi-threaded workloads. We choose fourteen single-threaded applications (shown in Table 2) that have at least one miss per kilo instructions (MPKI) on the baseline 2 MB 16-way set associative L2 cache when simulated for a representative set of one billion dynamic instructions selected by the SimPoint toolset [12]. All the applications are drawn from the SPEC 2000 and SPEC 2006 benchmark suites, and they run on the `ref` input set. We include both the versions of mcf, namely, 181.mcf and 429.mcf, in the evaluation because they exhibit very different cache behavior, as already discussed in Section 2.3.2 and shown in Table 2.

We use twelve four-way multiprogrammed workloads shown in the upper half of Table 3 (we only show the SPEC benchmark id for each member of a mix). Each workload mix is simulated until each thread commits the representative one billion dynamic instructions. A thread that completes this representative set early continues execution so that we can correctly simulate the cache contention for all the threads. However, all results are reported by taking into account only the first one billion committed instructions from each thread. We report normalized average CPI i.e.

$$(\frac{1}{4}\sum_i CPI_i^{NEW})/(\frac{1}{4}\sum_i CPI_i^{LRU}),$$

where the CPI of each thread i is computed based on the cycles taken by it to commit its first one billion instructions in the multiprogrammed mix. This metric captures the reduction in average turnaround time of a mix when a new policy "NEW" replaces the baseline LRU policy. We also evaluate a strong fairness metric, which captures the minimum benefit observed by a thread in a mix. It is given by $\max_i \frac{CPI_i^{NEW}}{CPI_i^{LRU}}$. A policy "NEW" is at least as fair as the baseline LRU policy if this fairness metric is less than or equal to one meaning that no thread in a mix observes a slowdown due to introduction of the policy "NEW". This metric captures single-stream performance in a multiprogrammed environment. Taken together, these two metrics summarize the overall performance of a mix as well as the lower bound performance of individual threads in the mix.

We also evaluate our policies on six multi-threaded programs shown in the lower half of Table 3. These applications use hand-optimized array-based queue locks and scalable tree barriers. All these applications are run to completion.

We evaluate dbpLIFO, peLIFO, and pcounterLIFO from the pseudo-LIFO family. The dbpLIFO policy uses a dead block predictor similar to the one proposed in [9]. This policy maintains 19 bits of extra state per cache block compared to the baseline: the lower eight bits of the PC (after removing the least significant two bits), six bits of reference count, a dead bit, and four bits for the fill stack position. Each 1 MB L2 cache bank is equipped with a 2048-entry history table per core for carrying out dead block prediction. The history table is indexed with eight bits of PC concatenated after three lower bits of the block address (after removing the bank id bits). Each history table entry contains a valid bit, six bits of reference count, six bits of filter reference count, and a hysteresis bit (see [9] for detail). The traditional dead block predictor-based replacement policy requires 37 KB, 232 KB, and 172 KB of auxiliary storage for our single-threaded, multiprogrammed, and multi-threaded simulation environments, respectively. For dbpLIFO, these requirements are 45 KB, 264 KB, and 198 KB, respectively.

The peLIFO policy maintains only nine bits of extra state per cache block compared to the baseline: one bit to record hit in the current fill stack position, four bits for last hit position, and four bits for the current fill stack position. This policy has less than 1% storage overhead. The pcounterLIFO policy requires four additional bits per cache block to remember the block's earliest dead point assigned to the block at the time of fill. Overall, the storage overhead of the peLIFO policy is 18 KB, 72 KB, and 36 KB for our single-threaded, multiprogrammed, and multi-threaded simulation environments, respectively. For the pcounterLIFO policy, these requirements are 26 KB, 104 KB, and 52 KB, respectively. The logic complexity of the peLIFO policy is expected to be higher than the pcounterLIFO policy because of the set sampling technique employed by the former.

The peLIFO policy dedicates sixteen set samples per bank-pair to each of the four competing policies (note that a bank-pair has 1024 sets). This policy uses four times the number of blocks in a bank-pair as its *epCounter* epoch length in the LRU mode (this is N of Section 2.3.4). This is 2^{16} for our configurations. In the peLIFO mode, the *epCounter* epoch length is set to 2^{13} (this is N' of Section 2.3.4). The phase change detection threshold D is set to four (see Section 2.3.4). The policy chooser counters are all 31 bits in size. Each entry of the *epCounter* array is 16 bits in size.

We compare our policies with configurations having a fully associative 16 KB victim cache per 1 MB L2 cache bank. Thus, an 8 MB shared L2 cache would get a total of 128 KB victim caching space. The victim cache exercises a non-most-recently-filled-random replacement policy. We also compare our policies with DIP, TADIP, UCP, PIPP, and ASP. For DIP, we use 1/32 as the BIP epsilon and 32 dedicated set samples per 1 MB bank. For TADIP, we use the same BIP epsilon and eight dedicated set samples per bank-pair per policy per thread. Even though the original TADIP proposal uses only one set sample per policy per thread, our experiments show that increasing the number of set samples per policy per thread up to a certain limit improves performance. For UCP, the partitions are recomputed every five million processor cycles using the greedy partitioning algorithm proposed in [10]. For PIPP, we use 3/4 as the promotion probability for non-streaming threads and 1/128 for the streaming threads. These probabilities are simulated with the help of the `random()` function of the `C` library. The streaming threads are identified using the same criteria and parame-

ters proposed in [15]. In ASP, the POP cache per thread is 32 KB for the four-way multiprogrammed workloads, while for the eight-way multi-threaded ones it is 16 KB. The total POP cache budget is kept fixed at 128 KB throughout the evaluation. The ownership hand-over threshold and confidence counter size are same as in the original proposal [13].

4. SIMULATION RESULTS

The simulation results are presented in four sections. Sections 4.1, 4.2, and 4.3 present the results for single-threaded applications, multiprogrammed workloads, and multi-threaded shared memory applications, respectively. Section 4.4 presents the performance of peLIFO in the presence of aggressive multi-stream stride prefetching.

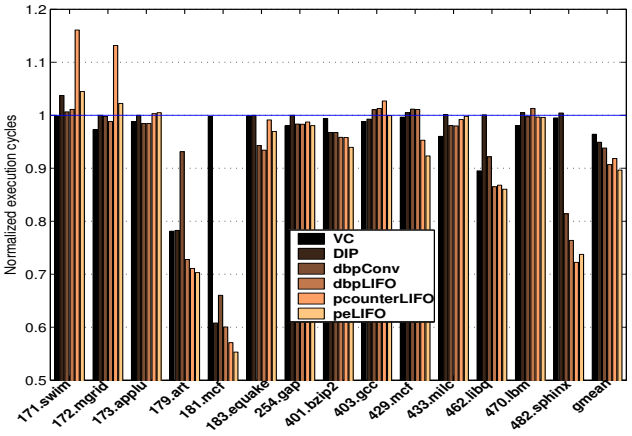

Figure 7: Performance of the single-threaded applications.

4.1 Single-threaded Applications

Performance analysis of the single-threaded applications, even in today's multiprocessor environments, has the potential to offer important insights into the working of the proposed policies. Figure 7 presents the execution time of the fourteen single-threaded applications running with six different L2 cache optimization schemes, namely, victim caching (VC), dynamic insertion policy (DIP), conventional dead block prediction (dbpConv) [9], dbpLIFO, pcounterLIFO, and peLIFO. The execution time is normalized to the baseline LRU policy. The difference between dbpConv and dbpLIFO is only in the selection of the dead block victim (see Section 2.2). As can be approximately deduced from Figure 2 in Section 1, traditional small fully associative victim caches are unlikely to offer much benefit. Overall, victim caching reduces execution time by 3.5% on average (see the gmean group of bars). Only 179.art enjoys a noticeable reduction (21.9%) in execution time. DIP is effective only in 179.art, 181.mcf, and 401.bzip2. This result can be explained with the L2 cache block reuse data presented in Figure 3(a) of Section 1. Overall, DIP is able to reduce the execution time by 5.1%, on average. As expected, dbpLIFO outperforms dbpConv. In fact, in 179.art, 181.mcf, 462.libquantum, and 482.sphinx3, these two policies have noticeable difference in performance. Overall, dbpConv reduces the execution time by 6.2%, while the corresponding saving in dbpLIFO is 9.3%. The pcounterLIFO policy comes quite close to dbpLIFO on average (8.2% reduction in execution time), but severely hurts the performance of 171.swim and 172.mgrid. Finally, peLIFO emerges the best policy across the board saving 10.3% execution time on average. Noticeable performance improvement can be seen in 179.art, 181.mcf, 401.bzip2, 429.mcf, 462.libquantum, and 482.sphinx3. The most encouraging result is that peLIFO performs equally well as the storage-heavy dbpLIFO.

Table 4: Top four escape points in peLIFO

App.	Top four escape points
171.swim	2 (0.26), 1 (0.26), 15 (0.22), 3 (0.14)
172.mgrid	15 (0.63), 13 (0.07), 14 (0.06), 11 (0.05)
173.applu	2 (0.31), 1 (0.28), 15 (0.23), 3 (0.16)
179.art	2 (0.84), 6 (0.09), 0 (0.02), 15 (0.01)
181.mcf	0 (0.59), 1 (0.31), 15 (0.03), 2 (0.02)
183.equake	1 (0.94), 15 (0.02), 5 (0.02), 2 (0.01)
254.gap	1 (0.49), 2 (0.42), 15 (0.08), 0 (<0.01)
401.bzip2	4 (0.51), 7 (0.25), 15 (0.02), 14 (0.02)
403.gcc	11 (0.22), 15 (0.13), 10 (0.13), 13 (0.08)
429.mcf	1 (0.78), 5 (0.12), 2 (0.04), 0 (0.01)
433.milc	1 (0.58), 15 (0.20), 2 (0.08), 14 (0.06)
462.libq	1 (0.95), 15 (0.03), 2 (<0.02), 0 (<0.01)
470.lbm	2 (0.87), 3 (0.05), 1 (0.05), 15 (0.02)
482.sphinx	2 (0.77), 1 (0.15), 15 (0.02), 6 (0.02)

To further understand the characteristics of peLIFO and its differences with DIP, in Table 4, we present the top four fill stack positions where most evictions took place in peLIFO during the simulations. We also show the fraction of evictions for each of these escape points within parentheses. Recall that the position zero corresponds to the top of the fill stack and the position fifteen corresponds to the bottom of the fill stack in a 16-way set associative cache. The pseudo-LIFO nature of peLIFO emerges clearly from the high eviction fractions seen in the upper part of the fill stack (except for 172.mgrid and 403.gcc). Only 181.mcf shows position zero as the most preferred escape point. However, in addition to a 59% eviction at position zero, a 31% eviction from position one in 181.mcf allows peLIFO to go beyond DIP in retaining working sets because DIP relies on evictions at position zero only.

The applications that successfully retain cache blocks in the lower part of the fill stack (indicated by a high fraction of evictions from the upper part of the fill stack), but fail to observe any benefit from peLIFO bring out an important drawback of this policy. Although the peLIFO policy successfully finds out cache space for capacity retention, it does not necessarily retain the "hot" blocks that would maximize the number of hits in the L2 cache. In other words, the current design does not do anything to improve the quality of retention; it only partitions each L2 cache set into two logical halves, one to satisfy the near-term uses and the other to satisfy a subset of far-flung uses.

4.2 Multiprogrammed Workloads

We present the detailed performance and fairness results for the multiprogrammed workloads in Figures 8 and 9. In part (a) of these figures, we include the results for victim caching (VC), thread-aware dynamic insertion policy with feedback (TADIP), utility-based cache partitioning (UCP), adaptive set pinning (ASP), and promotion/insertion pseudo-partitioning (PIPP). In part (b) of these figures, we evaluate dbpConv, dbpLIFO, pcounterLIFO, and peLIFO.

First, let us discuss the performance results. Traditional victim caching with small (16 KB) fully associative victim caches per 1 MB cache bank fails to reduce the average CPI much (1.1% on average). TADIP also turns out to be ineffective, as can be inferred from the L2 cache block reuse data in Figure 3(b). UCP performs the best across the board among the policies in Figure 8(a). On average, it reduces the average CPI by 15.4%. ASP is able to improve performance of MIX2 and MIX6 only, but severely hurts the performance of others. Overall, it increases the average CPI by 19.6% compared to the baseline LRU policy. We identified two major drawbacks in ASP. First, once a set is occupied by a thread, other threads cannot allocate blocks in that set

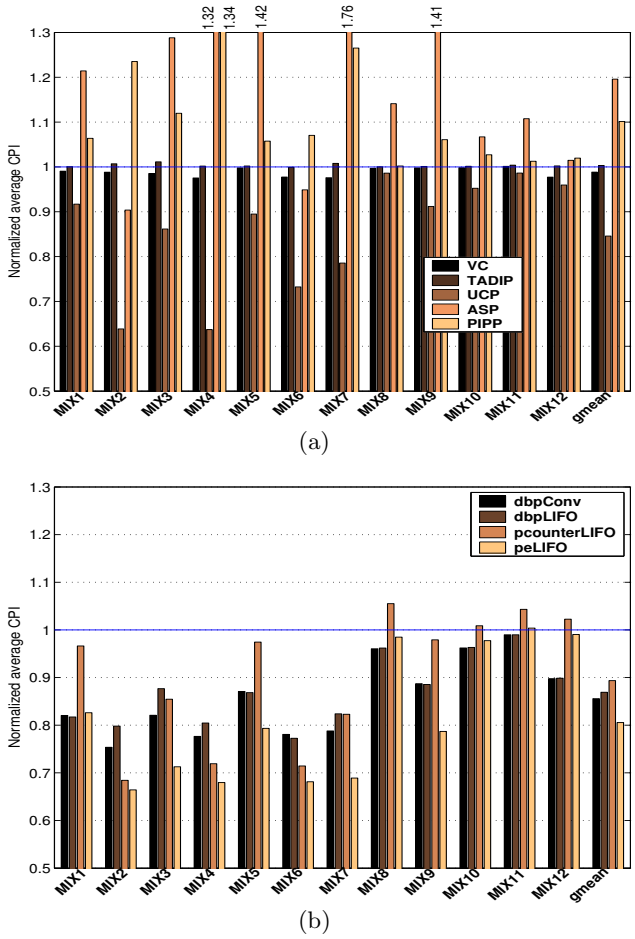

Figure 8: Performance of the multiprogrammed workloads.

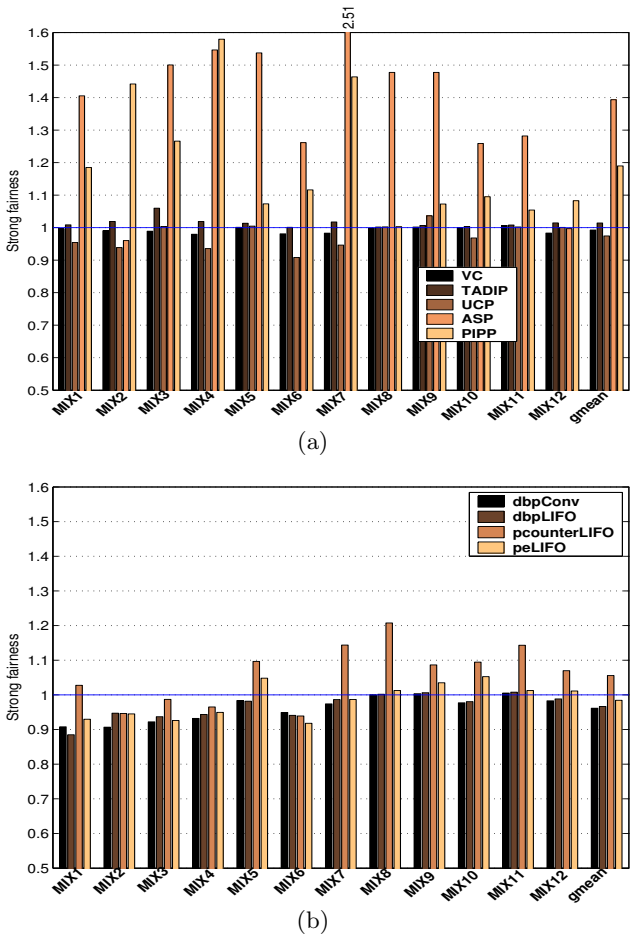

Figure 9: Fairness evaluation of the policies.

even though there are invalid ways in that set. This leads to very high congestion in the POP caches. Second, the set ownership hand-over policy must implement some fairness mechanism because the threads with moderate amount of cache demands most often lose a free set to another thread with high demands and have to remain satisfied with the POP cache space only. This is reflected in the poor POP cache hit rates, as shown in Table 5 (see the POP column, which lists the ratio of the total number of POP cache hits across all the threads to the number of POP cache accesses from all the threads). PIPP also exhibits poor performance. We observed that for these workloads, the congestion toward the tail of the access recency stack leads to high volume of conflict misses in PIPP. In such situations, it is important to get promoted out of the tail as soon as possible. Our preliminary experiments with the promotion probabilities and promotion steps point to a design that incorporates algorithms to dynamically learn these parameters.

Turning to Figure 8(b), we find that dbpConv and dbpLIFO perform equally well reducing the average CPI by about 14%, on average, dbpConv being slightly better. The peLIFO policy again emerges the best across the board reducing the average CPI by 19.4%, on average. It is important to note that peLIFO outperforms dbpConv and dbpLIFO on several workloads by large margins. We found that the auxiliary storage allocated to the dead block predictor is still not enough to realize the full potential of dbpConv and dbpLIFO on the selected set of one dozen workloads.

Table 5 shows the top four eviction positions in the fill stack for peLIFO. None of the workloads show position zero as the most preferred escape point, rendering TADIP ineffective for these workloads.

Finally, the fairness data shown in Figure 9 follow the trend of the performance data. UCP, dbpConv, dbpLIFO, and peLIFO are the policies with maximum fairness. It is important to note that pcounterLIFO loses out in fairness by a large margin, even though it delivers good average performance (10.6% reduction in average CPI).

Some of the policies evaluated so far were evaluated on cache hierarchies with one single block size across all levels of the hierarchy in their original proposals. This can have a great impact on performance of the policies that rely on a small number of block reuse in the L2 cache. For example,

Table 5: Characteristics of peLIFO and ASP

Mix	Top four escape points	POP
MIX1	4 (0.47), 1 (0.37), 0 (0.10), 10 (0.02)	0.62
MIX2	1 (0.51), 3 (0.26), 0 (0.16), 9 (0.02)	0.58
MIX3	2 (0.82), 1 (0.08), 0 (0.04), 7 (0.02)	0.41
MIX4	1 (0.63), 5 (0.27), 2 (0.05), 3 (0.01)	0.51
MIX5	6 (0.93), 1 (0.03), 14 (0.01), 15 (0.01)	0.53
MIX6	3 (0.71), 1 (0.16), 0 (0.08), 9 (0.02)	0.51
MIX7	3 (0.68), 2 (0.20), 1 (0.03), 5 (0.01)	0.47
MIX8	7 (0.43), 8 (0.43), 1 (0.03), 6 (0.02)	0.72
MIX9	1 (0.76), 6 (0.15), 0 (0.01), 2 (0.01)	0.72
MIX10	1 (0.79), 6 (0.08), 0 (0.06), 7 (0.01)	0.77
MIX11	3 (0.87), 9 (0.04), 1 (0.01), 8 (0.01)	0.66
MIX12	2 (0.70), 4 (0.17), 1 (0.04), 3 (0.04)	0.72

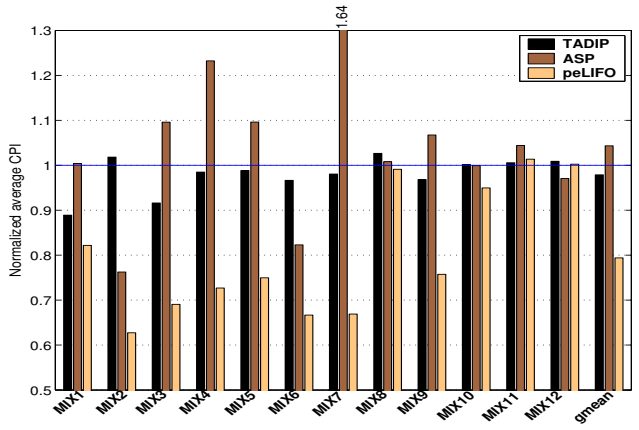

Figure 10: Performance of the multiprogrammed workloads with equal L1 and L2 cache block sizes (128 bytes each).

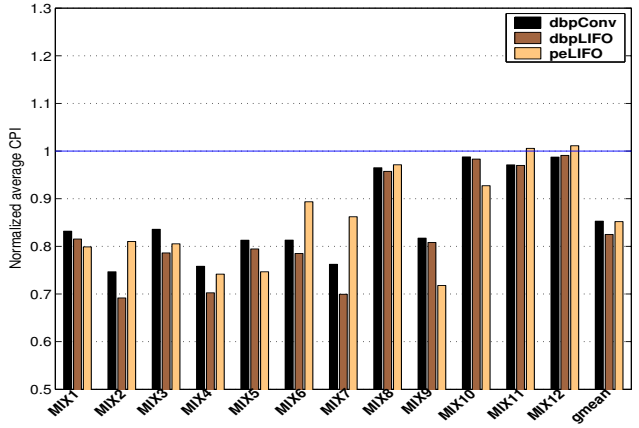

Figure 11: Performance of the multiprogrammed workloads with 16 MB L2 cache.

with a smaller number of L2 cache accesses per set, the set ownership hand-over in ASP gets accelerated. With equal L1 and L2 cache block sizes, the number of accesses per block (and per set) in the L2 cache usually drops because an entire L2 cache block is brought into the L1 cache in one shot. In Figure 10, we evaluate TADIP, ASP, and peLIFO on a cache hierarchy with L1 and L2 cache block size equal to 128 bytes.[3] Our other experiments with smaller equal block sizes, such as 64 bytes (not shown here), reveal that the baseline performance degrades sharply due to loss of opportunity in exploiting spatial locality in the L2 cache. In Figure 10, while the situation has dramatically improved in some of the workloads for TADIP and ASP, the overall performance is still nowhere close to peLIFO.

Finally, to confirm the hypothesis that dbpLIFO does gain importance over dbpConv as the cache capacity increases and the near-term conflicts reduce in volume, Figure 11 shows a performance comparison between dbpConv, dbpLIFO, and peLIFO on a 16 MB 16-way set associative shared L2 cache. Since we allocate a 2048-entry dead block history table for each core per 1 MB L2 cache bank (see Section 3), the total auxiliary storage allocated for dbpConv and dbpLIFO also increases proportionately with cache capacity. In several workloads, dbpLIFO performs better than dbpConv on a 16 MB cache. This is an interesting design point because dbpConv and peLIFO perform equally well. However, dbpConv uses 464 KB of auxiliary storage, while the requirement of peLIFO is only 144 KB. Once the dead block predictor has enough storage to perform well, the choice of the dead block victim within a set becomes important and is reflected in the performance difference between dbpConv and dbpLIFO. Although the average performance gap between dbpLIFO and dbpConv is still not significant, we expect this gap to grow as the L2 cache capacity increases.

[3] We also simulated PIPP on this configuration, but did not observe any noticeable change in the average performance.

4.3 Shared Memory Parallel Applications

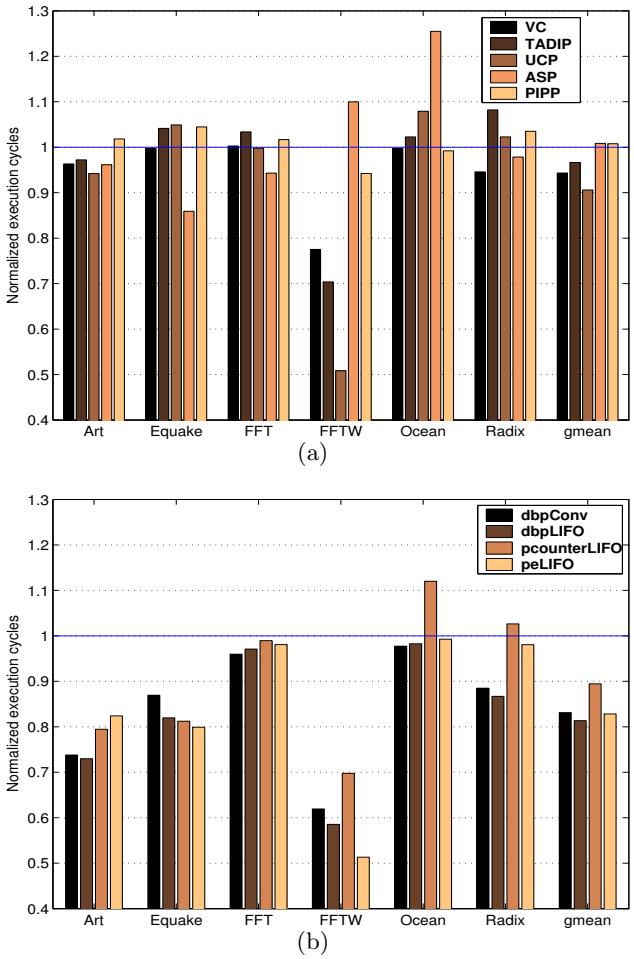

Figure 12: Performance of the multi-threaded workloads.

Figure 12 evaluates the performance of various policies on the multi-threaded applications. Victim caching (VC) performs reasonably well on FFTW reducing the execution time by 22.5% (see Figure 12(a)). TADIP also reduces the execution time of FFTW by nearly 30%. This is supported by the L2 cache block reuse data of Figure 3(b) in Section 1. UCP performs surprisingly well on FFTW reducing the execution time by almost 50%. ASP shows noticeable performance improvement in Art, Equake, and FFT. The dbpConv, dbpLIFO, and peLIFO policies perform remarkably well across the board (see Figure 12(b)). While dbpLIFO emerges the best among all the policies reducing execution time by 18.6% on average, peLIFO comes close saving 17.2% execution time on average. It is important to note that peLIFO has less than 1% storage overhead, while dbpLIFO is provisioned with large history tables for carrying out dead block prediction. In these applications, UCP falls significantly behind peLIFO in overall performance be-

cause the former slows down Equake, Ocean, and Radix due to thrashing within the threads' partitions.

4.4 Interaction with Prefetching

We conclude the discussion on the simulation results by presenting the performance of peLIFO in the presence of prefetching. We observe that the accesses seen by the L2 cache from a core are often a re-ordered version of the original stream. This re-ordering 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 enable proper learning of the stride pattern, we integrate a multi-stream stride prefetcher with each core's L1 cache controller (as opposed to L2 cache controller). Each prefetcher keeps track of sixteen simultaneous load and store streams and prefetches into the L1 caches. The stream size is chosen to be 4 KB matching 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 either on an L1 cache hit to a prefetched block or on an L1 cache miss and stays six strides ahead in each stream. Note that the L1 cache prefetch requests that miss in the L2 cache automatically prefetch into the entire cache hierarchy.

Our simulation results show that (details omitted in the interest of space), on average, peLIFO continues to reduce execution cycles by 12.2% for the single-threaded applications, 19.2% for the multiprogrammed workloads, and 15.3% for the multi-threaded applications compared to a baseline that uses the same multi-stream stride prefetcher. We briefly mention an interesting phenomenon that we observed in peLIFO after turning on prefetching. The prefetcher hurts the baseline performance of some of the single-threaded and multiprogrammed workloads. However, when the prefetcher is turned on in the presence of peLIFO, none of the workloads suffer from any slowdown compared to the baseline peLIFO. The peLIFO policy victimizes L2 cache blocks as early as possible from the upper part of the fill stack within a set. As a result, in the applications where accurate prefetching is hard, the blocks prefetched into the L2 cache due to wrong address prediction in the prefetcher quickly get evicted from the set because they do not get used. These blocks get evicted from the L1 caches as well due to L1-L2 inclusion. The end-result is that unnecessary blocks do not hold up cache space for long. This improves the overall effectiveness of prefetching in a number of single-threaded and multiprogrammed workloads.

5. SUMMARY

We have presented a new family of replacement policies named pseudo-LIFO for the last-level caches. The members of this family evict blocks from the upper part of the fill stack, thereby retaining a large fraction of the working set in the cache. We have discussed the design of three members of this family. The first member, named dead block prediction LIFO, leverages the existing dead block predictors to victimize the dead blocks residing close to the top of the fill stack. The second member, named probabilistic escape LIFO, dynamically learns the most preferred eviction positions within the fill stack and prioritizes the ones close to the top of the stack. The third member, named probabilistic counter LIFO, is a simple derivative of probabilistic escape LIFO. The probabilistic escape LIFO policy, while using less than 1% of extra storage for book-keeping, outperforms a large array of contemporary proposals on a selected set of single-threaded, multiprogrammed, and multi-threaded workloads. This preliminary study points to more rigorous evaluation of the pseudo-LIFO family on more realistic workloads.

6. REFERENCES

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

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

[3] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. In *Proc. of the 29th Intl. Symp. on Computer Architecture*, pages 209–220, May 2002.

[4] A. Jaleel et al. Adaptive Insertion Policies for Managing Shared Caches. In *Proc. of the 17th Intl. Conf. on Parallel Architecture and Compilation Techniques*, pages 208–219, October 2008.

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

[6] 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.

[7] A-C. Lai, C. Fide, and B. Falsafi. Dead-block Prediction & Dead-block Correlating Prefetchers. In *Proc. of the 28th Intl. Symp. on Computer Architecture*, pages 144–154, June/July 2001.

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

[9] H. Liu et al. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *Proc. of the 41st Intl. Symp. on Microarchitecture*, pages 222–233, November 2008.

[10] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proc. of the 39th Intl. Symp. on Microarchitecture*, pages 423–432, December 2006.

[11] M. K. Qureshi et al. Adaptive Insertion Policies for High Performance Caching. In *Proc. of the 34th Intl. Symp. on Computer Architecture*, pages 381–391, June 2007.

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

[13] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors. In *Proc. of the 13th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 135–144, March 2008.

[14] D. K. Tam et al. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *Proc. of the 14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 121–132, March 2009.

[15] Y. Xie and G. H. Loh. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-core Shared Caches. In *Proc. of the 36th Intl. Symp. on Computer Architecture*, pages 174–183, June 2009.

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