Re-Examining Cache Replacement Policies

Jason Zebchuk
Dept. of Electrical and Computer Engineering
University of Toronto
zebchuk@eecg.toronto.edu

Srihari Makineni and Don Newell
Systems Technology Lab
Intel Corporation
{*srihari.makineni, donald.newell*}*@intel.com*

Abstract— **The replacement policies commonly used in modern processors perform an average of 57% worse than an optimal replacement policy for commercial applications using large, shared caches in a chip-multiprocessor (CMP). Recent proposals that improve the performance of smaller, uniprocessor caches with SPEC CPU workloads do not achieve similar benefits with commercial workloads and larger caches , even though these caches still perform worse than optimal. The recently proposed Shepherd Cache replacement policy reduces miss-ratios by 7.3% on average, but it relies on an impractical LRU policy and requires 5.3% overhead relative to the total cache capacity. We propose two new, practical, low-overhead replacement policies that mimic Shepherd Cache with significantly less meta-data overhead. First, we propose a Lightweight Shepherd Cache design that reduces miss-ratios by 8% on average and up to 19%, while requiring only 1.9% meta-data overhead . We also propose an Extra-Lightweight Shepherd Cache design that reduces overhead to only 0.5% when combined with a practical Clock replacement policy while reducing miss-ratios by an average of 5.4% and up to 14%.**

I. Introduction

In the foreseeable future, the current trend towards chips with more cores and larger shared caches is expected to continue. Besides capacity and associativity, the replacement policy has an important impact on performance. Despite this, modern caches still use replacement policies that were tuned for relatively small uniprocessor caches. Even recent research on cache replacement policies (e.g. [1], [2], [3]) continues to use relatively small, 128KB to 2MB caches evaluated with SPEC CPU workloads. This work evaluates different replacement policies for 4MB to 32MB shared caches with commercial workloads. We focus on improving cache miss-ratios with practical, low-overhead replacement policies.

Our initial investigations evaluate a few common, well-known replacement policies, and demonstrate that they result in miss-ratios that are as much as twice the miss-ratio obtained when using Belady's MIN algorithm [4]. While the future knowledge used by this algorithm makes it unrealistic, it still indicates the potential for significantly lower miss-ratios over conventional replacement policies.

Recently, Rajan and Govindarajan introduced the Shepherd Cache (SC) design that attempts to mimic the decisions made by an optimal replacement policy [2]. SC requires a significant amount of meta-data overhead. For example, the meta-data overhead for a 4MB cache is 219KB, or 5.3% of the total cache capacity. This meta-data consumes not only significant on-chip area, but also significant power as a large portion of the replacement meta-data must be read for each cache access. In addition, the SC design incorporates a baseline LRU replacement policy that is impractical to implement for commercial designs with highly associative caches. Our new designs follow the same approach as the shepherd cache, but with a focus on reducing meta-data overhead and on making the design practical.

As an alternative to true LRU, commercial designs often approximate them with more practical pseudo-LRU replacement policies. However, we demonstrate that these approximations can increase miss-ratios up to 9% compared to true LRU. As a result, this work takes into consideration the practicality of implementing true LRU replacement policies and the performance impact of using more practical alternatives.

We present two new designs based on the original Shepherd Cache concept. The first design, the *Lightweight Shepherd Cache* (SC-L), directly mimics the original shepherd cache over 98% of the time, while requiring 63% less meta-data overhead. The second design, the *Extra-Lightweight Shepherd Cache* (SC-XL), attempts to capture only the essential characteristics of the original design while requiring minimal amounts of meta-data overhead. Comparing the different designs, the original Shepherd Cache requires 438 bits of meta-data per set, representing an overhead of 5.3% of the cache data capacity, and reduces miss-ratios by an average of 7.3% compared to LRU replacement. SC-L requires just 1.9% meta-data overhead and reduces miss-ratios by 8.2% on average. The SC-XL design combined with a baseline clock replacement policy requires just 0.5% overhead, while reducing miss-ratios by an average of 5.4% across all workloads and cache sizes.

The remainder of the paper is organized as follows: Section II describes several existing replacement policies and other related work. Section III describes the two new replacement policies. Section IV evaluates the different replacement policies. Finally, Section V summarizes and concludes the work.

II. Background and Related Work

A. Non-Adaptive Replacement Policies

Most caches use one of a small number of well-known cache replacement policies. The most well-known policy is the least-recently used (LRU) policy. Others include pseudo-LRU algorithms (PLRU) that approximate LRU, least-frequently used (LFU), the clock algorithm (Clock) [5], random replacement, and the "not most recently used" (Not-MRU) policy.

LRU uses a stack to track the order of the most recent accesses to a set of cache lines. Upon an access to a line, the line moves to the top of stack, indicating that it is the most recently used (MRU) line. When choosing a victim line for replacement, the line at the bottom of the stack, the least-recently used (LRU) line, is chosen. In the traditional LRU policy, new cache lines are placed at the top of the stack in the MRU position; we refer to this as the MRU insertion policy (MIP). For an N-way set-associative cache, each set requires at least $log_2(N!)$ bits to maintain an LRU stack. While this overhead is manageable for associativities of four or less, it quickly becomes impractical for higher associativities. As an alternative, a number of pseudo-LRU algorithms have been proposed that approximate the LRU stack with less overhead [6]. The LRU and pseudo-LRU algorithms exploit temporal and spatial locality in programs and attempt to keep recently accessed lines in the cache.

The LFU policy maintains an access count for each cache line in a set. When choosing a victim for replacement, the line with the lowest count (i.e., fewest accesses) is evicted. LFU policies typically also implement an aging policy that automatically reduces access counts over time to prevent cache pollution from stale lines.

The clock algorithm [5] is a well-known operating system page replacement policy, but it is equally applicable to processor caches. In this policy, the "hand" of the clock is a pointer to one of the cache lines in a set. Each cache line has a single *touched* bit. When a new line is allocated, the line pointed to by the hand is examined. If the touched bit for this line is not set, then it is chosen as the victim. Otherwise, the bit is reset and the hand is incremented to point to the next cache line. This process repeats until a line with a cleared bit is found and evicted. The *touched* bit for a line is set upon an access to that line, including the initial access. Despite being more complex than other algorithms, the Clock algorithm benefits from low area overhead that grows linearly with the cache associativity. Also, although conceptually Clock uses a sequential process, it can be implemented in a single cycle.

Random replacement and Not-MRU are two very simple policies. Random replacement, as its name suggests, simply selects a victim at random. This requires no additional metadata and can use a linear feedback shift register to select victim cache lines. As a slight improvement, the Not-MRU policy tracks the most recently used (MRU) line and evicts a random line other than the MRU line. This policy has been shown to out-perform random for some workloads.

B. Adaptive Insertion Policies

In a recent work, Qureshi et al. [1] propose modifying the traditional LRU replacement policy by changing where new cache lines are inserted into the LRU stack. They propose an LRU insertion policy (LIP), a bimodal insertion policy (BIP), and a dynamic insertion policy (DIP).

The work is motivated by the observation that the traditional LRU policy uses an MRU insertion policy (MIP), inserting new lines in the MRU position in the stack. Workloads with large working sets and cyclic access patterns are known to have poor hit-rates when using MIP. To overcome this problem, LIP instead inserts new lines in the LRU position at the bottom of the stack. This policy performs well for cyclic workloads, but penalizes many MIP-friendly workloads.

In the next policy, BIP, most lines are inserted in the LRU position, but with a small, fixed probability, ε, some lines are inserted in the MRU position. This provides an aging mechanism to help the policy adjust to changes in the working set. Finally, DIP dynamically chooses between using either BIP or MIP. A small number of sets are dedicated to using each policy, and a saturating counter compares which policy results in more misses. The remaining sets in the cache use whichever policy performs best in the dedicated sets.

C. Shepherd Cache

Rajan and Govindarajan [2] have proposed the Shepherd Cache (SC) design that attempts to emulate an optimal cache replacement policy for a conventional cache.

SC replaces a large, highly associative cache with a cache with lower associativity (the main cache, or MC), and uses a secondary, *shepherd* cache to approximate an optimal replacement policy in the main cache. The intuition is that the benefits of using optimal replacement will overcome the reduced associativity of the main cache. Blocks are initially allocated into the shepherd cache (SC), delaying the optimal decision that will be emulated. While a line is in the SC, it collects information about the order of accesses to each line in the same set in the main cache. A FIFO order is maintained within the SC, and when the SC is full each allocation evicts the oldest line from the SC. When a line is evicted from the SC, the information that was collected is used to approximate the replacement decision that an optimal replacement policy would have made when the line was first allocated.

An optimal replacement policy replaces the line with the least imminent access, that is the line whose next access is farthest in the future [4]. To determine this information, each line in the SC tracks the *imminence* order of each line in the set. Specifically, the imminence order is the order of the first access to each line after the initial allocation of the SC line. For this purpose, each SC line maintains an imminence counter for each line in the same set. These counters are initialized to a value representing *unknown* imminence, then the first line accessed sets its counter to 0, the next line to 1, and so on.

FIFO order is maintained within the SC. When a new block is allocated in a full SC, the oldest block in the SC tries to move to the main cache. At this point shepherd cache approximates the optimal replacement decision in the main cache. The imminence counters are used to evict the least imminent line, that is, the line with the highest value in its imminence counter.

The optimal replacement decision can only be determined when all lines have been accessed since the SC line was first allocated. Otherwise, the decision is approximated by evicting one of the lines with *unknown* imminence. If the line moving from the SC to the main cache has unknown

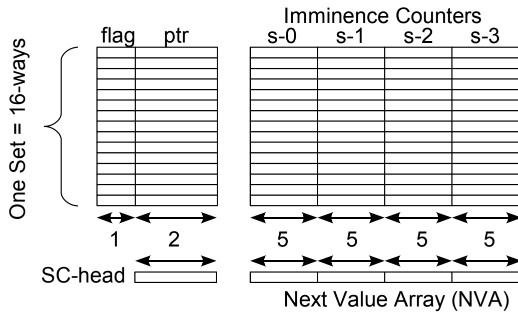

Fig. 1. Shepherd Cache metadata for one set in a 16-way set-associative cache with 4 SC-ways and 12 MC-ways.

imminence, it is directly evicted, otherwise, a baseline LRU replacement policy is used to evict one of the lines from the main cache which has unknown imminence. [1] Lines in the SC are not tracked by the baseline replacement policy, so the SC line is placed in the LRU position when it moves to the main cache.

To avoid moving lines between two distinct structures, the actual design maintains only a single cache array. Each line has a single bit *SC-flag* to indicate whether it is an MC-way or an SC-way, as well as an *SC-ptr* field that uses $log_2(SC\text{-}associativity)$ bits to indicate which imminence counters belong to each SC-way. Each line has *SC-associativity* imminence counters, each $log_2(Associativity)+1$ bits wide. In addition, each set has a *Next Value Array* (NVA) that stores the next counter value for each SC-way. Finally, $log_2(SC\text{-}associativity!)$ bits are used to maintain FIFO order amongst the SC-ways. For an SC design with four SC-ways and 12 MC-ways, all of this meta-data combined requires 393 bits of overhead per set, as shown in Fig. 1. When combined with a baseline LRU policy requiring 45 bits, this results in a total of 438 bits of meta-data per set. As a result of this large amount of meta-data, Rajan et al. [2] compare against a baseline with one additional block of data in each cache set.

D. Other Replacement Policies

Subramanian et al. [3] propose an adaptive mechanism similar to the DIP policy proposed by Qureshi et al. [1], except that it compares and selects from two entirely different replacement policies instead of just two insertion policies. This scheme is motivated by scenarios where different replacement policies perform significantly better for different workloads. In our results, we found the relative performance of different replacement policies was similar across different workloads. However, varying the insertion policy with LRU replacement does perform better for some workloads. Thus, we evaluate DIP but not the adaptive caches proposed by Subramanian et al.

[1]The implementation used in [2] reverses this order, only evicting the SC line if no line in the main cache has unknown imminence. However, the main effect of this difference is that a design with N SC lines per set using our implementation behaves similar to a design with $N-1$ SC lines per set using the original implementation.

The concept of a cache appears in many contexts in computer systems. Virtual memory paging systems, disk buffer caches, and web browser caches all act as caches, and they all employ replacement policies. A number of recent works have proposed new replacement policies in these other contexts [7], [8], [9]. These replacement policies could potentially be applied to processor caches. However, these algorithms typically require significant metadata overhead and complex logic that make them unattractive to implement in hardware.

III. NEW REPLACEMENT POLICIES

A. Lightweight Shepherd Cache (SC-L)

Our first new replacement policy simplifies the approach taken by the shepherd cache design. Instead of attempting to identify the imminence order of all lines in a set, our new Lightweight Shepherd Cache design (SC-L), divides each set into two groups: those with known imminence, and those with unknown imminence. When at least one line in the cache has unknown imminence, our new SC-L design follows the same replacement policy as the original SC design and uses a baseline replacement policy to select from the group of lines with unknown imminence. When all lines in the cache have known imminence, SC-L relies on the baseline policy to select a victim, instead of using the imminence counters to evict the least imminent line.

We described the metadata structures of the original SC design in Section II-C. In our new SC-L design, we replace the $log_2(associativity)+1$ bit imminence counters with single bit *known* flags. Where the original design would update the value of these imminence counters, the new SC-L design merely sets the *known* bit. The NVA is removed, and all the remaining metadata remains the same. This design reduces the total replacement policy overhead from 438 bits to only 162 bits per set when combined with a baseline LRU replacement policy.

B. Extra-Lightweight Shepherd Cache (SC-XL)

The SC-L design takes a simple approach to identifying whether a line has known or unknown imminence, and uses this information to approximate the behavior of the SC design. Our second design takes an even simpler approach and attempts to capture only the behavior of the SC design with a completely different mechanism.

When evicting a line, SC and SC-L look at three possibilities: 1) the SC line with unknown imminence; 2) an MC line with unknown imminence; and 3) a line with known imminence. We observe that lines with known imminence have necessarily been accessed more recently than lines with unknown imminence. Thus, if any lines in the MC have unknown imminence, then the LRU line should have unknown imminence. [2] Thus, if the *known* flags or imminence counters

[2]In practice, this is not strictly true. SC lines are placed in the LRU position when they move to the MC, even if they have known imminence relative to other SC lines. However, we ignore this slight discrepancy when motivating the SC-XL design.

are mainly used to identify lines with unknown imminence, the baseline LRU policy could potentially be used instead.

Based on these observations, we propose the Extra-Lightweight Shepherd Cache (SC-XL) that takes advantage of the inherent imminence ordering of LRU. The imminence counters of the SC design and the known bits of the SC-L design are all replaced with a single *known* bit for each SC-way. Newly allocated lines clear the known bit, and any subsequent access to that line will set the known bit. Like the previous SC and SC-L designs, newly allocated cache lines are placed into a small SC FIFO. If the oldest SC line does not have its known bit set, then it is evicted. Otherwise, the LRU line in the MC is evicted, and the oldest SC line becomes the LRU line in the MC.

Like the previous SC and SC-L designs, instead of maintaining separate SC and MC structures, metadata is used to identify which cache ways currently store SC-ways. We use a single pointer for each SC-way, as well as a single known bit for each SC-way, in addition to the metadata needed for the baseline replacement policy. Thus, for a 16-way set-associative cache with four SC ways, and a baseline LRU policy, SC-XL requires a total of 65 bits per set. This includes 45 bits for the baseline LRU replacement policy, four 4-bit pointers to identify the four SC-ways, and four 1-bit *known* flags.

C. Different Baseline Replacement Policies

Both of our new replacement policies require significantly less metadata overhead than the original SC design. However, they both still use an impractical LRU replacement policy. In theory, any replacement policy can be used as a replacement policy. Thus, the following two sections describe how we adapt SC-L and SC-XL to use more practical replacement policies.

1) Adapting SC designs to PLRU: Instead of the impractical LRU policy, commercial designs often use simpler pseudo-LRU [6] replacement policies. As an example pseudo-LRU algorithm, we consider the policy used in the IBM 3033 [6]. In this algorithm, a binary tree is used to represent relative orderings between groups of cache lines in a set. The leaves of the tree represent the order between adjacent cache lines; the next level of the tree represents the order between adjacent pairs, and so on until the root of the tree which represents the order between the two halves of the tree. On each access, a block becomes the most recently used block and adjusts the relative orderings at each level of the tree as necessary to indicate that that block was touched more recently than all others. As a small optimization, the top two levels of the tree are combined into a single 4-way branch and five bits are used to represent an exact LRU order between four sub-trees. This implementation requires $log_2(N)+1$ bits of metadata for each set in an N-way set-associative cache.

The binary tree implementation provides a total ordering between all cache lines, but this order usually differs from the actual LRU order. As an extreme example, when the MRU line and LRU line are in adjacent ways, the LRU line might be represented in the pseudo-LRU order as being the second most recently used line.

We adapt SC-L, and SC-XL to use pseudo-LRU as the baseline replacement policy by treating the total order represented by the tree as the true LRU order. When placing a line in the LRU position, we modify all levels of the tree to indicate that the given way is the LRU way. When selecting the LRU line from among a subset of all the lines in a set, we use the total order provided by the binary tree. Similar to our implementations with LRU, the oldest SC line is treated as being less recently used than all of the MC lines.

2) Adapting SC designs to Clock: Our evaluation in Section IV shows that a simple Clock algorithm performs better on average than SC-L or SC-XL when they use a baseline pseudo-LRU algorithm. Thus, it seems reasonable to adapt SC-L and SC-XL to use a clock replacement policy. To adapt our designs, we have to change the policy for selecting candidates for replacement decisions, as well as the policy for moving a line from an SC-way to an MC-way.

For both designs, if the oldest SC entry has not been touched since being allocated (i.e., it has unknown imminence), then this line is chosen as the victim. For the SC-L design, we then search for any lines with unknown imminence. While LRU and pseudo-LRU policies provide an order to select one of these lines, the Clock policy does not. Instead, we define a fixed order among ways, starting with way 0 and ending with way $associativity-1$. If all lines have known imminence, then we use the normal clock replacement policy, skipping any lines in SC-ways. For the SC-XL design, if the oldest SC-entry is not chosen, we simply use the baseline clock policy, again skipping any lines in SC-ways.

Finally, when moving a line from an SC-way to an MC-way, both designs treat this as a new allocation and clear the line's *touched* bit.

IV. EVALUATION

This section evaluates the new SC-L and SC-XL replacement policies. Section IV-A describes our evaluation methodology. Section IV-B evaluates a number of common replacement policies and demonstrates the potential improvement offered by the optimal replacement policy. Section IV-C re-examines the different insertion policies proposed by Qureshi et al. [1] and shows that they offer little benefit for commercial workloads when combined with practical PLRU replacement policies. Finally, Section IV-D compares SC, SC-L, and SC-XL designs when combined with LRU, PLRU, and Clock baseline replacement policies.

A. Methodology

We used the Casper [10] trace-based cache simulator to measure cache miss-ratios. We used bus traces collected from an eight processor system where each processor had a 256KB L2 cache. We simulated a four core CMP where each core used SMT to execute two threads, and the memory references in the traces were injected into the L2 caches. The

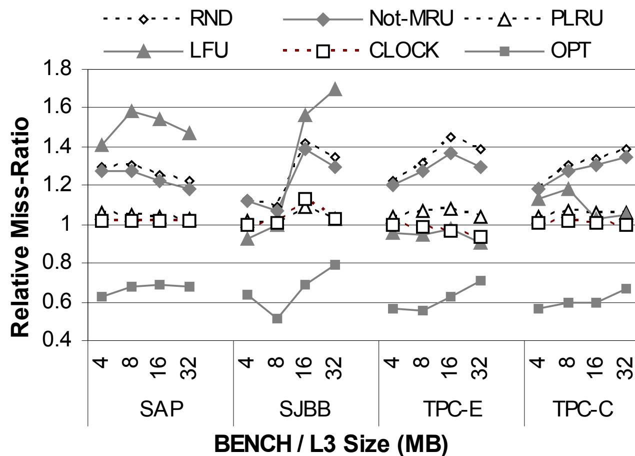

Fig. 2. Common replacement policies.

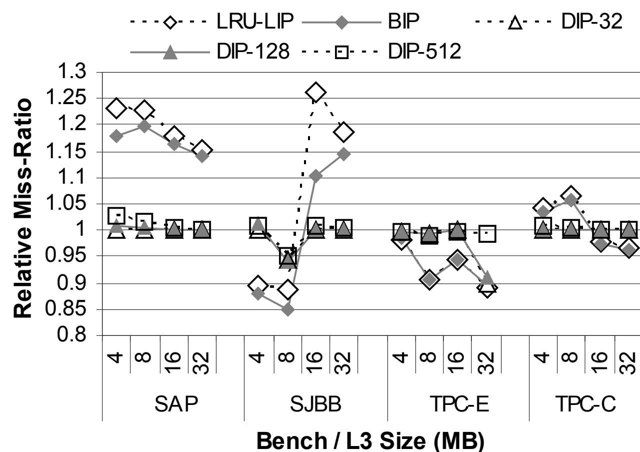

Fig. 3. LRU replacement with different insertion policies.

cache hierarchy consists of four private 256KB, 8-way set-associative L2 caches and a shared, 16-way set-associative L3 cache with a capacity ranging from 4MB to 32MB. All caches use 64-byte blocks. All experiments used an LRU replacement policy for the private L2 caches. We present results in terms of global off-chip miss-ratios for the shared L3 cache. The global miss-ratio is defined as the number of references that miss in the L3 divided by the total number of memory references in the trace. Results are presented relative to the baseline global miss-ratio for the same workload on the same cache with an LRU replacement policy with MRU insertion. All figures use this same baseline for comparison.

We used traces from four commercial workloads: SpecJBB (SJBB), a SAP workload, TPC-C, and TPC-E. For each trace, we used 50 million references to warm-up the caches and measured miss-ratios for the remaining references in the trace, which ranged from 200 to 250 million.

B. Simple Cache Replacement Policies

First, we examine the performance of various well-known replacement policies. Fig. 2 shows the relative miss-ratio of the L3 cache for all four workloads with cache sizes of 4MB, 8MB, 16MB and 32MB. The six curves show relative miss-ratios for: random replacement (RND), the Not-MRU policy, a pseudo-LRU algorithm (PLRU) based on the policy used in the IBM3033, least-frequently used (LFU) replacement, the clock algorithm (CLOCK), and finally, Belady's MIN algorithm for optimal replacement (OPT). The miss-ratios are normalized to the miss-ratio of LRU replacement with an MRU insertion policy. The two random replacement policies use the standard C library random number generator instead of modeling a specific LFSR implementation.

Fig. 2 shows that RND and Not-MRU perform very poorly, between 20% and 40% worse than LRU for most cases. Also, note that OPT performs an average of 36% better than LRU, indicating potential for new algorithms to improve performance.

Finally, we note the interesting behavior for SJBB when going from 8MB to 16MB. The LRU policy results in a sharp decline in absolute miss-ratios between these two points, representing an inflection point in the miss-ratio curve. This inflection point occurs for slightly larger cache sizes for other policies, resulting in the behavior shown in Fig. 2, where relative miss-ratios of different policies vary drastically. Many of our other results also show this same behavior for SJBB.

C. Insertion Policies

We next consider the effects of varying the insertion policy used with an LRU replacement policy. Fig. 3 shows the relative miss-ratios using an LRU insertion policy (LRU-LIP), a bimodal insertion policy (BIP) with $\varepsilon = 1/32$ [1], and dynamic insertion policies with 8-bit counters and 32, 128, and 512 dedicated sets (DIP-32, DIP-128, and DIP-512). To highlight DIP's ability to select between two different insertion policies, we select between LIP and MIP insertion policies (instead of BIP and MIP as in [1]). We use the method described in [1] for selecting dedicated sets. Fig. 3 shows that using BIP or LRU-LIP instead of the baseline LRU-MIP can reduce miss-ratios by up to 15% for SJBB and up to 10% for TPC-E. However, out of the seven points where LRU-LIP out-performs the baseline, DIP only sees improvement for two of these points. This behavior is a result of biased behavior in the dedicated sets. The sets dedicated to LIP have not only more absolute misses but also higher miss-ratios for many cases were LIP outperforms MIP in general (e.g., TPC-E with an 8MB cache). In such cases, the dedicated sets exhibit behavior counter to the overall cache trends, resulting in selecting a poorer performing insertion policy.

1) Prime-Modulo Set Indexing: To overcome the biasing of the dedicated sets, we implemented a prime-modulo set indexing scheme [11]. Fig. 4 shows the results of using prime-module set indexing for SJBB and TPC-E. The y-axis still shows relative miss-ratio compared to the baseline of LRU-MIP with conventional set-indexing. In addition to the three DIP configurations, we also show results for LRU-MIP

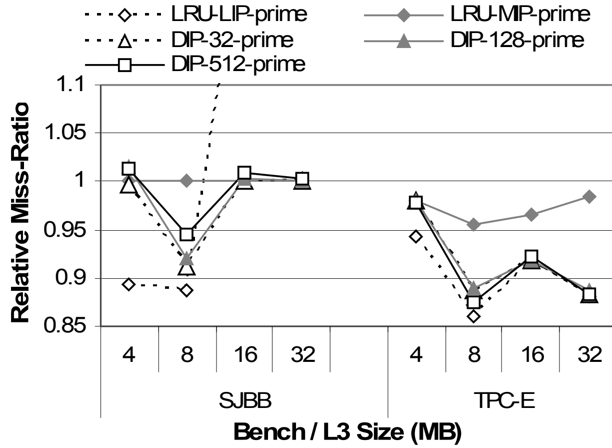

Fig. 4. Different insertion policies with prime-modulo set indexing.

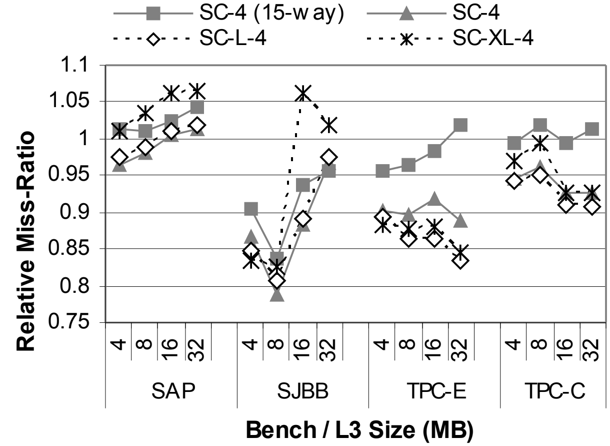

Fig. 6. SC, SC-L, and SC-XL designs with baseline LRU replacement.

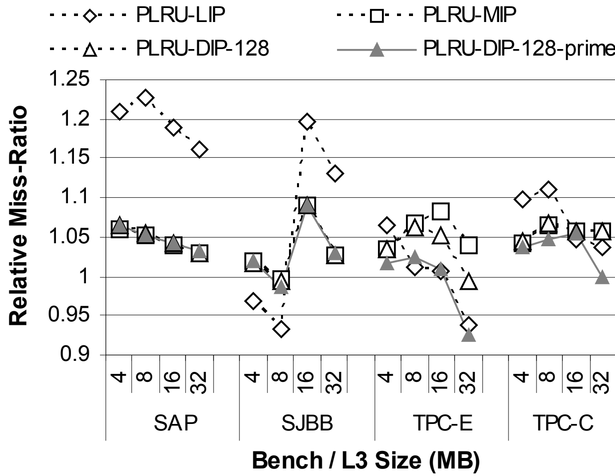

Fig. 5. Pseudo-LRU (PLRU) with different insertion policies.

and LRU-LIP with prime-modulo set indexing. Using prime-modulo set indexing does reduce the bias in the dedicated sets, as the DIP curves generally behave similarly to the best policy (LIP or MIP). However, for SJBB, we see that the number of dedicated sets has a significant effect on overall performance of DIP. Thus, while prime-modulo set-indexing improves the performance of DIP for our workloads, careful parameter tuning is still required for the best results.

2) Effects of Pseudo-LRU Replacement: Fig. 5 shows the effects of different insertion policies when used with the pseudo-LRU algorithm described in Section III-C.1. All results are shown relative to the same baseline as previous figures. The dashed curves with hollow markers show results for pseudo-LRU using LIP, MIP, and DIP with 128 dedicated sets. The solid curve shows results for DIP-128 using prime-modulo set indexing.

When individual curves are compared in Fig. 3 and Fig. 5, the results demonstrate that using pseudo-LRU performs worse than true LRU for any given insertion policy. It also reduces the benefits of varying insertion policies. Even the best policy shown in Fig. 5, PLRU-DIP-128-prime, increases the miss-ratio by an average of 2.7%, and in the best case it only sees a reduction in miss-ratio of 7.3%. Compare this with the results of the simple Clock algorithm shown in Fig. 2 which increases miss-ratios by only 0.9% on average. This demonstrates that for a practical low-overhead replacement policy, using a dynamic insertion policy offers little or no benefit compared to other well known replacement policies.

3) Insertion Policy Summary: Different insertion policies can reduce miss-ratios up to 15% in the best case. But in practice dynamic insertion policies are difficult to tune and require techniques such as prime-modulo set indexing to avoid problems with non-uniform set accesses. When more practical PLRU algorithms are used, the maximum benefit is reduced to only 7.3%, and the average miss-ratio is worse than simply using the Clock replacement policy.

D. Shepherd Cache Policies

Fig. 6 shows the relative miss-ratios of a number of shepherd cache designs compared to our baseline 16-way cache with LRU-MIP replacement. The first curve, indicated by grey squares, shows miss-ratios for an SC design with four SC-ways and eleven MC-ways, for a total of 15-way associativity. Despite having a smaller cache capacity, the additional metadata overhead for this design results in a total cache area comparable to the baseline design. The second curve, indicated by grey triangles, shows the miss-ratio for an SC design with four SC-ways and 12 MC-ways, for a total of 16-way set-associativity. This second design has the same data capacity as the baseline cache designs, but the extra replacement policy overhead is roughly equivalent to adding an extra way to the cache. The final two curves, indicated by hollow diamonds and asterisks, show SC-L and SC-XL designs, respectively, both with four SC-ways and 12 MC-ways. These two designs have the same data capacity as the baseline cache, and significantly less extra overhead than the original SC design.

These results show that our SC-L and SC-XL designs perform similarly to the original shepherd cache design for

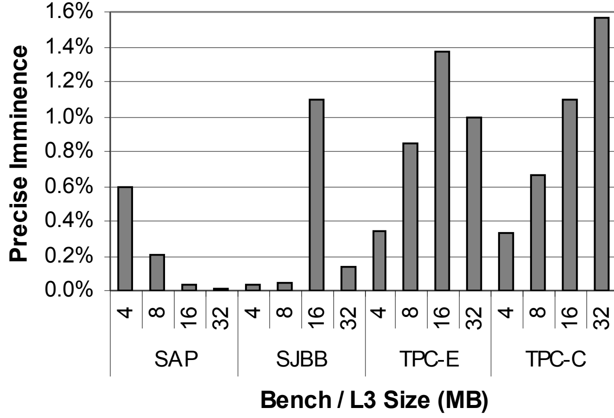

Fig. 7. Fraction of all replacements where all lines in the set have known imminence.

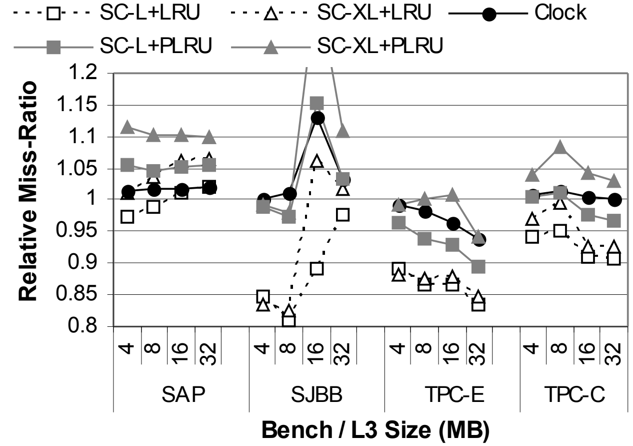

Fig. 8. SC-L, and SC-XL designs with baseline pseudo-LRU replacement.

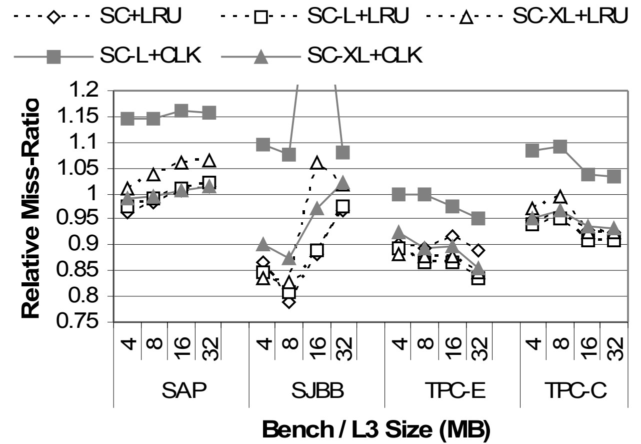

Fig. 9. SC-L, and SC-XL designs with baseline Clock (CLK) replacement.

our workloads. In fact, for TPC-E, our new designs perform better than the original shepherd cache design. As expected, SC-L performs better than SC-XL. Across all these cache sizes and workloads, the SC design offers a 7.3% reduction in miss-ratio while SC-L offers a slightly better 8.2% reduction in miss-ratio, and SC-XL offers an average of 4.9% reduction in miss-ratio.

The key difference between the SC and SC-L designs is that SC creates a precise imminence ordering, while SC-L simply tracks known or unknown imminence. Fig. 7 shows the fraction of all replacement decisions where all lines have known imminence for the 15-way SC-4 design shown in Fig. 6. The figure shows that 98.6% to 99.97% of the time there is at least one cache line that has unknown imminence and SC and SC-L should make the same replacement decision. [3]

1) Shepherd Cache + Pseudo-LRU: The SC, SC-L, and SC-XL designs considered so far relied upon a baseline LRU replacement policy in the MC-ways. Fig. 8 shows the results of implementing our SC-L and SC-XL designs with a pseudo-LRU baseline replacement policy (grey lines labeled SC-L+PLRU and SC-XL+PLRU). For comparison, the graph also shows the same designs using a baseline LRU policy. We also show the results of using a simple clock algorithm. All of the SC-L and SC-XL designs shown here have four SC ways and 12 MC ways. We omit the original SC design since it behaves similar to the SC-L design.

These results show that changing the baseline LRU policy to a pseudo-LRU policy eliminates much of the benefit of using shepherd cache. Averaging across all cache sizes and designs, the SC-L+PLRU shows a 0.3% increase in miss-ratio compared to our baseline of true LRU replacement, while the SC-XL+PLRU shows an even worse 6.2% increase in miss-ratio. Compare this to a simple clock algorithm which on average only increases the miss-ratio by 0.9% while requiring a total of 20 bits of overhead per set compared to 135 bits per set for SC-L+PLRU.

2) Shepherd Cache + Clock: Since the simple Clock algorithm performs better than SC-L and SC-XL with a pseudo-LRU algorithm, we have adapted our two new designs to use clock as a baseline replacement policy. The results are shown in Fig. 9, with the two grey curves labeled SC-L+CLK and SC-XL+CLK showing our designs with a baseline clock replacement policy, and the three dashed curves showing SC, SC-L, and SC-XL designs using LRU as the baseline replacement policy

The SC-XL design performs better than the SC-L design when combined with the Clock baseline replacement policy. This is a result of how the two designs were adapted to the new baseline replacement policy. In the end, the SC-XL+CLK design reduces the miss-ratio by an average of 5.4% and by up to 12.5% for the best case of SJBB with an 8MB cache. Note that this design requires only 40 bits of overhead per set compared to the next best design, SC-L+LRU, which requires 162 bits per set.

[3] In practice, the actual decisions can differ, e.g., the victim chosen by one design might have been previously evicted in the other design at a time when all lines had known imminence.

TABLE I

SUMMARY OF METADATA OVERHEAD AND MISS-RATIO REDUCTION.

	Overhead				
Policy	Bits / Set	% of Capacity	Min	Max	Average
LRU	45	0.5%	0	0	0
PLRU	17	0.2%	-9.0%	0.5%	-4.7%
Clock	20	0.2%	-13.0%	6.1%	-0.9%
DIP	45	0.5%	-1.0%	8.9%	0.1%
DIP-prime	189-237	2.3-2.9%	-1.5%	11.2%	3.2%
DIP-PLRU	17	0.2%	-9.1%	0.6%	-4.3%
DIP-PLRU-prime	161-209	2.0-2.6%	-9.0%	1.2%	-3.4%
SC-4 + LRU	438	5.3%	-1.3%	21.2%	7.3%
SC-L-4 + LRU	162	1.9%	-2.0%	19.2%	8.2%
SC-XL-4 + LRU	65	0.8%	-6.6%	17.4%	4.9%
SC-L-4 + PLRU	131	1.6%	-15.4%	10.6%	-0.3%
SC-XL-4 + Clock	40	0.5%	-2.3%	14.4%	5.4%

E. Result Summary

Table 1 provides a comparison of the miss-ratios and overhead of the best designs. The first column lists the different policies: LRU, pseudo-LRU (PLRU), clock, dynamic insertion policy with 128 dedicated sets with LRU replacement and normal set indexing (DIP) and prime modulo set-indexing (DIP-prim), DIP with 128 dedicated sets with pseudo-LRU replacement and normal (DIP-PLRU) and prime-modulo (DIP-PLRU-prime) set indexing, and different SC, SC-L, and SC-XL designs, all with four SC ways and 12 MC ways and with the baseline replacement policy indicated as LRU, PLRU or Clock. The second column shows the number of bits per set required for storing replacement policy metadata, and the third column shows this overhead as a fraction of the total data capacity of the cache. For SC, SC-L, and SC-XL designs, this overhead includes the overhead for the baseline replacement policy. For the four DIP policies, the small overhead of the policy select counters is ignored. The overhead shown for prime-modulo set indexing results from storing larger tags and assumes a 50-bit physical address and depends on the total number of sets in the cache, with larger caches incurring less overhead. The last three columns show the minimum, maximum and average reduction in miss-ratios for the different policies compared to the baseline LRU policy. These are arithmetic averages across all cache sizes and all workloads. Note that a negative reduction is equivalent to an increase in miss-ratio.

V. CONCLUSION

The basic replacement policies commonly used in modern processors perform an average of 57% worse than than optimal replacement policy for commercial applications using large shared caches in a CMP. Unfortunately, most work on cache replacement policies has been evaluated based upon SPEC CPU benchmarks running on uni-processors with relatively small caches. Recent work on dynamically changing the insertion policy while still using a traditional replacement policy [1] offers only limited potential improvements for commercial workloads and requires careful tuning, including a guarantee of even distribution of accesses across sets. The recently proposed shepherd cache policy offers significant potential for improvement, up to 21.2% reduction in miss-ratio for some cases and an average of 7.3% reduction for all cases we studied. However, this design has a complex replacement policy and requires over 46 bytes of metadata for each cache set.

Our newly proposed SC-L and SC-XL designs can achieve most of the potential improvement of the original shepherd cache with drastic reductions in the amount of metadata required. Our simple SC-L design combined with a baseline LRU replacement policy reduces miss-ratios by an average of 8.2% and up to 19.2% for some cases while requiring 162 bits of metadata per set, less than half the overhead of the original SC design. In addition, our SC-XL design, when combined with a practical clock replacement policy, further reduces overhead to only 40-bits per set while still reducing miss-ratios by an average of 5.4% and up to 14.4% for some cases. This design offers a promising new heuristic that achieves miss-ratio reductions for commercial applications using large caches with minimal additional overhead.

VI. ACKNOWLEDGMENTS

The authors would like to thank Andreas Moshovos, Kaveh Aasaraai, Ioana Burcea, and Myrto Papadopoulou for their valuable comments and feedback. We would also like to thank Kaushik Rajan for providing clarification of some details of the original Shepherd Cache design.

REFERENCES

[1] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. Emer, “Adaptive insertion policies for high-performance caching,” in *Proc. Intl Symposium on Computer Architecture (ISCA)*, San Diego, CA, 2007.

[2] K. Rajan and R. Govindarajan, “Emulating optimal replacement with a shepherd cache,” in *Proc. Intl Symposium on Microarchitecture (MICRO-40)*, Chicago, IL, Dec. 2007.

[3] R. Subramanian, Y. Smaragdakis, and G. H. Loh, “Adaptive caches: Effective shaping of cache behavior to workloads,” in *Proc. of the 39th International Symposium on Microarchitecture (MICRO0’39)*, Orlando, FL, December 2006.

[4] L. Belady, “A study of replacement algorithms for a virtual-storage computer,” *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[5] F. J. Corbato, “A paging experiment with the multics system,” MIT Project MAC Report MAC-M-384, May 1968.

[6] K. So and R. N. Rechtshaffen, “Cache operations by mru change,” *IEEE Transactions on Computers*, vol. 37, no. 6, pp. 700–709, June 1988.

[7] S. Jiang, F. Chen, and X. Zhang, “Clock-pro: An effective improvement of the clock replacement,” in *Proc. of the Annual Conference on USENIX Annual Technical Conference (ATEC05)*, Anaheim, CA, 2005.

[8] S. Jiang and X. Zhang, “Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance,” Marina Del Rey, CA, 2002.

[9] N. Megiddo and D. S. Modha, “Arc: A self-tuning, low overhead replacement cache,” in *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST03)*, San Francisco, CA, March 2003.

[10] R. Iyer, “On modeling and analyzing cache hierarchies using casper,” in *Proc of the 11th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS03)*, 2003.

[11] M. Kharbutli, Y. Solihin, and J. Lee, “Using prime numbers for cache indexing to eliminate conflict misses,” in *Proc. of the 10th Intl Symposium on High Performance Computer Architecture (HPCA-10)*, Madrid, Spain, Feb. 2004.