Counter-Based Cache Replacement and Bypassing Algorithms

Mazen Kharbutli, *Member*, *IEEE*, and Yan Solihin, *Member*, *IEEE*

Abstract—Recent studies have shown that, in highly associative caches, the performance gap between the Least Recently Used (LRU) and the theoretical optimal replacement algorithms is large, motivating the design of alternative replacement algorithms to improve cache performance. In LRU replacement, a line, after its last use, remains in the cache for a long time until it becomes the LRU line. Such *dead* lines unnecessarily reduce the cache capacity available for other lines. In addition, in multilevel caches, temporal reuse patterns are often inverted, showing in the L1 cache but, due to the filtering effect of the L1 cache, not showing in the L2 cache. At the L2, these lines appear to be brought in the cache but are never reaccessed until they are replaced. These lines unnecessarily pollute the L2 cache. This paper proposes a new *counter-based approach* to deal with the above problems. For the former problem, we predict lines that have become dead and replace them early from the L2 cache. For the latter problem, we identify never-reaccessed lines, bypass the L2 cache, and place them directly in the L1 cache. Both techniques are achieved through a single counter-based mechanism. In our approach, each line in the L2 cache is augmented with an event counter that is incremented when an event of interest such as certain cache accesses occurs. When the counter reaches a threshold, the line "expires" and becomes replaceable. Each line's threshold is unique and is dynamically learned. We propose and evaluate two new replacement algorithms: Access Interval Predictor (AIP) and Live-time Predictor (LvP). AIP and LvP speed up 10 capacity-constrained SPEC2000 benchmarks by up to 48 percent and 15 percent on average (7 percent on average for the whole 21 Spec2000 benchmarks). Cache bypassing further reduces L2 cache pollution and improves the average speedups to 17 percent (8 percent for the whole 21 Spec2000 benchmarks).

Index Terms—Caches, counter-based algorithms, cache replacement algorithms, cache bypassing, cache misses.

1 Introduction

Recent studies have shown that, in highly associative caches such as the L2 cache, the performance gap between the Least Recently Used (LRU) and Belady's theoretical optimal replacement algorithms is large. For example, the number of cache misses using LRU can be up to 197 percent higher than when using the optimal replacement algorithm [1], [2]. This suggests that alternative replacement algorithms may be able to improve cache performance significantly over LRU.

The LRU replacement algorithm tries to accommodate temporal locality by keeping recently used lines away from replacement in the hope that, when they are reused, they will still be in the cache. Unfortunately, two things work against LRU replacement. For one thing, each cache line will eventually be replaced after its last use. However, even after its last use, a line is not immediately replaced because it remains in the cache until it becomes the LRU line. Such *dead* lines unnecessarily reduce the cache capacity available for other lines. The *dead time*, that is, time between when a line becomes dead and when it is eventually replaced, becomes worse with larger cache associativities since it takes longer for a line that is recently last-used to travel down the LRU stack to become the LRU line. Hence, although a larger cache associativity improves cache performance in general, the performance gap between LRU and the optimal replacement algorithm also increases. Clearly, replacing dead lines promptly after their last use would improve cache performance by making the wasted capacity available for other cache lines that are not dead yet. In order to achieve that, a *dead line prediction* technique is needed to identify and replace dead lines early.

Another factor that works against LRU replacement is that temporal locality may *invert* when there are multiple levels of caches. There are many cache lines that exhibit bursty temporal reuses. This is often due to spatial reuses of different bytes of the same cache line, which tend to occur in burst. Current caches typically have large cache lines (64 or 128 bytes), amplifying this bursty reuse pattern. With a single-level cache, these bursty temporal reuses would be well accommodated by the LRU replacement algorithm. However, in multilevel caches, this bursty pattern often manifests at the L1 cache only and is filtered by the L1 cache. To the L2 cache, the line does not appear to have temporal reuse since it is brought into the cache but is not used until it is replaced. Hence, the lines are immediately dead after they are brought into the L2 cache. Such *never-reaccessed* lines unnecessarily waste the L2 cache capacity. Ironically, lines with less frequent temporal reuses cannot be filtered by a small L1 cache and such reuses will show up at the L2 cache. Note that, for never-reaccessed lines, immediately replacing them after they are brought into the L2 cache using a dead line prediction technique would only

- *M. Kharbutli is with the Department of Computer Engineering, Jordan University of Science and Technology, Irbid, Jordan 22110. E-mail: kharbutli@just.edu.jo.*
- *Y. Solihin is with the Department of Electrical and Computer Engineering, North Carolina State University, Campus Box 7256, Raleigh, NC 27695-7256. E-mail: solihin@ncsu.edu.*

Manuscript received 11 July 2006; revised 12 Mar. 2007; accepted 26 July 2007; published online 6 Sept. 2007.
Recommended for acceptance by A. Gonzalez.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0267-0706.
Digital Object Identifier no. 10.1109/TC.2007.70816.

be partially beneficial because the lines already cause cache pollution due to replacing other lines that may still be needed by the processor. A better approach is to identify them and avoid placing them in the L2 cache in the first place using a technique often referred to as *cache bypassing*.

The main contribution of this paper is that we show that a simple counter-based approach can be used to identify dead lines and replace them early and that the same approach can be simultaneously used for identifying and bypassing never-reaccessed lines. Our mechanism relies on counters that keep track of the number of relevant cache events in a cache line's history and use that to predict the cache line's future behavior. We call our approach *counter-based cache replacement* and *counter-based cache bypassing*. In our approach, each L2 cache line is augmented with an *event counter* that is incremented when an event of interest (such as certain cache accesses) occurs. For replacement decisions, when the counter reaches a *threshold*, the line *expires* and immediately becomes replaceable. We design and evaluate two alternative algorithms, which differ by the type of events counted and the intervals in which they are counted: The *Access Interval Predictor* (AIP) counts the number of accesses to a set in an interval between two consecutive accesses to a particular cache line, whereas the *Live-time Predictor* (LvP) records the number of accesses to a cache line in an interval in which the line resides continuously in the cache. For bypassing decisions, the same event counters can identify never-reaccessed lines and, in the future, they can be directly placed in the L1 cache without polluting the L2 cache.

Through a detailed simulation evaluation, AIP and LvP speed up 10 out of 21 Spec2000 applications that we tested by up to 48 percent or 15 percent on average (7 percent on average for the whole 21 Spec2000 applications) without slowing down the remaining 11 applications by more than 1 percent. Both AIP and LvP outperform other dead line predictors in terms of *coverage* (fraction of replacements initiated by the predictors), *accuracy* (fraction of replacements that agree with the theoretical OPT), and *IPC improvement* using comparable hardware costs. Furthermore, bypassing can be added to AIP and LvP without additional hardware, which further improves the average speedup to 17 percent (8 percent for the whole 21 applications). Both AIP and LvP only incur small overheads: Each cache line is augmented with 21 bits to store prediction information, equivalent to 4.1 percent storage overhead for a 64 byte line. In addition, a simple 40 Kbyte prediction table is added between the L2 cache and its lower level memory components. The prediction table is only accessed on an L2 cache miss and, thus, its access is overlapped with the L2 cache miss latency.

The rest of the paper is organized as follows: Section 2 discusses related work, Section 3 discusses our counter-based replacement algorithms, and Section 4 discusses the cache bypassing algorithm. Section 5 describes the evaluation environment, while Section 6 discusses the experimental results obtained. Finally, Section 7 concludes the paper.

2 Related Work

Dead-block replacement. To improve the cache replacement decisions, ideally, a line is replaced as soon as it has been last used. Previously, two approaches for dead line prediction have been proposed. In *sequence-based* prediction, a predictor records and identifies the sequences of memory events, such as the PC/addresses of the last few accesses, leading up to the *last use* of a line [3], [2]. In *time-based* prediction, a cache line's timing, measured in the number of clock cycles (or multiples of it), is used to predict when a line's last use has likely occurred [4], [5], [6], [7]. Note that no cache replacement policy has been proposed based on time-based prediction. Time-based prediction has only been evaluated in the context of cache leakage power reduction [4], [5], [7] and prefetching [8]. However, logically, it is applicable for aiding cache replacement decisions and, hence, it is compared against our techniques in this paper. Our counter-based approach is qualitatively compared to the sequence-based approach in Section 3.4 and to the time-based approach in Section 3.5. A quantitative comparison between the three approaches is in Section 6.

Takagi and Hiraki [9] recently proposed a counter-based replacement algorithm called IGDR which collects access interval distributions. In their algorithm, the virtual time counter is incremented on each cache access, regardless of the cache sets, thus requiring large counters. Moreover, the algorithm is costly to implement in hardware: Several per-line fields need to be updated, compared, and incremented on every access. In addition, the algorithm requires a large number of multiplications and divisions to regularly update tables. On the other hand, our predictors only require a 4-bit counter per cache line and only perform increment operations to the counter of the accessed line (LvP) or the counters of the lines in the accessed set (AIP). AIP and LvP only perform comparisons on a cache miss to identify dead lines. Finally, IGDR divides cache lines into classes and makes predictions based on a line's class. In AIP and LvP, each line's behavior is unique. A quantitative comparison with IGDR is provided in Section 6.

An alternative approach to early replacement of dead lines relies on the compiler to make such decisions [10], [11]. However, compiler techniques are less applicable to applications that have dynamic access pattern behavior or ones that are hard to analyze at compile time.

Cost-sensitive replacement algorithms have been proposed to reduce the cost of cache misses rather than miss rates [12], [13]. Such algorithms predict and replace the cache line that will have the lowest miss cost the next time it is accessed, even if it is still useful. On the other hand, our approach and dead-block predictors in general predict and replace a dead line that will not be accessed again while in the cache.

Finally, none of the dead-block prediction techniques support cache bypassing, whereas our predictors can also identify never-reaccessed lines and make bypassing decisions for them to further improve cache performance without extra hardware.

Cache bypassing. Several cache bypassing schemes have been proposed in the past. They can be categorized into static and dynamic approaches. Static approaches [14], [15] rely on

the profile-guided compiler to identify lines that should bypass the cache. However, compiler-based approaches are less applicable to applications that have dynamic access pattern behavior or ones that are hard to analyze at compile time due to the limitation of static analyses. In addition, memory profiling results are often too input dependent to be a reliable guide across multiple input sets [15].

Dynamic cache bypassing uses runtime behavior learning and prediction to identify bypassing opportunities. Our cache bypassing scheme falls into this category. Dynamic cache bypassing has been studied in [16], [17], [18], [19], [20], [21], [22], [23], [24], [25]. In these studies, cache bypassing decisions can be based on the program counter (PC) of the load instruction that generates a cache access [23], [22], [25] or on the memory address of the access [16], [18], [19], [20], [21]. A PC-based approach works well if the same memory address has a different locality behavior in different code segments, whereas an address-based approach works well if the same memory address has similar locality behavior in different code segments. Our counter-based algorithms use a hybrid PC and address information in making bypassing decisions. The main distinguishing feature of our bypassing techniques is that, while prior dynamic cache bypassing schemes implement hardware support specifically for bypassing, we show that our cache bypassing techniques can utilize the same hardware used for counter-based replacement algorithms. Hence, the extra performance gain bypassing techniques provide comes for free.

Other related work. Peir et al. [26] use bloom filters (probabilistic algorithms to quickly test membership in a large set using hashing into an array of bits) to predict if a load will hit in the cache or not and thus to make scheduling decisions about its dependent instructions. Keramidas et al. [27] use decaying bloom filters to select which ways in the set to check to avoid checking all ways and, thus, to save dynamic power. Cache hit/miss prediction and way prediction techniques are largely orthogonal to our work.

3 Counter-Based Dead-Line Prediction and Replacement

In this section, we will give an overview of how counter-based replacement algorithms work (Section 3.1), discuss the implementation of the AIP algorithm (Section 3.2) and the LvP algorithm (Section 3.3), qualitatively compare them with the sequence-based approach (Section 3.4) and the time-based approach (Section 3.5), and, finally, discuss some implementation issues (Section 3.6).

3.1 Overview

Fig. 1 illustrates the life cycle of a cache line A. Initially, a cache line A is brought into the cache, either through a demand fetch or by a prefetch. Although in the cache, A may be accessed several times before it is replaced. We call the time between the placement of A in the cache until it is replaced the *generation time*. Generation time is divided into the time from placement until last use (*live time* or LT) and the time from the last use until replacement (*dead time*). In the case where the cache line is never used after it is brought into the cache (such as when a prefetched line is not used), there will be no *live time* and the line would be dead the instant it is placed in the cache. Finally, we refer to the time between two consecutive uses of A as the *access interval* (denoted as AI).

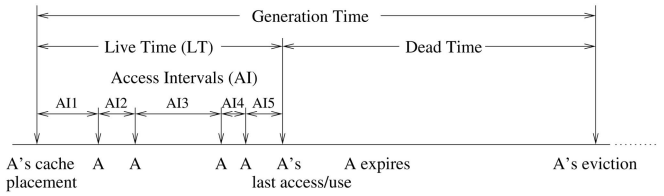

Fig. 1. The life cycle of a cache line A using terms defined in [28].

If the life cycle behavior, such as the live time or access interval of a cache line, is predictable, dead line prediction schemes can be devised to predict a cache line's future behavior. Three factors determine the design of such a predictor. The first factor is the *time interval* that serves as the basis of the prediction. Such an interval must allow early identification of dead lines so that the full performance benefit of replacing them early can be realized. The second factor is the *type of events* counted for each cache line in the chosen interval. The selection of the type of events to be counted affects the storage size of the event counters and affects the predictability of the intervals. The final factor is the *threshold selection*. A threshold value is used to identify a dead line whenever the line's event count reaches it. Obviously, threshold selection determines how aggressive a predictor is in identifying dead lines.

We choose the *access interval* and *live time* as our time intervals and call our predictors *AIP* and LvP, respectively. We will show later that both the access interval and live time predictors allow early identification of dead lines. In terms of the type of events counted during the time interval for a cache line, possible choices are accesses to the cache, accesses to the set that has the line or accesses to the line itself. For AIP, a reasonable choice would be to count the number of accesses to the *set that has the line*. Counting the number of accesses to the cache (regardless of the sets) would be a poor choice due to requiring large counters to count possibly many accesses during a line's access interval, not to mention that accesses to other sets are a noise since they do not affect the line's live time or dead time. For LvP, a reasonable choice of event to count is the number of accesses to the line itself during a single generation. Counting the number of accesses to the cache or to the set that has the line would require much larger and frequent counting, which incurs a high storage overhead. With their respective event types to count, we found that both AIP and LvP only require four-bit counters per cache line to count their respective events that occur in their respective time intervals. Detailed observations will be elaborated in Sections 3.2 and 3.3.

The final factor is threshold selection, which is to be learned from the past behavior of a cache line. Assuming there has been several access intervals and live times collected, thresholds for future predictions can be set by taking the statistical summary of past intervals. For example, the average or the maximum of event counts from past intervals could be used. We note, however, that

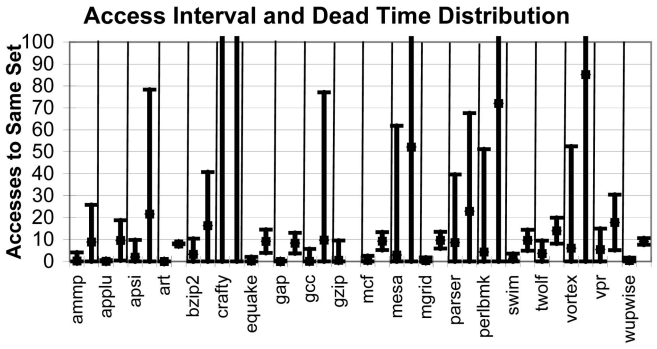

Fig. 2. Access interval (left bar for each application) versus dead time (right bar for each application) distribution, showing the average plus/minus its standard deviation. The cache parameters follow those in Table 4.

the penalty for replacing a line that is not dead yet is an extra cache miss on that line, which is quite significant. It is hard to justify trading off an extra cache miss for a slight increase in cache capacity. Hence, rather than using the average, we conservatively set the thresholds as the *maximum* of event counts in past intervals. For example, in AIP, the threshold is the maximum of all access intervals in the prior and current generations.

In the following sections, we will describe AIP and LvP in greater details.

3.2 AIP Algorithm Design and Implementation

Recall from previous discussion that, for each cache line, AIP works by counting the number of accesses to the set that has the line during the line's current access interval and identifies it as dead when the event count reaches the threshold. For AIP to reap the benefit of early replacement of dead lines, its access interval threshold value must be considerably smaller than its dead time. A previous study on L1 caches has shown that, indeed, access intervals of a line are typically a lot smaller than its dead time [8]. Our own experiments on L2 caches largely confirm the finding: The difference in a line's access interval duration and its dead time is often one or more orders of magnitudes (Fig. 2). This means that waiting for a typical access interval to elapse before concluding a line is dead will still allow the line to be replaced much sooner compared to when the line is replaced by LRU replacement.

3.2.1 Prediction Structures

Fig. 3 shows prediction structures used by AIP. Several fields are added to each L2 cache line in order to keep a count of events and its threshold value, while a separate *prediction table* is added between the L2 cache and its lower memory hierarchy components to keep the history of threshold values for lines that are not cached. The first field, *hashedPC*, is used to index a row in the prediction table when a line is replaced from the cache but has no role in counter bookkeeping. The size of *hashedPC* depends on the number of rows the prediction table has and, in our case, it is 8 bits. *hashedPC* is obtained by XOR-ing all 8-bit parts of the PC of the instruction that misses on the line. The second field is the *event counter* (C), which is incremented each time the set that has the line is accessed, regardless of whether the access was a hit or miss. The third and fourth fields are the counter thresholds for the

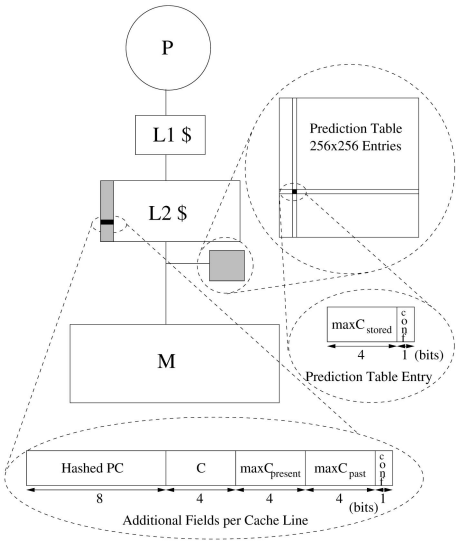

Fig. 3. Storage organization and overhead of AIP.

time intervals in the most recent generation $(maxC_{past})$ and in the current generation $(maxC_{present})$. The fifth field is a single confidence bit $(conf)$ that, if set, indicates that the line can expire and be considered for replacement. If it is not set, the line is not considered for replacement even if its threshold has been reached. Finally, to index a column in the prediction table, the line's block address is hashed by XOR-ing all 8-bit parts of it. For instruction lines, the block address is considered to be the PC.

The sizes of the counter and thresholds are determined based on profiling of likely counter values. In the profiling, we use infinite counter sizes and prediction table size and measure how many counters have their threshold values exceeding the value of 15 in the prediction table. A threshold value is measured as the maximum access interval in a single generation. Table 1 shows the result of

TABLE 1
Profile of Access Interval Counter *Threshold Values* in the Counter Prediction Tables

App.	Avg.	>15	=last
ammp	0.25	0.19%	97.6%
applu	0.11	0.00%	94.3%
apsi	1.99	2.96%	94.8%
art	0.00	0.00%	99.9%
bzip2	3.29	4.84%	50.8%
crafty	154.07	37.58%	14.5%
equake	0.26	0.26%	94.6%
gap	0.02	0.01%	91.0%
gcc	0.08	0.13%	99.5%
gzip	0.55	0.46%	86.7%
mcf	0.51	0.28%	94.1%
mesa	2.95	0.92%	81.1%
mgrid	0.43	0.04%	99.5%
parser	8.62	14.99%	38.9%
perlbmk	4.34	2.13%	76.6%
swim	1.19	0.02%	88.8%
twolf	3.64	4.27%	39.3%
vortex	6.05	4.85%	55.8%
vpr	5.44	9.96%	36.2%
wupwise	0.46	0.00%	51.4%

The cache parameters follow those in Table 4.

the profiling. During profiling, counter threshold values are simply recorded but are not used for cache replacement decisions. The table shows that the average counter threshold values are much smaller than the maximum value that can be stored using a 4-bit counter. Furthermore, fewer than 5 percent of counter threshold values are larger than 15, except for three applications. Therefore, using 4-bit saturating counters for both the event counter and thresholds is sufficient for most applications. With 4-bit counters, the storage overhead per cache line is $3 \times 4 + 8 + 1 = 21$ bits, quite reasonable compared to a typical L2 cache line size of 64 or 128 bytes. Finally, the table shows that the maximum counter values used as threshold values show repetitive patterns across generations. In all but seven applications, the threshold value discovered in the current generation is the same as the threshold value discovered in the prior generation more than 70 percent of the time. As described later, we will exploit this observation for deciding the confidence of the predictor.

The prediction table, located between the L2 cache and main memory, stores the 4-bit counter threshold values for lines that are not cached. The prediction table is a direct-mapped tagless table organized as a 256×256 two-dimensional matrix structure, with its rows indexed by the *hashedPC* and its columns indexed by an 8-bit hashed line address as described earlier. The PC used is the PC of the instruction that misses on the line causing it to be brought into the cache. It is used because the access behavior of a cache line is often correlated with the code section from where the line is accessed. The total table size is $256 \times 256 \times (4 + 1)$ bits $= 40$ Kbytes, also very reasonable compared to typical L2 cache sizes ranging from a half to a few megabytes. The prediction table is used to store threshold information of lines that are not cached and also to restore threshold information for lines that are brought into the cache. Hence, it is only accessed during a cache miss and its latency can be overlapped with the cache miss latency.

3.2.2 Algorithm Details

The AIP algorithm implementation is shown in Fig. 4. On a cache access to a line x in a set, the event counters of all lines in the set are incremented, including x's own counter (Step 1). If x is found in the cache, its counter value now represents a completed access interval. If x's counter value is larger than the maximum value in the current generation ($x.maxC_{present}$), it becomes the new maximum value, then is reset to 0 so that it can measure the next access interval (Step 2). Therefore, in a single generation, over time, $x.maxC_{present}$ may keep increasing as new access intervals with larger values are discovered.

If, on the other hand, x is not found in the cache (Step 3), a victim line from the set needs to be selected for replacement. To find the victim, all lines in the set are checked for expiration. A line b is said to have expired if its counter $b.C$ is larger than the maximum values in the current generation ($b.maxC_{present}$) and in the prior generation ($b.maxC_{past}$) and its confidence bit ($b.conf$) is "1" (Step 3a). The reason why $b.C$ is not only compared with $b.maxC_{present}$ is because the current maximum access interval may not have been fully discovered yet, so

On an access to line x in set s:
1. Increment the event counter of each line b in set s:
   $b.C = b.C + 1$;
2. If the access is a hit, reset x's counter after recording the new threshold maximum:
   $x.maxC_{present} = \max(x.C, x.maxC_{present})$;
   $x.C = 0$;
3. If the access is a miss,
3a. Identify all lines in set s that have expired.
   A line b has expired if
   $b.C > b.maxC_{present}$, $b.C > b.maxC_{past}$ and
   $b.conf == 1$.
3b. Find a replacement line y :
   if there is at least one expired line in the set
   y is chosen randomly from among the expired lines;
   else $y =$ the LRU line;
3c. Update the prediction table by y's information:
   Index the table using $y.hashedPC$ and y's line address
   $y.maxC_{stored} = y.maxC_{present}$;
   if$(y.maxC_{present} == y.maxC_{past})$
   $y.conf_{stored} = 1$;
   else $y.conf_{stored} = 0$.
3d. Place line x in the cache:
   Index the prediction table using $x.hashedPC$ and
   x's line address. $x.hashedPC$ is obtained by performing
   8-bit XOR to the instruction PC that causes the miss to x
   $x.C = x.maxC_{present} = 0$;
   $x.maxC_{past} = x.maxC_{stored}$;
   $x.conf = x.conf_{stored}$;

Fig. 4. AIP algorithm implementation.

$b.maxC_{past}$ provides a better guidance. Note that, if one of $b.maxC_{present}$ or $b.maxC_{past}$ is saturated, then the counter can never exceed that value, which means the line cannot expire. This is a design trade-off in which we favor small 4-bit counters at the expense of a slight reduction in prediction coverage. Recall that Table 1 shows that there are only fewer than 5 percent access interval thresholds that cannot be represented with 4-bit counters in most applications. Moreover, our experiments with larger counters do not produce noticeable extra performance compared to using 4-bit counters, further justifying the trade-off. Finally, $conf$ serves to prevent mispredictions when the behavior of a line is unstable, such as when it is transitioning from one behavior phase to another. $conf$ is set when a stable phase, defined as a single threshold value repeated over two generations, is detected.

After expired lines have been identified, a line y among them is chosen for replacement (Step 3b). If there are several expired lines, one of them is randomly selected for replacement. If no line has expired, the LRU line is selected for replacement.

When y is replaced, its counter information updates an entry in the prediction table (Step 3c). If the current maximum $y.maxC_{present}$ is equal to the prior generation maximum counter ($y.maxC_{past}$), then a stable phase in which threshold values are likely to be repeated has been found. Thus, the confidence bit in the table ($y.conf_{stored}$) is set. Otherwise, it is reset to 0. It is possible for multiple lines to hash to the same prediction table entry, potentially causing aliasing problems in the confidence bit and threshold value. However, in practice, we found this effect to be negligible, in part because the table indexing function is quite randomized due to utilizing both the $hashedPC$ and the line address to index the table, so consecutive PCs or addresses are unlikely to fall into the same entry. Moreover,

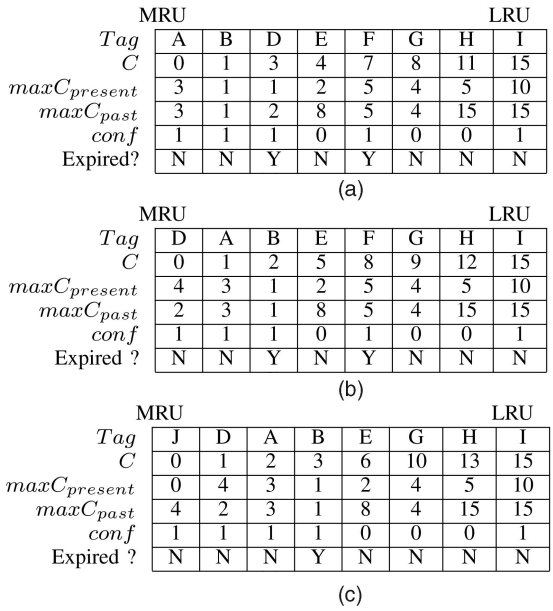

Fig. 5. Example of AIP implementation for an 8-way set. Lines are sorted from the MRU line (left) to the LRU line (right). (a) The initial states, (b) the states after an access (hit) to line D, and (c) the states after an access (miss) to line J.

we found that, in many applications, different lines hashing to the same prediction table entry exhibit similar behavior (counter threshold values), reducing the learning time of the algorithm. This is further evaluated in Section 6.4.

Finally, a newly fetched line x is placed in the cache (Step 3d). Its event counter $(x.C)$ and present maximum counter $(x.maxC_{present})$ are initialized to zero, whereas its past maximum counter $(x.maxC_{past})$ and confidence bit $(x.conf)$ are copied from the prediction table.

Note that cache latency critical paths are not affected much. Steps 1 and 2 can be performed after a cache access is completed. Steps 3a-d can be overlapped with the cache miss latency. In addition, the AIP algorithm only involves several 4-bit increment and compare operations, which require modest hardware support.

To illustrate the working of the AIP algorithm, we show an example in Fig. 5. The figure shows a set in an 8-way set associative cache. The lines are sorted from the MRU line (left) to the LRU line (right). The figure shows the different values of C, $maxC_{present}$, $maxC_{past}$, and $conf$ for each line. Fig. 5a shows the initial state with two expired lines (D and F) because their confidence bits are set and their C values are larger than their maximum counters ($maxC_{present}$ and $maxC_{past}$). Note that line G has not expired because $G.conf$ is zero. Fig. 5b shows the set after a cache access (hit) to line D. Line D now becomes the MRU line, the counter values for all the lines in the set are incremented, except for ones that have already saturated their 4-bit counters (line I). Updating the event counter values results in B becoming a new expired line. Since an access interval has just been completed for D, $D.maxC_{present}$ is updated with the value of $D.C$ (that is, 4) and, then, $D.C$ is reset to 0. Interestingly, line D is no longer expired since it is reaccessed. Finally, let us assume an access to line J results in a cache miss (Fig. 5c). In this case, because both lines B and F have expired, we randomly choose a line to replace from among them (F in the example). The values $J.maxC_{present}$ and $J.C$ are initialized to zero, whereas the values $J.maxC_{past}$ and $J.conf$ are initialized from the prediction table.

TABLE 2
Profile of the Counter Threshold Values in the Counter Prediction Tables for LvP

App.	Avg.	>15	=last
ammp	0.21	0.16%	97.8%
applu	0.23	0.00%	97.9%
apsi	2.46	0.86%	95.2%
art	0.00	0.00%	99.9%
bzip2	2.17	2.13%	54.7%
crafty	33.08	8.21%	24.1%
equake	0.21	0.10%	94.8%
gap	0.05	0.01%	91.8%
gcc	0.42	0.04%	99.4%
gzip	0.27	0.15%	87.4%
mcf	0.26	0.38%	95.4%
mesa	6.99	0.64%	89.2%
mgrid	0.28	0.00%	99.3%
parser	4.45	4.03%	43.8%
perlbmk	0.97	0.92%	78.0%
swim	0.56	0.17%	87.4%
twolf	1.76	2.00%	43.6%
vortex	11.07	1.70%	58.6%
vpr	2.39	2.00%	40.6%
wupwise	0.21	0.00%	54.1%

The cache parameters follow those in Table 4.

3.3 LvP Algorithm Design and Implementation

Recall that LvP works by counting the number of accesses to a cache line starting from when it was placed in the cache. If the count reaches a threshold, it is considered dead and becomes replaceable. For the LvP algorithm to work well, the live times have to be predictable across generations. Fortunately, this is the case. Table 2 shows in the last column how often the live time of a line from the previous generation is repeated in the current generation. With the exception of seven applications (bzip2, crafty, parser, twolf, vortex, vpr, and wupwise), in more than 70 percent of the cases, the live times are repeated. In addition, the table points out that the average live times are small for all applications except for crafty and vortex and that the event counts during live times hardly ever become larger than 15. Hence, as with AIP, we only need 4-bit counters to count events and to store threshold values.

3.3.1 Prediction Structures

The LvP algorithm is implemented in a similar way as the AIP algorithm in Section 3.2. However, there are several differences that make LvP implementation simpler. First, LvP counts events during the entire live time of a line. Thus, $maxC_{present}$ is not needed, reducing the storage overhead per cache line to $2 \times 4 + 8 + 1 = 17$ bits. Furthermore, since, on each cache access, only the accessed line's counter is incremented, LvP requires a smaller number of increment operations. Although both AIP and LvP's increment operations can be performed off the critical path of cache access latency, LvP is more power efficient due to much fewer increment operations.

On an access to line x in set s:
1. If the access is a hit:
   $x.C = x.C + 1$
2. If the access is a miss,
2a. Identify all lines in set s that have expired.
   A line b has expired if
   $b.C >= b.maxC_{past}$ and $b.conf == 1$.
2b. Same as Step 3b in AIP
2c. Update the prediction table by y's information:
   Index the table using $y.hashedPC$ and y's line address
   $y.maxC_{stored} = y.C$;
   if($y.maxC_{past} == y.C$)
   $y.conf_{stored} = 1$;
   else $y.conf_{stored} = 0$.
2d. Place line x in the cache:
   Index the prediction table using $x.hashedPC$ and
   x's line address. $x.hashedPC$ is obtained by performing
   8-bit XOR to the instruction PC that causes the miss to x
   $x.C = 0$;
   $x.maxC_{past} = x.maxC_{stored}$;
   $x.conf = x.conf_{stored}$;

Fig. 6. LvP algorithm implementation.

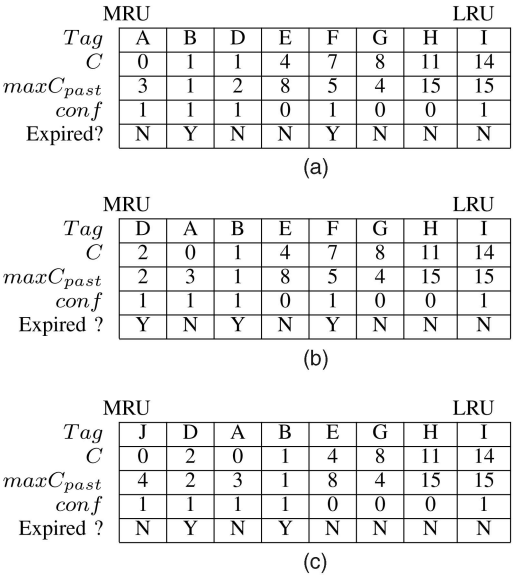

Fig. 7. Example of LvP implementation for an 8-way set. The figure shows (a) initial states, (b) the states after an access (hit) to line D, and (c) the states after an access (miss) to line J.

3.3.2 Algorithm Details

Fig. 6 shows the LvP algorithm in detail. Since LvP keeps track of the number of accesses to a line during its generation time, on a cache access, we only increment the line's own counter (Step 1). If the access is a cache miss, expired lines are identified if their event counters are larger than or equal to their thresholds and their confidence bits are set (Step 2a). Then, similarly to AIP, a line among the expired ones is randomly selected to be replaced (Step 2b). Then, the replaced line updates the prediction table (Step 2c). Finally, the new line is placed in the cache and its fields initialized (Step 2d).

To illustrate the working of the LvP algorithm, we show an example in Fig. 7. The figure shows a set in an 8-way set associative cache. The lines are sorted from the MRU line (left) to the LRU line (right). In Fig. 7a, there are two expired lines (B and F): Their counter values are equal to or larger than their thresholds ($maxC_{past}$) and their confidence bits are set. After a cache hit to line D in Fig. 7b, D becomes the MRU line and its event counter is incremented, which causes line D to expire. After an access to line J results in a cache miss in Fig. 7c, the selected line for replacement is randomly chosen to be F.

3.4 Comparison with Sequence-Based Replacement Algorithms

Sequence-based replacement algorithms record and learn sequences of memory events that lead up to the last use of a line. An event signature is often the PC and/or address of a memory access, whereas a sequence signature consists of a combination of past event signatures. Compared to sequence-based replacement algorithms, counter-based replacement algorithms are more storage efficient. Although it is not easy to encode an event signature in only four bits, we have shown that the counter of each line in our approach only needs to be 4 bits in size. In addition, in sequence-based algorithms, the longer a sequence is, the more storage is needed for keeping the sequence signature. With the counter-based approach, the *entire* past history of a line can be compactly summarized in a single threshold value, which only needs to be 4 bits in size. Consequently, for sequence-based algorithms to work best [3], [2], their authors use an on-chip prediction table that is several Mbytes in size. In contrast, our counter-based algorithms require only a 40 Kbytes prediction table and 21 or 17 bits per cache line overhead to store counters and threshold values, and larger structures do not yield noticeable extra performance improvement (Section 6.4).

Another significant difference is that sequence-based algorithms are often too aggressive. As soon as an event sequence pattern is identified, a line is identified as dead. In our counter-based approach, the choice of how aggressive the prediction is can be adjusted. For example, we have chosen to use a conservative prediction in which we take the maximum value of the access intervals or live times of a line. Consequently, although there is a slight increase in cases in which dead lines are not identified promptly, a cache line is seldom prematurely replaced. This choice contributes to the robustness of our AIP and LvP algorithms, which do not slow down any applications by more than 1 percent compared to LRU.

Finally, we will show in our evaluation results (Section 6.1) that, even with infinite prediction table sizes, compared to sequence-based algorithms, our counter-based algorithms achieve a higher coverage (79 percent versus 72 percent) and fewer premature replacements (4 percent versus 12 percent).

3.5 Comparison with the Time-Based Approach

As mentioned in Section 2, time-based mechanisms have only been evaluated for power optimizations [4], [5], [6], [7]. However, the basic idea in time-based mechanisms that a cache line's access interval or live time can be measured in time units (clock cycles) rather than event counts logically implies that they can be used as replacement algorithms. Unfortunately, time-based replacement is difficult to implement due to several factors.

First, the number of cache misses and replacements are more closely correlated with the number of cache accesses than with the number of clock cycles. For a time-based approach to work well, the timer has to accommodate cache timing variations of a wide range of applications. The time step needs to be fine grained enough to accommodate applications with frequent cache accesses (per unit time), but also needs to be coarse grained enough to accommodate applications with a low L2 cache access frequency. If the time step is too coarse, it cannot distinguish between an access interval with zero or with few accesses. On the other hand, if the time step is too fine, large timer storage is needed.

In addition, a time-based approach is harder to implement in real processors because the processor clock frequency frequently changes due to power/energy optimizations. Such changes do not change the time intervals that need to be measured in linear proportion. Hence, to work well in real implementations, a time-based approach will have to be made aware of many other system settings.

Finally, a time-based approach may not accurately identify never-reaccessed lines, which is essential for performing cache bypassing.

3.6 Other Implementation Issues

Counter bookkeeping implementation. To keep the hardware for incrementing counters very simple, we use a single 4-bit increment unit and a single compare unit per set for both AIP and LvP. In AIP, since all counters in a set are incremented on each access to any line in the set, we perform the increment operations sequentially. We note that, because the average time between two successive accesses to a cache set is often larger than 2,000 cycles, sequentially incrementing a set's counters results in negligible performance impact. Moreover, since counters need to be read and compared only when a cache replacement needs to be made on a cache miss, they can be performed in parallel with the cache miss latency. However, in terms of the number of increment operations, it is clear that LvP is a lot more power efficient than AIP.

Applicability to SMT/CMP systems. Since L2 cache sharing in SMT and CMP systems increases cache pressure and capacity misses for each thread that shares the cache, we believe that our counter-based replacement is even more applicable. However, we leave such a study for future work.

LRU and the underlying replacement algorithm. Although AIP and LvP divide cache lines into groups of expired and nonexpired lines, the selection of the victim line from each group follows the underlying replacement algorithm. As a result, AIP and LvP can be implemented over any cache replacement algorithms. For example, with random replacement, the replacement policy first chooses a victim line randomly from the group of expired lines. If no line has expired, then it chooses a victim line from the group of nonexpired lines. In our previous discussion and experiments, we assumed the underlying algorithm to be LRU, but note that there are no major reasons why other replacement policies cannot be used.

Prediction table initialization and resetting. When a new thread (application) starts running on the processor, the prediction table is reset by initializing all threshold values in it to 15 and all confidence bits to 0. Threshold values are not initialized to 0 to avoid misinterpreting a zero-threshold value as the case when a line, in its previous generation time, was not used after being brought into the cache. As we will see in Section 4, such lines will bypass the L2 cache the next time there is an attempt to bring them into the cache. Furthermore, since new threshold values are continuously being learned and recorded during runtime, there is no need to reset the table during execution. On context switches, we have a choice of whether to treat the prediction table as a process context (so that it is saved and restored) or simply reset it. Since our prediction table is small, it does not take long to warm it up and, hence, resetting it does not adversely impact prediction accuracy by much. To test this, we evaluated resetting the prediction table every 1 billion cycles and then every 100 million L2 accesses, but we only noticed negligible performance changes in both cases.

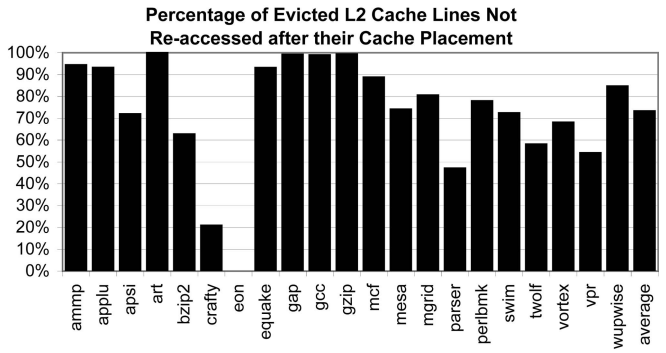

Fig. 8. Percentage of evicted L2 cache lines that were never reaccessed after being brought into the cache. The cache parameters follow those in Table 4.

4 Counter-Based Cache Bypassing

In addition to improving cache performance through early detection of dead lines, counter-based prediction can also be used to identify opportunities for bypassing. There are many cache lines that exhibit bursty temporal reuses. This is often due to spatial reuses of different bytes of the same cache line that tend to occur in bursts. Current caches typically have large cache lines (64 or 128 bytes), amplifying this bursty reuse pattern. With a single-level cache, this bursty temporal reuse would be well accommodated by the LRU replacement algorithm. However, in multilevel caches, this bursty pattern often manifests only at the L1 cache and is filtered by the L1 cache. To the L2 cache, the line does not appear to have temporal reuse since it is brought into the cache but is not used until it is replaced. Hence, the lines are immediately dead after they are brought into the L2 cache. Such *never-reaccessed* lines unnecessarily waste the L2 cache capacity.

Fig. 8 shows the fraction of evicted L2 cache lines that were never reaccessed after being brought into the cache. It ranges from 21 percent to almost 100 percent (there are no L2 cache evictions in the case of eon) and is above 70 percent in most cases. Note that immediately replacing them after they are brought into the L2 cache can be achieved with our counter-based replacement algorithms; however, it would only be partially beneficial because the lines already displaced other lines that may not be dead yet. A better

approach is to identify them and avoid placing them in the L2 cache in the first place using a technique referred to as cache bypassing. To support bypassing, it is necessary to relax the multilevel cache inclusion between the L1 and L2 caches. The cache inclusion property guarantees that the lines in an upper level cache are also stored in the lower level caches. Many modern cache implementations, including the AMD Opteron, no longer enforce the inclusion between L1 and L2 caches [29]. Typically, the cost of relaxing the inclusion property is in duplicating the L1 tags so that the tags can be checked against coherence requests without blocking the processor from accessing the L1 cache [30]. Since L1 caches are typically small in size (8-64 Kbytes), duplicating the L1 cache tags would incur negligible storage overhead.

Note that bypassing, if not performed conservatively, may easily degrade performance. Since bypassed lines are only placed in the L1 cache, if they are accidentally replaced from the L1 cache, they need to be reloaded from the memory (instead of from the L2 cache), incurring a higher latency. Since the L1 cache is small and has a low associativity, it is easy for a line to be replaced shortly after it is placed. Therefore, our design principle is that bypassing algorithms should not only *identify never-reaccessed lines*, but bypass them only if there is *no spare capacity* in the L2 cache because, if there is spare capacity at the L2 cache, it would not hurt the performance to also place the line in the L2 cache. Fortunately, our counter-based replacement algorithms provide mechanisms to achieve both objectives. First, both AIP and LvP already have the ability to detect never-reaccessed lines. A never-reaccessed line is detected when $maxC_{stored} = 0$ and $conf = 1$, indicating that, in the previous two generations, the line was not reaccessed while it resided in the L2 cache. Thus, it is quite likely that, in the next generation, the line will exhibit the same behavior. Second, spare capacity is detected when there is at least one dead line available to replace from the L2 cache, indicating that the processor no longer needs it, and replacing it with the requested line will not impact performance. In this case, bypassing is not performed, and the line is placed in the L2 cache, as well as the L1 cache. Note also that, in this case, a relearning opportunity is given to the line in case its reuse behavior has changed. However, if no spare capacity is detected (no expired line is found), all of the lines in the set are *live* and are still needed by the processor. In this case, bypassing is performed and the requested line that is predicted to be never-reaccessed is placed only in the L1 cache. Finally, when a modified line in the L1 cache is written back to the L2 cache and the line is not found in the L2 cache, the write back is forwarded to the L2's lower level memory hierarchy component.

In the case of write-through caches, the L1 cache will no longer filter out bursty temporal write operations to a single cache line from the L2 cache. We expect this to reduce the fractions shown in Fig. 8, thus bypassing opportunities. However, if a cache line does not exhibit bursty temporal write operations, bypassing opportunities will still exist. We leave such an evaluation as future work.

TABLE 3
The 21 Applications Used in Our Evaluation

Group A			
App	512KB L2 Misses	512KB L2 Miss Rate	1MB L2 Miss Rate
ammp	85147509	79.6%	69.5%
apsi	12142896	28.5%	11.9%
art	307862682	99.9%	72.7%
bzip2	6557733	31.1%	19.5%
gcc	89271818	70.3%	3.5%
mcf	234536895	79.2%	72.7%
mgrid	24649077	78.2%	42.0%
swim	63121429	64.3%	59.1%
twolf	25878177	34.7%	10.9%
vpr	15226842	24.2%	4.3%
Group B			
App	512KB L2 Misses	512KB L2 Miss Rate	1MB L2 Miss Rate
applu	23047662	80.9%	80.8%
crafty	392976	0.8%	0.2%
eon	1529	0.02%	0.02%
equake	29272671	82.6%	82.0%
gap	6619135	94.8%	94.8%
gzip	444278	1.1%	1.1%
mesa	1532404	11.6%	11.4%
parser	7499913	18.4%	13.7%
perlbmk	2457095	8.9%	7.8%
vortex	1585701	6.5%	5.3%
wupwise	7802399	82.7%	82.7%

5 Evaluation Environment

Applications. To evaluate the counter-based algorithms, we use 21 of the 26 Spec2000 applications. All Fortran90 applications (facerec, fma3d, galgel, and lucas) and sixtrack are excluded because our compiler does not support them. The applications and their L2 cache miss rates for both 512 KBytes and 1 MBytes L2 caches are summarized in Table 3. The applications were compiled with `gcc` using `-O3` optimization flag. The reference input set was used for all applications. Each application is simulated for three billion instructions after skipping the first two billion instructions.

The applications are divided up into two groups based on whether their performance is capacity-constrained at the L2 cache level. An application is said to be capacity-constrained if its number of L2 cache misses can be reduced by more than 5 percent when the L2 cache size is increased from 512 KBytes to 1 MByte. The reason for this categorization is that, because our counter-based replacement and bypassing free up wasted capacity due to dead lines and never-reaccessed lines, so, naturally, they would only benefit applications that are capacity-constrained. The capacity-constrained *Group A* consists of five SpecFP and five SpecINT applications. The remaining 11 applications that are not capacity-constrained are lumped into Group B. Our experiments confirmed that, under our algorithms, Group B applications' execution times are virtually unchanged (speedups range from -1 percent to 1 percent). Consequently, most of the results presented will focus on Group A applications.

Simulation environment. The evaluation is performed using a detailed execution-driven simulation environment that supports a dynamic superscalar processor model [31]. Table 4 shows the parameters used for each component of the architecture. The architecture is modeled cycle by cycle.

TABLE 4
Parameters of the Simulated Architecture

PROCESSOR
6-issue dynamic. 5 GHz. INT/FP/LD/ST FUs: 8/8/2/2
Pending LD/ST: 48/48. Branch penalty: 12 cycles
ROB entries: 256. INT/FP registers: 156/156
MEMORY
L1 data: WB, 16 KB, 2 way, 64-B line, 2-cycle hit RT
L1 inst: WB, 16 KB, 2 way, 64-B line, 2-cycle hit RT
L2 unified: WB, 512 KB, 8 way, 64-B line, 10-cycle hit RT
RT mem lat: 75 ns
Memory bus: split-trans, 8 B, 400 MHz
Dual channel DRAM. Each channel: 2 B, 800 MHz
PREDICTOR
Prediction table: 40-KB, tagless, direct-mapped, 4-cycle access time
Counter size: 4 bits.

Latencies correspond to contention-free conditions. RT *stands for round-trip* from the processor.

6 Evaluation

In this section, we present and discuss several sets of evaluation results. Section 6.1 shows the prediction accuracy and coverage of LvP and AIP compared to a sequence-based, a time-based, and another counter-based dead line prediction algorithm. Section 6.2 evaluates the performance of our counter-based replacement algorithms compared to a sequence-based algorithm, a time-based algorithm, and another counter-based algorithm using a limited prediction table size. Section 6.3 discusses the performance of the counter-based bypassing algorithms. Finally, Section 6.4 studies the effect of changing the cache parameters and the counters and prediction table sizes on the performance of the counter-based algorithms.

6.1 Coverage and Accuracy

Fig. 9 shows a limit study of sequence-based, time-based, and counter-based algorithms. The goal of the study is to evaluate the different algorithms' maximum ability to learn, store, and make correct predictions about the behavior of dead lines. The sequence-based algorithm that we choose is the DBP [3]. The time-based algorithm we choose is IATAC [4], which was recently proposed by Abella et al. to predict dead cache lines in the context of power reduction. Their implementation can be extended and used as a time-based replacement algorithm as well. IGDR [9] is a counter-based algorithm that differs from LvP and AIP in several implementation details (Section 2).

To explore the maximum performance potentials of all of the algorithms, we use *unlimited prediction structure sizes* and *full learning* for all algorithms. Under full learning, each algorithm is first trained during the first run of an application, in which Belady's OPT [32] is used for making replacement decisions. OPT is not implementable but represents the theoretical upper bound performance of a replacement algorithm because it takes into account future information. The studied algorithms simply observe and record what they learn from the OPT algorithm. During the application's second run, OPT is still used alone to make replacement decisions, but the other algorithms, already fully trained, are allowed to select victim lines to replace (without actually replacing them), and the selection is compared against the selection made by OPT. If an algorithm's selection of a line to replace agrees with OPT, it is categorized as *Correct Prediction*. If the selection does not match, it is categorized as *Wrong Prediction*. Finally, if the algorithms are unable to make a selection (no dead lines are predicted), it is categorized as *Not Predicted*. The bars are normalized to the original number of OPT replacements. Since the sum of *Correct Prediction* and *Not Predicted* is necessarily equal to the original number of OPT replacements, it stands at 100 percent in the figure. Therefore, *Wrong Prediction*s are the extra replacements due to mispredictions, which would often lead to premature cache line replacements and subsequent cache misses if replacement decisions are made by the algorithms.[1]

The figure shows that, on average, IATAC, IGDR, AIP, and LvP correctly predict more replacements than the sequence-based DBP (79 percent for both IATAC and LvP versus 78 percent for both IGDR and AIP versus 72 percent for DBP) while simultaneously achieving significantly fewer wrong replacements (4 percent for AIP and LvP versus 6 percent for IATAC and IGDR versus 12 percent for DBP). Comparing AIP, LvP, IGDR, and IATAC, we notice that they perform almost similarly, with slightly fewer wrong replacements in AIP and LvP. Note that being a time-based approach, IATAC is harder to implement than AIP and LvP (as discussed in Section 3.5). Moreover, the implementation of IGDR is costly as it requires several per cache line updates on every access in addition to many multiplications and divisions to compute and update tables regularly; hence, AIP and LvP present a more attractive solution. Let us consider applications that show low correct prediction rates: bzip2, twolf, and vpr. LvP and AIP are unable to predict many replacements because the counter threshold values are nonrepeating, as shown in Tables 1 and 2. However, due to the conservative threshold selection, AIP and LvP refrain from replacing cache lines, resulting in low *Wrong Prediction* rates. However, we note that these same applications are also problematic for the sequence-based DBP algorithm. DBP achieves low *Correct Prediction* rates for these applications. Unfortunately, even for these applications, DBP is still aggressive in replacing lines even though their behavior is not highly predictable. As a result, unlike our AIP and LvP, DBP also incurs high *Wrong Prediction* rates.

6.2 Performance of Counter-Based Replacement Algorithms

Although the previous section has discussed the upper bound performance potentials of AIP and LvP compared to sequence-based and time-based algorithms under unlimited prediction table sizes and full learning, this section compares their performance against other feasible alternatives under a more realistic implementation. Fig. 10 shows the percentage of IPC improvement of the different algorithms over LRU replacement. The total storage overhead of AIP and LvP, including the prediction table and

1. More conventional measures can be derived from the figure. *Coverage*, defined as the percentage of OPT replacements that are predicted, is equal to *Correct Prediction* since the total number of replacements is normalized at 100 percent. *Accuracy*, defined as the fraction of replacements that are correct, is equal to *Correct Prediction* divided by the sum of *Correct Prediction* and *Wrong Prediction*.

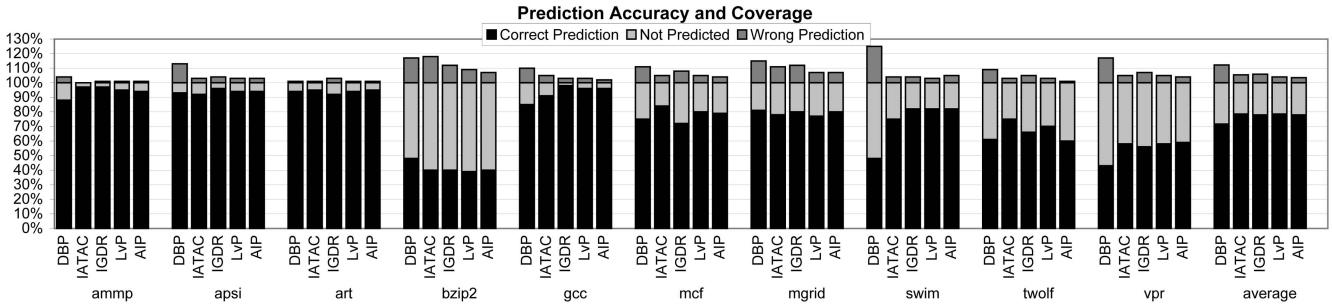

Fig. 9. Coverage and accuracy of the various algorithms with infinite table sizes.

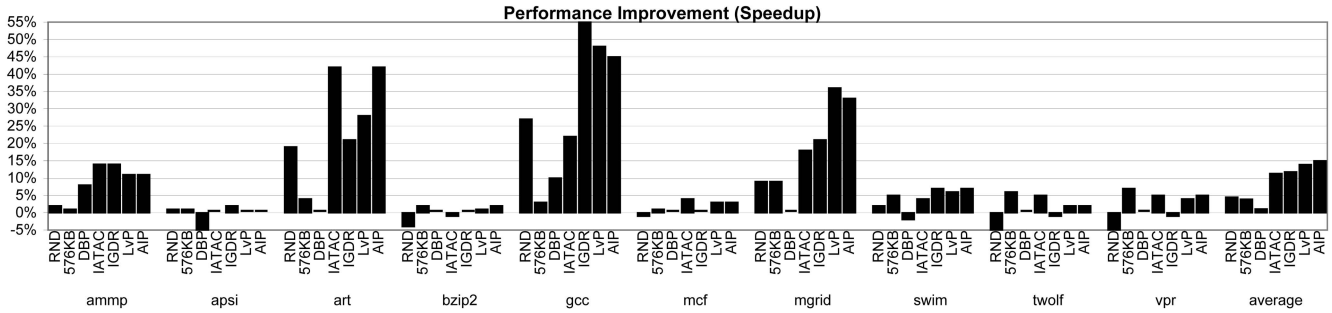

Fig. 10. Performance improvement results of various replacement algorithms.

extra fields per cache line is 61 KBytes. Thus, we compare our algorithms against several schemes. The first scheme is a random replacement algorithm (RND), which is used to demonstrate that the performance improvement of our algorithms is not due to the randomness in selecting a dead line to replace. The second scheme simply uses a larger $(512\ \text{KBytes} + 64\ \text{KBytes} = 576\ \text{KBytes})$ L2 cache organized as a 9-way LRU cache without any prediction mechanisms. The second group of schemes are DBP, IATAC, and IGDR using comparable prediction structure sizes (64 KBytes upper bound). For DBP and IATAC, the algorithms use the same schemes presented by their authors to predict dead lines; however, when a line has to be chosen for replacement, the line is chosen randomly from among the dead lines in the set, similarly to LvP and AIP.

Fig. 10 shows that, on average, AIP and LvP perform the best, achieving an average speedup of 14-15 percent without slowing down any applications, demonstrating their robust performance. *gcc* shows the greatest performance gain for AIP and LvP (45 percent and 48 percent speedup, respectively). IATAC and IGDR also perform well, but not as well as LvP and AIP, achieving an average speedup of 11 percent and 12 percent, respectively, and a maximum of 55 percent for gcc using IGDR. Due to the small prediction structures, the performance gap between DBP versus AIP, LvP, IATAC, and IGDR increases compared to when unlimited prediction tables are used (Fig. 9). Consequently, since AIP and LvP perform slightly better than IATAC and IGDR while, at the same time, having less implementation complexity (Section 3.5 and Section 2), we believe that AIP and LvP present a more attractive solution. DBP's average speedup is modest: 1 percent. With small predictors, DBP is unable to store enough sequence signatures to make many predictions and signature aliasing prevents it from making accurate predictions. The performance improvement from increasing the cache size to 576 Kbytes is insignificant (4 percent) or roughly only one fourth of the improvement obtained by our AIP and LvP algorithms. Moreover, an RND performs better than LRU for the shown applications. However, its improvement is less than one third the improvement of AIP and LvP, demonstrating that AIP and LvP's performance improvement is mainly due to their ability to correctly predict dead lines. Overall, the figure shows that AIP and LvP provide the best performance improvement. For the applications in group B, DBP, IATAC, IGDR, LvP, and AIP do not speed up or slow down any application by more than 1 percent. The RND slows down some of these applications by up to 5 percent.

6.3 Performance of Counter-Based Bypassing Algorithms

Fig. 11 shows the percentage of IPC improvement when we combine AIP and LvP with cache bypassing, compared to AIP and LvP without cache bypassing. The IPC improvement is over the LRU replacement. It also shows that, in general, bypassing adds roughly 2 percent performance gain compared to AIP and LvP alone, which is roughly 13 percent over the performance improvement of AIP and LvP alone. Bypassing brings the total speedup to 15 percent for LvP+Byp and 17 percent for AIP+Byp. Although an extra 2 percent improvement may seem minor, keep in mind that it is achieved for free, that is, without adding any new hardware over AIP and LvP alone. In addition, some of the benefits of bypassing are already realized by AIP and LvP because they can identify never-reaccessed lines as dead shortly after they are placed in the L2 cache.

We further analyze the performance improvement obtained by the algorithms just described, in terms of the number cache misses that are eliminated. Fig. 12 shows the normalized cache misses. The bars represent the four counter-based algorithms shown in Fig. 11. All bars are

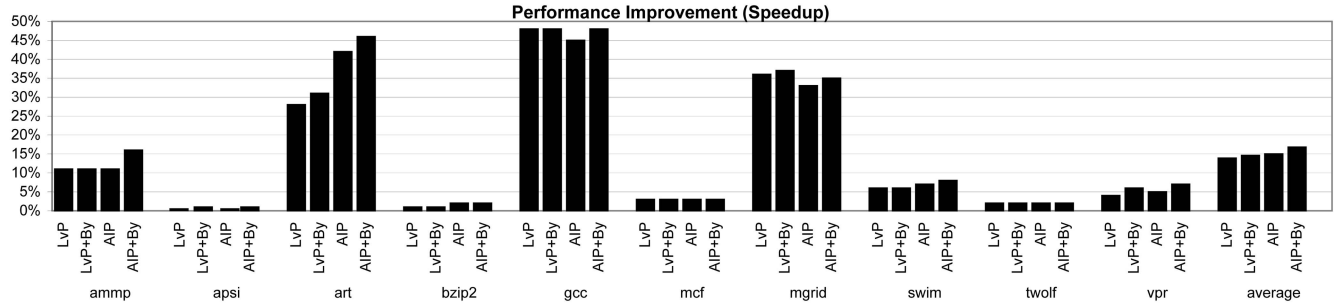

Fig. 11. Performance improvement results of various replacement and bypassing algorithms.

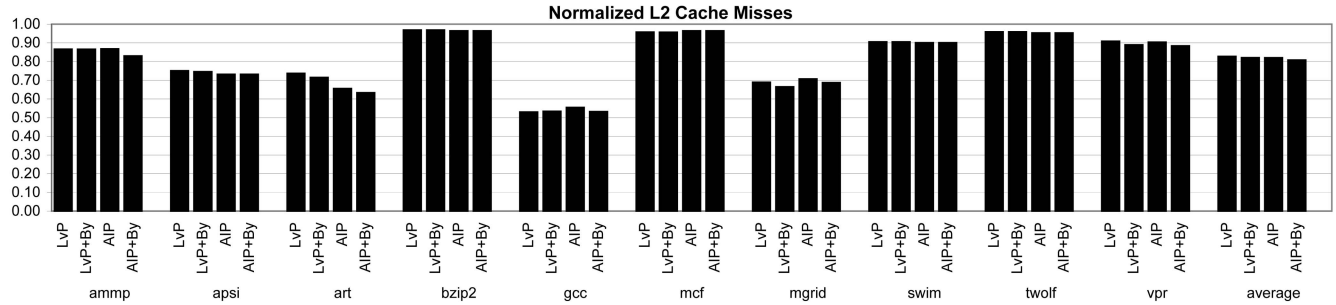

Fig. 12. Normalized L2 cache misses.

normalized to the number of cache misses when LRU is used alone. The number of misses and miss rates for the base (LRU) case are summarized in Table 3. The figure shows that the counter-based algorithms are capable of reducing a large fraction of misses. On average, AIP+By eliminates 19 percent of the cache misses.

To investigate the performance better, Fig. 13 shows the fraction of replacements that are triggered by the counter-based replacement algorithms (*Predicted*) and ones that are triggered by the LRU replacement algorithm (*LRU*) when AIP is implemented. A line is replaced using LRU replacement when there are no expired lines in the set to replace. All bars are normalized to the total number of replacements in the base case where the L2 cache uses the LRU replacement algorithm. Note that the sum of *Predicted* and *LRU* represents the new number of replacements (or number of misses) obtained by our algorithms normalized to when only the LRU replacement algorithm is used. Thus, the lower the sum, the fewer the cache misses. The figure shows that, on average, 19 percent of the original cache misses are eliminated, with the biggest gain being for art, gcc, and mgrid. In addition, the fraction of replacements triggered by the AIP algorithms is high and ranges between 41 percent to 91 percent, with an average of 68 percent.

Fig. 14 shows the percentage of fetched L2 cache lines with bypassing opportunities due to our counter-based bypassing algorithm when AIP+By is used. Note that not all such lines do bypass the cache because our design choice is not to bypass the cache if there is spare capacity due to expired lines. The figure shows that 24 percent (swim) to 78 percent (ammp and twolf) of lines bypass the L2 cache and are stored only in the L1 cache. On average, 58 percent of all lines bypass the cache. This is due to the high percentage of L2 cache lines that are not reaccessed after being brought into the cache until they are replaced, as illustrated in Fig. 8. Comparing Figs. 8 and 14, we see that our counter-based algorithms can effectively predict a large portion of lines that can bypass the cache.

Now, we would like to discuss why some applications can be sped up and why some applications cannot be sped up. So far, we have not discussed applications in Group B. Unlike applications in Group A, these applications do not achieve more than 1 percent speedups or 1 percent slowdowns with sequence-based, time-based, and counter-based algorithms.

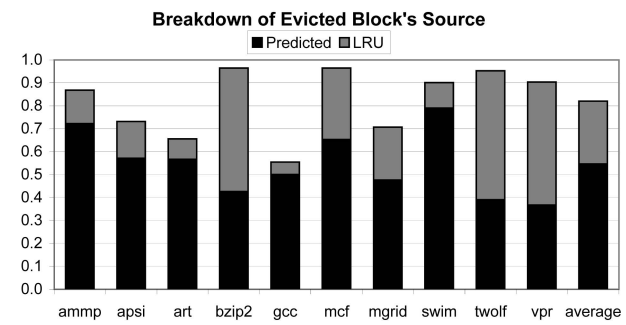

Fig. 13. Fraction of lines replaced by AIP versus by LRU.

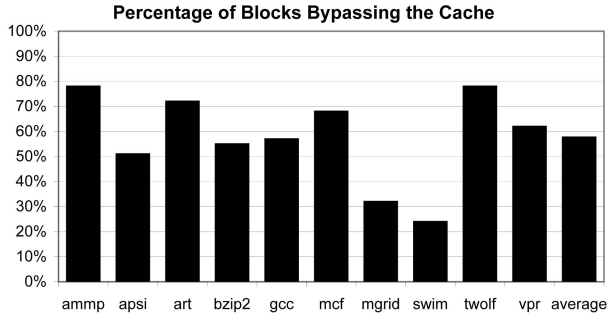

Fig. 14. Percentage of lines that bypass the L2 cache.

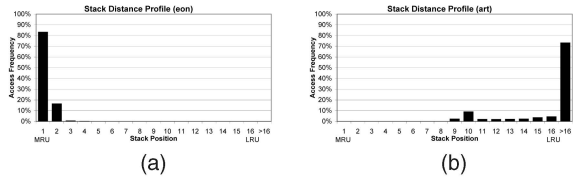

Fig. 15. Stack distance profiles of (a) eon and (b) art.

We found that their stack distance profiles [33], [34] reveal the reason for the low speedups. Fig. 15 shows two extremes: the stack distance profile for an application from group B (eon) and an application from group A (art). The stack distance profile collects the histogram of accesses to different LRU stack positions in the cache. For example, if there is an access to a line in the nth MRU position, the nth counter (nth stack position in Fig. 15) is incremented. When a cache has A associativity ($A = 8$ for our default cache configuration), a vertical line can be drawn between the Ath and $(A+1)$th stack positions. The sum of bars to the left of the line is the number of cache hits, whereas the sum of the bars to the right of the line is the number of cache misses. The stack distance profile can be used to compute the cache misses for different associativities (and, therefore, cache sizes). To see the impact of enlarging the cache size, we can move the vertical line to the right. To observe the impact of reducing the cache size, we can move the line to the left. Our counter-based algorithms can free up cache space occupied by lines that have expired, effectively increasing the cache space for other lines. However, Fig. 15a shows that, with eon, increasing the cache space does not affect its number of hits or misses. Hence, we do not obtain any performance improvement on eon even when the effective cache size is increased. On the other hand, Fig. 15b shows that, for art, if the cache size and associativity are increased, there are a significant number of cache misses that will turn into cache hits. Hence, our counter-based algorithms can and do improve the performance of art significantly. Most applications fall somewhere between art and eon and their stack distance profiles determine how much performance improvement can be expected by using the counter-based algorithms.

Overall, we found that if applications can benefit from a 10-50 percent larger L2 cache, they can benefit from the counter-based replacement algorithms.

6.4 Sensitivity to Cache Parameters

Figs. 16 and 17 study the sensitivity of AIP+By's performance to different cache sizes and associativities. Each bar represents the percentage of IPC improvement (speedup) of the AIP+By algorithm for applications in Group A over the base case where the LRU replacement algorithm is used. In Fig. 16, we try 256 Kbyte, 512 Kbyte, 1 Mbyte, and 2 Mbyte cache sizes while keeping the associativity at 8-way. Fig. 16 shows that AIP+By performs well over the different cache sizes, performing the best with a 2 Mbyte cache (25 percent speedup), followed by a 512 Kbyte cache (17 percent speedup). Examining individual applications, we find that the speedup is correlated with the application's working set size relative to the cache size. For example, going from a 512-Kbyte L2 cache to a 1-Mbyte L2 cache, the performance improvement of several applications (apsi, art, gcc, mgrid, swim, and twolf) decreases because, now, most of their working sets fit better in the cache. However, for other applications (ammp and mcf), the performance improvement increases because their working sets are now not too much larger than the cache.

In Fig. 17, we try 4-way, 8-way, and 16-way associativities while keeping the cache size constant at 512 Kbytes. Fig. 17 shows that AIP+By performs well across all associativities with performance gains ranging between 11 percent for a 4-way cache to 17 percent for an 8-way

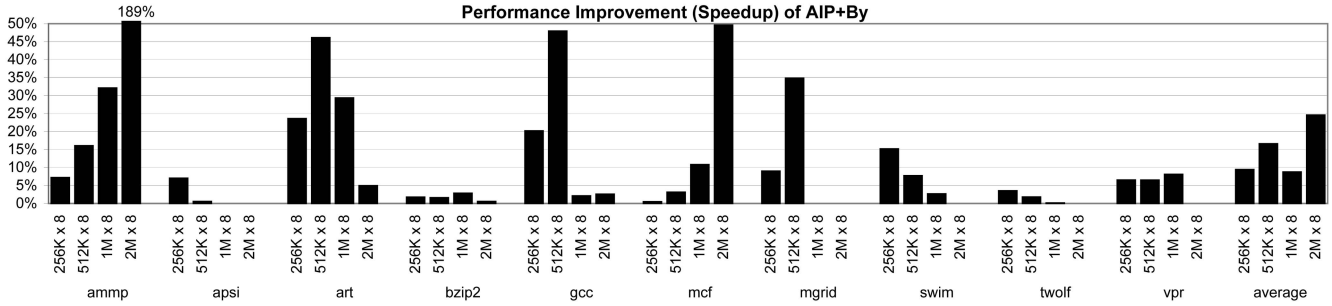

Fig. 16. Effect of changing the cache size on AIP+By's performance.

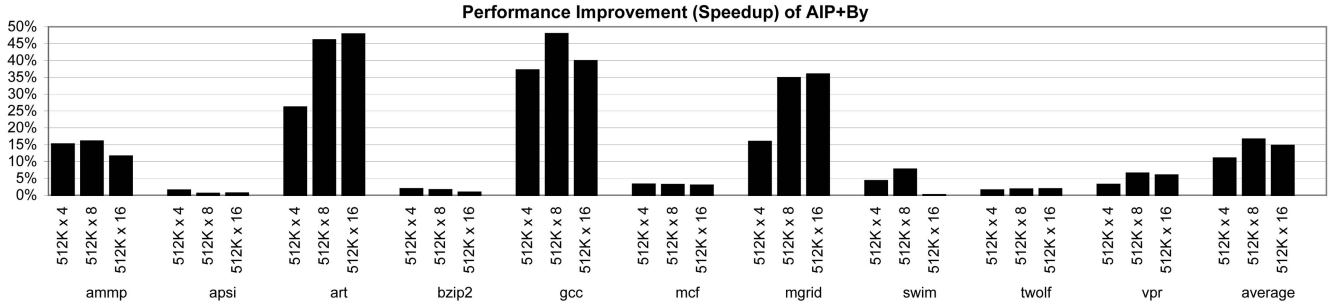

Fig. 17. Effect of changing the cache associativity on AIP+By's performance.

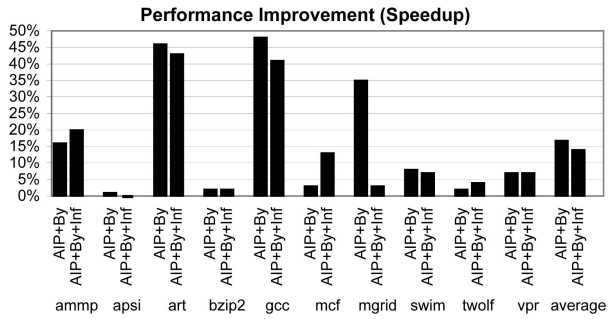

Fig. 18. Effect of using infinite counter and table sizes on AIP+By's performance.

cache. There is no clear trend between the cache associativity and the performance gain from AIP+By.

Finally, Fig. 18 studies the effect of limiting the counter and table sizes on the performance of AIP+By, including the effect of aliasing in the prediction table. Fig. 18 shows that, on average, AIP+By with limited counter and table sizes performs better than when infinite storage structures are used (AIP+By+Inf). We investigated this further and found that conflicts in the prediction table are, in many cases, beneficial as they reduce the learning time if the lines mapping to the same entry exhibit the same behavior. In the other case, where the lines mapping to the same table entry do not exhibit the same behavior, the confidence bit will remain 0, preventing incorrect predictions.

7 Conclusions

We have presented a new counter-based approach to deadline prediction and cache bypassing. In this approach, each line in the cache is augmented with an event counter that is incremented on each relevant event. When the counter reaches a threshold, the line is detected as dead and becomes replaceable. Replacing dead lines early makes space for lines that are not dead yet. For cache bypassing, lines that are brought into the L2 cache but are never reaccessed while they are resident are identified. Later, they are placed directly in the L1 cache, bypassing the L2 cache. By not placing dead lines and never-reaccessed lines in the L2 cache, we free up wasted capacity for other lines, hence improving the effective capacity of the L2 cache.

We found that the applications that benefit the most from our dead line replacement and bypassing are ones that are capacity constrained, for which an improvement in the effective capacity reduces their miss rates considerably. Our AIP and LvP perform almost equally well, speeding up 10 capacity-constrained (out of 21) Spec2000 benchmarks by up to 48 percent and 15 percent on average (7 percent on average for the whole 21 Spec2000 applications). Combined with bypassing, their speedups improve further to 17 percent on average. Due to using conservative design choices, the performance of our algorithms is robust: None of the 21 tested applications is slowed down by more than 1 percent for both algorithms on all cache parameter settings evaluated.

We compared our approach with sequence-based and time-based dead line prediction approaches. Due to the conservative design choices, our algorithms rarely suffer from replacing lines that do not agree with Belady's theoretical OPT, while, at the same time, achieving better coverage than the sequence-based approach, assuming unlimited prediction structure sizes for both approaches. In addition, due to the storage-efficiency of our counter-based approach, it significantly outperforms the sequence-based approach when comparable small prediction structures are used.

Acknowledgments

This work is supported in part by the US National Science Foundation through NSF Faculty Early Career Development Award Grant CCF-0347425, by North Carolina State University, and by the Jordan University of Science and Technology.

References

[1] W. Wong and J.-L. Baer, "Modified LRU Policies for Improving Second-Level Cache Behavior," *Proc. Sixth Int'l Symp. High-Performance Computer Architecture,* 2000.

[2] W.-F. Lin and S. Reinhardt, "Predicting Last-Touch References under Optimal Replacement," Technical Report CSE-TR-447-02, Univ. of Michigan, 2002.

[3] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-Block Prediction and Dead-Block Correlating Prefetchers," *Proc. 28th Int'l Symp. Computer Architecture,* 2001.

[4] J. Abella, A. Gonzalez, X. Vera, and M. O'Boyle, "IATAC: A Smart Predictor to Turn-Off L2 Cache Lines," *ACM Trans. Architecture and Code Optimization,* 2005.

[5] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power," *Proc. 28th Int'l Symp. Computer Architecture,* 2001.

[6] G. Chen, V. Narayanan, M. Kandemir, M. Irwin, and M. Wolczko, "Tracking Object Life Cycle for Leakage Energy Optimization," *Proc. ISSS/CODES Joint Conf.,* 2003.

[7] H. Zhou, M. Toburen, E. Rotenberg, and T. Conte, "Adaptive Mode Control: A Static-Power-Efficient Cache Design," *ACM Trans. Embedded Computing Systems,* 2002.

[8] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior," *Proc. 29th Int'l Symp. Computer Architecture* 2002.

[9] M. Takagi and K. Hiraki, "Inter-Reference Gap Distribution Replacement: An Improved Replacement Algorithm for Set-Associative Caches," *Proc. 18th Int'l Conf. Supercomputing,* 2004.

[10] K. Beyls and E. D'hollander, "Compile-Time Cache Hint Generation for EPIC Architectures," *Proc. Second Workshop Explicitly Parallel Instruction Computing Architecture and Compilers,* 2002.

[11] Z. Wang, K. McKinley, A. Rosenberg, and C. Weems, "Using the Compiler to Improve Cache Replacement Decisions," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* 2002.

[12] J. Jeong and M. Dubois, "Cache Replacement Algorithms with Nonuniform Miss Costs," *IEEE Trans. Computers,* vol. 55, no. 4, Apr. 2006.

[13] J. Jeong, P. Stenstrom, and M. Dubois, "Simple, Penalty-Sensitive Replacement Policies for Caches," *Proc. ACM Int'l Conf. Computing Frontiers,* May 2006.

[14] C. Chi and H. Dietz, "Improving Cache Performance by Selective Cache Bypass," *Proc. 22nd Ann. Hawaii Int'l Conf. System Sciences,* vol. 1, 1989.

[15] Y. Wu, R. Rakvic, L.-L. Chen, C.-C. Miao, G. Chrysos, and J. Fang, "Compiler Managed Micro-Cache Bypassing for High Performance EPIC Processors," *Proc. 35th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* 2002.

[16] T. Johnson, D. Connors, M. Merten, and W. Hwu, "Run-Time Cache Bypassing," *IEEE Trans. Computers,* vol. 48, no. 12, Dec. 1999.

[17] E. Tam, J. Rivers, V. Srinivasan, G. Tyson, and E. Davidson, "Active Management of Data Caches by Exploiting Reuse Information," *IEEE Trans. Computers,* vol. 48, no. 11, Nov. 1999.

[18] T. Johnson and W. Hwu, "Run-Time Adaptive Cache Hierarchy Management via Reference Analysis," *Proc. 24th Int'l Symp. Computer Architecture,* 1997.
[19] J. Rivers, E. Tam, G. Tyson, E. Davidson, and M. Farrens, "Utilizing Reuse Information in Data Cache Management," *Proc. 12th Int'l Conf. Supercomputing,* 1998.
[20] J. Rivers and E. Davidson, "Reducing Conflicts in Direct-Mapped Caches with Temporality-Based Design," *Proc. Int'l Conf. Parallel Processing,* 1996.
[21] V. Milutinovic, B. Markovic, M. Tomasevic, and M. Tremblay, "The Split Temporal/Spatial Cache: Initial Performance Analysis," *Proc. Int'l Workshop SCI-Based High-Performance Low-Cost Computing,* 1996.
[22] A. Gonzalez, C. Aliagas, and M. Valero, "A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality," *Proc. Ninth Int'l Conf. Supercomputing,* 1995.
[23] G. Tyson, M. Farrens, J. Matthews, and A. Pleszkun, "A New Approach to Cache Management," *Proc. 28th Int'l Symp. Microarchitecture,* 1995.
[24] L. Li, I. Kadayif, Y.-F. Tsai, N. Vijaykrishnan, M. Kandemir, M. Irwin, and A. Sivasubramaniam, "Leakage Energy Management in Cache Hierarchies," *Proc. 11th Int'l Conf. Parallel Architectures and Compilation Techniques,* 2002.
[25] H. Dybdahl and P. Stenström, "Enhancing Last-Level Cache Performance by Block Bypassing and Early Miss Determination," *Proc. Asia-Pacific Computer Systems Architecture Conf.,* 2006.
[26] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai, "Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching," *Proc. Int'l Conf. Supercomputing (ICS '02),* 2002.
[27] G. Keramidas, P. Xekalakis, and S. Kaxiras, "Applying Decay to Reduce Dynamic Power in Set-Associative Caches," *Proc. Int'l Conf. High Performance Embedded Architectures and Compilers,* 2007.
[28] D. Wood, M. Hill, and R. Kessler, "A Model for Estimating Trace-Sample Miss Ratios," *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems,* 1991.
[29] Advanced Micro Devices, "AMD Opteron Product Data Sheet," http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/23932.pdf, June 2004.
[30] D. Culler, J. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, 1999.
[31] V. Krishnan and J. Torrellas, "A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* 1998.
[32] L. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," *IBM Systems J.,* vol. 5, 1966.
[33] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems J.,* vol. 9, no. 2, 1970.
[34] J. Lee, Y. Solihin, and J. Torrellas, "Automatically Mapping Code on an Intelligent Memory Architecture," *Proc. Seventh Int'l Symp. High-Performance Computer Architecture,* 2001.

Mazen Kharbutli received the BS degree in electrical and computer engineering from the Jordan University of Science and Technology (JUST) in 2000, the MS degree in electrical and computer engineering from the University of Maryland, College Park, in 2002, and the PhD degree in computer engineering from North Carolina State University in 2005. He is currently an assistant professor in the Department of Computer Engineering at JUST. His research interests include computer architecture, memory hierarchy design, architectural support for security and reliability, and computer security. He has released HeapServer—a secure heap management library. He has received several scholarships and awards and is a member of the academic honor society Phi Kappa Phi. He is a member of the IEEE. More information can be found at http://www.kharbutli.com.

Yan Solihin received the BS degree in computer science from the Institut Teknologi Bandung in 1995, the MASc degree in computer engineering from Nanyang Technological University in 1997, and the MS and PhD degrees in computer science from the University of Illinois at Urbana-Champaign in 1999 and 2002. He is currently an associate professor with the Department of Electrical and Computer Engineering at North Carolina State University. He has graduated two PhD students and seven master's degree students. He has published more than 35 papers in computer architecture and image processing, which cover memory hierarchy design, architecture support for quality of service, security, and software reliability. He has released ACAPP—a cache performance model toolset, HeapServer—a secure heap management library, Scaltool—a parallel program scalability pinpointer, and Fodex—a forensic document examination toolset. He was a recipient of the 2005 IBM Faculty Partnership Award, 2004 NSF Faculty Early Career Award, and 1997 AT&T Leadership Award. He is a member of the IEEE and ACM SigMicro. More information can be found at http://www.ece.ncsu.edu/arpers.