To appear in the Proceedings of the International Conference on Computer Design, San Jose, Oct 2-5, 2005

Counter-Based Cache Replacement Algorithms *

Mazen Kharbutli and Yan Solihin
Department of Electrical and Computer Engineering
North Carolina State University
{mmkharbu,solihin}@eos.ncsu.edu

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, suggesting that alternative replacement algorithms can improve the performance of the cache. One main reason for this performance gap is that in the LRU replacement algorithm, a line is only evicted after it becomes the LRU line, long after its last access/touch, while unnecessarily occupying the cache space for a long time.

This paper proposes a new approach to deal with the problem: **counter-based L2 cache replacement** *. In this approach, each line in the L2 cache is augmented with an event counter that is incremented when an event of interest, such as a cache access to the same set, occurs. When the counter exceeds a threshold, the line "expires", and becomes evictable. When expired lines are evicted early from the cache, they make extra space for lines that may be more useful, reducing the number of capacity and conflict misses. Each line's threshold is unique and is dynamically learned and stored in a small 40-Kbyte counter prediction table. We propose two new replacement algorithms: Access Interval Predictor (AIP) and Live-time Predictor (LvP). AIP and LvP speed up 10 (out of 21) SPEC2000 benchmarks by up to 40% and 11% on average.*

1. Introduction

Recent studies have shown that in a highly associative caches such as the L2 cache, the performance gap between the Least Recently Used (LRU) and the theoretical optimal replacement algorithm is large (up to 197% increase in the number of misses) [14, 10], suggesting that improving replacement algorithms may improve the cache performance. One main reason for this performance gap is that with LRU replacement, a line that is no longer needed occupies the cache for a long time, because it has to become the LRU line in its set before it can be evicted. This *dead time* becomes worse with larger cache associativities. Although larger cache associativities in general improve performance, the larger dead time increases the performance gap between LRU and optimal replacements.

This paper proposes a new approach, **counter-based cache replacement** that significantly improves highly-associative cache replacement performance. In this approach, each L2 cache line is augmented with an *event counter* that is incremented when an event of interest (such as a cache access to a block or to any blocks in a set) occurs. When the counter exceeds a *threshold*, the line "expires", and immediately becomes evictable. We observe that data blocks in applications show different reuse patterns. A block can be reused frequently in a *bursty* or *non-bursty* manner, or may be infrequently reused. Blocks with bursty reuse pattern benefit typically result in cache hits, whereas blocks with infrequent reuse pattern typically result in cache misses. However, many frequently-reused blocks with a non-bursty reuse pattern often incur capacity cache misses if their *reuse distance* is large, i.e. they are reused soon after they are evicted from the cache. If such blocks are kept a little longer in the cache, their subsequent reuses will result in cache hits. Counter-based replacements predict blocks that will not be reused and evict them early. Thus, non-bursty reused blocks can be kept longer in the cache, turning their reuses into cache hits. Counter-based replacements distinguish different reuse patterns by keeping a unique and dynamically-learned expiration threshold per block in a small prediction table.

We propose two counter-based replacement algorithms, which differ by the type of intervals during which the events are counted. The first algorithm is called the *Access Interval Predictor* (AIP). For each cache line, it records the number of accesses to the set where the line resides, between consecutive accesses to the line. The event count threshold is selected conservatively as the maximum of such numbers. The second algorithm is called the *Live-time Predictor* (LvP). For each cache line, it records the number of accesses to itself during the interval in which the line resides continuously in the cache. The event count threshold is selected conservatively as the maximum of such numbers.

Both AIP and LvP only incur small storage and latency overheads: each cache line is augmented with 21 bits to store prediction information, equivalent to 4.1% storage overhead for a 64B line. In addition, a fixed-size 40-KB tagless direct-mapped prediction table is added between the L2 cache and its lower level memory components. The L2 cache access time is not impacted because the prediction table is always accessed in parallel with the latency of an L2 cache miss.

AIP and LvP speed up 10 out of 21 Spec2000 applications that we tested by up to 40% and 11% on average, while not slowing down any of the remaining eleven applications by more than 1%. We found that AIP and LvP outperform other predictors, both in terms of coverage (fraction of total evictions that are initiated by the predictors) and accuracy (percentage of counter-based evictions that agree with the theoretical optimal replacement), while using simpler hardware support compared to other predictors.

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

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

2. Related Work

To predict when to evict a cache block, two hardware approaches have been proposed. We refer to the first approach as *sequence-based* prediction, since it records and predicts the sequence of memory events leading up to the *last touch* of a line [8, 10]. Because for a given block, there may be many such sequences, such predictors require a large hardware table (a few Mbytes) to record the event sequences.

The second approach, which we refer to as *time-based* approach, tracks a line's timing in processor clock cycles, to predict when a line's last touch has likely occurred [5, 3, 16]. Although the time-based approach has only been evaluated in the context of cache leakage power reduction [5, 16] and prefetching [4], we apply them for cache replacement. Compared to our counter-based approach, time-based approach is much more difficult to implement. First, since clock frequency is affected by many power/energy optimizations, it is hard to ensure the effectiveness of time-based predictors. Secondly, since applications greatly vary in their number of cache hits or misses per second, to be effective, the timing counters must be large and incremented frequently.

As an alternative to hardware techniques, compiler techniques have been proposed by Wang et al. [13], and Beyls and D'Hollander [2] to identify data that can be evicted early. However, compiler techniques are less applicable to applications that have dynamic access pattern behavior, or ones that are hard to analyze at compile-time.

Concurrent to our work, Takagi and Hiraki [12] recently proposed counter-based replacement called IGDR which collects access interval distribution where each per-block counter is incremented on each cache access, regardless of the sets. Because accesses to the entire cache occur frequently, IGDR requires very large counters. In addition, the access interval information captured is coarse-grain due to the interference between different sets. In contrast, because our AIP algorithm only collects the number of cache accesses per set, it does not suffer interference between different sets and only requires much smaller (4-bit) counters. In addition, whereas they do not use live time prediction, we provide such a prediction through our LvP algorithm.

3. Counter-Based Replacement Algorithms

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 replacement approach (Section 3.4) and the time-based approach (Section 3.5), and discusses some implementation issues (Section 3.6).

3.1. Overview

Figure 1 illustrates the life cycle of a cache line A. It begins when A is first accessed after it is brought into the cache, either by a demand fetch or by a prefetch. While in the cache, A is accessed several times before it is evicted. The time between the placement of A in the cache until it is evicted is called the *generation time*, which is further divided into the *live time* (LT) and the *dead time*. The *live time* of a line A is the time from the placement of A to the last access of A, whereas the *dead time* is the time from the last access of A to the time A is evicted from the cache. The time between two consecutive accesses to A is called the access interval (Δ).

There are two counter-based replacement algorithms that we propose. The first algorithm, *Access Interval Predictor* (AIP),

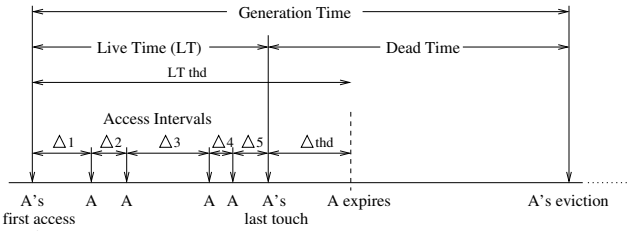

Figure 1: The life cycle of a cache line A, using terms defined in [15].

counts the number of accesses to the same set during an access interval of a line. The reason for counting only the accesses to the same set (as opposed to accesses to any set in the cache) is to avoid using large counters to count many more accesses, and to avoid behavior interference from other sets. If the event count since the last access to A exceeds an access interval threshold (Δ_{thd}), the line expires and becomes a candidate for replacement. The second approach, *Live Time Predictor* (LvP), bases its prediction on the live time of the line. For a line A, it counts the number of accesses to itself during a single generation. If the event count since A's placement in the cache reaches the live time threshold (LT_{thd}), the line expires and becomes a candidate for replacement.

The thresholds can be learned from the past behavior of the line. For example, Δ_{thd} is conservatively chosen as the maximum of all access intervals of the prior and current generations. Likewise, LT_{thd} is conservatively chosen as the maximum live times of the prior generation. This is a *unique feature of counter-based predictors: the behavior of a cache line can be compactly summarized in a single Δ_{thd} or LT_{thd}, resulting in compact prediction information.*

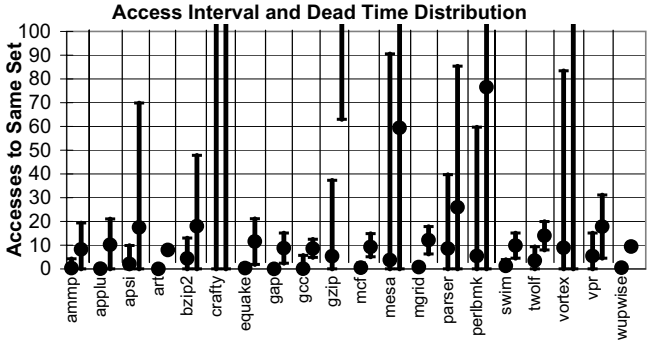

Figure 2: Access interval distribution. The y-axis represents the number of L2 cache accesses to the same set containing the block. For each application, the line on the left represents the access interval while the line to the right represents the dead time. The cache parameters follow those in Table 3.

For the AIP algorithm to work well, access intervals of a line must be considerably smaller than the dead time, so that expired lines are evicted soon enough to make room for other lines. Figure 2 shows that this is the case. The figure shows the average access interval $\pm$ its standard deviation and the average dead time $\pm$ its standard deviation for each of the applications studied. The unit on the y-axis is the number of L2 cache accesses to the same set containing the block. The figure shows that the access interval is typically more than one order of magnitude smaller than the dead time. Similar observations were made by other researchers when clock cycles, instead of event counts, were used [4]. Dead times tend to increase with larger and higher associativity caches because the time for a line to move from the MRU position to the LRU position increases.

For the LvP algorithm to work well, the live times have to be predictable. Fortunately, this is the case. To measure that, we profile how often the live time of a line is the same as the live

time in the previous generation. With the exception of six applications (bzip2, crafty, gzip, parser, twolf, and vpr), in more than 70% of the cases the live times are repeated. The complete data will be presented and discussed later in Table 1.

3.2. AIP Algorithm Implementation

3.2.1. Storage Organization and Overhead

Figure 3 shows the organization and storage overhead of the proposed AIP algorithm. First, each cache line is augmented with five extra fields. The first field, *hashedPC*, stores an 8-bit hashed value of the PC of the memory instruction that misses on the line. The *hashedPC* field is not used for counter bookkeeping, it is merely used to index a row in the prediction table, when a line is replaced from the cache. Since the prediction table has 256 rows, the *hashedPC* needs to be 8 bits, obtained by XOR-ing all 8-bit parts of the PC of the instruction that misses on the line. A similar 8-bit hashing of the block address is used to index a column in the prediction table. The second field is the *event counter* (C). The third and fourth fields are the counter thresholds, computed by taking the maximum of event counts in the prior 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 block can expire and be considered for replacement. If it is not set, the block cannot expire.

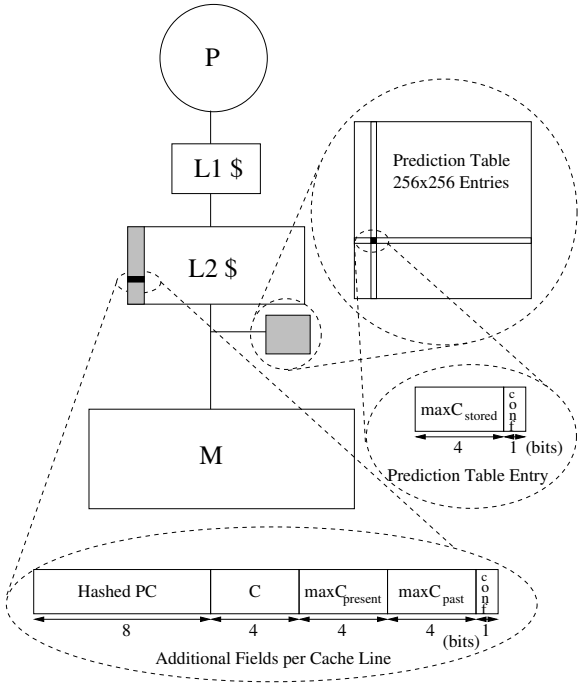

Figure 3: Storage organization and overhead of AIP.

The sizes of the counter and thresholds are determined based on profiling of likely counter values. Table 1 shows the average ("average" column) and the percentage of counter threshold values that are larger than 15 ("> 15" column) in the prediction table for both the AIP and LvP algorithms. In obtaining data in Table 1, we use profiling with infinitely-large prediction tables, event counters, and counter thresholds. The counter threshold values are simply recorded and profiled. They are not used for replacement decisions. The table shows that most counter threshold values are small. Fewer than 5% of counter threshold values are larger than 15, except for five applications. Therefore, using 4-bit saturating counters for both the event counter and counter thresholds is sufficient for most applications. The storage overhead per cache line is a reasonable $3 \times 4 + 8 + 1 = 21$ bits, equivalent to 4.1% overhead compared to a 64-byte line or 2.1% overhead compared to a 128-byte line.

	AIP				LvP		
app.	avg	avg cycles	>15	=last	avg	>15	=last
ammp	0.26	27,220	0.2%	99.9%	0.21	0.2%	99.9%
applu	0.11	7,410	0.0%	99.3%	0.24	0.0%	99.5%
apsi	2.1	131,206	2.9%	92.2%	2.48	0.9%	92.3%
art	0	1	0.0%	99.9%	0	0.0%	99.9%
bzip2	4.4	519,234	7.6%	48.8%	2.57	2.3%	48.3%
crafty	139	8,342,991	35.5%	28.8%	30.9	7.95%	27.7%
eon	N/A	N/A	N/A	N/A	N/A	N/A	N/A
equake	0.26	43,929	0.3%	89.1%	0.21	0.1%	89.1%
gap	0.02	5,130	0.0%	94.9%	0.04	0.0%	94.5%
gcc	0.09	9,350	0.1%	99.0%	0.49	0.2%	98.0%
gzip	5.32	702,624	4.0%	58.9%	1.44	1.7%	57.3%
mcf	0.51	60,919	0.3%	93.9%	0.26	0.4%	94.7%
mesa	3.7	354,970	0.9%	70.2%	8.81	0.6%	80.0%
mgrid	0.82	55,870	0.0%	97.0%	0.69	0.0%	97.0%
parser	8.62	1,173,352	14.9%	35.1%	4.5	4.03%	52.6%
perlbmk	5.37	839,526	2.6%	71.3%	1.12	1.0%	70.9%
swim	1.32	61,944	0.1%	87.5%	0.6	0.2%	86.2%
twolf	3.51	502,711	4.2%	45.8%	1.74	2.0%	45.9%
vortex	8.9	700,481	5.6%	72.1%	17.18	1.8%	73.7%
vpr	5.41	620,291	9.8%	36.8%	2.38	2.0%	40.9%
wupwise	0.46	185,225	0.0%	99.1%	0.21	0.0%	99.1%

Table 1: Profile of the counter threshold values in the counter prediction tables for AIP and LvP. The counter is represented as the number of accesses in the same set for AIP, and as the number of accesses to the *self* line for LvP. *average* is the average of all values, *avgCycles* is the average of all values in cycles, *>15* is the percentage of values larger than 15. *=last* is the percentage of new values equal to the prior values for the same cache line and PC. eon has no L2 cache evictions and thus no learned counter threshold values. The cache parameters follow those in Table 3.

The prediction table stores 4-bit counter threshold values. The prediction table is organized as a 256×256 two-dimensional direct-mapped tagless table structure, with its rows indexed by the *hashedPC*, and its columns indexed by an 8-bit hashed line address. The motivation behind using both the hashed line address and the *hashedPC* of the instruction that misses on the line is that the access behavior of a cache line is quite correlated with the code section where the line is accessed from. The total table size is $256 \times 256 \times (4 + 1)$ bits $= 40$ Kbytes, a reasonable 8% overhead compared to a 512 Kbyte L2 cache and 4% overhead compared to a 1 Mbyte L2 cache. Also, since the table is only accessed on a cache miss, and since its access can be overlapped with the cache miss latency, it is not latency sensitive. Furthermore, since the table is small, it can be located on chip.

3.2.2. Algorithm Details

The AIP algorithm implementation is shown in Figure 4. On a cache access to a block x in a set, the event counters of all lines in the set are incremented including x's counter (Step 1). If the access is a hit, the event counter value represents an access interval of block x that has just completed. The counter value is then compared to the maximum access interval in the current generation ($x.maxC_{present}$) and the maximum of them is used as the new value for $x.maxC_{present}$ (Step 2). Therefore, in a single generation, $x.maxC_{present}$ may keep on increasing until the maximum access interval is found. In addition, the event counter $x.C$ is reset so that it can measure the next access interval.

If the access to x is a miss, then a line y needs to be selected for eviction to make room for x. At this time, all lines in the set (except for the MRU line) are checked for expiration. A line b is determined to have expired if its counter $b.C$ is larger than both $b.maxC_{present}$ and $b.maxC_{past}$, and the confidence

On an access to block x in set s:

1. Increment the event counter of each block b in set s:
   $b.C = b.C + 1$;
2. If the access is a hit, reset x's counter after recording the new maximum:
   $x.maxC_{present} = \max(x.C, x.maxC_{present})$;
   $x.C = 0$;
3. If the access is a miss,

3a. Identify all blocks in set s that have expired. A block b has expired if $b.C > b.maxC_{present}$, $b.C > b.maxC_{past}$ and $b.conf == 1$.

3b. Find a replacement block y
   if there is at least one non-MRU expired block in the set
   y = the expired block **closest** to the LRU position;
   else y = the LRU block;

3c. Update the prediction table by y's information:
   Index the table using $y.hashedPC$ and y's block 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 block x in the cache:
   Index the prediction table using $x.hashedPC$ and x's block address. $x.hashedPC$ is obtained by performing 8-bit XOR to the instruction PC that causes the miss to x
   $x.C = x.maxC_{present} = 0$; $x.maxC_{past} = x.maxC_{stored}$; $x.conf = x.conf_{stored}$;

Figure 4: AIP algorithm implementation.

bit ($b.conf$) is '1' (Step 3a). The reason why $b.C$ is also compared with $b.maxC_{past}$ is because the current generation may not have completed, and hence $b.maxC_{present}$ may not represent the maximum access interval in the current generation yet. Finally, the confidence bit indicates how much confidence we have in the prediction, and therefore must be '1' for the block to expire. The $conf$ bit serves to prevent mispredictions when the behavior of a line is transitioning from one behavior phase to another. Although the $= last$ column in Table 1 shows that access interval values are highly predictable across generations, the $conf$ bit detects the unpredictable cases and prevents possible mispredictions.

After expired lines have been identified, a line y among them is chosen for eviction (Step 3b). If there are several expired lines, the line that is the closest to the LRU position is selected for replacement. It is also possible that no line has expired. This case may arise in several scenarios. First, it happens when no line has its event counter exceeding the maximum values. Secondly, it could also happen if one or more of the maximum counters ($b.maxC_{present}$ or $b.maxC_{past}$) is saturated. In this case, the event counter can never exceed the maximum counters. This case does not happen frequently because there are not many lines with such a large access interval. Table 1 shows that there are only fewer than 5% large access intervals that cannot be represented with 4 bit counters in most applications. Our experiments confirmed that we do not gain noticeable performance improvement when we use larger counters. Finally, if the confidence bit is '0', then the line cannot expire. This may be an initial condition where the line is still learning its access interval, or may be because the maximum access intervals across generations are not that predictable.

After the block-to-replace y has been determined, the counter prediction table is updated (Step 3c). If the current maximum counter $y.maxC_{present}$ is equal to the prior generation maximum counter ($y.maxC_{past}$), then the maximum access interval is likely to be predictable. In this case, the confidence bit in the table ($y.conf_{stored}$) is set. Otherwise, it is reset to 0. Since multiple addresses may map to the same entry in the prediction table, they may have different maximum access intervals. This will cause the confidence bit to be reset to '0', preventing the algorithm from applying counter-based replacement to the affected lines. However, we found that this effect is negligible in reality because by using both the $hashedPC$ and the block address to index the prediction table, the accesses to the prediction table are randomized. Finally, a newly fetched block x is placed in the cache (Step 3d). Its event counter ($x.C$) and present maximum counter ($x.maxC_{present}$) are initialized to zero, while its past maximum counter ($x.maxC_{past}$) and confidence bit ($x.conf$) are copied from the table.

Note that Steps 3a, 3b, 3c, and 3d can be performed in parallel with the cache miss latency. Therefore, their latencies can be completely hidden. In addition, the AIP algorithm only involves several 4-bit increment and compare operations. Therefore, they can be implemented in hardware easily.

MRU LRU

Tag	A	B	D	E	F	G	H	I
C	0	1	3	4	7	8	11	15
$maxC_{present}$	3	1	1	2	5	4	5	10
$maxC_{past}$	3	1	2	8	5	4	15	15
$conf$	1	1	1	0	1	0	0	1
Expired?	N	N	Y	N	Y	N	N	N

(a)

MRU LRU

Tag	D	A	B	E	F	G	H	I
C	0	1	2	5	8	9	12	15
$maxC_{present}$	4	3	1	2	5	4	5	10
$maxC_{past}$	2	3	1	8	5	4	15	15
$conf$	1	1	1	0	1	0	0	1
Expired ?	N	N	Y	N	Y	N	N	N

(b)

MRU LRU

Tag	J	D	A	B	E	G	H	I
C	0	1	2	3	6	10	13	15
$maxC_{present}$	0	4	3	1	2	4	5	10
$maxC_{past}$	4	2	3	1	8	4	15	15
$conf$	1	1	1	1	0	0	0	1
Expired ?	N	N	N	Y	N	N	N	N

(c)

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

To illustrate the working of the AIP algorithm, we show an example in Figure 5. The figure shows a set in an 8-way set associative cache. The blocks are sorted from the MRU block (left) to the LRU block (right). The figure shows the different values of C, $maxC_{present}$, $maxC_{past}$, and $conf$ for each block. Figure 5a shows the initial state with two expired blocks (D & F) because their confidence bits are set and their C values are larger than their maximum counters ($maxC_{present}$ and $maxC_{past}$). Note that block G has not expired because $G.conf$ is zero. Figure 5b shows the set after a cache access (hit) to block D. Block D now becomes the MRU block, the counter values for all the blocks in the set are incremented, except for ones that have already saturated the 4-bit counters (block I). Updating the event counter values results in B becoming a new expired block. Since an access interval has just been completed for D, $D.maxC_{present}$ is updated with the value of $D.C$ (i.e. 4), and then $D.C$ is reset to 0. Interestingly, block D is now no longer expired since it is reaccessed. This demonstrates that by conservatively evicting an expired line only when it is necessary to make room for a new cache miss, we give a chance for the line to learn new (larger) access intervals. Finally, let us assume an access to block J results in a cache miss (Figure 5c). In this case, because both blocks B and F have expired, we choose to replace F, which is closer to the LRU position than B. The values $J.maxC_{present}$ and $J.C$ are initialized to zero while the values $J.maxC_{past}$ and $J.conf$ are initialized from the prediction table.

3.3. LvP Algorithm Implementation

The LvP is implemented in a similar way as the AIP algorithm in Section 3.2. However, there are several differences.

3.3.1. Storage Organization and Overhead

The storage organization is the same as in AIP algorithm (Figure 3), with one exception. The LvP algorithm counts events during the entire live time of a line. Thus, $maxC_{present}$ is redundant and can be eliminated. Similar to AIP, Table 1 shows that LvP counters only need to be 4-bit wide, because usually there are only 5% or less live time event counts that have values larger than 15. Without the $maxC_{present}$ field, the storage overhead in each cache line is slightly reduced: 3.3% overhead in a cache with 64-byte line, or 1.7% overhead in a cache with 128-byte line.

3.3.2. Algorithm Details

Figure 6 shows the LvP algorithm steps. The LvP algorithm keeps track of the number of accesses to a line during its generation time. Therefore, on a cache access, it only increments its own counter (Step 1). If the access results in a cache miss, a line that has expired is chosen for replacement (Step 2a) by comparing the line's event counter C with its $maxC_{past}$ and checking the confidence bit ($conf$). After a block has been chosen for eviction (Step 2b), the prediction table is updated (Step 2c) and the new block is placed in the cache and its fields initialized (Step 2d).

On an access to block 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 blocks in set s that have expired. A block b has expired if $b.C >= b.maxC_{past}, b.conf == 1$.
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 block address
   $y.maxC_{stored} = y.C$;
   if $y.maxC_{past} == y.C$
   $conf_{stored} = 1$;
   else $conf_{stored} = 0$.
2d. Place block x in the cache:
   Index the prediction table using $x.hashedPC$ and x's block address. $x.hashedPC$ is obtained by performing 8-bit XOR to the instruction PC that causes the miss to x
   $x.C = 0$;
   $x.maxC_{past} = x.maxC_{stored}$;
   $x.conf = x.conf_{stored}$;

Figure 6: LvP algorithm implementation.

3.4. Comparison with Sequence-Based Replacement Algorithms

Counter-based replacement algorithms differ in several ways with sequence-based replacement algorithms. First, sequence-based replacement algorithms mark a cache line evictable right at the time of the last touch, whereas counter-based replacement algorithms wait until the line expires. Therefore, counter-based algorithms tend to be more conservative and have a slight delay in marking lines evictable (except for LvP). However, with access intervals approximately one order of magnitude smaller than the dead times in a highly associative L2 cache, the timing advantage of sequence-based replacement algorithms is small.

The storage overhead of the prediction table in sequence-based algorithms is much larger than counter-based algorithms. In [8, 10], the authors an on-chip prediction table that is several Mbytes in size. Our counter-based algorithms require only a 40 Kbytes prediction table. The reason for the space efficiency is that for each line, only one 4-bit threshold value needs to be maintained. In contrast, with sequence-based algorithms, if there are n sequence of events that lead to the last touch of a cache line, they must be recorded and learned by the predictor. Therefore, sequence-based predictors also require longer training compared to counter-based predictors. to that observed in the time-based

Finally, our counter-based predictors are more conservative because the counter threshold is taken as the maximum value of the access intervals or live times of a line, whereas sequence-based predictors may prematurely evict a line as soon as a predicted event sequence occurs.

The differences pointed above cause our counter-based algorithms to achieve a higher coverage and accuracy (76% vs. 70% coverage, and 4-5% vs. 12% premature evictions) when infinite prediction tables are used (Section 5.1). Our counter-based algorithms also achieve higher speedups (11% vs. 4%) when fixed-size prediction tables are used.

3.5. Comparison with the Time-Based Approach

Although the time-based approach has not been implemented for cache replacement before, it is potentially applicable. In contrast to event counting, a time-based approach would measure the access intervals in multiples of clock cycles. A time-based approach can be thought of as a counter-based approach where the counters count the number of clock cycles. There are several important differences however. First, the number of cache misses and evictions are more closely correlated with the number of cache accesses than with the number of clock cycles. Since time-based predictors update a cache line's timer at each timestep which is a multiple of processor cycles, the timer has to accommodate cache timing variations of a wide range of applications. For example, the timestep has to be small enough to accommodate applications with a high L2 cache access frequency, but the timer storage should be large enough to accommodate applications with a low L2 cache access frequency. For example, in the counter-based approach, an access interval is always detectable, however, in the time-based approach, if the time unit is too coarse, it cannot distinguish between a zero access interval and a small access interval. On the other hand, if the time unit is too fine, large timer storage is needed to accommodate large access intervals in some applications.

Finally, a time-based approach is harder to implement in real processors because the processor clock frequency is affected by many factors, such as many energy saving optimizations. For example, when fetch throttling is activated, the frequency of load instructions per cycle may decrease, affecting the duration of access intervals and live times in the L2 cache. However, a counter-based approach is not much affected because the number of L2 cache accesses per access interval is likely to remain the same. In general, to work well, a time-based approach will have to be made aware of many other system variables.

3.6. Other Implementation Issues

Counter book-keeping implementation. To keep the hardware for incrementing counters very simple, we use a narrow (4-bit) increment and a compare unit per set for both AIP and LvP. In AIP, since all counters in a set are incremented on each access to any block in the set, we perform the increments sequentially. We note that because the average time between two successive accesses to a cache set is larger than 2000 cycles, sequentially incrementing a set's counters results in negligible performance impact. Moreover, since counters need to be read and compared only when cache replacement needs to be made on a cache miss, they are performed in parallel with cache miss latency.

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.

4. Evaluation Environment

Applications. To evaluate the counter-based algorithms, we use 21 of the 26 Spec2000 applications. The four Fortran90 applications (facerec, fma3d, galgel, and lucas) along with sixtrack are not included due to limitations in our compiler infrastructure. The applications, and their L2 cache miss rates for both 512 KByte and 1 MByte L2 caches are summarized in Table 2. The applications were compiled with the -O3 flag. The REF input set was used for each application. Each application was fast forwarded for 2 Billion instructions and then run for another 3 Billion instructions. The applications are divided up into two groups. Group A consists of 10 applications whose number of misses are reduced by more than 5% when the L2 cache size is doubled from 512KB to 1MB. Group A indicates applications that have capacity misses that can be reduced when the effective cache size is increased. The remaining 11 applications are lumped into Group B. *For Group B, LTHF32, DBP, AIP and LvP do not speed up any application by more than 2% or slow down any by more than 1%. Consequently, most of the evaluations in Section 5 will focus on applications in Group A*.

Group A			Group B		
App	512KB L2 Miss Rate	1MB L2 Miss Rate	App	512KB L2 Miss Rate	1MB L2 Miss Rate
ammp	86.7%	80.4%	applu	76.9%	76.9%
apsi	35.1%	15.2%	crafty	0.9%	0.2%
art	100.0%	71.6%	eon	0.0%	0.0%
bzip2	28.4%	16.8%	equake	78.1%	77.2%
gcc	90.7%	0.7%	gap	89.4%	89.3%
mcf	79.5%	73.4%	gzip	2.6%	2.4%
mgrid	63.5%	42.4%	mesa	9.6%	9.4%
swim	62.6%	57.5%	parser	15.8%	13.9%
twolf	35.2%	12.3%	perlbmk	8.0%	6.8%
vpr	24.2%	4.4%	vortex	4.4%	3.5%
			wupwise	79.9%	79.9%

Table 2: The 21 applications used in our evaluation.

Simulation Environment. The evaluation is performed using a detailed execution-driven simulation environment that supports a dynamic superscalar processor model [6]. Table 3 shows the parameters used for each component of the architecture. The architecture is modeled cycle by cycle.

PROCESSOR
6-issue dynamic. 5 GHz. Int, fp, ld/st FUs: 8, 8, 2/2
Pending ld, st: 24, 24. Branch penalty: 12 cycles
ROB entries: 156, int/fp registers: 128/128
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

Table 3: Parameters of the simulated architecture. Latencies correspond to contention-free conditions. *RT* stands for round-trip *from the processor*.

Prediction Parameters. To study the performance of counter-based replacement algorithms, we compare them against two sequence-based replacement algorithms: the Last Touch History Predictor proposed by Lin and Reinhardt (LTHF32) [10], and the Dead Block Predictor (DBP) proposed by Lai and Falsafi [8]. When comparing the coverage and accuracy potentials of the predictors, we use unlimited prediction table sizes in the comparison in Section 5.1. However, when comparing the performance of each of the schemes in Sections 5.2 and 5.3, we use fixed size predictors. For LvP and AIP, we use a 40-KB prediction table size, while the history and signatures table sizes used in LTHF32 and DBP are fixed at 64KB each. (as described in Section 3.2) when we evalute the counter-based predictor performance in Sections 5.2 and 5.3. The AIP and LvP prediction table access latency is four cycles.

5. Evaluation

In this section, we present and discuss several sets of evaluation results. Section 5.1 shows the prediction accuracy and coverage of LvP and AIP compared to those of DBP and LTHF32. Section 5.2 presents the performance of the counter-based algorithms compared to sequence-based algorithms and a timer approach using a limited prediction table size. Finally Section 5.3 studies the effect of changing the cache parameters on the performance of the counter-based algorithms. Although most of the discussion will focus on applications in Group A, we will also discuss why no speedup or slowdown is obtained for applications in Group B (Section 5.2).

5.1. Coverage and Accuracy

Figure 7 shows the coverage and accuracy of sequence-based and counter-based algorithms assuming an infinite prediction table size, for applications in group A. Assuming infinite table size allows us to find out the upper bound performance of each algorithm. A trace of L2 cache accesses is first collected. The trace is then used in a trace simulator that implements an optimal replacement algorithm (OPT). OPT assumes perfect future knowledge and replaces the line in the set that will be accessed the farthest in the future [1]. Although in real machines OPT is not implementable due to the lack of future information, it gives an upperbound performance for cache replacement algorithms. In the first pass of the trace simulation, the alternative replacement algorithms are merely learning and filling their prediction tables. In the second pass, the algorithms are run again using the same trace. They are allowed to make predictions based on the prediction tables they learned earlier and mark lines that they would evict early, but without actually evicting them. We compare whether the lines that are marked for eviction are the same as the lines that would have been replaced using OPT.

Each bar in the figure is broken down into three components. The first component shows the fraction of predicted evictions that agree with OPT evictions (*Correct Prediction*), ones that disagree with OPT evictions (*Wrong Prediction*), and OPT evictions that are not predicted (*Not Predicted*). The bars are normalized to the original number of OPT evictions. Since the sum of *Correct Prediction* and *Not Predicted* is necessarily equal to the original number of evictions, it stands at 100% in the figure. *Wrong Prediction*s are the extra evictions due to mispredictions, which correlate with extra cache misses due to evictions that are too early. *Coverage* is equal to *Correct Prediction*, which measures the percentage of evictions that are predicted correctly. *Accuracy* is equal to the fraction of eviction predictions that are correct, which is equal to *Correct Prediction* divided by the sum of *Correct Prediction* and *Wrong Prediction*.

The figure shows that on average, the counter-based AIP and LvP correctly predict more evictions than the sequence-based LTHF32 and DBP (76% for both AIP and LvP vs. 62% for LTHF32 and 70% for DBP). LvP and AIP also have significantly less wrong eviction predictions (4% for AIP and 5% for LvP vs. 15% for LTHF32 and 12% for DBP).

Let us consider highly unpredictable applications: bzip2, twolf, and vpr. For these applications, all the algorithms in general have low correct prediction rates, representing the worst case scenario for all the algorithms. LvP and AIP are unable to predict many of the evictions because the counter thresh-

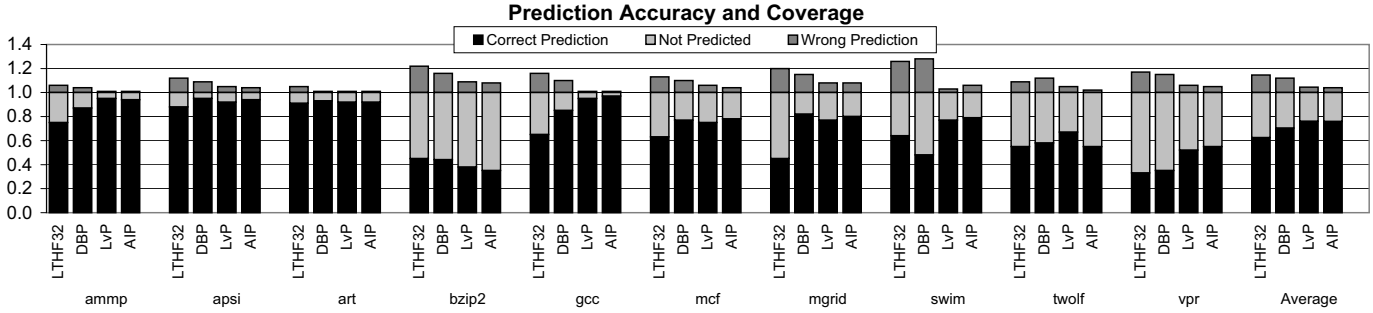

Figure 7: Coverage and accuracy of the various algorithms with infinite table sizes.

old values vary, resulting in the lines being evicted by OPT without being predicted by LvP and AIP. Large counter threshold values force LvP and AIP to predict evictions conservatively, resulting in very low wrong predictions (less than 9%) across all benchmarks. As discussed in Section 3.4, by not predicting the evictions, the performance of counter-based algorithms degrades gracefully to that of the LRU replacement algorithm. This result points out to the robustness of the performance of counter-based replacement algorithms. Sequence-based algorithms behave in the opposite manner. On these applications, sequence-based predictors continue to make many eviction predictions, and many of them are wrong because the applications are highly unpredictable. These wrong predictions result in an extra 9-22% evictions compared to the number of OPT evictions, despite using infinite table sizes and confidence mechanisms. As discussed in Section 3.4, mispredictions in the sequence-based approach cause lines to be evicted too early, potentially incurring many extra cache misses.

To summarize, even with infinite prediction table sizes, counter-based algorithms outperform sequence-based algorithms with slightly higher coverage but with much higher accuracy (i.e., few mispredictions). Note that the coverage and accuracy for sequence-based algorithms are not directly comparable to those in past studies [8, 7], because they are collected for the L2 cache, of which a lot of accesses have been filtered out by the L1 cache. Also note the coverage and accuracy will not be directly proportional to performance improvement because of many other factors (the base L2 miss rates and whether they are in group A or B).

5.2. Performance of Counter-Based Algorithms

Figure 8 shows the percentage of IPC improvement (speedup) of the different algorithms for applications in group A over the base case where the LRU replacement algorithm is used. For AIP and LvP, the prediction table structure is 40-KB, while extra fields per cache line occupy 21 bits as described in Section 3.2. Thus, the storage overhead of our algorithms (including the prediction table and extra cache fields) is 61 KB. Therefore we also compare the performance of our algorithms against a 512KB + 64KB = 576KB 9-way LRU cache. We also evaluate the performance of LTHF32 and DBP using 64KB history and signatures tables for each scheme. In addition, we evaluate the performance of a time-based approach by using a counter-based replacement algorithm (AIP) with a time counter (AIP.Time) instead of an event counter. To choose how frequently the time counters are incremented, we profile the access intervals of all applications. For each application, we then compute the average access interval (in number of clock cycles) minus the standard deviation of it. Then, we choose the minimum value across the applications as the time counter increment unit, to ensure that the unit is fine-grain enough to capture small access intervals. We found this unit to be 10,000 cycles.

Each group of bars represent the 576KB 9-way LRU cache (576KB), the last-touch history predictor (LTHF32), the dead-block predictor (DBP), the time-based access interval predictor (AIP.Time), the live time predictor (LvP), and the access interval predictor (AIP).

Figure 8 shows that on average, AIP and LvP perform the best, achieving an average speedup of 11% while not slowing down any of the applications, demonstrating their robust behavior. gcc shows the greatest performance gain for AIP and LvP (40% speedup). On the other hand, the sequence-based algorithms (LTHF32 and DBP) don't perform well and achieve modest speedups of 0.2% and 4%, respectively. This is due to the fact that with small fixed-size predictors, they are unable to store enough signatures to make many predictions. LTHF32 slows down four application (bzip2, mgrid, swim, and vpr) by up to 3%. DBP also slows down two applications (bzip2 and swim) by up to 5%. The slow-downs are a result of the sequence-based algorithms' low accuracy on those applications as illustrated in Section 5.1. AIP.Time also does not perform very well and achieves an average speedup of about 5% while not slowing down any application. This is lower than the speedups achieved by our counter-based algorithms but higher than the ones achieved by sequence-based algorithms. The performance improvement from increasing the cache size is insignificant (4%) justifying the extra storage overhead needed by our counter-based algorithms. Therefore, considering that our algorithms have low hardware complexity, the results show that they are promising techniques to implement.

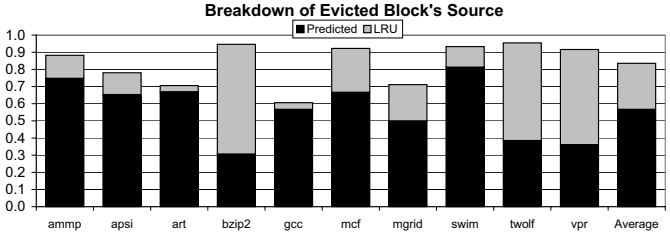

Figure 9: Fraction of lines evicted by AIP versus by LRU.

To investigate the performance better, Figure 9 shows the fraction of evictions 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 evicted using LRU replacement when there are no expired lines in the set to evict. All bars are normalized to the total number of evictions 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 evictions (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, 17% of the original cache misses are eliminated, with the biggest gain being for art, gcc, and mgrid. In addition, the fraction of evictions triggered by the AIP algorithms is high and ranges between 32% to 95%, with

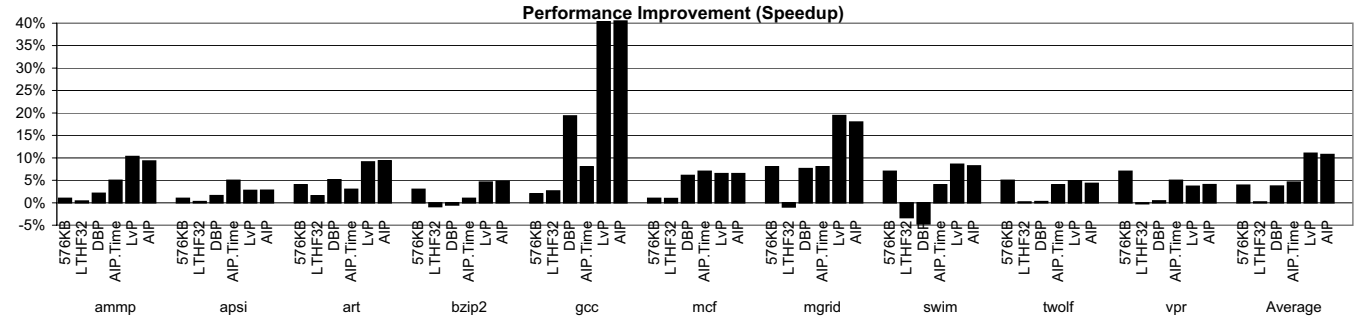

Figure 8: Performance improvement results of various replacement algorithms.

Configuartion	min	avg	max
256KB 8-way	0%	10%	28%
512KB 8-way	0%	11%	40%
1MB 8-way	0%	7%	25%
512KB 4-way	0%	10%	54%
512KB 16-way	0%	9%	27%

Table 4: Minimum, average, and maximum speedup of AIP for different L2 cache configurations for applications in Group A.

an average of 68%.

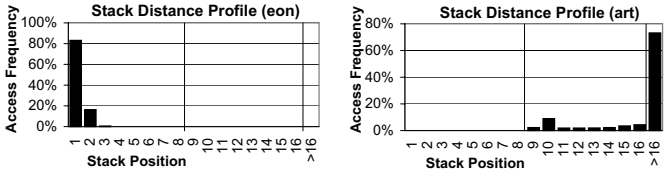

Figure 10: Stack distance profile of eon and art.

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 2% speedups or slowdowns with sequence-based and counter-based algorithms. We found that their stack distance profiles [11, 9] reveal the reason for the low speedups. Figure 10 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. Our counter-based algorithms can free up cache space occupied by lines that have expired, effectively increasing the cache space for other lines. However, Figure 10a shows that with eon, increasing the cache space by several associativities 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, Figure 10b shows that for art, if the cache size and associativity are increased, there is a significant number of cache misses that will turn into cache hits. Hence, our counter-based algorithms can 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% larger L2 cache, they can benefit from the counter-based replacement algorithms. This is why applications in group B do not show performance improvement: one type of applications has a small working set that fits in the L2 cache, while the other type of applications has a working set much larger than the L2 cache.

5.3. Sensitivity to Cache Parameters

Table 4 summarizes the minimum, average, and maximum speedup achieved when AIP is used with different L2 cache configuration using applications in Group A. The table demonstrates our counter-based algorithms' robustness over various cache configurations.

6. Conclusions

We have proposed two new **counter-based L2 cache replacement** algorithms: the Access Interval Predictor (AIP) and Live-time Predictor (LvP). AIP and LvP perform almost equally well, speeding up ten (out of 21) Spec2000 benchmarks by 11% on average. We have compared our approach with two existing approaches: sequence-based approach and time-based approach. Compared to the sequence-based approach, our counter-based algorithms achieve better prediction coverage and accuracy, lead to better performance improvement, and are more space efficient. Compared to the time-based approach, our counter-based algorithms perform better and are easier to implement because they are more architecture independent. We also found that the size of an application's working set relative to the L2 cache size has a strong impact on how much performance benefit it experiences.

References

[1] L. A. Belady. A study of replacement algorithms for virtual-storage computers. *IBM Systems Journal*, 1966.

[2] K. Beyls and E. D'Hollander. Compile-Time Cache Hint Generation for EPIC architectures. In *2nd Workshop on Explicitly Parallel Instruction Computing Architecture and Compilers*, 2002.

[3] G. Chen, N. Vijaykrishnan, M. Kandemir, M. Irwin, and M. Wolczko. Tracking Object Life Cycle for Leakage Energy Optimization. In *Proc. of the ISSS/CODES joint conference*, 2003.

[4] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *29th Intl. Symp. on Computer Architecture*, 2002.

[5] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *Intl. Symp. on Computer Architecture*, pages 240–251, 2001.

[6] V. Krishnan and J. Torrellas. A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 286–293, 1998.

[7] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *27th Intl. Symp. on Computer Architecture*, 2000.

[8] A.-C. Lai, C. Fide, and B. Falsafi. Dead block prediction and dead block correlating prefetchers. In *28th Intl. Symp. on Computer Architecture*, 2001.

[9] J. Lee, Y. Solihin, and J. Torrellas. Automatically Mapping Code on an Intelligent Memory Architecture. In *7th Intl. Symp. on High Performance Computer Architecture*, 2001.

[10] W.-F. Lin and S. K. Reinhardt. Predicting last-touch references under optimal replacement. *University of Michigan Tech. Rep. CSE-TR-447-02*, 2002.

[11] R. L. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2), 1970.

[12] M. Takagi and K. Hiraki. Inter-Reference Gap Distribution Replacement: an Improved Replacement Algorithm for Set-Associative Caches. *Proc. of the 18th Intl. Conf. on Supercomputing*, 2004.

[13] Z. Wang, K. McKinley, A. Rosenberg, and C. Weems. Using the Compiler to Improve Cache Replacement Decisions. In *Proc. of Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2002.

[14] W.A.Wong and J-L.Baer. Modified LRU Policies for Improving Second-Level Cache Behavior. In *HPCA*, 2000.

[15] D. A. Wood, M. D. Hill, and R. E. Kessler. A Model for Estimating Trace-Sample Miss Ratios. In *Proc. of the ACM SIGMETRICS*, 1991.

[16] H. Zhou, M. Toburen, E. Rotenberg, and T. Conte. Adaptive Mode Control: A Static-Power-Efficient Cache Design. *ACM Trans. on Embedded Computing Systems*, 2(3), 2002.