Appears in the Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture

Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency

Haiming Liu
Dept. of Computer Sciences
The University of Texas at Austin
hmliu@cs.utexas.edu

Michael Ferdman
Dept. of Elec. & Comp. Eng.
Carnegie Mellon University
mferdman@ece.cmu.edu

Jaehyuk Huh
Advanced Micro Devices
Jaehyuk.Huh@amd.com

Doug Burger
Microsoft Research
One Microsoft Way Redmond, WA
dburger@microsoft.com

Abstract

Data caches in general-purpose microprocessors often contain mostly dead blocks and are thus used inefficiently. To improve cache efficiency, dead blocks should be identified and evicted early. Prior schemes predict the death of a block immediately after it is accessed; however, these schemes yield lower prediction accuracy and coverage. Instead, we find that predicting the death of a block when it just moves out of the MRU position gives the best tradeoff between timeliness and prediction accuracy/coverage. Furthermore, the individual reference history of a block in the L1 cache can be irregular because of data/control dependence. This paper proposes a new class of dead-block predictors that predict dead blocks based on bursts of accesses to a cache block. A cache burst begins when a block becomes MRU and ends when it becomes non-MRU. Cache bursts are more predictable than individual references because they hide the irregularity of individual references. When used at the L1 cache, the best burst-based predictor can identify 96% of the dead blocks with a 96% accuracy. With the improved dead-block predictors, we evaluate three ways to increase cache efficiency by eliminating dead blocks early: replacement optimization, bypassing, and prefetching. The most effective approach, prefetching into dead blocks, increases the average L1 efficiency from 8% to 17% and the L2 efficiency from 17% to 27%. This increased cache efficiency translates into higher overall performance: prefetching into dead blocks outperforms the same prefetch scheme without dead-block prediction by 12% at the L1 and by 13% at the L2.

1. Introduction

Prior studies have shown that data caches have low efficiency [2], [26]; only a small fraction of cache lines actually hold data that will be referenced before eviction. Traditionally, a cache line that will be referenced again before eviction is called a *live block*; otherwise it is called a *dead block*. Cache efficiency can be improved if more live blocks are stored in the cache without increasing its capacity. Improved cache efficiency reduces cache-miss rate and improves system performance.

The root cause of low cache efficiencies is that blocks die, reside in the cache for a long period of time with no accesses, and then are finally evicted. With LRU replacement, upon the last access to a block, multiple replacements to that set must occur before the dead block is evicted [7], [19], which can take thousands of cycles. This last access may occur after only several accesses, or worse, immediately upon the first time the block is loaded into the cache.

To achieve better efficiency, dead blocks should be identified early. The earlier a dead block is identified, the more opportunity there is to improve cache efficiency. A block turns dead on its last access before its eviction from the cache. The identification of a dead block should be done between the last access to the block and its eviction from the cache. Since the hardware does not know with certainty which access to a block is the last access, the identification of a block as dead is a speculative action called *dead-block prediction*.

Three approaches for dead-block prediction have been proposed: trace-based, counting-based, and time-based. Lai et al. were the first to propose the concept of dead-block prediction and a trace-based predictor [16], which predicts a block dead once it has been accessed by a certain sequence of instructions. They use the predictor to trigger prefetches into the L1 data cache. Hu et al. later proposed a time-based predictor [7], also to trigger prefetches into the L1 data cache. Time-based predictors predict a block dead once it has not been accessed for a certain number of cycles. Kharbutli et al. proposed a counting-based predictor [14], which predicts a block dead once it has been accessed a certain number of times. They use the predictor to optimize cache replacement policy and to bypass zero-reuse blocks.

Most prior dead-block predictors predict the death of a block immediately after the block is accessed, as shown in Figure 1(a), which shows a sequence of accesses to three blocks, A, B, and C, in the same set of a two-way associative cache. P(A) in the figure indicates a prediction about whether

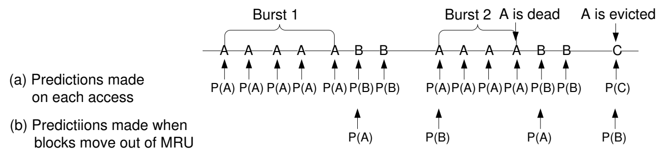

Burst 1
Burst 2
A is dead
A is evicted
A A A A A B B A A A A B B C
(a) Predictions made on each access
P(A) P(A) P(A) P(A) P(A) P(B) P(B) P(A) P(A) P(A) P(A) P(B) P(B) P(C)
(b) Predictiions made when blocks move out of MRU
P(A) P(B) P(A) P(B)

Figure 1. Predicting dead blocks at different times

block A has died. While this approach identifies dead blocks as early as possible, it sacrifices prediction accuracy and coverage because a block just accessed may be accessed again soon. There is a tradeoff between the timeliness and accuracy/coverage of dead-block prediction. The earlier the prediction is made, the more useful it is. On the other hand, the later the prediction is made, the less likely it is to mispredict. In this paper, we quantify this tradeoff by making dead-block predictions at different points during the dead time of a block. Making dead-block predictions when a block just becomes non-MRU, as shown in Figure 1(b), gives the best tradeoff between timeliness and prediction accuracy/coverage.

Prior dead-block predictors also update the history of a block *every time* the block is referenced. A prediction about whether a block has died is made based on the individual reference history of each block. However, how a block is accessed in the L1 cache may depend on the control-flow path the program takes, the value or offset of the referenced data in the block, and other parameters. These variations can cause the individual reference history of a block to be irregular and cause problems for existing dead-block predictors. To address this problem, we propose a new class of dead-block predictors for the L1 cache that predict dead blocks using the *cache burst* history of each block. A cache burst begins when a block moves into the MRU position and ends when it moves out of the MRU position. In these new dead-block prediction schemes, the contiguous references a block receives in the MRU position are grouped into one cache burst. In Figure 1, block A receives two cache bursts. A prediction about whether a block has died is made only when it becomes non-MRU, using the block's cache burst history. Because cache burst history hides the irregularity in individual references, it is easier to predict than individual reference history for L1 caches. The downside of this approach is that dead-block predictions are made later than the point at which blocks actually die (the last reference). Cache bursts can be used with trace-based, counting-based, or time-based predictors. This paper evaluates cache bursts as a strategy for improving counting-based and trace-based predictors.

Cache bursts only work well at the L1 cache. For the L2, counting-based predictors work best. We improve upon a previously proposed counting-based prediction scheme by addressing its limitations caused by reference count variation.

Compared to prior schemes, the new predictors show significant improvement in prediction accuracy and coverage while only lose approximately $1/n^{th}$ of the dead time, where n is the associativity of the cache. When used in a two-way L1 cache, a trace-based cache burst predictor can correctly identify 96% of the dead blocks with a 96% accuracy and a counting-based cache burst predictor can correctly identify 86% of the dead blocks, also with a 96% accuracy. For a 16-way L2 cache, the improved counting-based predictor can identify 67% of the dead blocks with a 89% accuracy.

These improved dead-block predictors are used in several ways to improve cache efficiency. Like prior work, they are used for replacement optimization and for bypassing zero-reuse blocks at the L2 cache and for prefetching into dead blocks at the L1 cache. They are also used for prefetching into dead blocks at the L2 cache, which has not been studied by prior work. The results show bypassing and replacement optimization offer mostly overlapped benefit, with both techniques achieving similar results of approximately 5% performance improvement on the same set of applications. In contrast, prefetching into dead blocks increases the L1 efficiency from 8% to 17% and the L2 efficiency from 17% to 27%. The improved cache efficiency translates into higher overall performance: prefetching into dead blocks outperform the same prefetch scheme without dead block prediction by 12% at the L1 and by 13% at the L2.

2. Prior Work on Dead Block Prediction

Dead block prediction can be performed in software [23], [28] or in hardware [1], [7], [14], [16]. Software solutions pass hints about dead-block information collected through profiling or compiler analysis [23], [28] to the hardware. They are more accurate but usually have lower coverage. Hardware solutions can be classified into two categories: data-address based [7] and PC based [1], [14], [16]. Compared to data-address based approaches, PC-based approaches require much lower storage overhead. Based on the state the predictor maintains to make predictions, hardware solutions can be classified into three categories: *trace-based*, *counting-based*, and *time-based*.

Lai et al. were the first to propose the concept of dead-block prediction [16] and a trace-based dead-block predictor for the L1 cache, called DBP. Because we use DBP in this paper to refer to dead-block prediction in general, to avoid confusion, we use the name Reference Trace Predictor (RefTrace) to denote this predictor.[1] RefTrace records the sequence of instructions that have referenced a block by hashing the PCs of these instructions together. A history table is used to learn which trace values (sequences of references) result in dead blocks by observing the trace value of each evicted block. Blocks brought into the cache by the same instruction

1. RefTrace was evaluated on directly mapped caches in [16]. This paper uses it on set-associative caches. In contrast to this study which evaluates RefTrace in the MRU position, [4] evaluated RefTrace in the LRU position.

but referenced along different paths will have different trace values upon eviction. The different sequences of references conceptually form a tree embedded in the history table, with the root of the tree being the instruction that caused the miss and each leaf indicating dead blocks. Each entry in the history table indicates the likelihood that the corresponding trace value will result in a dead block. Aliasing can occur if one sequence, which results in dead blocks in some cases, is a prefix of other longer sequences.

Kharbutli and Solihin later proposed a counting-based dead-block predictor called Live Time Predictor [14], for L2 caches. In this paper, we use the name RefCount to denote that it is a counting-based predictor. In RefCount, each block in the cache is augmented with a counter that records both how many times the block has been referenced and the PC of the instruction that first missed on the block. When the counter reaches a threshold value, the block is predicted dead. The threshold is dynamically learned using a history table by observing the reference count and recorded PC of each evicted block. Compared to RefTrace, RefCount uses only the PC of the instruction that brought a block into the cache to make predictions, and can not distinguish blocks that are brought into the cache by the same instruction but are referenced by different instruction sequences.

Hu et al. proposed a time-based dead-block predictor, Timekeeping (TK) [7], for the L1 cache. TK dynamically learns the number of cycles a block stays alive and if the block is not accessed in more than twice this number of cycles, it is predicted dead. Abella et al. proposed [1] another time-based predictor to turn off dead blocks dynamically in the L2 cache. They observed that both the inter-access time between hits to the same block and the dead time correlate with the reference counts of a block. They also predict a block dead if it has not been accessed in a certain number of cycles, but the cycle count is derived from how many times the block has been accessed. Compared to time-based predictors, trace-based and counting-based predictors are easier to implement in hardware and incur less overhead. Also, the traces and reference counts of blocks are more closely correlated to the memory-reference behavior of a program than the cycle count between accesses to the same block.

These predictors are used in various cache optimizations, including prefetching, replacement, bypassing, power reduction, and coherence protocol optimizations.

Prefetching: Lai et al. [16] and Hu et al. [7] used dead-block prediction to trigger prefetches into dead blocks in the L1 data cache. They found triggering prefetches on dead-block predictions improves the timeliness of prefetching compared to triggering prefetches on cache misses. Ferdman and Falsafi later extended the work in [16] to store correlation patterns off-chip and stream them on-chip as needed [4], which makes it possible to perform correlation-prefetching with large correlation tables.

Replacement: Kharbutli and Solihin [14] used dead-block prediction to improve the LRU algorithm by replacing dead blocks first, and also for bypassing the cache. Other approaches optimize LRU replacement without dead-block prediction: Wong and Baer modified the LRU algorithm by replacing blocks with no temporal locality first [29], Kampe et al. proposed an Self-Correcting LRU algorithm [12] to correct LRU replacement mistakes, whereas Qureshi et al. proposed to adaptively place missing blocks into the LRU instead of the MRU position when the working set is larger than the capacity of the cache [20].

Bypassing: Prior work has also used bypassing [6], [10], [11], [22], [27] to improve cache efficiency. Tyson et al. proposed bypassing based on the hit rate of the missing load/store instruction [27]. Johnson et al. proposed bypassing based on the reference frequency of the data being referenced [11] but put bypassed blocks in a separate buffer parallel to the cache. Jalminger and Stenström proposed bypassing based on the reuse distance of the missing block [10]. González et al. proposed to bypass L1 data cache blocks with low temporal locality [6].

Power reduction: Dead block prediction has also been used to reduce leakage by turning off dead blocks. Kaxiras et al. used dead-block prediction to turn off blocks in the L1 D-cache [13]. Abella et al. proposed to turn off blocks in the L2 cache dynamically [1]. Both schemes predict how many cycles have to pass before a block can be turned off without affecting performance. Dead block prediction can also be used in drowsy caches [5], to decide which blocks should switch to the drowsy state.

Coherence protocol optimization: Cache coherence protocols can also benefit from dead-block prediction. Lebeck and Wood proposed dynamic self-invalidation [17] to reduce the overhead of the cache coherence protocol by invalidating some of the shared cache blocks early. Lai and Falsafi later proposed a last-touch predictor [15] that uses PC traces to predict when shared cache blocks should be invalidated. Somogyi et al. studied using PC-traces to identify last stores to cache blocks [25].

3. Cache Efficiency

The concept of cache efficiency was first proposed by Burger et al. in [2], where cache efficiency is defined as the average fraction of the cache blocks that store live data. For any cycle during the execution of a program, some fraction of the blocks in the cache are live. The arithmetic mean of these live block fractions across the execution time of a program, or cache efficiency, can be computed as:

$$E = \frac{\sum_{i=0}^{A \times S-1} U_i}{N \times A \times S} \quad (1)$$

In Equation 1, A is the associativity of the cache, S is the number of sets, N is the execution time in cycles, and U_i is the total number of cycles for which cache block i is live.

Application	DL1 efficiency		L2 efficiency	
	Baseline	Optimized	Baseline	Optimized
swim	0.02	0.36	0.06	0.10
mgrid	0.08	0.24	0.18	0.23
applu	0.07	0.23	0.03	0.07
gcc	0.05	0.10	0.34	0.55
art	0.01	0.10	0.12	0.69
mcf	0.04	0.07	0.05	0.14
ammp	0.08	0.14	0.05	0.08
lucas	0.01	0.06	0.01	0.04
parser	0.33	0.33	0.32	0.33
perlbmk	0.40	0.40	0.17	0.21
gap	0.07	0.12	0.07	0.09
sphinx	0.09	0.10	0.34	0.52
corner_turn	0.02	0.12	0.04	0.05
stream	0.01	0.21	0.03	0.16
vpenta	0.01	0.01	0.80	0.80
GeoMean	0.08	0.17	0.17	0.27

Table 1. Cache efficiency of a 64KB, 2-way DL1 and a 1MB, 16-way L2 measured using sim-alpha

Cache efficiency measures the portion of the cache that actually holds useful data; the remaining portion holds useless data and can be vacated to store useful data. Table 1 shows the cache efficiency of a set of SPEC2000 benchmarks and several other benchmarks measured using *sim-alpha* [3], which models an Alpha 21264 processor. The geometric mean of the baseline cache efficiency for the L1 data cache and L2 cache, shown in the columns labeled "Baseline" of Table 1, is only 0.08 and 0.17 respectively, indicating the poor utilization of the caches and significant opportunities for improvement.

The reason cache efficiency is low is that the time a block stays alive in the cache is usually much shorter than the time that it is dead. The interval between the last access to a block and its eviction from the cache is called the dead time of the block. Likewise, the interval between the first access to a block, i.e., the access which brings the block into the cache, and the last access before its eviction, is called the live time. Prior work has shown that the dead time is usually at least one order of magnitude longer than the live time [7].

To improve cache efficiency, a cache must identify dead blocks early and replace them with useful blocks. By identifying dead blocks early with the best dead-block predictors we evaluated for the L1 and L2 caches and replacing them with new blocks through prefetching (discussed later in this paper), the L1 efficiency more than doubles and the L2 efficiency improves by 60%, as shown in the columns labeled "Optimized" in Table 1.

4. Identifying Dead Blocks Based on Cache Bursts

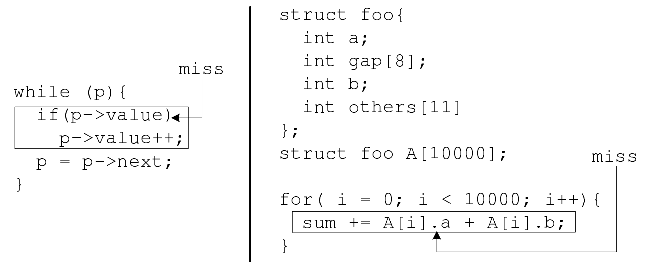
Accesses to the L1 and L2 caches have different characteristics. For example, accesses to the L2 cache are filtered by the L1 so L2 accesses have little spatial locality within a block while L1 accesses can have high spatial locality. These differences should be considered when designing dead-block predictors for each cache level. In this section, we propose new dead-block predictors for the L1 and L2 respectively.

4.1. Cache Burst Predictors: Tolerating Irregularity of Individual References in the L1 Cache

All prior dead-block predictors try to find regular patterns in the individual reference history of each block. However, individual reference histories can be volatile and irregular because how a block is accessed may depend on the control flow path the program takes, the value or offset of the referenced data in the block, and other parameters, all of which can change dynamically and may not show any regular patterns (RefTrace can handle control flow dependence to some extent). This is especially true for the L1 cache because the irregularity can be filtered out by the L1 cache and may not be observed by the L2 cache. Figure 2 shows two examples of reference variance.

Figure 2(a) shows how control-flow irregularity can lead to irregular reference history. Suppose the first access to $p\rightarrow value$ always misses and $p\rightarrow value$ will not be referenced after the iteration. Depending on whether $p\rightarrow value$ is zero, the block which has $p\rightarrow value$ can be accessed either once or twice. However, it is not possible to find a regular pattern in the individual reference history of each block because some of the blocks are referenced only by the load instruction while others are referenced by both the load and the store.

Figure 2(b) shows how data alignment variation can cause the same problem. Suppose the cache block size is 64 bytes and the access to *A[i].a* always misses. Because of data alignment differences, *A[i].a* and *A[i].b* can be located in the same block or in two adjacent blocks. If they are located in the same block, the block will be accessed twice before



```
while (p){
  if(p->value)
    p->value++;
  p = p->next;
}
```

(a) Control flow irregularity

```
struct foo{
  int a;
  int gap[8];
  int b;
  int others[11]
};
struct foo A[10000];

for( i = 0; i < 10000; i++){
  sum += A[i].a + A[i].b;
}
```

(b) Data alignment variation

Figure 2. Examples of irregular accesses to L1 blocks

eviction. Otherwise, the block that has *A[i].a* will only be accessed once. Again, it is not possible to find a regular pattern in the individual reference history of each block that has *A[i].a* because some blocks will be accessed only by one load instruction and others will be accessed by both loads.

This irregularity in individual reference history can cause problems for existing dead-block predictors: neither RefCount nor RefTrace can handle the two examples in Figure 2 well because neither can predict exactly after which access a block becomes dead.

The problem with trying to find regular patterns in the individual reference history of each block is that the predictor observes events at excessively fine granularity. Because L1 cache accesses tend to be bursty in the sense that several accesses to the same block are usually clustered in a short interval, an effective strategy is to predict dead blocks by cache bursts instead of individual references. We formally define cache bursts as follows:

Definition A *cache burst* is the contiguous group of cache accesses a block receives while it is in the MRU position of its cache set with no intervening references to any other block in the same set.

Although the references within a cache burst may be irregular, the cache-burst history can still be regular. Examining the two examples using bursts, there still is a regular pattern. In Figure 2(a), the block containing $p{\rightarrow}value$ will become dead after exactly one cache burst, regardless of whether $p{\rightarrow}value$ is zero. In Figure 2(b), the block containing *A[i].a* will also become dead after exactly one cache burst, regardless of whether *A[i].b* is located in the same block.

Based on this observation, we propose a new class of dead-block predictors that predict based on cache bursts, not references, of each block. Cache bursts begin when a block moves into the MRU position and end when it moves out of the MRU position, at which point a dead block prediction is made, typically $1/n^{th}$ into the dead time, where n is the set associativity.

A Burst Counting Predictor (BurstCount) uses the same structure as a reference counting predictor except that it counts cache bursts instead of individual references. When a block is filled into the MRU position of its set, its burst count is set to 0. The burst count is incremented only when the block moves from a non-MRU position into the MRU position. If the block is accessed in the MRU position, the burst count does not change. A prediction is made only when a block becomes non-MRU.

Similarly, a Burst Trace Predictor (BurstTrace) uses the same structure as a reference trace predictor. The difference is that BurstTrace predicts dead blocks based on the sequence of cache bursts a block has received. In BurstTrace, the trace of a block is updated only when the block moves into the MRU position. If it is accessed in the MRU position, the trace does not change. A prediction is made only when the block becomes non-MRU.

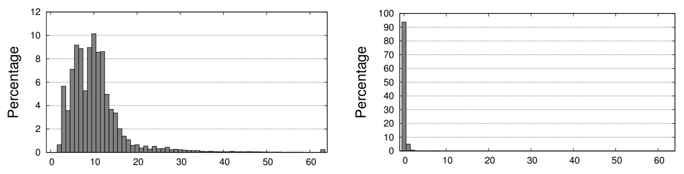


Figure 3. (a) Reference count distribution; (b) Burst count distribution

Figure 3 shows that burst history is more regular than reference history in the L1 cache. Figure 3(a) shows the reference count distribution of the blocks brought into the L1 D-cache by the same instruction in *sphinx*. This particular instruction causes the most misses in the L1 D-cache. The X axis is the reference count. The Y axis shows for a given reference count, what percentage of the blocks (out of all the blocks brought into the cache by this instruction) die after that number of references. Figure 3(b) shows the corresponding burst count distribution for the same instruction. The figures indicate burst count is much more predictable than reference count in the L1 D-cache.

Besides the higher dead-block prediction accuracy and coverage, burst-based predictors also have much lower power overhead than reference-based predictors. A reference-based predictor needs to read the history table and update the state of the accessed block on every cache access. In contrast, a burst-based predictor only reads the history table when a block becomes non-MRU and updates the state when it becomes MRU.

		By references	By bursts
Prediction metric	Trace	RefTrace [16]	BurstTrace
	Counting	RefCount [14]	BurstCount
	Time	TimeKeeping [7], IATAC [1]	Future work

Table 2. A taxonomy of dead-block prediction schemes

The introduction of cache bursts adds a new dimension to the design space of dead-block predictors. Based on the metric used to make dead-block predictions, dead-block predictors can be classified into trace-based, counting-based, and time-based. Based on how the state of a block is updated, dead-block predictors can be classified into reference-based and burst-based. Table 2 classifies the possible dead-block predictors using this taxonomy.

4.2. Tuning Reference Counting for the L2 Cache

While burst-based predictors work well for the L1 cache, they do not benefit the L2 cache because most of the irregularity in individual references has already been filtered out by the L1. Prior work [14] found counting-based predictors are better suited for the L2 than trace-based predictors because the filtering effect of the L1 prevents trace-based predictors from seeing the complete reference history of a block. One problem with counting-based dead-block predictors is reference count variation: blocks brought into the cache by the same instruction can receive different number of references in the cache.

To handle reference count variation, RefCount uses a confidence bit in each entry of the history table: when a block is evicted from the cache, its reference count is compared with the threshold stored in the history table. The confidence bit is set if the new reference count equals the old threshold and cleared otherwise. The threshold in the history table will not be used for prediction if the confidence bit is cleared.

One problem with this mechanism is that it can clear the confidence bit unnecessarily: the confidence bit will be cleared whenever a smaller reference count follows a larger reference count. In an extreme case where the reference count alternates between two different values, the confidence bit will never be set. A better way to handle such cases is to continue to use the larger reference count as the threshold without clearing the confidence bit, if the smaller reference count is only temporary. This can be achieved by an additional counter, *filter_cnt*, and a saturating counter, *sat_cnt*. A smaller reference count is first stored in *filter_cnt* and changes the threshold only when *sat_cnt* saturates.

Another issue is how to keep the history information current. As shown in Figure 4(a), in RefCount, each block copies the threshold and confidence bit from the history table when the block is filled into the cache and uses the copied information to make predictions thereafter. However, the threshold and confidence bit stored in each block can become outdated as the history table gets updated. A better approach is to remove the threshold and confidence bit stored in each block. Instead, when the predictor predicts, it uses the threshold and confidence bit from the history table, which has the most up-to-date information. This optimization reduces the area overhead but increases the frequency the history table is accessed. The increased accesses to the history table adds little energy overhead because the L2 cache is accessed only when L1 caches miss. Additionally, when used at the L1

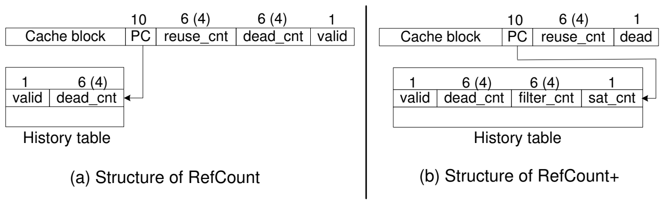


Figure 4. Differences between RefCount and RefCount+. RefCount+ stores threshold and confidence bit in history table to keep them up to date; also uses *filter_cnt* and *sat_cnt* to filter out noise

cache, the frequency of history table lookups are mitigated in the burst scheme because predictions are made only when a block becomes non-MRU.

With the inclusion of these two changes, we call the resulting predictor RefCount+, as shown in Figure 4(b).

4.3. Timeliness vs. Accuracy/Coverage: When to Predict

One question not answered by prior work is the best time to make dead-block predictions. The dead time of a block begins with the last access to the block and ends with its eviction from the cache. Dead block prediction can be made at any point in this interval. Almost all prior dead-block prediction schemes predict whether a block has died immediately after it is referenced, when the block is still in the MRU position. Higher prediction accuracy and coverage can be achieved if dead-block predictions are made later because it is less likely to make premature predictions. At the same time, predictions made closer to the end of a block's dead time are less useful because they leave the majority of the dead time exposed.

Figure 5 shows the accuracy and coverage of the RefCount+ predictor when dead-block predictions are made at different depths of the LRU stack after a block's last access. The results are obtained using *sim-alpha* with a 4-way, 64KB L1 cache. Other parameters of the simulation are listed in Table 4. The X axis shows the average number of cycles between the last access to a block and its movement into each position of the LRU stack. The last number on the X axis is the average number of cycles between the last access to a block and its eviction from the cache, i.e., the dead time. As expected, accuracy increases as predictions are made later. Coverage also increases because delaying the prediction does not miss any opportunity to identify dead blocks and the increase in accuracy causes more dead blocks to be correctly identified. The "knee" of the curves is located at way one of the LRU stack, indicating that predicting when a block just becomes non-MRU gives the best tradeoff between timeliness and accuracy/coverage. The same study of a 16-way L2 cache shows a similar trend except that the difference in prediction accuracy and coverage between way one and the LRU position is larger because of the higher associativity.

4.4. Evaluation: Dead Block Prediction Accuracy and Coverage

We compare the prediction accuracy and coverage of various dead block predictors. Coverage is measured as the number of blocks evicted from the cache that are correctly predicted dead divided by the total number of cache evictions. Accuracy is measured as the number of correct dead-block predictions divided by the total number of dead-block predictions ever made by each predictor. The evaluation uses both

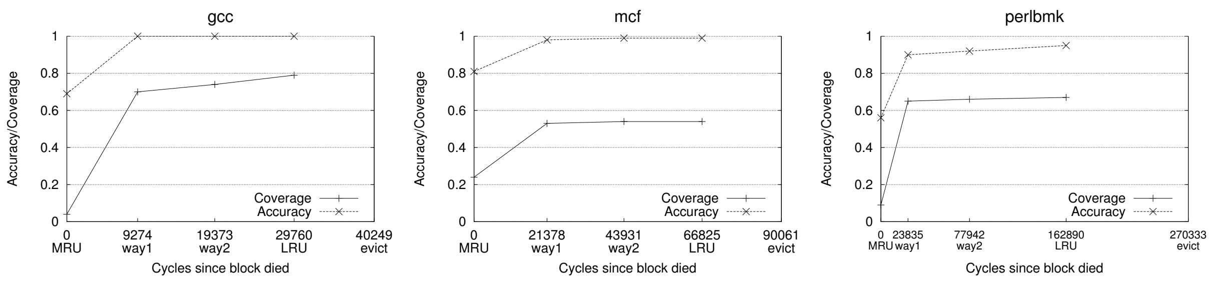


Figure 5. Prediction accuracy/coverage when predictions are made at different depths of the LRU stack for a 4-way L1 cache

Overhead	L1					L2		
	RefTrace	BurstTrace	RefCount	RefCount+	BurstCount	RefTrace	RefCount	RefCount+
History table entries	1K	1K	2K	1K	1K	64K	2K	2K
History table (bits)	2K	2K	14K	14K	14K	128K	10K	20K
Per-block (bits)	10	10	21	17	17	16	17	13
Total overhead(bytes)	1.5K	1.5K	4.4K	3.9K	3.9K	48K	35K	29K

Table 3. Overhead of different dead-block predictors

Issue width	6-way out of order(4 integer, 2 floating point)
Inst. window	80-entry reorder buffer, 32-entry Load/Store queue each
L1 I-cache	64KB, 2-way LRU, 64B cacheline, 1-cycle w/ set prediction
L1 D-cache	64KB, 2-way LRU, 64B cacheline, 3-cycle
L2 cache	1MB, 16-way LRU, 64B cacheline, 12-cycle
Main memory	200-cycle, 16B bus width

Table 4. Configuration of simulated SP machine

single-threaded benchmarks running on a single processor and multi-threaded benchmarks running on a CMP.

We first compare the overhead of each predictor, listed in Table 3. The overhead of each predictor includes the history table and the extra bits added to each block. The size of RefCount is scaled down from [14] to make it comparable with other predictors. It uses a 2K-entry history table; the index into the table is a hash with 8 bits from the PC and 3 bits from the block address. When calculating the predictor overhead, we assume a 64KB L1 D-cache and a 1MB L2 cache, both with 64-byte blocks.

4.4.1. Single-threaded workloads. The results for single-threaded workloads are collected using *sim-alpha* [3]. Table 4 shows the configuration of the simulated machine.

Besides the 11 benchmarks from SPEC 2000, we also use two benchmarks from Versabench [21] (*corner_turn* and *vpenta*), a speech recognition application (*sphinx*), and *stream* [18]. For each benchmark, we simulate up to 2 billion instructions identified by SimPoint [24].

Table 5 lists the prediction coverage and accuracy of each predictor used at the L1 D-cache. We can draw several conclusions from Table 5. First, both burst-based predictors (BurstTrace, BurstCount) significantly outperform the corresponding reference-based predictors (RefTrace, RefCount+): BurstTrace makes 50% more correct predictions than RefTrace with higher accuracy, and BurstCount makes 25% more correct predictions than RefCount+ with the same accuracy. The improvement in dead-block prediction comes with much reduced power consumption and no increase in area. Second, the optimizations to RefCount lead to better prediction coverage with higher accuracy: RefCount+ makes 13% more correct predictions than RefCount, with higher accuracy (96% vs. 91%). Third, of the five predictors listed in Table 5, BurstTrace incurs the smallest overhead but has the best coverage and accuracy, making it the best predictor for the L1 D-cache.

Table 6 shows the coverage and accuracy of the L2 dead-block predictors. We only compare reference-based predictors because burst-based predictors do not work as well at the L2 cache. Here, the two counting-based predictors (RefCount, RefCount+) both outperform the trace-based predictor (RefTrace), corroborating the findings in [14]. Of the two counting-based predictors (RefCount, RefCount+), RefCount+ has significantly higher accuracy (89% vs. 64%) and also higher coverage because of its ability to handle reference count variation better, as discussed in subsection 4.2. Additionally, RefCount+ also incurs the smallest overhead, making it the best choice for the L2 cache.

4.4.2. Multi-threaded workloads. We also present results for a set of server and parallel workloads collected using MP-sauce [9]. MP-sauce is an execution-driven, full-system simulator derived from IBM's SimOS-PPC. The timing model is based on sim-outorder in SimpleScalar with additional changes to model CMPs. The main parameters of the simulated machine are listed in Table 7.

We evaluate three commercial applications (SPECWeb99, TPC-W, and SPECjbb) and five scientific applications from SPLASH-2 [30].

Application	RefTrace		BurstTrace		RefCount		RefCount+		BurstCount	
	coverage	accuracy	coverage	accuracy	coverage	accuracy	coverage	accuracy	coverage	accuracy
swim	0.90	0.96	1.00	1.00	0.78	1.00	0.97	1.00	1.00	1.00
mgrid	0.68	0.82	0.98	0.97	0.65	0.96	0.83	1.00	0.91	0.99
applu	0.45	0.74	0.98	0.96	0.75	0.93	0.78	1.00	0.95	0.99
gcc	0.65	0.94	0.97	0.99	0.69	0.97	0.74	1.00	0.93	0.99
art	0.96	0.95	0.99	0.99	0.91	1.00	0.90	1.00	0.97	0.99
mcf	0.75	0.82	0.99	0.97	0.47	0.98	0.54	0.99	0.93	0.98
ammp	0.54	0.69	0.94	0.90	0.55	0.95	0.68	0.95	0.77	0.95
lucas	0.90	0.92	0.97	0.98	0.99	1.00	0.96	1.00	0.88	0.99
parser	0.17	0.45	0.85	0.84	0.20	0.69	0.29	0.78	0.54	0.83
perlbmk	0.28	0.85	0.85	0.92	0.54	0.57	0.57	0.80	0.60	0.77
gap	0.42	0.77	0.96	0.98	0.39	0.97	0.41	1.00	0.88	0.99
sphinx	0.47	0.66	0.95	0.93	0.27	0.81	0.37	0.89	0.79	0.92
corner_turn	1.00	0.96	1.00	1.00	0.98	1.00	1.00	1.00	1.00	1.00
stream	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
vpenta	0.98	1.00	1.00	1.00	0.79	0.99	0.98	1.00	0.99	1.00
GeoMean	0.61	0.82	0.96	0.96	0.61	0.91	0.69	0.96	0.86	0.96

Table 5. Coverage and accuracy of DL1 DBPs (Single-threaded workloads)

Application	RefTrace		RefCount		RefCount+	
	cov.	accu.	cov.	accu.	cov.	accu.
swim	0.61	0.75	0.94	0.99	0.96	1.00
mgrid	0.69	0.80	0.74	0.93	0.85	0.98
applu	0.67	0.80	0.82	0.97	0.88	0.98
gcc	0.23	0.20	0.40	0.34	0.47	0.86
art	0.91	0.97	0.89	0.89	0.92	1.00
mcf	0.51	0.72	0.61	0.93	0.73	0.95
ammp	0.58	0.54	0.52	0.43	0.51	0.72
lucas	0.73	0.68	0.95	1.00	0.98	1.00
parser	0.18	0.21	0.15	0.14	0.23	0.56
perlbmk	0.88	0.69	0.80	0.91	0.85	0.92
gap	0.38	0.46	0.98	0.99	0.98	1.00
sphinx	0.28	0.54	0.40	0.33	0.37	0.79
corner_turn	0.41	0.40	0.55	0.43	0.40	0.85
stream	0.78	0.76	0.98	1.00	0.99	1.00
vpenta	N/A	N/A	N/A	N/A	N/A	N/A
GeoMean	0.51	0.55	0.63	0.64	0.66	0.89

Table 6. Coverage and accuracy of L2 DBPs (Single-threaded workloads)

# of processors	16
Issue width	4-way out of order
Instruction window	64-entry RUU, 32-entry Load/store queue
L1 I-cache	64KB, 2-way LRU, 64B cacheline, 2-cycle
L1 D-cache	64KB, 2-way LRU, 64B cacheline, 2-cycle
L2 cache	1MB private per core, 8-way LRU, 64B cacheline, 13-cycle
Coherence protocol	Snoop-based MOESI
Main memory	200-cycle

Table 7. Configuration of simulated MP machine

Because of the cache coherence protocol, the definition of prediction coverage for multi-threaded workloads differs slightly from the definition used for single-threaded workloads. Multiprocessor coverage is measured by the total number of blocks predicted dead when evicted from the cache or invalidated by the coherence protocol divided by the total number of evictions and invalidations.

Table 8 shows the prediction coverage and accuracy of each predictor used at the L1 data cache. For multi-threaded benchmarks, the benefit of using burst history over individual access history is more pronounced: BurstTrace makes 70% more correct predictions than RefTrace and BurstCount makes 40% more correct predictions than RefCount+, both with higher accuracy. Again, RefCount+ significantly outperforms RefCount with higher coverage and accuracy because of its ability to handle reference count variation better. Of the five predictors, BurstTrace is still the best choice because of its highest coverage, lowest overhead, and close to highest accuracy.

Table 9 shows the prediction coverage and accuracy of the L2 dead-block predictors. For the two counting-based predictors (RefCount, RefCount+), the prediction coverage and accuracy are much lower compared to those for the single-threaded workloads. This effect results from cache invalidations caused by the coherence protocol, which makes prediction harder. Although RefCount+ still has the highest accuracy and significantly outperforms RefCount, its coverage is only about 27%. Another phenomenon is that RefTrace has the highest coverage of the three predictors. This effect is also caused by the cache coherence protocol: the L2 caches in a CMP sees more accesses (upgrade requests, for example) from the L1 which would otherwise be filtered by the L1 cache in a single processor.

Application	RefTrace		RefCount		RefCount+	
	cov.	accu.	cov.	accu.	cov.	accu.
SPECweb	0.59	0.78	0.35	0.44	0.22	0.86
SPECjbb	0.46	0.58	0.32	0.40	0.23	0.67
TPC-W	0.45	0.74	0.31	0.38	0.27	0.89
barnes	0.32	0.69	0.20	0.09	0.19	0.81
FFT	0.38	0.58	0.10	0.29	0.37	0.57
lu	0.62	0.69	0.52	0.10	0.37	0.92
ocean	0.50	0.83	0.33	0.55	0.34	0.92
radix	0.42	0.59	0.11	0.38	0.23	0.54
GeoMean	0.46	0.68	0.25	0.28	0.27	0.76

Table 9. Coverage and accuracy of L2 DBPs (Multi-threaded workloads)

Application	RefTrace		BurstTrace		RefCount		RefCount+		BurstCount	
	coverage	accuracy	coverage	accuracy	coverage	accuracy	coverage	accuracy	coverage	accuracy
SPECweb	0.30	0.85	0.57	0.86	0.26	0.78	0.37	0.93	0.52	0.93
SPECjbb	0.15	0.69	0.59	0.89	0.24	0.65	0.28	0.88	0.51	0.92
TPC-W	0.09	0.62	0.41	0.89	0.14	0.42	0.16	0.82	0.37	0.88
barnes	0.36	0.68	0.81	0.93	0.29	0.59	0.24	0.75	0.50	0.85
FFT	0.67	0.91	0.73	0.94	0.58	0.90	0.62	0.99	0.60	0.95
lu	0.81	0.85	0.81	0.90	0.81	0.88	0.66	0.98	0.75	0.98
ocean	0.56	0.88	0.78	0.95	0.32	0.93	0.61	0.98	0.73	0.97
radix	0.86	0.82	0.78	0.89	0.58	0.84	0.58	0.90	0.61	0.88
GeoMean	0.37	0.78	0.67	0.91	0.35	0.72	0.39	0.90	0.56	0.92

Table 8. Coverage and accuracy of DL1 DBPs (Multi-threaded workloads)

5. Using Dead Block Prediction To Improve Performance

There are several distinct ways to use dead-block prediction to improve performance. A conservative approach, including replacement optimization and bypassing, only evicts dead blocks early to give other blocks more opportunities to get reused. A more aggressive approach prefetches new blocks into dead blocks to reduce future demand misses.

5.1. Evicting Dead Blocks Early

With LRU replacement, blocks with poor locality can stay in the cache too long and cause blocks with good locality to be replaced. Once a block is dead, it can be evicted from the cache before it becomes LRU. Early dead-block eviction gives other blocks that are located lower on the LRU stack more opportunities to get reused. However, if these blocks do not receive additional references, evicting dead blocks early does not improve performance.

One form of early dead-block eviction, replacement optimization, has been studied by prior work [14]. In replacement optimization, on a cache miss, the hardware first checks if any block in the set has already been predicted dead. If so, the dead block that is closest to the LRU is picked for replacement. If no dead block in the set is found, the LRU block is replaced.

A more aggressive form of early dead-block eviction, cache bypassing, targets blocks that will not be referenced if they are placed into the cache. Programs that exhibit poor locality or have a working set larger than the capacity of the cache have many such zero-reuse blocks. 32% of the blocks brought into the L1 cache and 40% of the blocks brought into the L2 cache for the single-threaded benchmarks studied in this paper are never reused. Dead block prediction can be used to identify these zero-reuse blocks; if a block causing a miss is predicted dead, it will not be written into the cache.

We evaluate several early dead-block eviction schemes, the speedups of which are shown in Figure 6. Like [14], all schemes are applied to the L2 cache because evicting dead blocks early at the L1 cache gives only marginal

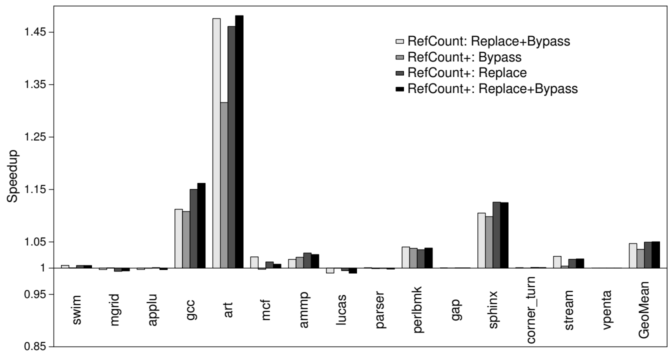


Figure 6. Speedups of using dead-block prediction for replacement/bypassing with a 1MB L2 cache

speedups. The "RefCount:Replace+Bypass" scheme, as described in [14], uses the RefCount dead block predictor for replacement and bypassing: on a miss, it first tries to find a dead block for eviction; if no dead block exists, it bypasses the missing block if it is predicted dead; otherwise, the LRU block is chosen for eviction. The "RefCount+:Replace" scheme uses the RefCount+ dead-block predictor just for replacement: on a miss, it first tries to find a block that is predicted dead; if no such block exists, the LRU block is replaced. The "RefCount+:Bypass" scheme uses RefCount+ just for bypassing: if a missing block is predicted dead, it is not written into the cache. The "RefCount+:Bypass+Replace" scheme is similar to "RefCount:Replace+Bypass" but uses RefCount+ for both replacement and bypassing.

Figure 6 indicates the four schemes achieve similar speedups of approximately 5% on average. The figure also indicates that the benefits of using dead-block prediction for bypassing and replacement are mostly overlapped: if a program benefits from bypassing, it also benefits similarly from the replacement optimization. Doing bypassing and replacement optimization at the same time does not bring much additional performance improvement.

5.2. Improving Prefetching

Early dead-block eviction by itself has the limitation that successful bypassing or early replacement of dead blocks does not always reduce the cache-miss rate. For programs that do not benefit from early dead-block eviction, a more aggressive technique, which replaces dead blocks with prefetched blocks, can be used.

While prefetching can be performed without dead-block prediction, using dead-block prediction to trigger prefetches has two benefits. First, dead blocks provide ideal space to store prefetched blocks without causing pollution. Second, the long dead time gives sufficient time for prefetched blocks to arrive at the cache before they are referenced.

One issue ignored by prior work that uses dead-block prediction for prefetching is how to track prefetched blocks so that the dead-block predictor can predict when these blocks become dead. The prefetch engine can bring many blocks into the cache and these prefetched blocks are not associated with any instruction in a program. Since all the dead-block predictors we study in this work use the PC to make predictions, prefetched blocks will not be predicted dead, preventing further prefetches from being triggered. To address this problem, an extra bit, *pc_valid*, is added to each block to differentiate prefetched blocks from blocks that are caused by demand misses. For prefetched blocks, the *pc_valid* bit is initially set to zero. When a prefetched block is accessed for the first time, its *pc_valid* bit is set to one and the PC of the current instruction is used to update the hashed PC stored along with the block.

Next, we investigate using dead-block prediction to prefetch into dead blocks at the L1 and L2 caches.

5.2.1. Baseline prefetch engine. We use an existing prefetching scheme, tag correlating prefetching (TCP) [8] as the baseline prefetch engine. TCP is a correlating prefetcher that was proposed to reduce the penalty of L1 misses but places prefetched data in the L2 cache to avoid polluting the L1 cache. With dead-block prediction at the L1 cache, prefetched data can be directly placed into the L1 cache. Figure 7 shows how TCP works. Each set maintains the two most recent tags that caused misses to the set. On a miss, a hash of the two tags in the miss history of the accessed set is used as index into the correlation table. If a match is found, the predicted tag is used with the index of the set to form a prefetch address. The correlation table is updated on every cache miss.

As a correlating prefetcher, TCP can learn arbitrary repetitive miss patterns. TCP also exploits the property that the same sequence of tags are often accessed in different sets, an effect known as constructive aliasing. Constructive aliasing enables TCP to learn access patterns more quickly.

The prefetch engine is configured as follows: both the L1 and L2 correlation tables are 2-way associative. The L1 table has 1024 sets and the L2 table has 8192 sets. Each entry in the table is 36 bits. Hence, the L1 correlation table is 9KB and the L2 correlation table is 72KB. Table 4 shows the parameters of the simulated machine.

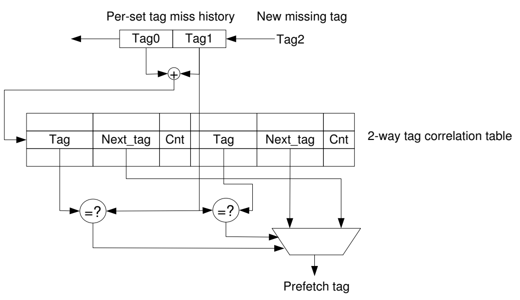


Figure 7. Baseline Tag Correlating Prefetch Engine

5.2.2. Using Dead Block Prediction to Improve L1 Prefetching. Using dead-block prediction to trigger prefetches into the L1 cache was first proposed by Lai et al. in a scheme called Dead Block Correlating Prefetching (DBCP) [16]. DBCP triggers prefetches when dead blocks are identified in the L1 cache, not when the L1 cache misses, to reduce pollution and improve the timeliness of prefetching. DBCP requires a large correlation table (a 2MB table for a 32KB directly-mapped L1 D-cache) because it records correlation of full block addresses, compared to Tag Correlating Prefetching, which records correlation of cache-line tags and needs a much smaller table (9KB for a 64KB two-way L1 cache). Tag correlation is smaller because one entry of tag correlation can represent multiple entries of full block address correlation, at the cost of potential aliasing.

Figure 8 compares the speedups of three L1 prefetching schemes. The baseline TCP prefetches on L1 misses and places prefetched blocks into the LRU position. The second scheme uses the RefTrace dead-block predictor with the baseline TCP. It prefetches when blocks become dead and places prefetched blocks into the space of the dead blocks. This scheme resembles the DBCP scheme because it uses the same dead-block predictor but differs from DBCP in the prefetch engine. The third scheme is similar to the second one except it uses BurstTrace, which works best at

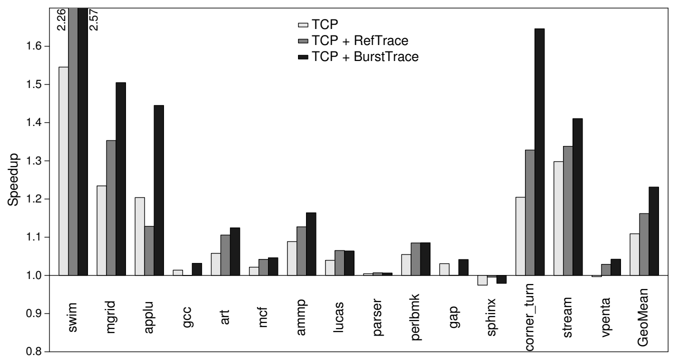


Figure 8. Speedups of L1 prefetching schemes with a 64KB L1 cache

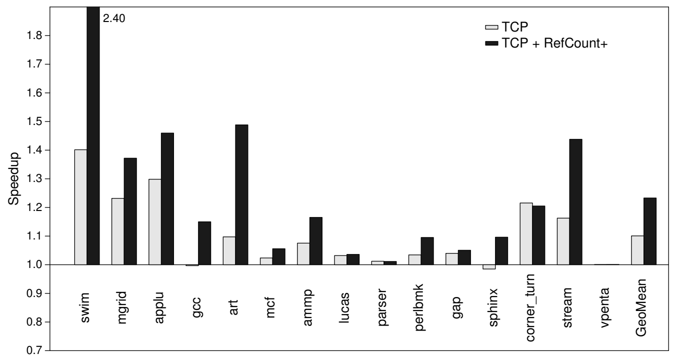


Figure 9. Speedups of L2 prefetching schemes with a 1MB L2 cache

the L1 cache. Figure 8 shows using dead-block prediction to trigger prefetches improves performance for almost all the applications. It also shows BurstTrace outperforms RefTrace when used with the baseline prefetch engine, because of its better dead-block prediction capability. On average, the baseline prefetch engine improves performance by 11%, adding RefTrace improves performance by 16%, and adding BurstTrace improves performance by 23%.

5.2.3. Using Dead Block Prediction to Improve L2 Prefetching. Dead block prediction can also be used to trigger prefetches into L2 caches, which has not been evaluated in prior dead-block prediction work. Applying dead-block prediction to L2 prefetching differs from L1 prefetching in several ways. First, the L2 cache is more tolerant of pollution but L2 misses are much more expensive. Therefore L2 prefetching should be more aggressive. Second, dead-block prediction at the L2 cache has much lower coverage (66%) than at the L1 (96%). This means one third of the dead blocks are not identified by dead-block prediction and triggering prefetches only when dead blocks are identified will miss many opportunities to prefetch. Therefore, besides issuing prefetches when dead blocks are identified in the L2 cache, additional prefetches are issued when the L2 cache misses, to cover the otherwise missed opportunities of those dead blocks that are not identified by dead-block prediction.

Figure 9 shows the speedups of two L2 prefetching schemes: the baseline TCP, which prefetches on L2 misses, and the baseline TCP augmented with RefCount, which prefetches both when L2 misses and when blocks become dead in the L2 cache. The figure shows using RefCount to trigger additional prefetches improves performance by 23% compared to the performance improvement of 10% by the baseline prefetch engine.

6. Conclusion

The efficacy of the cache is determined by the amount of useful data it stores, not the capacity of the cache. In this paper, we propose several dead-block predictors that identify dead blocks with better accuracy and coverage than prior schemes and use these predictors to eliminate dead blocks and increase the efficacy of the cache.

For both the L1 and L2 caches, predicting the death of a block when it becomes non-MRU, not immediately after it is accessed, gives the best tradeoff between timeliness and prediction accuracy/coverage. Because of the differences in L1 and L2 accesses, a dead-block predictor should maintain different state in each block to make better dead-block predictions at the L1 and L2 cache.

For the L1 cache, a dead-block predictor should maintain state about cache bursts, not individual references, to make predictions. Cache bursts are more predictable because they hide the irregularity of individual references. Therefore, burst-based predictors can correctly identify more dead blocks while making fewer predictions. The best burst-based predictor can identify 96% of the dead blocks in the L1 D-cache with a 96% accuracy.

For the L2 cache, a dead-block predictor should maintain state about reference counts to make predictions. To cope with reference count variation, we optimize an existing predictor by using more up-to-date history information to increase prediction accuracy and filtering out sporadic smaller reference counts to increase prediction coverage. The improved predictor can identify 66% of the dead blocks in the L2 cache with a 89% accuracy.

We used dead-block prediction to improve performance through replacement optimization, bypassing, and prefetching. Replacement optimization and bypassing eliminate dead blocks only on demand misses whereas prefetching aims to eliminate dead blocks whenever they are identified. On average, replacement optimization or bypassing improves performance by 5% while prefetching into dead blocks brings a 12% performance improvement over the baseline prefetching scheme for the L1 cache and a 13% performance improvement over the baseline prefetching scheme for the L2 cache. These results indicate that it is possible to increase cache efficiency by storing useful data in the space of dead blocks. On the other hand, even after these optimizations, the average cache efficiency is still low (17% for the L1 and 27% for the L2), due to the following reasons: dead blocks identified too late, wrong dead-block predictions, dead blocks not identified, the time spent waiting for correctly prefetched blocks to arrive, and useless prefetches. It remains to be seen how better prefetching schemes can push the cache efficiency even higher.

Acknowledgments

The authors would like to thank Babak Falsafi, who contributed a lot to this work. The authors also thank members of the CART research group and the anonymous reviewers for their valuable feedback on this paper. This work is supported by the Defense Advanced Research Projects Agency under contract F33615-03-C-4106 and by NSF grant EIA-0303609.

References

[1] J. Abella, A. Gonzàlez, X. Vera, and M. F. P. O'Boyle. IATAC: a smart predictor to turn-off L2 cache lines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(1):55–77, March 2005.

[2] D. Burger, J. R. Goodman, and A. Kägi. The declining effectiveness of dynamic caching for general-purpose microprocessors. Technical Report 1261, Computer Sciences Department, University of Wisconsin, Madison, 1995.

[3] R. Desikan, D. C. Burger, S. W. Keckler, and T. M. Austin. Sim-alpha: a validated, execution-driven alpha 21264 simulator. Technical Report TR-01-23, Computer Sciences Department, University of Texas at Austin, 2001.

[4] M. Ferdman and B. Falsafi. Last-touch correlated data streaming. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2007.

[5] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[6] A. González, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proceedings of the 9th International Conference on Supercomputing*, 1995.

[7] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

[8] Z. Hu, M. Martonosi, and S. Kaxiras. TCP: Tag correlating prefetchers. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, 2003.

[9] J. Huh. Hardware techniques to reduce communication costs in multiprocessors, 2006. Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin.

[10] J. Jalminger and P. P. Stenström. A novel approach to cache block reuse prediction. In *Proceedings of the 2003 International Conference on Parallel Processing*, 2003.

[11] T. L. Johnson, D. A. Connors, M. C. Merten, and W. W. Hwu. Run-time cache bypassing. *IEEE Transactions on Computers*, 48(12):1338–1354, 1999.

[12] M. Kampe, P. Stenström, and M. Dubois. Self-correcting LRU replacement policies. In *Proceedings of the 1st conference on Computing frontiers*, 2004.

[13] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[14] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4):433–447, 2008.

[15] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 139–148, 2000.

[16] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 144–154, 2001.

[17] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48–59, 1995.

[18] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995.

[19] A. Mendelson, D. Thiébaut, and D. Pradhan. Modeling live and dead lines in cache memory systems. *IEEE Transactions on Computers*, 42(1):1–14, 1993.

[20] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr, and J. Emer. Adaptive insertion policies for high-performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[21] R. Rabbah, I. Bratt, K. Asanovic, and A. Agarwal. Versatility and versabench: A new metric and a benchmark suite for flexible architectures. Technical Report MIT-LCS-TM-646, MIT Techincal Memo, June 2004.

[22] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing resue information in data cache management. In *Proceedings of the 12th International Conference on Supercomputing*, 1998.

[23] J. Sartor, S. Venkiteswaran, K. S. McKinley, and Z. Wang. Cooperative caching with keep-me and evict-me. In *The 9th IEEE Annual Workshop on the Interaction between Compilers and Computer Architectures*, 2005.

[24] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architecture and Compilation Techniques*, 2001.

[25] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Memory coherence activity prediction in commercial workloads. In *3rd Workshop on Memory Performance Issues*, 2004.

[26] P. Stenström. Chip-multiprocessing and beyond. In *Keynote at 12th International Symposium on High-Performance Computer Architecture*, 2006.

[27] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995.

[28] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2002.

[29] W. A. Wong and J.-L. Baer. Modified LRU policies for improving second-level cache behavior. In *Proceedings of 6th International Symposium on High Performance Computer Architecture*, 2000.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*.