Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior

Zhigang Hu[†] Stefanos Kaxiras[‡] Margaret Martonosi[†]
[†]Department of Electrical Engineering, Princeton University, {hzg, mrm}@ee.princeton.edu
[‡] Communication Systems and Software, Agere Systems, kaxiras@agere.com

Abstract

Techniques for analyzing and improving memory referencing behavior continue to be important for achieving good overall program performance due to the ever-increasing performance gap between processors and main memory. This paper offers a fresh perspective on the problem of predicting and optimizing memory behavior. Namely, we show quantitatively the extent to which detailed timing characteristics of past memory reference events are strongly predictive of future program reference behavior. We propose a family of timekeeping techniques *that optimize behavior based on observations about particular cache time durations, such as the cache access interval or the cache dead time. Timekeeping techniques can be used to build small, simple, and high-accuracy (often 90% or more) predictors for identifying conflict misses, for predicting dead blocks, and even for estimating the time at which the next reference to a cache frame will occur and the address that will be accessed. Based on these predictors, we demonstrate two new and complementary time-based hardware structures: (1) a time-based victim cache that improves performance by only storing conflict miss lines with likely reuse, and (2) a time-based prefetching technique that hones in on the right address to prefetch, and the right time to schedule the prefetch. Our victim cache technique improves performance over previous proposals by better selections of what to place in the victim cache. Our prefetching technique outperforms similar prior hardware prefetching proposals, despite being orders of magnitude smaller. Overall, these techniques improve performance by more than 11% across the SPEC2000 benchmark suite.*

1 Introduction

For several decades now, memory hierarchies have been a primary determinant of application performance. As the processor-memory performance gap has widened, increasingly aggressive techniques have been proposed for understanding and improving cache memory performance. These techniques have included prefetching, victim caches, and more [3, 7, 8, 12, 15, 16, 17, 18]. More recently, the static and dynamic power dissipation of cache structures has also been a vexing problem, and has led to another set of techniques for improving cache behavior from the power perspective [6, 9, 13, 19, 23].

For most of these cache power and performance optimizations, a key hurdle for their success lies in classifying behavior such that hardware and software can deduce which optimizations to apply and when. For example, classification schemes that identify conflict misses or that predict upcoming reuse may be useful in determining which items would be best to store in a victim cache.

Most prior work on caches has focused on time-independent reference activity. In these approaches, event ordering and interleaving are of prime importance. In contrast, the time durations between events play a lesser role. For example, Tam et al. propose managing multi-lateral caches using cache-line access information [20]. They take into account the number of times a cache line is accessed but not any timing information about the accesses themselves. Charney and Reeves were first to propose address correlation in hardware for data prefetching [2]. Likewise, Joseph and Grunwald propose a Markov-based predictor to guide prefetch, but they use a time-independent Markov model; it tracks the sequence of accesses but not the time durations between them [7].

In contrast, this paper shows quantitatively that the timing characteristics of reference events can be strongly predictive of program reference behavior. Time-based tracking of memory references in programs can be a powerful way of understanding and improving program memory referencing behavior. Just as one example, tracking the time duration between when a cached item is last successfully used and when it is evicted (i.e., the *dead time*) can be an accurate predictor of whether the cached data is involved in a mapping conflict for which a victim cache may be helpful.

Mendelson et al. proposed an analytical model for predicting the fraction of live and dead cache lines [14] of a process, as a function of the process' execution time and the behavior of other processes in a multitasking environment [11]. Wood et al. introduced one notion of time-based techniques as a way of improving the accuracy of cache sampling techniques in simulations [22]. They showed that one can deduce the miss rates of unprimed references at the beginning of reference trace samples by considering the proportion of cycles a cache line spends *dead* or waiting to be evicted. More recently, work on cache decay is another example of time-based methods for managing cached data [9]. Here each cache line has a 2-bit timer associated with it to track recency of accesses to it. That work used this tracking to propose that cache lines that have not been recently accessed be turned off, or "decayed", in order to save leakage energy.

Our work studies the predictive and classificatory role of timing statistics more broadly. In particular, this paper shows that time-based techniques are quite effective in deducing aspects of cache behavior that have previously been either hard to discern on-the-fly in a single program run, or hard to discern at all. For example, identification of conflict misses often uses multiple simulation runs [5] or elaborate hardware structures [3]. Our work here shows that conflict misses can be identified with good accuracy simply based on small cache line counters.

The paper makes contributions at three levels:

- First, we construct an expanded set of useful metrics regarding generational behavior in cache lines, and we provide quantitative characterizations of the SPEC2000 benchmarks for these metrics.
- Second, using these metrics, we introduce a fundamentally different approach for on-the-fly categorization of application reference patterns. We give reliable predictors of conflict misses, dead blocks and other key aspects of reference behavior.
- Third, based on our ability to discover these reference patterns on-the-fly, we propose hardware structures that exploit this knowledge to improve performance. In particular, we propose and evaluate mechanisms to manage victim caches and prefetching. Our victim cache technique improves over previous proposals by better selecting what to place in it. Our hardware prefetching technique also outperforms prior related proposals and is orders of magnitude smaller as well. These techniques improve performance by more than 11% across the SPEC2000 benchmark suite.

More broadly, we expect that these metrics and methods will offer researchers even more intuition for further hardware structures in the future.

The remainder of the paper is structured as follows. Section 2 presents the experimental methods used in the paper. Section 3 outlines terminology and gives overview statistics regarding the generational aspect of cache behavior which is fundamental to our approach. Then, in Section 4, we present a hardware-efficient method to improve victim cache performance using timekeeping techniques. Section 5 then discusses an even more aggressive application of timekeeping techniques: we propose a prefetching system based on (i) dead block prediction, (ii) reuse analysis, and (iii) reference timing, all mainly discerned from simple-to-implement timekeeping metrics. Finally, Section 6 discusses some remaining details and offers our conclusions.

2 Methodology and Modeling

2.1 Simulator

To evaluate our proposals, we use a modified version of Simplescalar 3.0 [1] to simulate an aggressive 8-issue out-of-order processor. The main processor and memory hierarchy parameters are shown in Table 1. Because contention can have important influence on performance, we have incorporated a simulator modification that accurately models contention at the L1/L2 and memory buses [10]. As in [10], the busses always give processor memory requests priority over hardware prefetch requests.

2.2 Benchmarks

We evaluate our results using the SPEC CPU2000 benchmark suite [21]. The benchmarks are compiled for the Alpha instruction set using the Compaq Alpha compiler with SPEC *peak* settings, which include aggressive software prefetching.

Processor Core	
Clock rate	2GHZ
Instruction Window	128-RUU, 128-LSQ
Issue width	8 instructions per cycle
Functional Units	8 IntALU,3 IntMult/Div, 6 FPALU,2 FPMult/Div, 4 Load/Store Units
Memory Hierarchy	
L1 Dcache Size	32KB, 1-way, 32B blocks 64 MSHRs
L1 Icache Size	32KB, 4-way, 32B blocks
L1/L2 bus	32-byte wide, 2GHZ
L2 I/D	each 1MB, 4-way LRU, 64B blocks,12-cycle latency
L2/Memory bus	64-byte wide, 400MHZ
Memory Latency	70 cycles
Prefetcher	
Prefetch MSHRs	32
Prefetch Request Queue	128 entries

Table 1: Configuration of Simulated Processor

In our simulation, we treat these software prefetches as normal memory reference instructions. In section 5, we also experiment with ignoring all the software prefetches to evaluate the effect of software prefetching on our timekeeping prefetching.

For each program, we skip the first 1 billion instructions to avoid unrepresentative behavior at the beginning of the program's execution. We then simulate 2 billion instructions using the reference input set. We include some overview statistics here for background. Figure 1 shows how much the performance (IPC) of each benchmark would improve if *all* conflict and capacity misses in L1 data cache could be eliminated. This is the target we aim for in our memory optimizations. The programs are sorted from left to right according to the amount they would speed up if conflict and capacity misses could be removed. Figure 2 breaks down the misses of these programs into three stacked bar segments denoting cold, conflict and capacity misses. An interesting observation here is that the programs that exhibit the biggest potential for improvement (RHS of Figure 2) also tend to have comparatively more capacity misses than conflict misses. Thus, we expect that eliminating capacity misses will result in larger benefit than eliminating conflict misses. This is confirmed in later sections. Although we focus mainly on those benchmarks that benefit the most from the elimination of conflict and capacity misses, we also present results for other benchmarks for completeness.

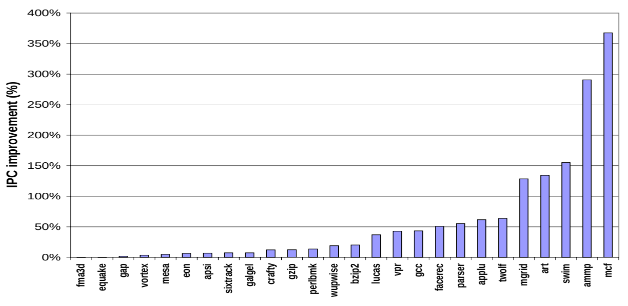


Figure 1: Potential IPC improvement if all conflict and capacity misses in L1 data cache could be eliminated for SPEC2000 benchmarks.

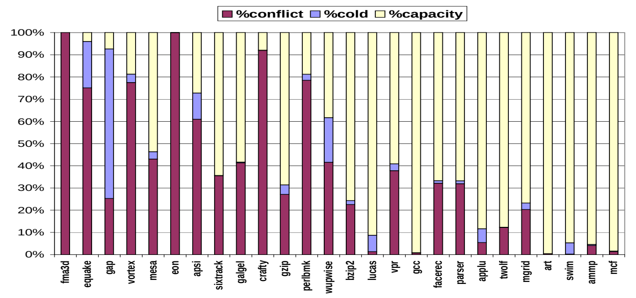


Figure 2: Breakdown of program L1 data cache misses into three categories: conflict, cold and capacity.

3 Generational Behavior of Cache Lines: Metrics and Motivation

The generational behavior of cache lines is a well-established phenomenon. As illustrated in Figure 3, each generation is defined as beginning with a cache miss that brings new data into this level of the memory hierarchy. A cache line generation ends when data leaves the cache because a miss to some other data causes its eviction. Each cache line generation is divided into two parts: the *live time* of the cache line, where the line is actively accessed by the processor and the *dead time*, awaiting eviction.

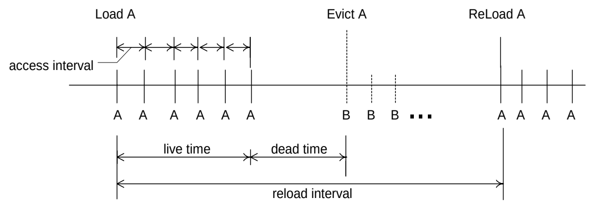


Figure 3: Timeline depicting a generation of the cache line with A resident, followed by A's eviction to begin a generation with B resident. Eventually, A is re-referenced to begin yet another generation.

The live time of a cache line starts with the miss that brings the data into the cache and ends with the last successful hit before the item is evicted. The dead time is defined as the time duration where the cached data will not be used again successfully *within this generation*. That is, the dead time is the time between the last hit and when the data is actually evicted. Many times we see only a single miss and then eviction. We consider these cases to have zero live time; therefore the generation time equals to the dead time. Such cases are important for classifying misses, as we will show in later sections.

There are further metrics that also turn out to be of practical interest. These include the access interval and reload interval. Access interval refers to the time intervals between successive accesses to the same cache line *within the live time of a generation*. In contrast, reload interval is used to denote the time duration between the beginnings of two generations that involve the same memory line. The reload interval in one level of the hierarchy (eg, L1) is actually the access interval in the next lower level of the hierarchy (eg, L2) assuming the data is resident there.

We examine four of the metrics illustrated in Figure 3 (live time, dead time, access interval and reload interval) and give an initial sense of their distributions and how they compare.

Ultimately, the goal of these data is the following. Imagine that execution is at some arbitrary point along the timeline depicted in Figure 3. We can know something about past history along that timeline, but we wish to predict what is likely to happen soon in the future. First, we wish to deduce *where we currently are*. That is, are we currently in live time or dead time? Dead time cannot be perfectly known until it is over, but if we can predict it accurately, we can build power and performance optimizations based on this dead block prediction. Second, we wish to deduce *what will happen next*: a re-reference of the current cached data? Or a reference to new data that begins a new generation? Accurately deducing what will be referenced next *and when* is a crux issue for building effective victim caches and prefetchers. This paper demonstrates quantitatively that our timekeeping techniques allow effective predictions and optimizations that address both of these questions.

Overview Distributions: Live Time, Dead Time, Access Interval and Reload Interval Figure 4 illustrates a distribution for SPEC2000 benchmarks of *live times* and *dead times* within cache generations. Recall that Live time is defined as the time duration between when a memory line arrives in cache, and when it experiences its last successful use (last hit) before it is evicted to end the generation. Dead time is defined as the time duration between when an item in the cache is last used successfully as a cache hit, and when it is evicted from the cache. An effective prefetching technique might be based on deducing when a line will be dead, and then proactively prefetching the next line that will map there. We discuss such strategies in Section 5.

Dead times are in general much longer than average live times. For example, over all of the SPEC suite, 58% of live times are 100 cycles or less. In contrast, only 31% of dead times are less than 100 cycles. This is a useful observation because it hints that we can succeed in discerning dead versus live times a priori based on the durations observed.

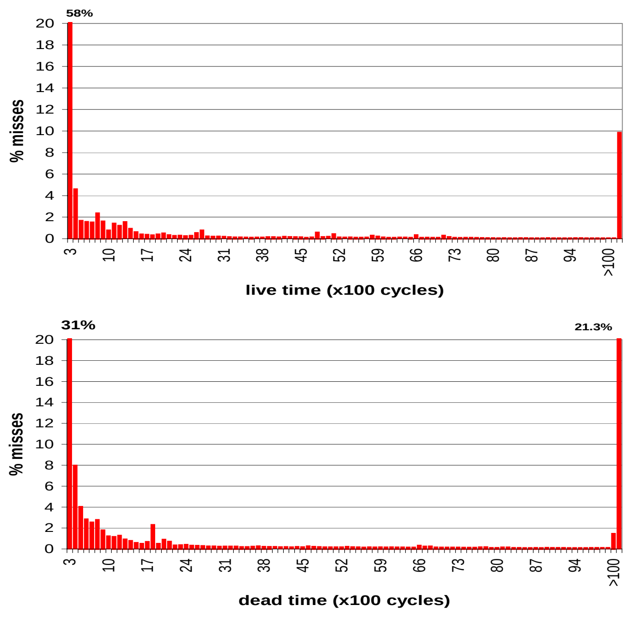
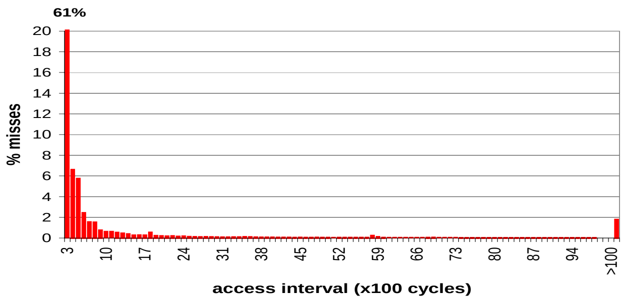


Figure 4: Distribution of live times and dead times for all generations of cache lines in the SPEC2000 simulations.



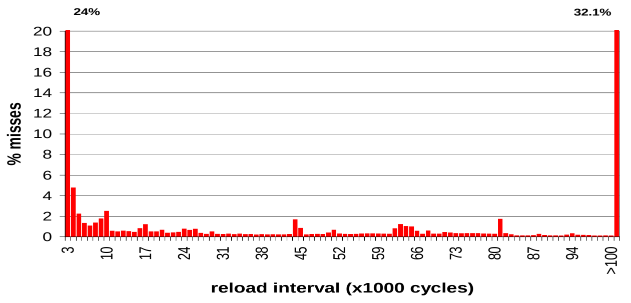


Figure 5: Distribution of access intervals and reload intervals for all generations of cache lines in the SPEC2000 simulations.

Two additional metrics: access interval and reload interval, also help to classify references. Access interval is the time duration between successive references (hits) within a cache live time. In contrast, reload interval is the time between the beginnings of two successive generations involving the same memory line. For data resident in L2 cache, a reload interval in the L1 cache corresponds to the access interval of the same data in the L2 cache, down one level in the hierarchy. Figure 5 illustrates access interval and reload interval distributions. Note that reload intervals are plotted with the x-axis 1000X space cycles rather than 100X as in the access interval graph. The distributions here are even more distinct. 91% of access intervals are less than 1000 cycles, while only 24% of reload intervals are in that range. In addition to the distributions of absolute values for these metrics, their variability over time is also of interest. We revisit this issue in Section 5.

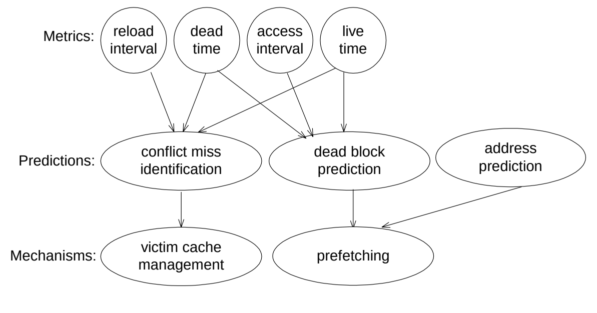


Figure 6: Timekeeping metrics, some resulting predictions based on them, and related hardware mechanisms.

From Statistics to Hardware Figure 6 depicts the stages one goes through in using timekeeping metrics to build mechanisms for improving memory behavior. In the top layer are basic metrics, all of which appear in Figure 3. They include reload interval, dead time, access interval, and live time. As subsequent sections will show, these *metrics* can be used to identify *predictions* about program behavior. These predictions might include identifying conflict misses or deducing dead cache blocks. Interestingly, sometimes more than one metric can be used as a predictor of the same behavior. For example, reload interval, dead time, and live time can each be turned into a conflict miss predictor (see Section 4), each with different tradeoffs in terms of predictive accuracy and coverage. Finally, these predictions can, in turn, be composed into *mechanisms* that actively respond and optimize based on the prediction. Tracking the timekeeping metrics requires little hardware; essentially just coarse-grained simple counters that are ticked periodically (but not necessarily every cycle) from the global cycle counter provided on most microprocessors. We discuss mechanisms and their implementations in the sections that follow.

4 Timekeeping Metrics to Identify and Avoid Conflict Misses

Canonically, cache misses are classified into 3 categories: cold miss, conflict miss and capacity miss [4]. Cold misses occur when a cache line is loaded into the cache the first time. Conflict misses are those misses which can be eliminated by a fully-associative cache. Capacity misses are those which will miss even with a fully-associative cache.

Interpreting Hill's definitions with generational behavior, a conflict miss occurs because its last generation was unexpectedly interrupted—something that would have not happened in a fully associative cache. Similarly, a capacity miss occurs because its last generation was ended because of lack of space—again, something that would not have happened in a larger cache. In this section we quantitatively correlate the miss types with timekeeping metrics. When we correlate metrics to a miss type we always refer to the timekeeping metrics of the last generation of the cache line that suffers the miss. In other words, what happens to the current generation of a cache line tells us something about its next miss.

4.1 Identifying Conflict Misses

By Reload Interval While Figure 5 showed reload intervals over all generations, Figure 7 splits the reload interval distribution into two graphs for different miss types. These statistics show vividly different behavior for conflict and capacity misses. In particular, reload intervals for capacity misses are overwhelmingly in the tail of the distribution. In contrast, reload intervals for conflict misses tend to be fairly small: an average of roughly 8000 cycles. The average reload interval for a capacity miss is one to two orders of magnitude larger than that for a conflict miss! Large reload intervals for capacity misses make sense: for an access to an item to be classified a capacity miss, there must be at least 1024 (total number of blocks in the cache) unique accesses to drive the item out of a fully-associative cache after its last access. In a processor that typically issues 1-2 memory accesses per cycle, the time for 1024 accesses is on the order of a thousand cycles, and it may take much longer before 1024 *unique* lines have been accessed. On the contrary, a conflict miss has no more than 1024 unique cache accesses after their last access; this leads to their small reload intervals.

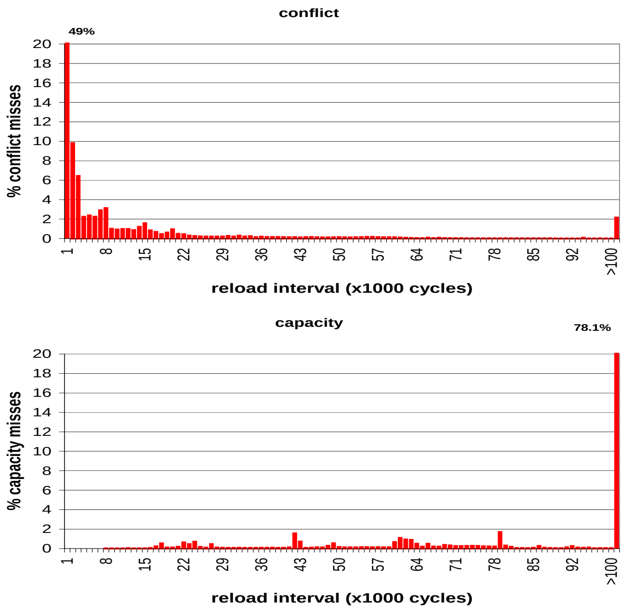


Figure 7: Distribution of reload interval for conflict (Top) and capacity (Bottom) misses

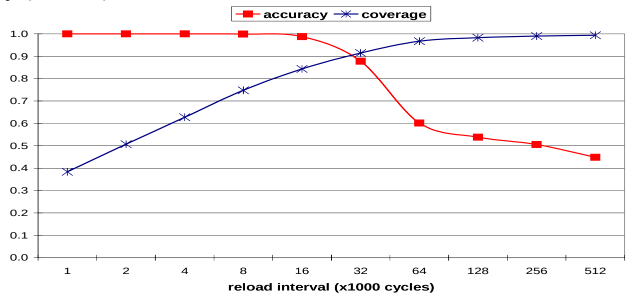


Figure 8: Accuracy and coverage for conflict miss predictions based on *reload interval*. Each data point indicates what the accuracy or coverage would be for predicting conflict misses to be all instances where the *reload interval* is less than the quantity on the x-axis.

Reload intervals make excellent predictors of conflict misses. Figure 8 shows accuracy and coverage when reload interval is used as predictor. For each point on the x-axis, one curve gives the accuracy of predicting that reload intervals less than that x-axis value denote conflict misses. The other curve gives the coverage of that predictor: i.e., how often it makes a prediction.

When conflict misses are defined as small reload intervals (about 1000 cycles or less) prediction accuracy is close to perfect. Coverage, the percent of conflict misses captured by the prediction, is low at that point, however, about 40%. The importance of reload interval, though, shows in the behavior of this predictor as we increase the threshold: up to 16K cycles, accuracy is stable and nearly perfect, while coverage increases to about 85%. This is appealing for selecting an operating point because it means we can walk out along the accuracy curve to 16K cycles before accuracy sees any substantive drop. The clear drop there makes that a natural breakpoint for setting up a conflict predictor based on reload intervals smaller than 16K.

By Dead Time Figure 9 shows the distribution of dead time divided by miss types. Again, we see trends similar to reload interval distribution, though not as clear cut. That is, dead times are typically small for conflict misses, while much larger for capacity misses. These observations about dead times hint at a phenomenon one could exploit. Namely, one can deduce that an item has been "prematurely" evicted from the cache due to a conflict miss, if its dead time is quite short. Where dead times are quite large, it hints at the fact that the item probably left the cache at the end of its "natural lifetime"; that is, it was probably evicted as a capacity miss at the end of its usage.

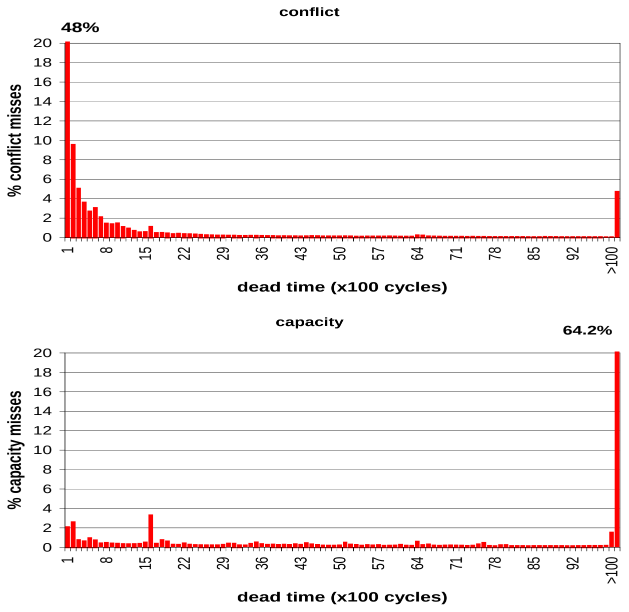


Figure 9: Distribution of dead time for conflict (Top) and capacity (Bottom) misses

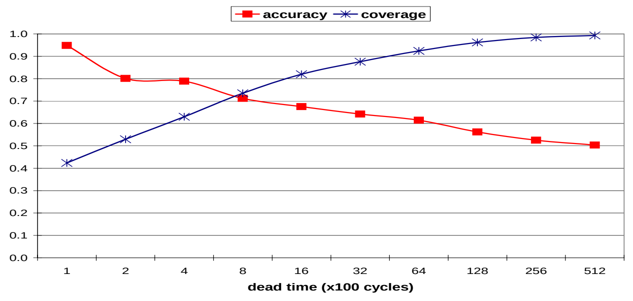


Figure 10: Accuracy and coverage for conflict miss predictions based on *dead time*. Each data point indicates what the accuracy or coverage would be for predicting conflict misses to be all instances where the *dead time* is less than the quantity on the x-axis.

Figure 10 shows accuracy and coverage of a predictor that predicts an upcoming conflict miss based on the length of the dead time of the current generation. Namely, for a point on the x-axis, accuracy and coverage data indicate what the prediction outcome would be if one considered dead times less than that value as indicators of conflict misses. Coverage is essentially the fraction of conflict misses for which we make a prediction. Accuracy is the likelihood that our prediction is correct, for the instances where we do make a prediction.

As Figure 10 shows, predicting a conflict miss if the dead time of its last generation is smaller than a given threshold is very accurate (over 90%) for small thresholds (100 cycles or less). But coverage is only about 40% (attesting to the fact that most dead times are large). Increasing the dead-time threshold degrades accuracy but increases coverage. A likely method for choosing an appropriate operating point would be to walk down the accuracy curve (i.e., walk out towards larger dead times) until just before accuracy values drop to a point of insufficient accuracy. One can then check that the coverage at this operating point is sufficient for the predictor's purpose. In Section 4.2, we describe a hardware mechanism that uses dead-time predictions of conflict misses to filter victim cache entries.

By Live Time Live time is also highly biased between conflict (very small live times) and capacity misses (larger live times). A very important special case here is when we have a live time equal to zero. This special case makes for a simple and fairly accurate predictor of conflict misses. In fact, a single ("re-reference") bit in each L1 cache line is all that is needed to distinguish between zero and non-zero live times.

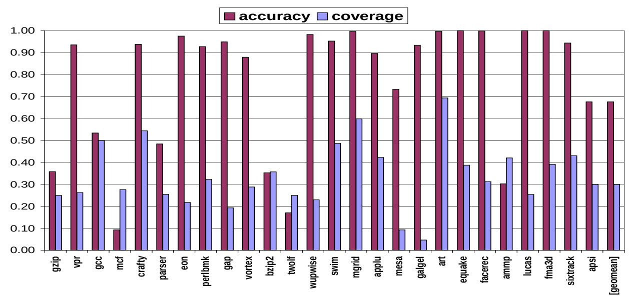


Figure 11: Accuracy and coverage of using "live time = 0" as a predictor of conflict misses.

Figure 11 shows the accuracy and coverage of such a prediction. Accuracy is very high: for many programs, accuracy is close to one. The geometric mean for all SPEC2000 is 68% accuracy, but coverage is low. Coverage varies from benchmark to benchmark with a geometric mean of roughly 30%. In contrast to the previous approaches this prediction has no knobs to turn to trade accuracy for coverage. Because of the low coverage and its specialized nature, live-time conflict prediction is likely to be useful in few situations. We include it here mainly to demonstrate how different metrics can classify or predict behavior.

Prediction Location Conflict predictors based on dead times (or live times) rely only on L1-centric information. In contrast, conflict predictors based on reload intervals would most likely be implemented by monitoring access intervals in the L2 cache. As a result, one's choice of how to predict conflict misses might depend on whether the structure using the predictor is more conveniently implemented near the L1 or L2 cache.

4.2 Utilizing Conflict Miss Identification: Managing a Victim cache

A victim cache is a small fully-associative cache that reduces L1 miss penalties by holding items evicted due to recent conflicts. Victim caches help with conflict misses, and in the previous subsection we discussed timekeeping metrics that can be used as reliable indicators of conflict misses, particularly dead time and reload interval.[1] Here, we propose using these conflict indicators to manage the victim cache. In particular, we want to avoid entering items into the victim cache that are unlikely to be reused soon.

Small reload intervals are highly correlated to conflict misses and they are an effective filter for a victim cache. The intuition that ties reload intervals to victim caches is the following: Since the size of victim cache is small, a victim block will stay only for a limited time before it is evicted out of the victim cache. In terms of generational behavior, this means that only victim blocks with small reload intervals are likely to hit in the victim cache. Blocks with large reload intervals will probably get evicted before their next access so it is wasteful to put them into the victim cache. Unfortunately, reload intervals are only available for counting in L2. This makes it difficult for their use as a means to manage an L1 victim cache.

Besides short reload intervals, short dead times are also very good indicators of conflict misses. Dead times are readily available in L1 at the point of eviction and as such are a natural choice for managing a victim cache associated with the L1. We use a policy in which the victim cache only captures those evicted blocks that have dead times of less than a *threshold* of 1K cycles. Figure 9 shows that these blocks are likely to result in conflict misses.

The hardware structure of the dead-time victim filter is shown in Figure 12. A single, coarse-grained counter per cache line measures dead time. The counter is reset with every access and advances with global ticks that occur every 512 cycles. Upon a miss the counter contains the time since the last access, i.e., the dead time. An evicted cache line is allowed into the victim cache if its counter value is less than or equal to 1 (giving a range for the dead time from 0 to 1023 cycles). Our experiments show that for a 32-entry victim cache managed in this way, the traffic into the victim cache is reduced by 87%. This reduction is achieved without sacrificing performance, as seen in Figure 13.[2]

Collins et al. [3] suggest filtering the victim cache traffic by selecting only victims of possible conflict misses. Their solution requires storing an extra tag for each cache line (remembering what was there before) to distinguish conflict misses from capacity misses. Comparing our approach with a Collins-style filter in Figure 13, we see similar traffic reduction, but our timekeeping based filter leads to higher IPC for most of the benchmarks. Note that as in Figure 1, in Figure 13 the potential for speedup increases to the right, but the ratio of conflict misses to total misses increases to the left. In Figure 13, the programs that experience the largest speedups

[1]We do not further examine the zero-live-time predictor because of its relatively low coverage and significant overlap with the technique based on dead time.

[2]The traffic reduction by our filter also implies power reduction but we have not examined this in depth since it is beyond the scope of this paper.

with our timekeeping victim filter are clustered in the middle of the graph. Programs to the far left have little room for improvement. Programs to the far right whose misses are overwhelmingly capacity misses are *negatively* affected with an *unfiltered* victim cache, but they retain their performance if a conflict filter (either Collins-style or timekeeping) is employed.

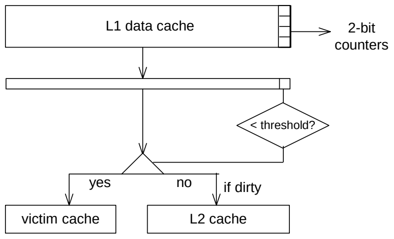


Figure 12: Implementation of a timekeeping victim cache filter

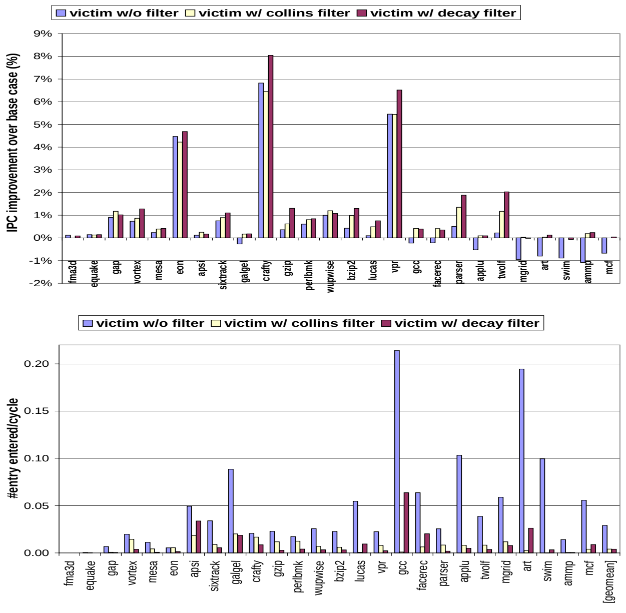


Figure 13: IPC improvement (Top) and fill traffic to victim cache (Bottom) for timekeeping victim cache filter compared to Collins filter.

The performance of our timekeeping victim filter indicates that the parameters (dead-time threshold, cache sizes, etc.) are well matched. This is not coincidental. What makes our timekeeping techniques invaluable is that they provide a sound framework to reason about such parameters rather than to revert to trial-and-error. We will informally "prove" that the optimal dead-time threshold actually stems from the size of the victim cache and the reuse characteristics of the program. It is essentially a form of Little's Law, from queueing theory. The reasoning is as follows:

1. We can think of a victim cache as providing associativity for some frames of the direct-mapped cache. Without any filtering, associativity is provided on a first-come, first-served basis: every block that is evicted gets an entry in the victim cache.
2. Our timekeeping filtering based on dead time results in a careful selection of the frames for which the victim cache is allowed to provide associativity. Timekeeping filtering culls out blocks with dead times greater than the threshold. In turn, the dead-time threshold controls the *number* of frames for which associativity is provided for.
3. Our filtering ensures that the victim cache will provide associativity *only* for the "active" blocks that are fairly-recently used at the time of their eviction.
4. Since the victim cache cannot provide associativity to more frames than its entries, the best size of the victim cache relates to the amount of cache in active use. A larger set of "active" blocks dilute the effectiveness of the victim cache associativity. In the data here, with a 1K cycle dead time threshold, only about 3% of cache blocks resident at any moment meet the threshold. Since 3% of 1024 total cache blocks is 30.72, a 32-entry victim cache is a good match.

The relation of the dead-time threshold and the size of the victim cache not only gives us a good policy to statically select an appropriate threshold, but also points to adaptive filtering techniques. Although beyond the scope of this paper, adaptive filtering adjusts the dead time threshold at run-time so the number of candidate blocks remains approximately equal to the number of the entries in the victim cache. With a modest amount of additional hardware an adaptive filter would perform even better than static filter shown above, which already outperforms previous proposals.

5 Timekeeping Metrics for Dead-Block Prediction and Prefetch

In addition to identifying conflict misses, timekeeping metrics are also of much broader use. We show them here used as a guide in coordinating data prefetches. Managing data prefetch can be thought of in terms of three sub-problems:

- Identifying dead blocks in cache, into which we should prefetch the next block to be referenced.
- Identifying which next block should be prefetched
- Identifying *when* the prefetch must occur, in order for it to be timely.

Timekeeping metrics can form the basis of building efficient hardware prefetch mechanisms. Predicting prefetch targets can be orthogonal to the timeliness of the prefetch, but as we will show in this section it can also be integrated well with time predictions.

5.1 Dead Block Prediction

Dead block prediction is important for prefetch, because prefetches that arrive into the cache before the resident block is dead will induce extra cache misses. Live time, dead time and access interval statistics are closely related to the liveness of a block. In this section, we explore these correlations to construct predictors for dead blocks. First, we discuss a simple predictor which simply differentiates among large dead times and short access intervals. We then propose a dead block predictor that is based on the regularity of live times.

5.1.1 By Dead Time

One method of identifying dead blocks is by measuring time between accesses [9]. If we measure an inordinately large time since the last access, and we have yet to encounter the next access then chances are that we are within dead time.

Both access intervals and dead times refer to time between consecutive accesses to the same cache frame. The difference is that during the dead time the current block in the frame turns out to be dead. A dead block predictor can be constructed by dynamically distinguishing dead times from access intervals. From Figure 5, we observe that most access intervals are very short and clustered around zero. On the other hand, a number of dead times are quite large and thus clearly distinguishable from access intervals. Based on this observation, a dead block predictor could be constructed as follows: if the idle time of a block exceeds some threshold, we predict that it is in its dead time. Figure 14 shows the accuracy and coverage of this predictor with different threshold values. To get high accuracy, the threshold must be more than 5120 cycles. At this point the coverage (the percent of the blocks for which we do make a prediction) is only about 50%.

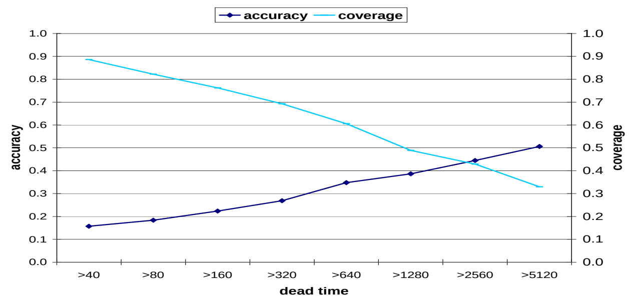


Figure 14: Dead block prediction based on dead time: Accuracy and coverage.

5.1.2 By Live Time

Dead block prediction based on dead time is very successful in reducing leakage energy in caches, because that application benefits from the many CPU cycles spent in the heavy tail of the dead time distribution. But for prefetching, dead-time prediction does not have enough coverage to be useful. Even more important, once one waits a long decay interval to make a decision about deadness, it could potentially be too late to do a timely prefetch. In this section, we will look at an alternate way of determining that a block is dead (and therefore ready for something to be prefetched on top of it.) If we can accurately predict how long a live time is, then we will know that prefetches arriving just after this live time ends should be timely. This paper is the first to show the degree to which live times can be predicted; we then use this to schedule timely prefetches.

As with other predictors, past history is our guide in predicting the next live time of a block. The simplest history-based predictor is to predict that the live time of a block will be the live time of its previous generation. To test this, we profiled variability of consecutive live times per block using counters with a resolution of 16 cycles. Figure 15 shows the profiling results for selected SPEC2000 programs (which have significant speedup potential and are discussed in detail in section 5.2.3) and for the geometric mean for all SPEC2000 programs. A significant percentage (more than 20%) of the differences are less than 16 cycles.

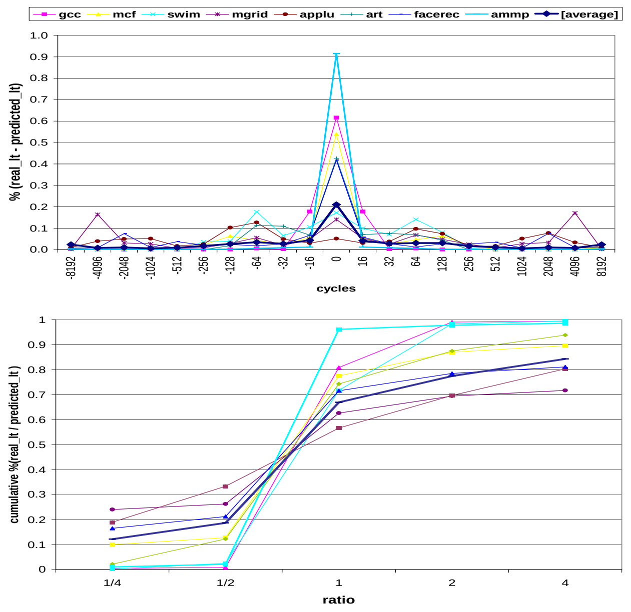


Figure 15: Distribution of absolute difference (Top) and cumulative distribution of relative ratio (Bottom) of consecutive live times.

Based on this regularity of the live times we can construct a predictor for dead blocks as follows: at the start of a block's generation we predict that its live time is going to be similar to its last live time. After its predicted live time is over, we wait a brief interval and then we predict the block to be dead. The question is: how long should we wait before predicting the block is dead? To account for some variability in the live time we could add a fixed number of cycles to the predicted live time. Because live times have a wide range in magnitude, however, we chose instead to scale the added time to the predicted live time. To choose an appropriate scaling factor, the second graph of Figure 15 shows the cumulative distribution of the ratio of the current live time divided by the previous live time. As we can see in this graph, on average, about 80% of the current live times are less than twice the previous live time.

Thus, a simple heuristic is to predict that a block is dead at a time twice its previous live time from the start of its current generation. Additional justification for this predictor comes from our observation in Section 2 that dead times are significantly larger than live times. Using this dead block predictor, Figure 16 shows the accuracy and coverage for the SPEC2000 programs. Coverage in this case refers to the percentage of blocks for which we do make a prediction. Blocks with a generation *shorter* than twice their predicted live time have already been evicted by the time of the prediction so they are not covered by our predictor. On average (for all SPEC2000), accuracy is around 75% and coverage about 70%, both better than those of the dead-time dead block predictor in the previous section. There is also a discernible trend for increased accuracy and coverage to the right of the graph towards the programs with significant percentage of capacity misses and significant potential for speedup.

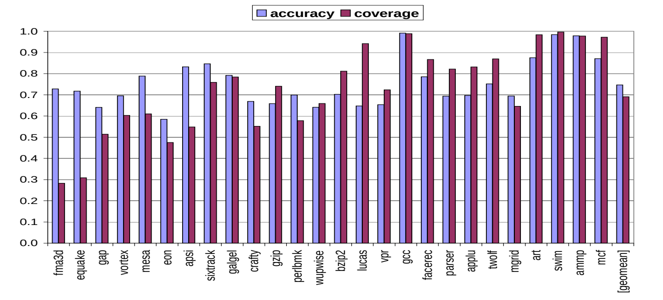


Figure 16: Accuracy and coverage of live time based dead block prediction

5.2 Timekeeping Prefetch

A full-fledged prefetching mechanism needs to establish both what to prefetch and when to prefetch it. Regarding what to prefetch, a history-based predictor can help to provide accurate address prediction, as discussed, for example, in [2], [7] and [10]. Regarding when to prefetch, a live-time dead-block prediction is an efficient mechanism to schedule prefetches but this also requires a predictor structure to predict live times.

In this section we show that the *same* structure that can predict addresses can also predict live-times (or vice-versa), unifying the two predictors into a single structure. We propose a compact, history-based, predictor for both addresses and live-times that outperforms previous proposals for most of the SPEC2000 benchmarks. Furthermore, it requires only a tiny fraction of the area compared to prior proposals: about two orders of magnitude smaller than [10].

5.2.1 Address and Live-Time Predictor

Our predictor is a correlation table not unlike the Dead Block Correlating Predictor (DBCP) table in [10]. In our case, the reference history used by our predictor is just the most recent miss address per frame, which is readily available in L1. In contrast, the DBCP approach also requires a PC trace which, in many cases, is complex to obtain from within the out-of-order core. We use miss address per cache frame, rather than the global miss trace of the cache. This means that a prediction that refers to a specific frame takes into account only the miss trace of this frame. The issue is complicated somewhat in set-associative caches where we use *per set* miss trace history but we still perform all timekeeping and accounting on a per frame basis. Per set miss trace history removes some of the conflict misses that are dispersed within the history of the capacity misses. This is an advantage for our prefetching mechanism since it caters mostly to capacity misses as we will show later in this section.

Each predictor entry stores both the prediction for the prefetch address and the predicted live time of the block to be replaced by the prefetch. We use a 1-miss history to get these predictions. For example, assume that block A occupies a cache frame. At the point when block B replaces A we access the predictor using the (per-frame) history (A,B). The predictor returns a prediction that block C should replace B *and* a prediction for the live time of B. Using the predicted live time of B, we apply our live-time-based dead-block prediction and we "declare" B to be dead at a time twice its predicted live time. At that point in time, we schedule the prefetch to C to occur. (One could also estimate when C needs to arrive, and exploit any slack to save power or smooth out bus contention.)

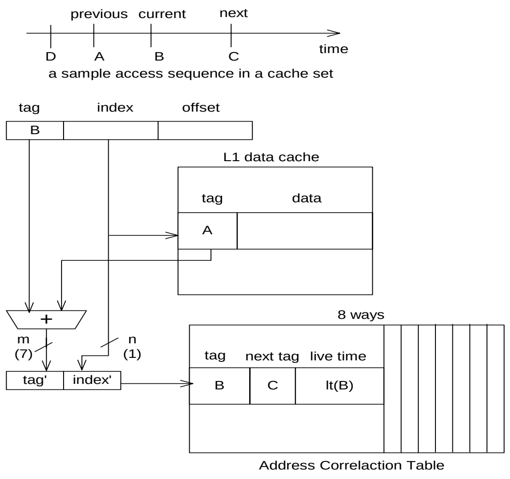


Figure 17: Structure of Timekeeping Address Correlation Table

5.2.2 Implementation Details

Figure 17 shows how we access the address correlation table and in particular the indexing mechanism we use. When block B replaces block A in a cache frame we add the *tags* of A and B (using truncated addition as per [10]). When combined with A and B's common index, the sum of the tags gives us a pointer to the correlation table. The pointer is constructed by taking *m* bits from the sum of the tags and *n* bits from the index. The correlation table is typically set-associative so the pointer selects a set in this table. We then select the correct entry in the set by matching the tag of the block B to the identification tag in the predictor entry. The selected predictor entry predicts the *tag* of the block to be prefetched. The index is implied and is the same as in A and B. The same entry also gives a prediction for the live time of B. This live-time prediction is at the crux of our ability to do timely prefetch.

We have tested several sizes of this table ranging from megabytes to just a few kilobytes. Even very small tables work surprisingly well. An interesting observation arises when we index this table using mainly tag information and only partial index information (*n* less than 10). In this case, histories from different cache frames (or sets) may map to the same entry. This results in *constructive* aliasing and allows our table to have much smaller size than the table in [10]. The intuition behind this constructive aliasing is that often multiple distinct data structures are traversed similarly. If accesses in one data structure imply accesses to another data structure, it does not matter what particular element is accessed in the one or the other. A contrived and simplistic example is to imagine a loop that adds elements of two arrays and stores them in a third array. Many triads of elements accessed can share a single entry in the correlation table as long as their tags remain the same! They all exhibit the same access pattern.

Within every cache line, we need the following timekeeping hardware for our prefetch: two counters, a register, and two extra tag fields, as shown in Figure 18. The two counters and the register are only 5 bits long each. The results we present in this paper are for an 8KB, 8-way set-associative correlation table. We index the table using seven bits from the sum of tags (*m*=7) and one bit from the cache index (*n*=1).

One counter (gt_counter) and one register (lt_register) are needed to track live time as follows. The gt_counter is initialized at the beginning of a generation and is continuously incremented by the global tick until the next miss. At this point the counter contains the generation time. At every intermediate hit the gt_counter is copied over to the lt_register so at any point in time the lt_register trails the gt_counter by one access. Thus, when a generation ends, the value of the lt_register is the time from the start of the generation to the last access, i.e., the live time. An additional counter (prefetch_counter) and a tag field (next_tag) are needed to schedule a prefetch while another tag (prev_tag) is needed for predictor update as discussed below.

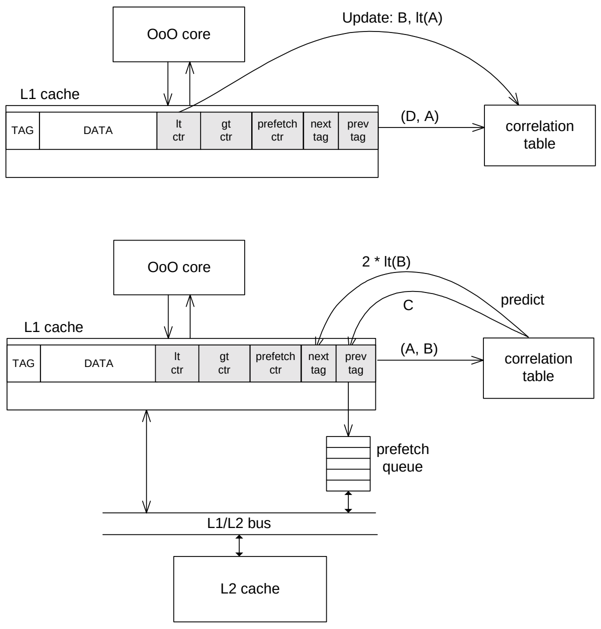


Figure 18: Update of a predictor entry (Top) and predictor access to make a prediction (Bottom)

Figure 18 shows the overall structure for timekeeping prefetch. The correlation table sits besides the L1 cache. Assume we have the following sequence of blocks in a cache frame: D,A,B,C, as shown in the top part of Figure 17. When a miss on address B attempts to replace block A, the lt_counter contains the live time of A, the prev_tag contains D, and the following actions occur:

1. A demand fetch is sent to the L2 for block B.
2. Predictor update (top diagram of Figure 18). An index is computed from A and its precursor D. The predictor table is accessed with history (D,A) and the entry corresponding to A is updated with B as the predicted next tag and lt(A) as the next prediction for the live time of A. Then A is installed in prev_tag.
3. Predictor access (bottom diagram of Figure 18). An index is computed from B and the currently evicted block A. The predictor is accessed with history (A,B) and an entry is selected that corresponds to B. Predictions for the live time of B and the next tag C are obtained and installed in the prefetch_counter and the next_tag respectively. (The live time is doubled by shifting one bit before it is installed in the prefetch counter.) The prefetch counter is decremented with every tick. When it reaches zero the prefetch to C is put into an 128-entry prefetch queue (which is modeled as in other prefetching work).

5.2.3 Results

Figure 19 shows the IPC improvement over the base configuration. We include results for our 8KB timekeeping correlation table and we compare to the prior proposed DBCP table of 2MB size. Our timekeeping based prefetch achieves higher IPC improvement than DBCP in all SPEC2000 benchmarks except mcf and ammp. In addition, it improves performance for all but four of the SPEC2000 programs. Overall, our prefetch mechanism achieved 11% IPC improvement while DBCP only achieved about 7% improvement. Referring back to Figure 1, we see that our prefetching mechanism achieves significant speedups for many of the programs with a very large percentage of capacity misses (programs to the right of the graph in Figures 1 and 19) without harming those heavy on conflict misses (programs to the left of the graphs). The best performers are gcc, facerec, applu, mgrid, art, swim, ammp, and mcf. From the programs with the highest potential for speedup only two, twolf and parser, do not benefit from prefetch. These two programs exhibit very low accuracy in address prediction which results in a slight performance loss for twolf; the same programs are problematic even with a 2MB DBCP.

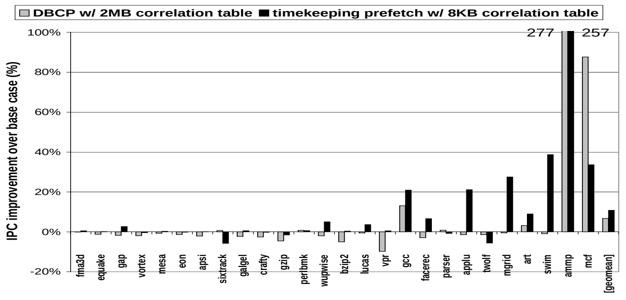


Figure 19: IPC improvement using timekeeping prefetch with an 8KB correlation table vs. DBCP prefetch with a 2MB correlation table

Considering only the eight best performers we see that, although the achieved speedups are significant, there is still room for improvement when compared to the ideal case. The differences are explained by close examination of the accuracy and coverage of our address prediction and the *timeliness* of our prefetches. Figure 20 shows the address accuracy and coverage for our 8KB address correlation table. Coverage in this case refers to the hit rate of the predictor; if we miss in the predictor we cannot make an address prediction. Figure 21 classifies the timeliness of the prefetches for the correct and wrong address predictions. Each bar (from bottom to top) shows prefetches that are:

- “early,” arrived early and displaced the current *live* block
- “discarded,” thrown out of the prefetch queue before been issued to the L2 to make space for new prefetches
- “timely,” arrived within the dead time and before the next miss

- “started_but_not_timely,” issued, but arrived late (after the next miss)
- “not_started,” did not even issue before the next miss

From Figures 20 and 21 we can deduce the following. For two programs mgrid and facerec, while their address accuracy and coverage are fair, only 40% and 30% respectively of their correct prefetches are timely, while most of their other prefetches are late. This is because these two programs have short generation times and it is difficult to pinpoint their dead times. Two programs, art and to a lesser extent gcc, have a lot of discarded prefetches because of burstiness. This was also observed in DBCP prefetching. In addition art suffers from low address accuracy. The reason why mcf does not achieve its full potential is because of its low address accuracy. This program benefits from very large address correlation tables and this is the reason why it is doing well with a 2MB DBCP. We observed better performance for mcf with our timekeeping prefetch when we used a larger address correlation table of 2MB. Finally, ammp which speeds up by 257% — almost all of its potential — shows very good address accuracy and coverage and in addition shows very timely prefetches. As a general observation, the timeliness of our prefetches, especially with respect to earliness, correlates well with the accuracy of the address prediction: when we predict addresses well, we tend not to displace live blocks (Figure 21).

The results shown in this section are obtained using SPEC2000 binaries compiled with *peak* compiler settings, which aggressively employ software prefetching. We observed similar results when ignoring all compiler inserted prefetches. The interaction between compiler prefetch and timekeeping prefetch is out of the scope of this paper, but makes an interesting topic for future work.

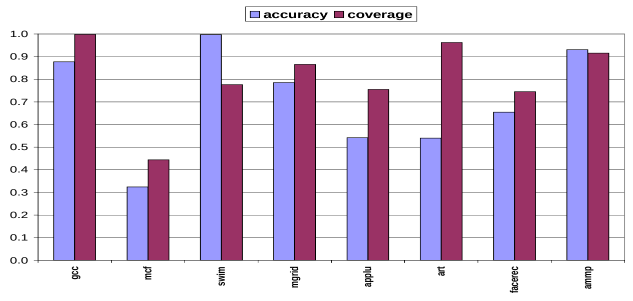
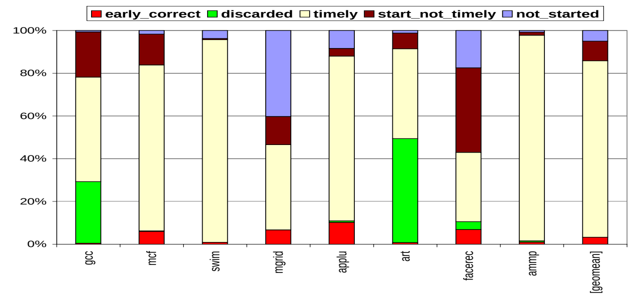


Figure 20: Address accuracy of the 8KB correlation table for the eight best performers

6 Conclusions

This paper demonstrates the predictive power of using time-based techniques to identify and optimize for various aspects of memory referencing behavior. We show that timing generational characteristics such as live times, dead times, access intervals, and reload intervals allows one to classify misses and deduce other characteristics of reference behavior with high accuracy. Only few, small counters per cache line are needed to obtain the time measurements we use for miss classification or prediction.



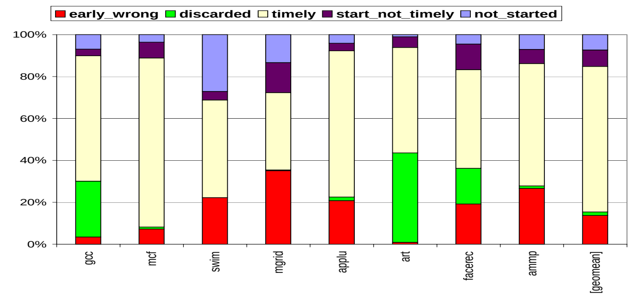


Figure 21: Timeliness of prefetches for correct (Top) and wrong (Bottom) address predictions

We first observe that our time-based metrics can be used to distinguish among conflict and capacity misses. Typically a block’s generation is expected to exhibit a short live time followed by a long dead time, a characteristic exploited in cache decay. However, conflict misses are “catastrophic” to the typical generation of a block in that they cut its live time or dead time short. A generation resulting from a conflict exhibits either a zero live time or an inordinately short dead time. Furthermore, since the block is thrown out of the cache despite being alive, its reload interval (the access interval in the next lower level of the cache hierarchy) is also very short. These observations lead to three different run-time predictions of conflict misses. We explore the accuracy and the coverage of these predictions and we propose a mechanism to take advantage of such run-time prediction. Specifically, we propose filtering a victim cache so as to feed it only with blocks evicted as a result of a conflict. We demonstrate this filter using a dead time conflict predictor. Our victim cache filtering results in both IPC improvements and significant reduction in victim cache traffic. Our filter outperforms a previous proposal that predicts conflict misses remembering previous tags in the cache.

We then use our timekeeping techniques to tackle the problem of predicting when a block is dead and prefetch another block in anticipation of the next miss. With this technique we attempt to address the problem of capacity misses. Cache decay readily offers such a dead block prediction based on the time difference among access intervals and dead times. Unfortunately, the accuracy and coverage of decay, although fine for leakage control, are not ideally suited for prefetching. One of the contributions of this paper is the discovery that live times, when examined on a per cache-frame basis, exhibit regularity. Predicting the live time of the current block (based on its previous live times) allows us to schedule a prefetch to take place shortly after the block “dies.” To implement a timekeeping prefetch we need both an address and a live time predictor. We propose a new history-based predictor that provides both predictions simultaneously. Our predictor is a correlation table accessed using the history of the previous and current miss in a frame. It predicts the live time of the current block, and the address to prefetch next. Because we index this predictor using mostly tag information we observe significant *constructive aliasing* both for addresses and live times. This allows us to outperform a 2MB DBCP predictor [10] using just 8KB of predictor state for all SPEC2000 with an average IPC improvement (over the base configuration) of 11%.

To summarize how our contributions affect the performance of the programs in the SPEC2000 suite we present an overview in the form of a Venn diagram in Figure 22. The diagram depicts three intersecting sets of programs:

1. programs with few memory stalls (negligible speedup is expected for these programs),
2. programs helped by timekeeping victim cache filter,
3. programs helped by timekeeping prefetch

Next to each program we indicate the IPC improvement (for programs in the intersections of the sets we indicate the maximum improvement). We note that, for the most part, our timekeeping victim cache filter helps with programs that suffer mostly from conflict misses while our timekeeping prefetch helps with programs suffering from capacity misses. Few programs benefit from both mechanisms. Overall, Figure 22 shows that our mechanisms are complementary in handling potential memory stalls.

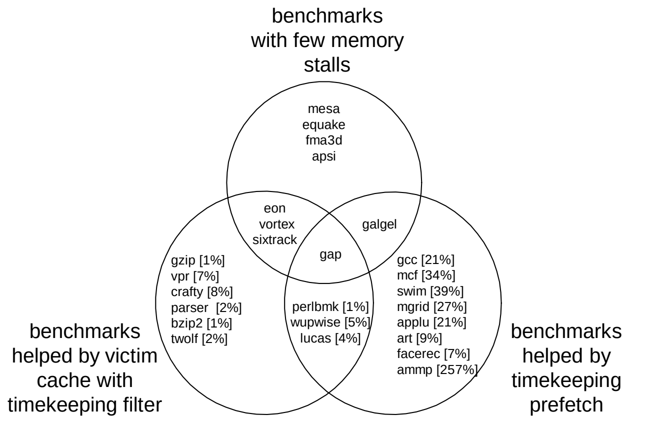


Figure 22: Effect of timekeeping victim cache and timekeeping prefetch on SPEC2000 benchmarks

Although our mechanisms cover many of the programs for which speedup can be expected, there is still some room for improvement. Timekeeping techniques can offer new perspective for predicting memory behavior and improving program memory performance. Using these techniques, we expect that researchers will be able to uncover other effective, hardware-efficient performance and power optimizations.

7 Acknowledgments

We would like to thank An-Chow Lai for insightful discussions and for providing us with detailed bus models for SimpleScalar. Our thanks to the anonymous referees for providing helpful comments as well. This work was supported in part by NSF ITR grant 0086031 and by donations from IBM and Intel.

References

[1] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *Computer Architecture News*, pages 13–25, June 1997.

[2] M. J. Charney and A. P. Reeves. Generalized correlation-based hardware prefetching. Technical Report EE-CEG-95-1, School of Electrical Engineering, Cornell University, 1995.

[3] J. Collins and D. Tullsen. Hardware identification of cache conflict misses. In *Proc. 32nd Intl. Symp. on Microarchitecture*, pages 126–135, 1999.

[4] M. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California at Berkeley, Nov. 1987.

[5] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[6] M. G. Johnson Kin and W. H. Mangione-Smith. The filter cache: An energy efficient memory structure. In *Proc. Micro-30*, Nov. 1997.

[7] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[8] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proc. ISCA-17*, May 1990.

[9] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *Proc. 28th Annual Intl. Symp. on Computer Architecture*, 2001.

[10] A.-C. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction and Dead-Block Correlating Prefetchers. In *Proc. 28th Annual Intl. Symp. on Computer Architecture*, 2001.

[11] A. Mendelson, D. Thiébaut, and D. K. Pradhan. Modeling live and dead lines in cache memory systems. *IEEE Transactions on Computers*, 42(1):1–16, Jan. 1993.

[12] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proc. ASPLOS-V*, pages 62–73, Oct. 1992.

[13] M. D. Powell et al. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *ISLPED*, 2000.

[14] T. R. Puzak. *Analysis of cache replacement algorithms*. PhD thesis, University of Massachusetts, Amherst, 1985.

[15] V. Santhanam, E. H. Gornish, and W.-C. Hsu. Data prefetching on the HP PA-8000. In *Proc. ISCA-24*, pages 264–73, June 1997.

[16] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, Sept. 1982.

[17] S. Srinivasan, R. Ju, A. Lebeck, and C. Wilkerson. Locality vs. criticality. In *Proc. 28th Annual Intl. Symp. on Computer Architecture*, 2001.

[18] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *International Symposium on Microarchitecture*, pages 148–159, 1998.

[19] C. Su and A. Despain. Cache Designs for Energy Efficiency. In *Proceedings of the 28th Hawaii Int'l Conference on System Science*, 1995.

[20] E. S. Tam, J. A. Rivers, V. Srinivasan, G. S. Tyson, and E. D. Davidson. Active management of data caches by exploiting reuse information. *IEEE Transactions on Computers*, 48(11):1244–1259, 1999.

[21] The Standard Performance Evaluation Corporation. WWW Site. http://www.spec.org, Dec. 2000.

[22] D. A. Wood, M. D. Hill, and R. E. Kessler. A Model for Estimating Trace-Sample Miss Ratios. In *ACM SIGMETRICS*, pages 79–89, June 1991.

[23] H. Zhou, M. Toburen, E. Rotenberg, and T. Conte. Adaptive mode control: A static-power-efficient cache design. Sept. 2001.