Predicting Last-Touch References under Optimal Replacement

Wei-Fen Lin and Steven K. Reinhardt

Advanced Computer Architecture Laboratory
EECS Department
University of Michigan
Ann Arbor, MI 48109-2122
{wflin,stever}@eecs.umich.edu

Abstract

Effective cache replacement is becoming an increasingly important issue in cache hierarchy design as large set-associative caches are widely used in high-performance systems. This paper proposes a novel approach to approximate the decisions made by an optimal replacement algorithm (OPT) using last-touch prediction. The central idea is to identify, via prediction, the final reference to a cache block before the block would be evicted under OPT—the "OPT last touch". Given perfect prediction, replacing the referenced block immediately after each OPT last touch would give optimal replacement behavior.

This paper evaluates the feasibility of this approach by studying, at a fundamental level, the predictability of OPT last-touch references, and the applicability of these predictions to improving replacement decisions. We show that trace-based predictors, previously proposed for LRU last-touch prediction, can predict OPT last touches as well. We enhance these predictors using future information, but find that their performance degrades significantly as cache size and associativity increase. We introduce a new class of predictors based on last-touch history, which significantly outperform trace-based predictors on large set-associative secondary caches. Across eight SPEC CPU2000 benchmarks on a 1MB 16-way associative secondary cache, an idealized history-based OPT last-touch predictor can potentially eliminate 39% of LRU misses—eliminating 63% of the gap between LRU and OPT.

1. Introduction

The effectiveness of a cache is strongly influenced by its replacement policy. Although the common least recently used (LRU) replacement algorithm performs well in many situations, comparing its performance with that of optimal replacement (OPT) [2] often indicates significant room for improvement. For example, computations that cycle through a working set slightly larger than the cache cause LRU to exhibit notorious worst-case behavior. In addition, LRU is not guaranteed to provide decreasing miss rates with increasing associativity; in fact, the opposite behavior—increased associativity leading to increased misses—is not uncommon. As a result, caches with higher associativities, such as the 16-way set-associative level-two cache on the AMD Athlon processor [1], provide greater opportunities for improving replacement beyond LRU. As processor performance continues to outstrip reductions in off-chip memory access latency, maximizing the effectiveness of these large on-chip caches is of growing importance.

Unfortunately, optimal replacement is not generally achievable in practice, as it requires knowledge of future accesses. However, many applications exhibit repetitive reference patterns; we may be able to exploit this repetition to approach OPT. For example, Figure 1 plots reference addresses vs. time (in references) for portions of four applications from the SPEC CPU2000 suite. Repetitive patterns are clearly visible in *ammp*, *art* and *mcf*, while *twolf*'s access pattern appears

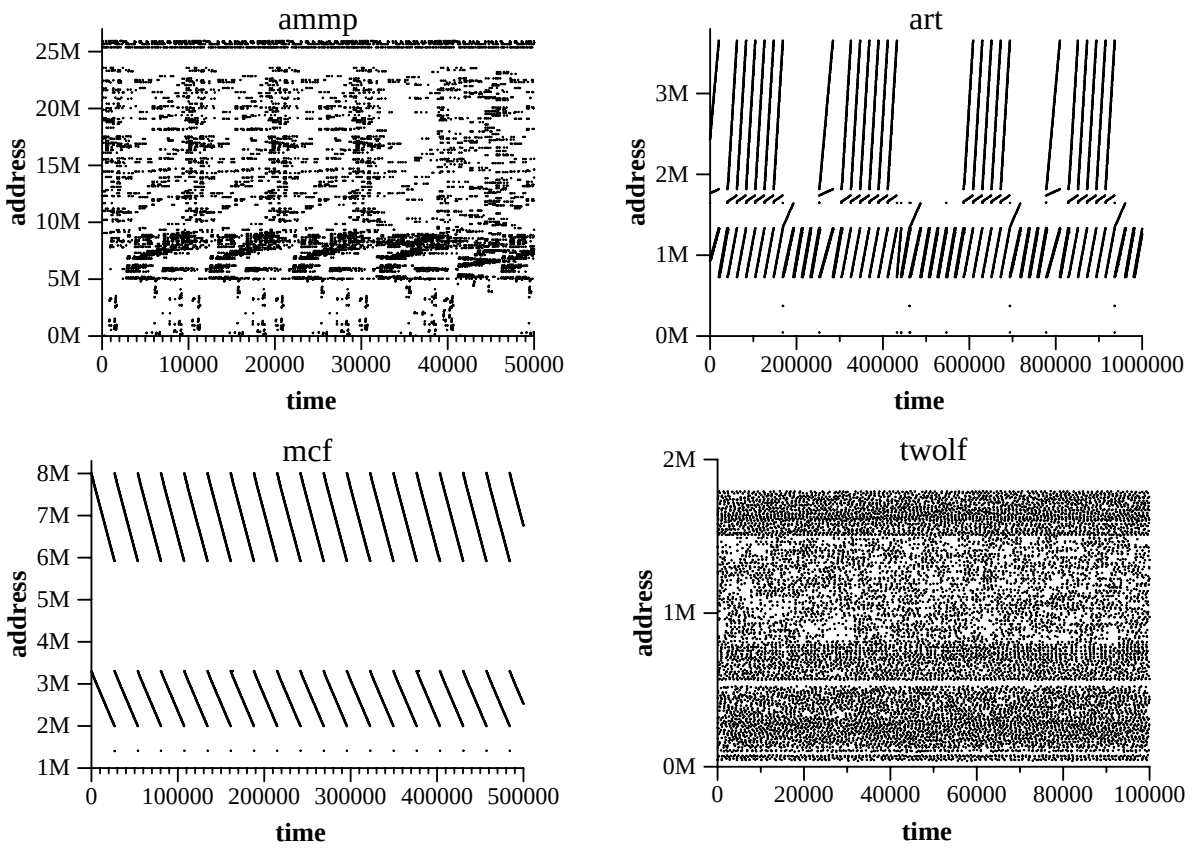

Figure 1. Memory reference behavior

random. Of the eight benchmarks we study in this paper—those from SPEC CPU2000 having the largest gap between LRU and OPT behavior—we find that five exhibit clear repetitive patterns when plotted in this fashion. If we can determine the optimal replacement decisions for one iteration of the repetitive pattern—a feasible task once that iteration completes—we may then apply those decisions in future iterations to approximate optimal replacement.

A practical implementation of such a scheme must address numerous issues. In this paper, we focus on one significant and crucial question: Given knowledge of *past* optimal replacement decisions (i.e., those that can be computed using an off-line algorithm on the references that have already been observed), can we correlate those past decisions with *future* references in order to "replay" those optimal decisions at the appropriate points?

This question has two facets. First, how do we represent past replacement decisions in such a way that they can be reapplied in the future? Second, how do we correlate the multiple points in the reference stream at which the same replacement decision should be applied? The second question depends on the first, as the particular representation of a replacement decision will determine the set of points at which that decision should be applied.

We address both these facets using techniques based on last-touch prediction [10]. We record OPT behavior by identifying the final reference to each cache block before OPT would have replaced the block from the cache (i.e., the "OPT last touch" reference). We then collect signatures (e.g., program counter and/or address information) for these OPT last-touch references, which ideally serve to distinguish them from non-last-touch references. If a current reference's

signature is among the set that correlates with previous OPT last-touch references, we can then reenact OPT's previous decision by replacing the referenced block at the first opportunity.

This paper investigates the feasibility of this approach by analyzing the predictability of OPT last-touch references. This work makes four primary contributions:

First, we find that reference signatures are capable of identifying OPT last-touch references. Most significantly, signatures can effectively distinguish OPT last touches from references that are last touches under LRU but are *not* last touches under OPT. Separation of these two reference types is crucial to providing improvement over LRU replacement. Previous work [10, 11] indicated only that LRU last touches could be distinguished from LRU non-last-touch references.

Second, we propose a variety of signatures and investigate their effectiveness in correlating OPT last-touch references across a range of cache configurations. As in earlier work [10, 11], we find that trace-based signatures (combining several prior reference PCs and an address) are effective for small, direct-mapped, first-level caches. However, as cache size and associativity increase, and/or when the reference stream is filtered (as for a second-level cache), the accuracy of these signatures degrades significantly. We show that the accuracy of trace-based signatures can be enhanced by including "future" trace information, i.e., by retroactively identifying a last-touch reference based on subsequent reference information. This technique is valuable in our context, as much information subsequent to the reference in question is available in advance of the next replacement decision. We also introduce a more accurate class of signatures based on the historical pattern of last-touch vs. non-last-touch references to the same block, and show that these signatures correlate most strongly with last-touch references, particularly in large, highly associative, second-level caches.

Third, we identify and compare two methods of using OPT last-touch prediction to approximate OPT given a base LRU replacement algorithm. One approach, outlined above, identifies likely OPT last touches, and evicts the referenced blocks before they reach the LRU position. An alternate approach predicts LRU last touches that are *not* OPT last touches (which we refer to as *LRU non-OPT*, or *LNO*, last touches), and attempts to retain the referenced blocks past the point where they would normally be evicted by LRU. These two schemes correlate with two general strategies found in the literature: one focusing on *early eviction* of blocks unlikely to be re-referenced, the other focusing on *late retention* of blocks likely to be re-referenced. An LNO last-touch scheme is attractive because the number of LNO last touches is generally smaller than the number of OPT last touches, and ideally could be tracked with less state. However, we discover that LNO last touches are generally less predictable than OPT last touches. In addition, mispredictions in late-retention schemes (including LNO last-touch prediction) can be more costly than early-eviction mispredictions. The combination of these factors makes a scheme based on LNO last-touch prediction less effective, and implies that early-eviction schemes in general may be preferable to late-retention schemes.

Finally, we demonstrate that, under idealistic assumptions, OPT last-touch prediction can provide significant improvements in miss rates—from 5% to 55%, with an average of 39%, on a 1MB 16-way associative secondary cache—on several of the SPEC CPU2000 benchmarks. These positive results motivate the pursuit of a practical implementation. We leave the exploration of the implementation space for future work, but note that off-line, profile-driven identification of OPT last-touch signatures, coupled with previously proposed last-touch prediction structures [10, 11], is readily feasible and could come close to achieving the idealized results presented here.

We begin the rest of the paper by presenting related work. In Section 3, we discuss the early-eviction and late-retention strategies approximating OPT replacement, and how prediction of OPT

or LNO last-touch references can be used in these approaches. Section 4 addresses the predictability of OPT last-touch references, including trace-based analysis of a variety of signature types on a range of cache configurations. Section 4 also introduces two novel signature schemes, trace-based signatures with future information and last-touch-history signatures. We estimate the achievable miss-rate benefits of replacement algorithms driven by OPT last-touch prediction in Section 5, including a comparison of early-eviction and late-retention strategies. Section 6 presents our conclusions and future work.

2. Related work

This paper ties together two current research themes: the pursuit of improved cache replacement policies and last-touch prediction. We discuss related work in each of these areas in turn.

Puzak's thesis [17] explores the difference between LRU and OPT replacement policies for hardware caches, and served as an inspiration for our work. He found that most references that were OPT hits but LRU misses in an n-way associative cache were to the top $2n$ blocks in the LRU stack. To capture these hits, he proposed a *shadow directory*, which maintains an additional n address tags for each set of an n-way cache. These shadow-directory tags, in conjunction with the n conventional tags, track the $2n$ least-recently-used blocks mapped to that set. When a miss matches one of the shadow-directory tags, the referenced block is marked in the cache. The replacement policy keeps marked blocks in preference to unmarked blocks, even when the marked block is less recently used. In our terminology, this shadow-directory scheme is a late-retention approach, using the block address as a signature to identify (a subset of) LRU non-OPT last touches.

More recently, Wong and Baer [22] note, as we do, that the increasing capacity and associativity of on-chip level-two caches, coupled with the increasing performance impact of off-chip accesses, motivates replacement policies more sophisticated than LRU. They propose a scheme that, like Puzak's, marks blocks predicted to exhibit temporal locality, and preferentially retains these blocks over unmarked blocks regardless of their LRU stack position. A block's temporal locality is predicted based on the program counter (PC) of the most recent reference to the block. PC values are marked as temporal either statically through profiling (using off-line OPT) or dynamically based on on-line observation of whether blocks referenced from a particular PC are re-referenced before eviction. In our terminology, their scheme is also a late-retention scheme, using the reference PC as a signature to predict which references are *not* OPT last touches.

Several other cache researchers have proposed schemes for identifying blocks unlikely to exhibit temporal locality (called "non-temporal" blocks) based on reference PCs or block addresses. The non-temporal category includes long sequential reference streams and loops much larger than the cache, patterns that are well known to degrade LRU performance. These efforts focus on reducing conflicts in small, low-associativity level-one caches by segregating predicted non-temporal blocks into separate structures [18, 7] or having them bypass the cache completely [21], rather than on improving replacement in a large, highly associative level-two cache.

The pursuit of improved replacement algorithms has been more active in the area of virtual-memory systems and file and database buffer caches, likely due to the higher latencies of disk accesses, the flexibility of software algorithm implementations, and the larger gap between LRU and OPT due to full associativity. Several approaches seek to identify blocks (pages) without exploitable temporal locality for early eviction, paralleling the temporal/non-temporal work in hardware caches. Specific schemes identify non-temporal blocks in a variety of ways: e.g., by disregarding initial block references in the LRU algorithm [14, 8], or by recognizing sequential

address patterns [6]. Other algorithms attempt to measure a block's temporal locality along a more continuous spectrum, by considering its reference frequency [19, 13] or by detecting patterns in the inter-reference intervals of specific blocks [16].

Unlike other approaches, the EELRU algorithm [20] does not predict the temporal locality of specific pages. EELRU uses a scheme similar to Puzak's shadow directory to detect misses to recently evicted LRU pages. When a number of these references are detected, the algorithm evicts some pages early to allow others to remain longer, in the expectation that the retained pages will convert some of these miss references to hits. However, the selection of which pages to evict early is arbitrary.

Perhaps the most sophisticated algorithm proposed to date is Kim et al.'s UBM [9], which attempts to detect and characterize specific sequential and looping reference patterns—including the length and period of individual loops—and applies optimal replacement based on these identified patterns. This scheme uses aggregate loop size and period information to partition available space between looping and non-looping references, and can prioritize among loops of similar size based on their periods. We attempted to mimic UBM in a hardware cache and found that it is difficult to measure loop parameters accurately, and that the algorithm does not handle loops with non-constant periods (for example, see the reference plot for *art* in Figure 1).

Another class of techniques relies on application-level hints regarding future reference patterns [4, 15]. While these techniques are powerful, they require substantial programmer effort or sophisticated pre-execution support [5].

Last-touch prediction was first proposed by Lai and Falsafi [10] in the context of multiprocessor cache coherence protocols. Building on earlier work by Lebeck [12], they use predicted last touches to identify points at which a processor can proactively "self-invalidate" cached copies of shared data, avoiding the overhead of later coherence actions. Lai, Fide, and Falsafi [11] subsequently applied last-touch prediction under LRU replacement to uniprocessor caches. In this case, they used early eviction to free up space for prefetched blocks, reducing the potential cache pollution effect of the prefetches. Both of these earlier works used last-touch prediction only to predict which blocks would subsequently be evicted in the normal course of execution, gaining efficiency by moving that eviction earlier in time. However, as will be discussed in the following section, this style of last-touch prediction cannot directly improve miss rates.

3. Using last-touch information to approximate OPT replacement

In this section, we discuss the application of last-touch information to approximating OPT replacement. We describe and compare the early-eviction and late-retention approaches to approximating OPT, then discuss the applicability of last-touch information to these strategies.

Attempts to improve upon LRU generally fall into two categories. Some seek to identify blocks likely to be referenced shortly after they reach the LRU eviction point, and attempt to retain those blocks for an additional amount of time. Others seek to identify blocks that have not yet reached the LRU eviction point, but are unlikely to be referenced again before that time, and attempt to evict those blocks in advance. We label these strategies *late retention* and *early eviction*, respectively. Although all non-LRU algorithms evict some blocks earlier and retain others later than LRU, the difference in these strategies lies in which action is performed proactively. A late-retention strategy identifies potentially useful blocks, and gambles that the other blocks forced to leave the cache early are not even more useful. An early-eviction strategy identifies potentially useless blocks, and gambles that the other blocks that are allowed to remain in the cache longer are not equally useless.

A key practical effect of this distinction involves the miss-rate penalty for making an incorrect prediction and the potential benefit of a correct prediction. To retain a block marked as useful, a late-retention strategy must evict one unmarked block prematurely for every miss that occurs while the block is retained. If the marked block is not re-referenced (i.e., the prediction of utility was incorrect), then the strategy will incur an additional miss relative to LRU for every evicted block that would have been re-referenced prior to its LRU eviction point. Even if the prediction of utility was correct, there will be no gain relative to LRU if one evicted block would have been an LRU hit, and a loss if more than one. In contrast, an early-eviction strategy suffers at most one additional miss relative to LRU for every misprediction. This penalty may be erased if the block that is retained due to the early eviction converts an LRU miss into a hit. If the prediction of (lack of) utility was correct, then the strategy is guaranteed to do no worse than LRU, and has the potential to gain one hit. We will show in Section 5 that this effect contributes to making our last-touch-based late-retention algorithm less effective than a similar early-eviction algorithm.

We now turn our attention to approximating OPT replacement using last-touch information. We first show that OPT last touches are a strict subset of LRU last touches. As a result, we can partition all references into three disjoint subsets: those that are not last touches under either policy, those that are last touches under both policies, and LRU non-OPT (LNO) last touches. We then discuss the use of OPT and LNO last-touch information in approximating OPT replacement.

To show that OPT last touches are a strict subset of LRU last touches, we first note that a reference can never be a hit under LRU but a miss under OPT [17]. By definition, an LRU hit always references a block at some depth d in the LRU stack, where $d < n$, the associativity of the cache. Thus fewer than n unique blocks were referenced since the previous reference to the same block. Therefore, at the time of this previous reference, fewer than n other blocks had next reference times earlier than the block in question. Thus OPT would not have replaced the block after its previous reference, and the current reference must also be a hit under OPT.

Now consider an OPT last-touch reference. By definition, the next reference to the same block is a miss under OPT. As this next reference cannot be an OPT miss but an LRU hit, the next reference must also be a miss under LRU. Therefore the current OPT last-touch reference must also be a last-touch reference under LRU. Thus OPT last touches are a subset of LRU last touches.

Of course, the converse does not hold, and it is possible for an LRU last touch to not be an OPT last touch. In fact, each of these LRU non-OPT (LNO) last touches implies a later reference that is an OPT hit but an LRU miss; thus the number of LNO last touches is equal to the number of additional misses incurred by LRU over OPT.

Given complete and accurate last-touch information, we can implement OPT replacement using either a late-retention or an early-eviction approach. We assume a base LRU policy, which automatically distinguishes LRU last touches (OPT+LNO) from non-last-touch references—all we have to do is wait and see if LRU replaces the block. Thus given information identifying either LNO or OPT last touches, we can infer the other set. The late-retention strategy requires knowledge of LNO last touches; we retain blocks referenced by LNO last touches until they are re-referenced, regardless of their LRU stack depth. Given perfectly accurate and complete LNO last-touch information, all of the blocks evicted early as a result must have been referenced by OPT last touches (i.e., they must be dead under OPT). Conversely, the early-eviction strategy requires knowledge of OPT last touches; we simply evict a block immediately after each OPT last touch. Again, given perfectly accurate and complete information, all of the blocks retained past the LRU eviction point in this fashion must have been referenced by LNO last touches.

In the abstract, prediction of all LRU (OPT+LNO) last touches, as was the goal of previous last-touch prediction work [10, 11], is not sufficient to derive improved replacement behavior. Blocks subject to LRU last touches can be evicted early, freeing up space in the cache. However, given perfect prediction, these evictions will not affect the miss rate, as any blocks that could have used that space to stay past their LRU eviction point (thus converting an LRU miss to a hit) have themselves already been evicted. Any actual miss-rate improvement would be the result of a predictor inaccuracy that failed to identify an LNO last touch as an LRU last touch.

Thus the first question that must be answered is whether last-touch prediction mechanisms, having previously shown to distinguish LRU last touches from non-last-touch references, are also capable of distinguishing OPT last touches from both non-last-touch and LNO last touches. We address this question in the following section.

4. Analysis of last-touch predictability

Determining the fundamental predictability of OPT last touches is the first step in analyzing the utility of OPT last-touch prediction for improving replacement. After describing our methodology in Section 4.1, we then examine the effectiveness of a variety of trace-based signatures—i.e., those using reference PCs and/or addresses—for distinguishing OPT last touches. Signatures of this type have been proposed previously for last-touch prediction [10] and for enhancing replacement decisions [17, 22]. We show that these trace-based signatures can be improved by including future trace information. Section 4.3 introduces a more accurate class of signatures based on historical last-touch patterns. We end this section with an analysis of the relative predictability of OPT vs. LNO last touches.

4.1. Methodology

We examined last-touch predictability by analyzing reference traces from a subset of the SPEC CPU2000 benchmarks. Because we are interested in exploring fundamental properties of application reference patterns, the overhead of timing simulation was not justified. We used the SimpleScalar toolset [3] to collect reference traces from optimized Compaq Alpha binaries of SPEC CPU2000 applications. We skipped the initialization portion of each benchmark, then collected two 50-million-reference traces: one of references to the L1 data cache (i.e., all program data accesses), and one of references to the unified secondary (L2) cache (i.e., 50 million L1 cache misses) using 64KB 4-way associative L1 data and instruction caches. Throughout the paper, we use a 64-byte block size, 64KB L1 caches, and 1MB L2 caches, unless stated otherwise. For this study, we selected the subset of SPEC CPU2000 benchmarks for which LRU replacement exhibits more than 15% additional misses over OPT in a 1MB, 4-way set-associative L2 cache. The specific benchmarks, and the number of L2 misses incurred by OPT and LRU on our 50-million-reference trace, are shown in Table 1.[1]

To analyze the effectiveness of a particular signature type on a particular cache configuration, we simulate the cache under both LRU and OPT replacement. As each reference is processed, we calculate its signature value. When the simulation progresses to the point where we can determine whether or not a particular reference was a last touch under LRU or OPT, we look up the signature value in a hash table and increment an associated counter accordingly. At the end of the simula-

1. In addition to the benchmarks listed, *vpr* shows a 50% miss-rate difference between OPT and LRU. However, we encountered simulation problems with this benchmark, so we were forced to exclude it at present. We hope to resolve these problems and include *vpr* in our analysis in the future.

	Benchmark	Description	OPT misses	LRU misses	LRU/ OPT
Floating Point	ammp	Computational Chemistry	8965877	13289270	1.48
	art	Image Recognition / Neural Networks	10819942	26024370	2.41
	galgel	Fortran 90 Computational Fluid Dynamics	3978119	11796704	2.97
	sixtrack	Fortran 77 High Energy Nuclear Physics Accelerator Design	16282407	35858293	2.20
Integer	bzip	Compression	5919139	9166266	1.55
	mcf	Combinatorial Optimization	25143734	49742535	1.98
	twolf	Place and Route Simulator	4020636	7034244	1.75
	vortex	Object-oriented Database	5343935	7310515	1.37

Table 1: Benchmark characteristics

tion, the hash table indicates, for each signature value that occurred, the number of references of each type (OPT last touch, LNO last touch, or non-last-touch) for that signature. Because we are focusing on the fundamental efficacy of various signature types, we do not limit the amount of signature storage.

4.2. Trace-based signatures

We focus first on trace-based signatures, incorporating program counter and/or address values, as these have been shown in previous work [10, 11] to be effective for predicting LRU last-touches in small, direct-mapped L1 caches. Other work on predicting reference behavior for improving replacement has used PCs or addresses—degenerate forms of trace-based signatures—to correlate with predicted reference behavior [17, 22].

Table 2 lists the various trace-based signature types we studied. We assume that a reference PC value has half the storage requirement of a reference address, as the number of memory reference instructions in a program is generally far less than the number of data locations (particularly for applications that stress a large cache). We found that incorporating more than two reference addresses in a signature is impractical, as the total number of signature values quickly becomes unmanageable, even on a machine with more than 1GB of memory. In addition, as the number of unique signature values increases, we must also increase trace lengths (and thus simulation time) to observe a significant number of references per signature. We thus restrict ourselves to signatures with storage cost no more than 2.5 times a singe address, as shown in the rightmost column of Table 2. In our naming scheme, Puzak [17] used 1Addr and Lai and Falsafi [10] use 1Addr3APC.

We analyze the efficacy of a particular signature by examining the trade-off between accuracy and coverage provided by that signature type. The *accuracy* of a particular signature value is the fraction of all references associated with that signature value that are of the type of interest. We define the *coverage* of a signature type at a particular threshold accuracy as the fraction of references of the type of interest that are associated with signature values whose accuracy is equal to or greater than the threshold value. That is, if we choose the set of signature values whose accuracy is at or above the threshold to identify a particular type of reference—for example, OPT last touches—what fraction of those references will we identify?

We begin by examining last-touch predictability in a 64KB, direct-mapped L1 cache, to correlate our results with those of previous work [11]. We assume cache bypassing is not allowed, i.e., referenced blocks must be loaded into the cache; thus, OPT and LRU policies are indistinguishable. We also exclude signatures such as 2SAddr and 1Addr2SPC that use information from mul-

Signature	Elements	Estimated Relative Cost
1Addr	address of current ref	1
2Addr	address of current ref, address of previous ref (any address)	2
2SAddr	address of current ref, address of previous ref to the same cache set	2
1Addr1PC	address + PC of current ref	1.5
1Addr2PC	address + PC of current ref, PC of previous ref (any address)	2
1Addr2APC	address + PC of current ref, PC of previous ref to same address	2
1Addr2SPC	address + PC of current ref, PC of previous ref to same cache set	2
1Addr3PC	address + PC of current ref, PCs of 2 previous refs (any address)	2.5
1Addr3APC	address + PC of current ref, PCs of 2 previous refs to same address	2.5
1Addr3SPC	address + PC of current ref, PCs of 2 previous refs to same cache set	2.5
1PC2Addr	address + PC of current ref, address of previous ref (any address, any PC)	2.5
1PC2PAddr	address + PC of current ref, address of previous ref from same PC (any address)	2.5
1PC2SAddr	address + PC of current ref, address of previous ref to same cache set	2.5

Table 2: Trace-based signatures

tiple references to the same cache set: for direct-mapped caches, seeing two references to different blocks in the same cache set guarantees that the first reference is a last touch.

Figure 2 shows the full accuracy/coverage trade-off curves for a 64KB, direct-mapped L1 cache. These curves plot the coverage of each signature type (on the vertical axis) for a given accuracy threshold (on the horizontal axis). Due to space constraints, we show only a representative subset of our benchmarks at this level of detail. In general, higher curves imply better predictability. A "perfect" prediction results in a straight flat line at the y-value of 100%, as seen for several signature types in *art*.

We can draw a number of conclusions from Figure 2. Overall, using only addresses is less accurate than using PC-address combinations, as indicated in earlier work [10, 11]. However, once the PC and address of the reference are included, incorporating an additional address is slightly more useful than incorporating one or even two additional PCs. The 1PC2PAddr signature provides the best predictability in some benchmarks (such as *twolf*), while the 1PC2Addr signature performs best for the others (not shown). As in [11], we see very good predictability for *ammp*, *art*, and *mcf*. However, predictability varies significantly across benchmarks, and is much worse for some of the benchmarks, such as *twolf*, not analyzed in [11]. If we choose the best signature for each benchmark, with accuracy threshold of 80%, coverage varies from 15% to 100%. In general, floating-point benchmarks appear to have higher predictability than integer benchmarks, with the exception of *mcf* and *galgel* (as will be shown in Figure 3).

Because we focus on improving replacement in large, highly associative, secondary caches, we are interested in how last-touch predictability changes in this different environment. Figure 3 shows the predictability of OPT last touches for the 1PC2PAddr signature, on average the best performer in a 64KB direct-mapped L1, as we increase associativity and size and switch to the filtered reference stream seen by an L2 cache. To show these results compactly for all benchmarks, we sample the accuracy/coverage curves of Figure 2 at 100%, 80%, 50% and 30% and display the results for each configuration in a stacked bar. We omit bzip here because the 1PC2Paddr signature generates too many unique values, making simulation impractical.

Significantly, we found that the predictability of OPT last touches in associative caches is only slightly worse than that of LRU last touches, and follows very similar trends across signature

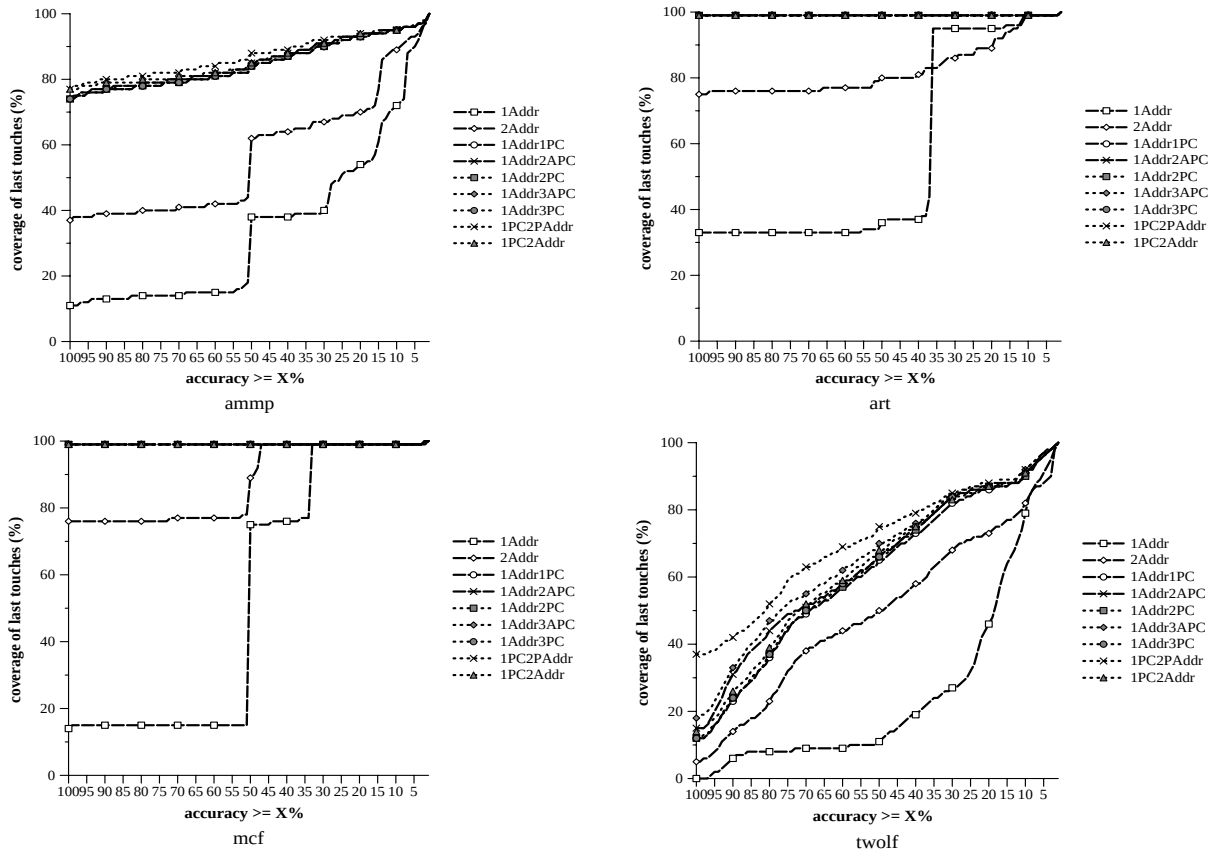

Figure 2. Last-touch predictability of trace-based signatures for direct-mapped L1 cache

types and cache configurations. This result is important for the feasibility of last-touch-based replacement. To conserve space, we show only the OPT results.

The first three bars for each benchmark in Figure 3 show the effect of increasing associativity to four-way, then increasing size to 1MB. For four of the seven benchmarks shown, increases in cache size and associativity both degrade predictability significantly. The same trend is evident in *twolf*, but is less pronounced. *Vortex*'s predictability improves slightly at higher associativity, but degrades with a larger cache. *Galgel* degrades with associativity, but sees a large boost in predictability in the larger cache. As the cache size increases from 64K to 1M, *galgel*'s miss rate improves dramatically, eliminating over 85% of the OPT last touches. Most of these belong to low-accuracy signatures. In addition, the number of unique signatures increases. As a result, the predictability improves significantly.

Comparing the third and fourth bars in each plot shows the effect of L1 filtering on predictability. The effects are mixed; four benchmarks see a significant decrease in predictability, while two (*art* and *vortex*) are basically unchanged, and *ammp* sees a large increase. Increasing the L2 associativity from 4-way to 16-way (the rightmost bar) tends to hurt predictability further, though again *galgel* is a notable exception (as is *ammp*, to a lesser extent).

Nevertheless, the cumulative effect of increasing size and associativity and introducing L1 filtering is a marked decrease in predictability on every benchmark except *galgel*. If we limit ourselves to signatures with accuracy of at least 80%, we cannot cover more than 50% of OPT last touches on any benchmark in a 1M 4-way L2 cache; coverage is less than 5% in all but two of the

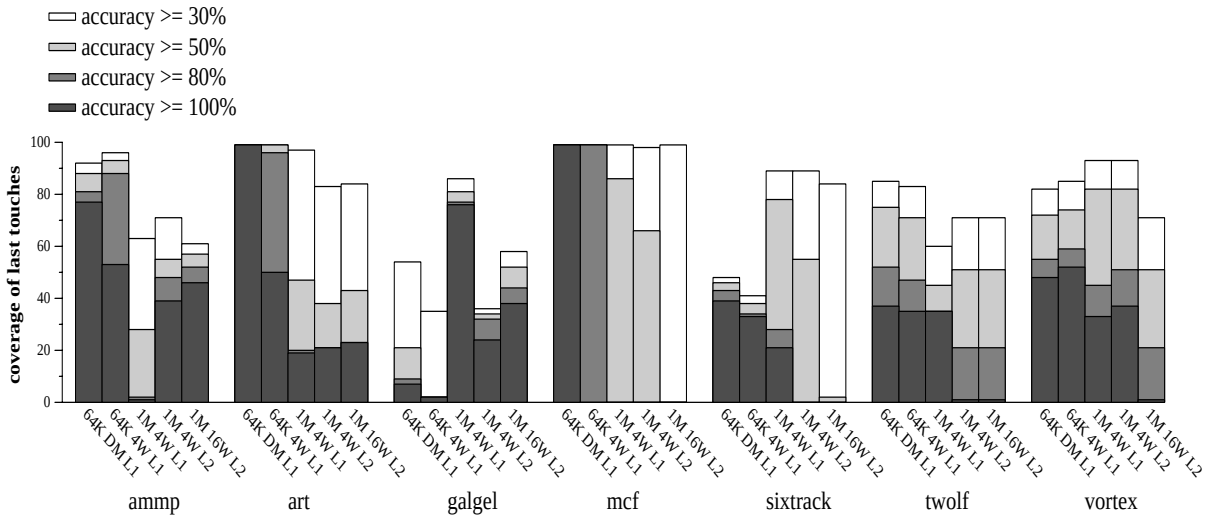

Figure 3. Effects of associativity, cache size, and L1 filtering on predictability (1PC2PAddr)

benchmarks. Unfortunately, trace-based signatures do not provide the desired level of predictability in large set-associative caches.

We discovered that the accuracy of trace-based signatures can be enhanced by the inclusion of "future" information, that is, information regarding program references that occur after the reference being predicted as a last touch. In contrast, all of the signature types listed in Table 2 include only information from the current reference and prior references. Although using this future information delays the last-touch prediction, the prediction is not needed until the next replacement in our context. As long as the future information that we incorporate is available before then, we can usefully apply it to our last-touch prediction.

We re-examined all of the signature types listed in Table 2 that incorporate prior information, modifying them to use future information instead. For example, we altered the 1PC2Addr signature to use the address and PC of the current reference plus the address of the *subsequent* reference, generating a new signature type we called 1PC2AddrF. In general, modifying a signature in this fashion improves predictability (i.e., increases coverage at a given accuracy threshold) by 3% to 10% on most benchmarks. (Detailed results are omitted due to space restrictions.) After examining this new set of signatures, we identified 1PC2SAddrF—comprising the address and PC of current reference, plus the address of the subsequent reference to the same cache set—as the best overall trace-based signature type. The results for 1PC2SAddrF are shown in the leftmost bar of Figure 4.

4.3. Last-touch-history signatures

Even with the addition of future information, we are still unsatisfied with the predictability of trace-based signatures is still unimpressive. Apparently, short sequences of PC and address information are inadequate to reliably locate a reference within the repetitive access patterns that we see in Figure 1. In this section, we introduce a new class of signatures that attempt to exploit this repetitive behavior directly to identify OPT last touches. That is, if these repetitive access patterns do lead to repetitive replacement behavior—the crux of our motivation to study this topic—then we should be able to use the past patterns of last-touch references to predict future last touches.

Specifically, we use a bit vector, called *last-touch-history* (LTH) vector, to record the last-touch history for each unique cache block address. For each reference to a block, a '1' or '0' bit is

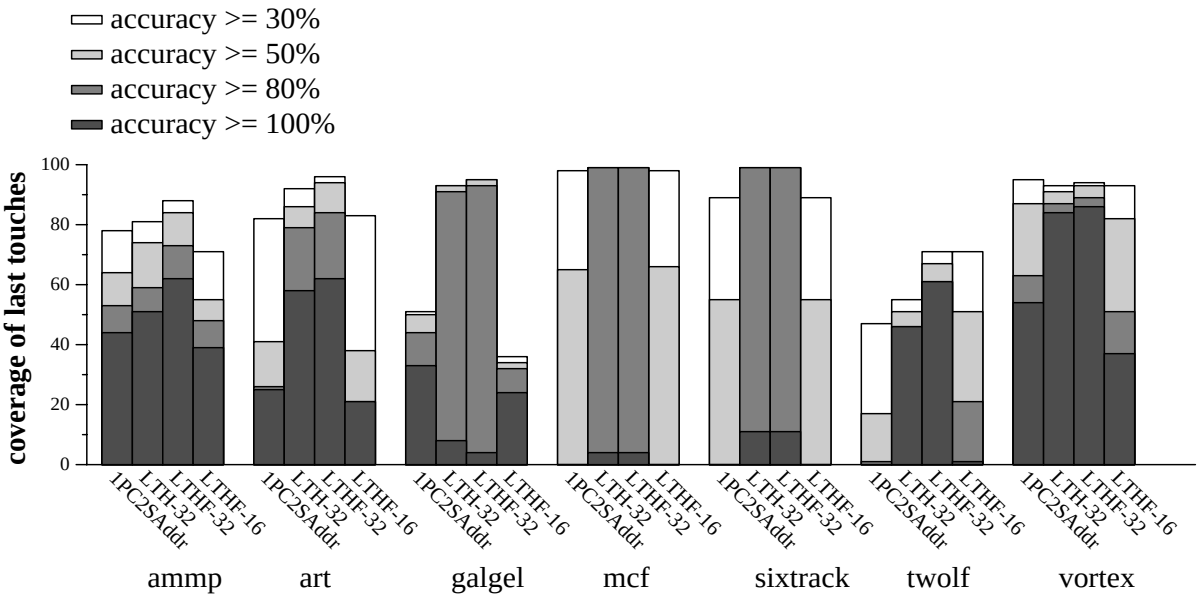

Figure 4. Last-touch-history signatures vs trace-based signatures

shifted into the block's LTH vector, indicating whether the reference was a last touch. Note that the determination of whether a particular reference is a last touch cannot be made until either the block is either replaced or re-referenced, so the last-touch bit for a particular reference is not shifted in until well after the reference occurs. However, at the time that a reference occurs, the LTH bit for the block's *previous* reference is always available. The LTH vector value for a block's previous references, through the immediately preceding reference, is then used as the signature value for predicting whether the current reference is a last touch.

The predictability of last-touch-history signatures is a function of the LTH vector length. To capture the last-touch behavior in one iteration of a repetitive reference pattern, the LTH vector must have as many bits as there are references to each cache block within an iteration. However, we have found empirically that the number of references to a cache block in an iteration often exceeds 32 or even 64. One technique to reduce the number of bits consumed by a given reference pattern is to filter out MRU hits before recording bits in the LTH vector. A significant fraction of references are MRU hits, and the first n–1 references in a sequence of n consecutive MRU hits are certainly not last touches. If we predict whether a reference was a last touch only after a block leaves MRU position, we need to record the last-touch history for only the last reference in a sequence of MRU hits. Ideally, we do not lose any last-touch information, while a lot of redundant data—non-last-touch MRU hits—is filtered out of the LTH vector.

Figure 4 shows results for 32-bit LTH signatures with and without MRU-hit filtering (LTH-32 and LTHF-32, respectively) and 16-bit LTH signatures with MRU-hit filtering (LTHF-16) on a 1MB 4-way associative L2 cache. For comparison, the first bar shows the coverage for the best trace-based signature using future information (1PC2SAddrF). LTH-32 outperforms 1PC2SAddrF for all benchmarks. Even in *galgel*, where the coverage at 100% accuracy declines relative to 1PC2SAddrF, the coverage at 80% accuracy is nearly twice as great. (We will show in Section 5 that 80% accuracy is more than adequate to achieve real gains in replacement.) In fact, the LTH predictor provides higher accuracy and coverage while generating fewer unique signatures than 1PC2SAddrF. The MRU-hit filtering of LTHF-32 further improves predictability, particularly on *ammp* and *twolf*. The LTHF-16 results show that, even with MRU-hit filtering, 16 bits

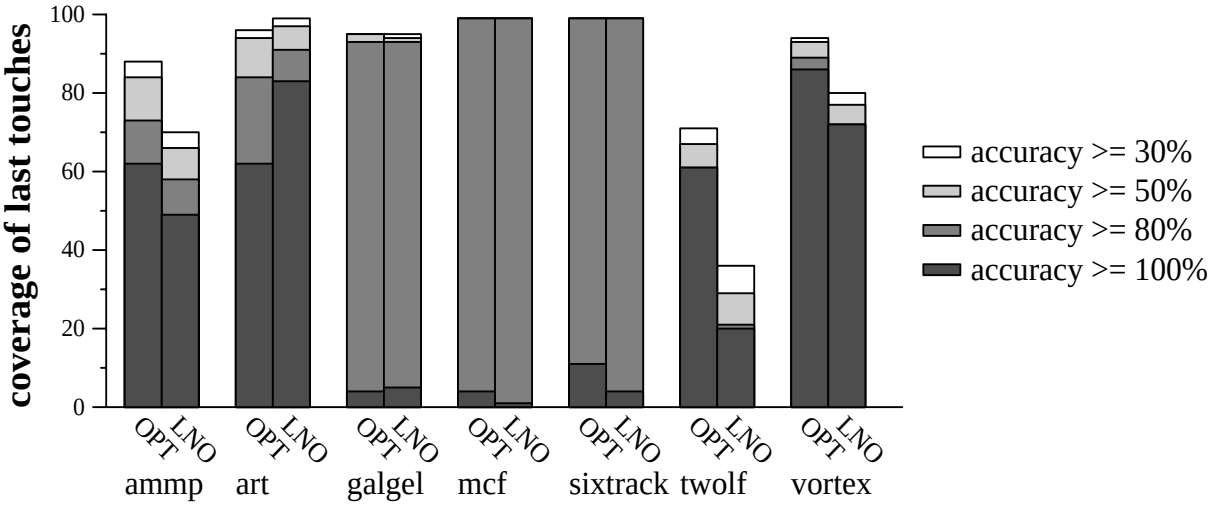

Figure 5. OPT vs. LNO last touches

are not enough to accommodate sufficient last-touch history. LTHF-16 is less effective than 1PC2SAddrF in most cases. We use LTHF-32 signatures for the rest of the paper.

4.4. Comparing the predictability of OPT and LNO last touches

The results presented so far have indicated the predictability of OPT last touches. As discussed in Section 3, prediction of either OPT or LRU non-OPT (LNO) last touches can be used to approximate OPT replacement. As mentioned above, we examined the predictability of LNO and LRU last touches for all the signatures discussed previously, and found similar predictability trends; i.e., 1PC2SAddrF and LTHF-32 are the best overall trace-based and history-based signatures, respectively, for predicting any of these categories of last touches. Nevertheless, an interesting question remains: which class of last touches, OPT or LNO, is more predictable? The answer is worth pursuing because, as discussed in Section 3, it partially determines the relative effectiveness of early-eviction vs. late-retention approaches to approximating OPT.

Figure 5 compares the predictability of OPT and LNO last touches using LTHF-32 signatures, again on a 1MB 4-way associative L2 cache. Most benchmarks exhibit higher predictability of OPT last touches, with *art* being a notable exception. In particular, the less regular integer benchmarks (*twolf* and *vortex*) show significantly higher predictability for OPT. From these results, we might guess that early-eviction approaches are more effective than late-retention approaches. However, the evidence is not conclusive. In the next section, we measure the miss-rate reduction achievable under idealized circumstances for early-eviction and late-retention replacement policies, using OPT and LNO last-touch prediction respectively.

5. Estimating potential miss-rate improvement

The preceding analysis of OPT last-touch predictability indicates potential for improving replacement using last-touch prediction. With LTHF-32 signatures, even in the worst case (*twolf*), about 70% of the OPT last touches can be captured by signatures with accuracy higher than 80%. To provide a more concrete indication of the possible miss-rate improvement achievable through this approach, we simulated the performance of two last-touch-driven replacement policies on our traces. Although this study is still idealized—it assumes we have full knowledge of the last-touch/non-last-touch bias of every signature value, and unlimited storage for signatures—it does repre-

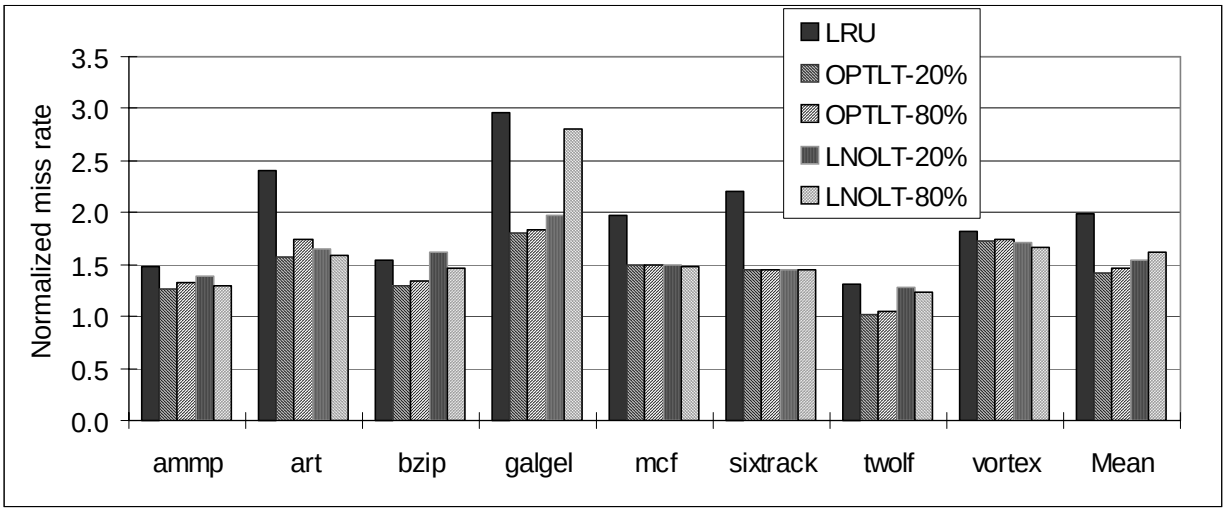

Figure 6. OPTLT vs. LNOLT replacements, 4-way 1MB L2 cache

sent a feasible on-line replacement algorithm, given the ability to identify signature biases via profiling. Other practical issues, such as the applicability of signature information collected on a small profiling run to varying input data, and the effect of finite signature storage, are left for future work.

We propose two replacement algorithms, OPTLT and LNOLT, based on prediction of OPT and LNO last touches, respectively. Both algorithms replace only non-MRU blocks to compensate for the MRU-hit filtering of the LTHF-32 signature. OPTLT is an early-eviction algorithm: on a miss, it searches for the first non-MRU block whose last reference was predicted to be an OPT last touch. The search is performed in MRU order, to quickly evict the block with the most recent OPT last touch. LNOLT uses late retention: on a miss, it will *not* evict a block whose last reference was predicted to be an LNO last touch if there are any non-MRU blocks whose last reference was predicted not to be an LNO last touch. Both algorithms fall back on LRU if there are no non-MRU blocks with predicted last-touch references (or, in the case of LNOLT, if all non-MRU blocks were predicted to have last-touch references).

For both algorithms, the last-touch predictor is simply a static table of signature values; any reference whose signature is found in the table is predicted to be a last touch. The table is generated by making an initial pass through the trace, as in the Section 4, and recording all signature values with accuracy above a predetermined threshold. We ran each algorithm with a variety of thresholds; the impact of the threshold value on algorithm performance is discussed further below.

Figure 6 shows miss rates for LRU, OPTLT, and LNOLT, normalized to OPT, for a 1MB 4-way associative L2 cache. For the proposed algorithms, results are shown for signature thresholds of 20% and 80%. LRU's relative miss rate varies from 1.36 to 2.96 with an average of 1.96. OPTLT with a threshold of 20% is the best performer on average, and the best or close to the best performance on each individual benchmark as well. Overall, it eliminates nearly 30% of LRU misses, closing 58% of the gap between LRU and OPT. OPTLT with an 80% threshold is nearly as good; *art* is the only benchmark that exhibits a noticeable degradation at the higher threshold. LNOLT also improves miss rates significantly, but is slightly inferior to OPTLT overall. The choice of threshold for LNOLT has a dramatic effect on *galgel*, but is not significant otherwise.

Figure 7 shows the equivalent results for a 16-way cache. The higher associativity increases the LRU/OPT ratio to 2.67. Again, OPTLT at 20% threshold is the best performer, now by a wider

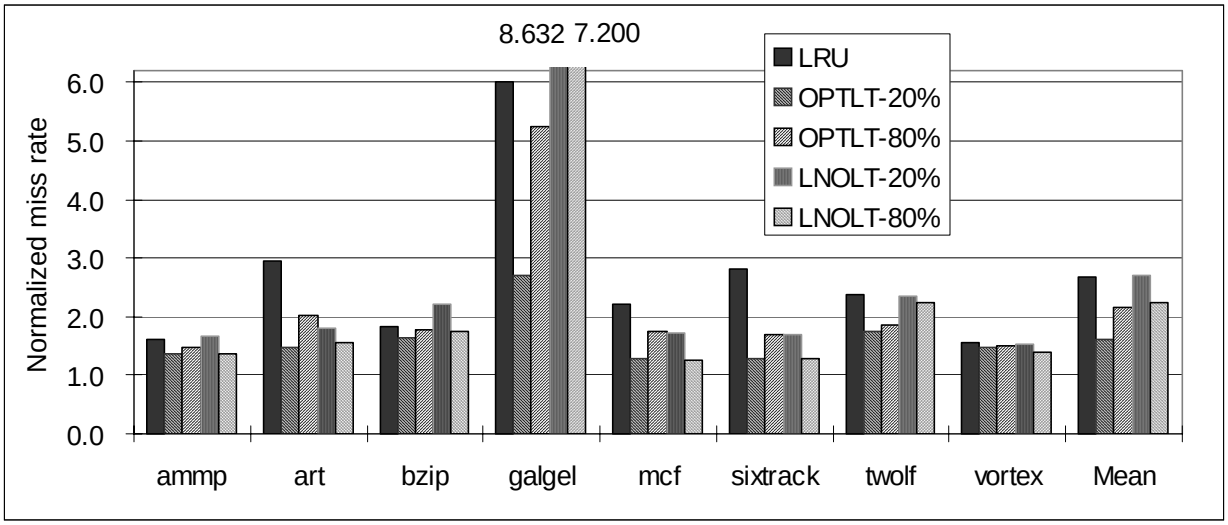

Figure 7. OPTLT vs. LNOLT replacements, 16-way 1MB L2 cache

margin. It eliminates 39% of LRU misses and closes 63% of the OPT-LRU gap. Although LNOLT at 80% threshold is competitive with OPTLT on many benchmarks, it performs very poorly on *galgel*, actually increasing misses over LRU by 20%. LNOLT-20% also performs worse than LRU on *bzip*. LNOLT underperforms OPTLT on *twolf*, regardless of threshold.

These results indicate that OPTLT is more effective than LNOLT, especially when the associativity increases. The reason is two-fold. First, as we have showed, OPT last touches have better predictability than LNO last touches. Second, as discussed in Section 3, the misprediction penalty of LNOLT is higher than that of OPTLT. Apparently the latter factor is dominant; note that *galgel*, the worst case for LNOLT, actually exhibits slightly higher predictability for LNO over OPT last touches in Figure 5.

Although we show results for only 20% and 80% threshold values, we simulated the full range of thresholds from 20% to 100% in 20% increments. For OPTLT, most benchmarks have an optimal (minimum miss-rate) threshold of 20% (or lower); the exception, *vortex*, reaches the minimum at 40%. However, for LNOLT, the optimal threshold always resides at 80% (or higher) for the 16-way cache, and for most benchmarks on the 4-way cache as well. These results indicate that LNOLT's high misprediction penalty is a significant factor, and thus LNO last-touch prediction accuracy must be higher than OPT last-touch accuracy to provide similar benefits. In addition, LNOLT is much more susceptible then OPTLT to degrading replacement performance below that of LRU. We believe these results imply that early-eviction strategies are in general preferable to late-retention strategies for improving replacement beyond LRU.

6. Conclusions and future work

We have shown that repetitive reference patterns can be exploited to improve replacement by tracking what optimal replacement (OPT) would have done at specific program points, and replaying these replacement decisions when the corresponding situation arises again. Last-touch prediction provides a useful framework for recording OPT decisions and identifying where and when they should be replayed.

Previous last-touch predictors focused on predicting LRU last touches; however, to improve replacement, predictors must distinguish OPT from LRU non-OPT last touches. We have shown that last-touch predictors are quite capable of making this distinction. Unfortunately, previously

proposed trace-based predictors, which perform well on small, direct-mapped, primary caches, do not fare as well on large, highly associative, secondary caches, even when extended to include future reference information. We introduced a new class of last-touch predictors based on last-touch history vectors that significantly outperform trace-based predictors, and provide reasonable accuracy and coverage even on large secondary caches.

We also identified and contrasted two general approaches to improving LRU replacement: early eviction and late retention. We proposed and evaluated two replacement algorithms based on last-touch prediction, one using early eviction and one late retention. Under idealized conditions, on a 16-way associative 1MB L2 cache, the early-eviction algorithm eliminates 39% of LRU misses and closes 63% of the gap between LRU and OPT, on average across 8 SPEC CPU2000 benchmarks. Our results show that, although the late-retention algorithm is competitive with early eviction on some benchmarks, its higher misprediction penalty can produce dramatic worst-case performance scenarios in which the algorithm underperforms LRU by a significant margin.

We have shown that last-touch prediction has significant potential for improving replacement. However, many implementation issues need to be addressed for last-touch prediction to become an effective approach in practice. Our future work includes practical signature classification using both profile-based and approximate on-line algorithms, as well as efficient hardware schemes for storing and managing signature/prediction information. Although this paper focused on replacement in hardware caches, we believe the concepts presented here are likely to be applicable to virtual memory and file-system buffer management as well.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. 9734026 and 0105503. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Advanced Micro Devices, Inc. AMD Athlon processor and AMD Duron processor with full-speed on-die L2 cache. White Paper, June 2000. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/cache_wp.pdf.

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

[3] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: the SimpleScalar tool set. Technical Report 1308, Computer Sciences Department, University of Wisconsin–Madison, July 1996.

[4] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. In *First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–177, November 1994.

[5] F. Chang and G. Gibson. Automatic I/O hint generation through speculative execution. In *Third USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, February 1999.

[6] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–126, May 1997.

[7] Teresa L. Johnson and Wen-mei W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 315–326, June 1997.

[8] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Databases*, pages 439–450, 1994.

[9] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. A low-overhead, high-performance unified buffer management scheme that exploits sequential and looping references. In *Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 119–134, October 2000.

[10] An-Chow Lai and Babak 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, June 2000.

[11] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 144–154, June 2001.

[12] Alvin R. Lebeck and David 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, June 1995.

[13] Donghee Lee, Jongmoo Choi, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the lru and lfu policies. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–143, May 1999.

[14] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD Conference*, pages 296–306, May 1993.

[15] R. H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles (SOSP)*, pages 79–95, December 1995.

[16] Vidyadhar Phalke and Bhaskarpillai Gopinath. An inter-reference gap model for temporal locality in program behavior. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 291–300, May 1995.

[17] Thomas R. Puzak. *Analysis of Cache Replacement Algorithms*. PhD thesis, Univ. of Massachusetts, February 1985.

[18] Jude A. Rivers and Edward S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *Proceedings of the 1996 International Conference on Parallel Processing (Vol. I)*, pages I–151–162, August 1996.

[19] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–142, May 1990.

[20] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: Simple and effective adaptive page replacement. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 122–133, May 1999.

[21] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A modified approach to data cache management. In *28th Annual International Symposium on Microarchitecture*, pages 93–103, December 1995.

[22] Wayne A. Wong and Jean-Loup Baer. Modified LRU policies for improving second-level cache behavior. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 49–60, January 2000.