Journal of Instruction-Level Parallelism 11 (2009) 1-24 Submitted 10/08; published 4/09

Enhancing LTP-Driven Cache Management Using Reuse Distance Information

Wanli Liu WANLI@UMD.EDU
Donald Yeung YEUNG@ENG.UMD.EDU
Department of Electrical and Computer Engineering
University of Maryland at College Park
College Park, MD 20742 USA

Abstract

Traditional caches employ the LRU management policy to drive replacement decisions. However, previous studies have shown LRU can perform significantly worse than the theoretical optimum, OPT [1]. To better match OPT, it is necessary to aggressively anticipate the future memory references performed in the cache. Recently, several researchers have tried to approximate OPT management by predicting *last touch references* [2, 3, 4, 5]. Existing last touch predictors (LTPs) either correlate last touch references with *execution signatures*, like instruction traces [3, 4] or last touch history [5], or they predict *cache block life times* based on reference [2] or cycle [6] counts. On a predicted last touch, the referenced cache block is marked for early eviction. This permits cache blocks lower in the LRU stack–but with shorter reuse distances–to remain in cache longer, resulting in additional cache hits.

This paper investigates three mechanisms to improve LTP-driven cache management. First, we propose exploiting reuse distance information to increase LTP accuracy. Specifically, we correlate a memory reference's last touch outcome with its *global reuse distance history*. Second, for LTPs, we also advocate selecting the most-recently-used LRU last touch block for eviction. We find an MRU victim selection policy evicts fewer LNO last touches [5] and mispredicted LRU last touches. Our results show that for an 8-way 1 MB L2 cache, a 54 KB RD-LTP which combines both mechanisms reduces the cache miss rate by 12.6% and 15.8% compared to LvP and AIP [2], two state-of-the-art last touch predictors, and by 9.3% compared to DIP [7], a recent insertion policy. Finally, we also propose predicting actual reuse distance values using *reuse distance predictors* (RDPs). An RDP is very similar to an RD-LTP except its predictor table stores exact reuse distance values instead of last touch outcomes. Because RDPs predict reuse distances, we can distinguish between LNO and OPT last touches more accurately. Our results show an 64 KB RDP can improve the miss rate compared to an RD-LTP by an additional 2.7%.

1. Introduction

The performance of the cache memory hierarchy is critical to the overall performance of modern computer systems. In particular, the policies used to manage the contents of caches can have a major impact on hit rates, and hence, memory hierarchy effectiveness. Traditional caches employ the LRU policy to drive replacement decisions. However, previous studies have shown LRU can perform significantly worse than the theoretical optimum, OPT [1], especially for large and highly associative caches commonly found at the L2 level [5, 8]. These studies suggest an opportunity exists for more sophisticated replacement algorithms to provide higher performance.

To improve upon LRU and better match OPT, it is necessary to aggressively anticipate the future memory references performed in the cache. In the case of OPT, perfect knowledge about the *reuse distance* of memory references is available to the replacement algorithm, allowing it to always evict the block used furthest in the future. Recently, several researchers have tried to approximate such omniscient OPT management by predicting *last touch references* [2, 3, 4, 5]. On a predicted last touch, the referenced cache block is marked for early eviction since it is unlikely to be re-referenced prior to becoming the LRU block. This permits cache blocks lower in the LRU stack–but with shorter reuse distances–to remain in cache longer, resulting in additional cache hits.

At the heart of such sophisticated replacement algorithms are the last touch predictors (LTPs) used to predict last touch references and drive replacement decisions. To date, two major approaches have been considered for LTPs. The first approach correlates last touch references with *execution signatures*. Signature types that have shown the greatest promise include instruction traces [3, 4] and last touch history [5]. The second approach identifies last touches by predicting *cache block life times* based on either reference [2] or cycle [6] counts. In this approach, a last touch is assumed whenever the predicted life time of a block in the cache expires.

This paper investigates several novel mechanisms for improving LTP-driven cache management. First, we propose exploiting reuse distance information to increase LTP accuracy. Like last touches, reuse distances associated with individual memory references also exhibit repeating patterns which can be captured by a hardware predictor. Specifically, we correlate a memory reference's last touch outcome with its *global reuse distance history* and the memory instruction's PC, and store the correlation in a hardware table. We call a hardware structure with both these mechanisms a *reuse distance last touch predictor* (RD-LTP). To determine reuse distances, RD-LTPs observe a cache block's position in the LRU stack at reference time. Consequently, RD-LTPs can track reuse distances for blocks that remain in the cache. By augmenting the cache with *shadow tags* [9], RD-LTPs can also monitor the reuse distances of recently evicted cache blocks. This enables RD-LTPs to track LRU last touches even when cache management deviates from a true LRU policy due to early evictions. For RD-LTPs, we also advocate selecting the most-recently-used LRU last touch block for eviction. We find an MRU victim selection policy often picks the best block to evict.

Our results show that for an 8-way 1 MB L2 cache, a 54 KB RD-LTP can reduce the cache miss rate by 12.6% and 15.8% compared to LvP and AIP [2], two state-of-the-art last touch predictors, and by 9.3% compared to DIP [7], a recent insertion policy. These performance gains are achieved for two reasons. First, RD-LTPs exhibit a much higher prediction rate, predicting 71.2% of the LRU last touches compared to only about 19% for LvP and AIP. Second, we find the MRU victim selection policy avoids evicting LNO last touches [5] (*i.e.* the evictions that are last touches under LRU but not under OPT) as well as mispredicted LRU last touches, thus increasing the proportion of OPT last touches that are evicted.

Finally, we also investigate techniques to further improve how we distinguish between LNO and OPT last touches, thus enabling even higher quality early eviction decisions. Specifically, we propose predicting actual reuse distance values using *reuse distance predictors* (RDPs). An RDP is identical to an RD-LTP except its predictor table stores exact

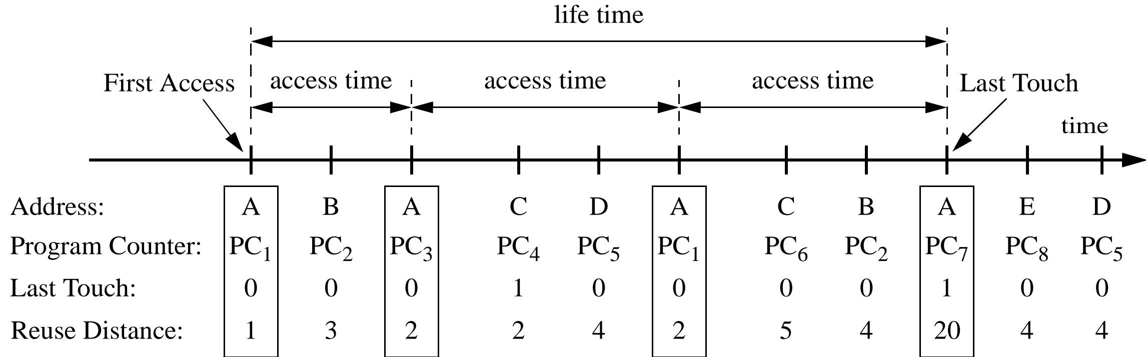

Figure 1: A memory address trace, and various information associated with the trace used to predict last touch and/or reuse distance. Information includes memory addresses, memory reference program counters, last touch history, and reuse distance. Access and live times are indicated for a sequence of references to the memory address A.

reuse distance values instead of last touch outcomes. To be effective, RDPs must track reuse distances larger than the cache's natural LRU stack depth. We rely on the same shadow tags in RD-LTPs to provide the deeper reuse distance information. Because RDPs predict reuse distances, we can more exactly determine which cache blocks are used the farthest in the future, thus identifying OPT last touches more accurately. Our results show an 64 KB RDP can improve the miss rate compared to an RD-LTP by an additional 2.7%.

The remainder of this paper is organized as follows. After discussing related work in Section 2, Section 3 introduces our predictors. Then, Section 4 presents our experimental methodology. Next, Section 5 evaluates cache management policies that use RD-LTPs, and Section 6 evaluates cache management policies that use RDPs. Finally, Section 7 concludes the paper.

2. Related Work

This paper is closely related to previous work on last touch prediction. Several proposals for predicting a cache block's last touch have been explored. To illustrate the different approaches, Figure 1 shows a timeline of memory references performed on the memory location A (indicated by the boxes), beginning with a cache-missing reference that allocates A's block in cache, and ending with a last touch reference. Memory references performed on other locations in between references to A are also shown, and various runtime information associated with all the memory references is indicated below the timeline.

Existing LTPs predict last touches based on either signatures, life times, or access times. Signature-based LTPs associate each last touch reference with an execution signature that captures the runtime context for the last touch. In particular, Lai's LTP [3, 4] forms a signature from instruction traces. Truncated addition is performed on the sequence of program counter values for each memory reference to a cache block, thus encoding the trace

of memory instructions leading up to the block's last touch. For example, in Figure 1, the sum $PC_1 + PC_3 + PC_1$ truncated to the desired length is an instruction trace signature for the last touch reference performed by PC_7. While effective for L1 caches, Lin and Reinhardt [5] show instruction trace signatures are far less accurate for large and highly-associative caches commonly found at the L2 level. Instead, they find L2 last touches are better correlated to last touch history. For each memory reference, the last touch history specifies a single bit–"0" for not a last touch and "1" for last touch–as indicated in Figure 1. A memory reference's last touch signature is formed by concatenating the history bits from the N preceding memory references to the same cache block. For high prediction accuracy, $N = 16$ to 32 is required [5].

The Inter-Reference Gap (IRG) model [10] is another signature-based predictor similar to our approach. IRGs correlate predicted future reuse length values with histories of previous reuse length values. In that regard, it's similar to our RDP. The main difference is our signatures and predicted values are *reuse distances*, not reuse lengths. Also, IRGs only accumulate history locally to a single memory block whereas our approach uses global history.

In contrast to signatures, Kharbutli and Solihin [2] predict last touches by observing either cache block life times or access times. When the life time of a block in the cache expires, the memory reference at the time of expiration is predicted as a last touch. Alternatively, if no reference to a cache block occurs after the access time elapses, then the most recent reference is predicted as a last touch. To quantify life and access times, Kharbutli and Solihin count memory references. In particular, the number of references to a cache block from the first access to the last touch quantifies life time, while the number of interceding references between two touches to the same cache block quantifies access time. For example, in Figure 1, the cache block containing location A has a life time of 4 and an access time of 2 (the worst-case time value is chosen). The counters for predicting life and access times are stored in hardware predictors, called LvP and AIP, respectively.

In addition to using memory reference counts, cache block life times (and hence last touches) can also be predicted based on cycle counts. Cache Decay [6] and Adaptive Mode Control [11] observe the number of cycles that have elapsed since a block's most recent reference, and marks the block as dead when the elapsed time exceeds a certain threshold. Another approach ties cache block liveness to program or runtime system execution [12]. For example, after a method referencing a block's data terminates or the block's data has been garbage collected, the cache block is assumed to have received its last touch. Such cycle-based and program-based approaches have been used to save energy (*e.g.*, predicted dead blocks are powered down to eliminate leakage). Lastly, while all the techniques mentioned thus far are hardware techniques, Wang *et al* propose identifying last touch references in the compiler, and providing hints to the architecture through special memory operations.

The predictors studied in this paper are signature-based predictors, but they differ from existing techniques in two major ways. First, instead of using instruction traces, last touch history, or reuse lengths, we form signatures from sequences of reuse distance values. Moreover, our signatures are global in that they aggregate information from different memory locations; previous signatures contain information from a single memory location only. And second, compared to previous LTPs, we also predict more detailed reuse information. LTPs essentially predict a binary reuse outcome: the distance to the next use of a cache block

is either greater than or less than the cache associativity, implying the current memory reference is a last touch or not a last touch, respectively, assuming LRU. For LRU last touches, we also predict how far into the future the next reference will be. This more exact information can be used to help distinguish between LNO and OPT last touches.

Finally, while our approach (along with all other LTPs) identify dead blocks for *early eviction*, some techniques identify soon-to-be-referenced blocks for *late retention*. In particular, Puzak [9] uses shadow tags to identify LNO blocks and retains them in cache longer. Similar to Puzak, we too use shadow tags–not for late retention but rather for keeping accurate reuse information to form signatures. More recently, Qureshi *et al* [7] propose the Dynamic Insertion Policy (DIP) which places certain incoming cache blocks in the *MRU* stack position so as to retain already resident cache blocks. Compared to LTP techniques, DIP requires less hardware support, but only targets a specific reuse pattern.

3. Predicting Last Touch References

This section presents our predictors. We begin by discussing reuse distance prediction using reuse distance history (Section 3.1.), and motivate intuitively why it can work well (Section 3.2.). Then, we describe how predictions are performed in RD-LTPs (Section 3.3.). Finally, we discuss LNO last touches, and introduce RDPs (Section 3.4.).

3.1. Global Reuse Distance History

All our predictors correlate predicted outcomes with sequences of reuse distance values, called *reuse distance history*. Moreover, these sequences are *global* because they are formed from back-to-back memory references, not just references to the same memory location. The bottom of Figure 1 shows the sequence of reuse distance values for our example memory reference timeline. The sequence labels each memory reference with its reuse distance, *i.e.* the number of unique memory locations referenced before the next reference to the same location. For example, the first reference to B has a reuse distance of 3 because A, C, and D are referenced before B is referenced again. Let us define the *previous reuse distance* (PRD) of a memory reference as the reuse distance of the most recent reference to the same location. For example, the second reference to C has a previous reuse distance of 2 because its previous reference to C (performed by PC_4) has a reuse distance of 2. Given these definitions, a memory reference's global reuse distance history contains the N previous reuse distance values immediately preceding the reference, where N is the history length. For example, the global reuse distance history of the last reference to A, assuming $N = 2$, is $\{2, 3\}$ because the two preceding memory references to C and B have previous reuse distances of 2 and 3, respectively.

To enable the use of global reuse distance history in hardware predictors, we make two simplifications. First, we consider reuses at cache block granularity rather than individual memory words–*i.e.*, A–D in Figure 1 represent cache block addresses. This makes sense since cache management decisions are made at the cache block level anyways. And second, we compute each reuse distance value across references to the same cache set rather than across all memory references–*i.e.*, A–D in Figure 1 map to the same set.

With these simplifications, we can easily compute the previous reuse distance for certain memory references in hardware, and hence form global reuse distance histories, as long as

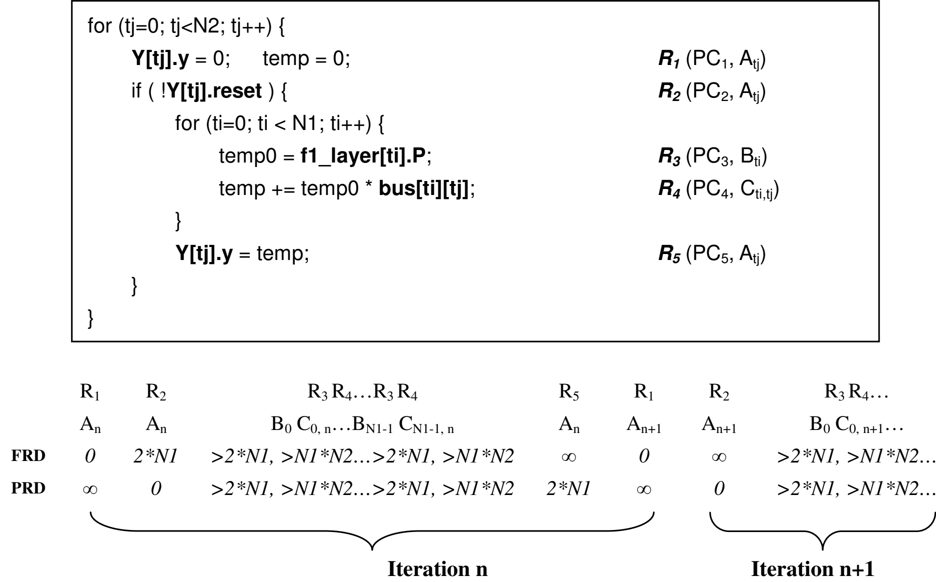

Figure 2: Memory reference pattern from the ART benchmark.

the cache maintains LRU ordering between cache blocks. In particular, on a cache hitting memory reference, the per-set reuse distance of the previous memory reference to the same cache block is simply the block's position in the cache set's LRU stack. This permits us to track memory references' reuse distances so long as the associated cache blocks remain in cache. (Equivalently, we can observe any reuse distance between 0 and $CA - 1$, where CA is the cache associativity). Unfortunately, we cannot track the reuse distance for memory references whose blocks leave the cache since their associated LRU stack information is lost. When a cache block leaves the cache, we assign a reuse distance of CA to the last memory reference performed on the block (*i.e.*, its last touch), signifying the true reuse distance is unknown but is at least CA. For example, in Figure 1, the global reuse distance history for the last reference to D, assuming $N = 2$, $CA = 8$, and the reference to E (which is a cache miss) is $\{2, 8\}$.

3.2. Motivating Example

Having defined global reuse distance history, we now show why it can be effective for reuse distance prediction. As a motivating example, we use a frequently executed loop nest from *art*, a memory-intensive SPEC2000 benchmark. Figure 2 shows the loop nest code. Within the outer loop of the code, there are five memory references, R_1 to R_5, performed by five different instructions, PC_1 to PC_5. The memory reference trace for two consecutive iterations of the outer loop is shown under the code, along with each reference's *future* reuse distance (FRD) and PRD. The FRD and PRD values are computed assuming each element of the Y array is smaller than a cache block so that R_1 can access the same cache block as R_2 in the same iteration of the outer loop, as well as R_5 if $Y[tj].reset$ is false.

From the sequence of PRD values in Figure 2, we can identify a few reuse distance history patterns. In particular, assuming a history length of 3, possible patterns include $[\infty, 0, > 2 * N1]$ for R_4, $[0, > 2 * N1, > N1 * N2]$ for R_3, $[> 2 * N1, > N1 * N2, > 2 * N1]$ for R_4, and $[> N1 * N2, 2 * N1, \infty]$ for R_2. As described in Section 3.1., these pattern histories–along with the associated referencing PCs–can be used to predict memory references' FRD values. Take R_2 for example. The FRD of the first R_2 instance is 2*N1 because the if condition is met for iteration n. This information can be captured by a predictor and used to predict the following iteration, where the reuse distance history is the same for the second R_2 instance. Notice, however, reuse distance history alone is not enough to distinguish certain cases. For example, the reuse distance history for the last instance of R_3 is identical to the reuse distance history for R_5. In such cases, we must augment the ambiguous reuse distance histories with memory references' PCs to distinguish between them. At the same time, PC alone (without reuse distance history) clearly cannot provide good predictability either. For example, the FRD of R_2 depends on the outcome of the if statement. Predicting solely on the PC value, PC_2, cannot disambiguate between the two possible if statement outcomes, whereas the reuse distance history can provide the context for performing such disambiguation. This is why we combine reuse distance history with instruction PCs.

Not only does Figure 2 show how an RD-based predictor works, it also illustrates why it can be effective. Despite executing a large number of iterations (N1 is typically very big), there are a relatively small number of reuse distance history patterns that arise in the `art` code. This is because the code contains only a few memory references, and exhibits fairly simple control flow. Hence, a small predictor table can capture all of the important patterns. In contrast, signatures that track individual cache blocks, like those used in LvP and AIP, can generate significantly more patterns due to the enormous number of cache blocks referenced by the code. For per-block signatures, much larger predictor tables are required to store the prediction state. The compact prediction state associated with RD-based predictors also facilitate very fast training. For example, after only a single outer-loop iteration, the reuse distance history for R_2 can be captured and used to make predictions for the second instance of R_2 in the following iteration. However, for per-block predictors, enough accesses to each cache block must occur to generate sufficient history to train a predictor. For example, in the *art* code, it is difficult to predict for cache block A_n since there are only 3 references (at most) to the block each outer-loop iteration.

Although our analysis of reuse distance prediction does not take cache organization into consideration (essentially, we assume a fully associative cache), we find the same behavior for reuse distance history patterns occurs in individual sets of set-associative caches as well. Hence, our motivating example also illustrates why global reuse distance history can be effective for reuse distance prediction in a set-associative cache.

3.3. Predictor Hardware

Now, we present our predictors in detail. We begin by describing how we predict last touches using RD-LTPs. Our other predictor, the RDP, employs very similar hardware, and will be discussed in Section 3.4..

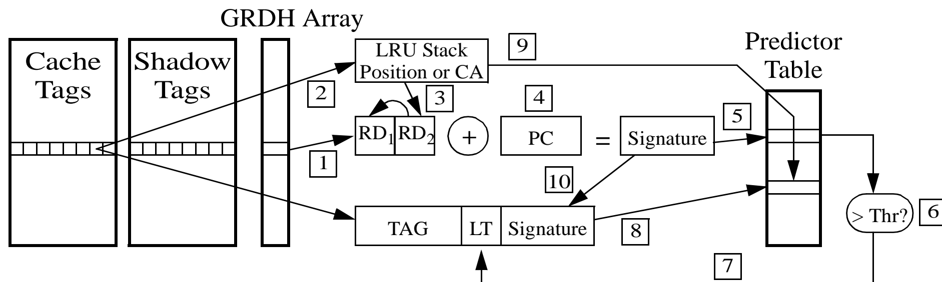

Figure 3: Reuse distance last touch predictor organization and actions. Steps 1–7 perform a prediction. Steps 8–10 update the predictor.

Figure 3 shows the hardware organization of an RD-LTP, and illustrates the different steps involved in performing predictions and updating predictor state. An RD-LTP requires four additions to a conventional cache. First, a *global reuse distance history array* (GRDH array) is needed to store the per-set global (previous) reuse distance histories. Second, each cache tag is augmented with a *last touch (LT) bit* as well as a *signature field* containing the signature observed during the block's most recent reference. (In Figure 3, we assume an 8-way set associative cache, so there are 8 tag, LT, and signature fields in the main tag array). Third, a *shadow tag array* [9] is included to extend the LRU stack depth of the cache. Like normal tags, each shadow tag also includes a signature field as well (but no LT bit). Finally, a central *predictor table* is needed to store the prediction outcomes.

Labels "1" – "7" in Figure 3 illustrate the different steps for a last touch prediction on a cache hit. First, we read the GRDH array entry corresponding to the referenced cache set (label "1"). This entry contains the concatenated previous reuse distances for the N memory references prior to the current reference that map to the referenced set. We construct the N-lengthed global reuse distance history for the current memory reference by observing the LRU stack position of the referenced cache block (label "2"), and append it to the GRDH entry (label "3"). (In Figure 3, we assume $N = 2$). Next, we XOR the global reuse distance history with the memory reference's PC (label "4") to form the signature for the current reference. This signature is used to index the predictor table (label "5"), producing a saturating counter whose value is compared against a threshold (label "6"). If the counter value is greater than the threshold, a last touch outcome is predicted; otherwise, a not lost touch outcome is predicted. The predicted outcome is written into the cache block's LT field (label "7"). In addition to cache hits, RD-LTPs also make predictions on cache misses. The same 7 steps in Figure 3 are performed, except CA is appended to the GRDH entry instead of the cache block's LRU stack position (see Section 3.1.).

Labels "8" – "10" illustrate the different steps for updating the predictor. In particular, the hit/miss outcome of the current memory reference validates the correctness of the last touch prediction for the previous reference to the same cache block. To permit updating the predictor with this actual outcome information, the signature associated with the previous prediction is stored along with the tag of the referenced cache block. This signature is used to index into the predictor table (label "8") so that the previous prediction's saturating

counter can be updated (label "9"). If the current reference is a cache hit, the update can occur at reference time; in this case, the counter is decremented to reflect the cache hit. If the current reference is a cache miss, then the cache block (and hence its signature) is no longer in cache, and thus the update must occur at eviction time before the signature is lost. In this case, the counter is incremented to reflect the impending cache miss. Lastly, the current memory reference's signature is stored with the cache tag (label "10") to enable a predictor update on the next reference to the same cache block.

As Figure 3 shows, the ability to monitor the position of cache blocks in LRU stacks is critical to RD-LTPs. Unfortunately, once cache blocks leave the cache, RD-LTPs cannot track their reuse distances. This can become problematic when the cache acts on predictions to evict blocks early. In particular, if an incorrect last touch prediction leads to an early eviction, it is impossible to detect the misprediction and update the predictor accordingly since the cache block is no longer in cache when the next reference to the block (which would have been a cache hit) occurs. The cache not only suffers an additional miss, but the predictor will likely make the same misprediction in the future. Worse yet, the additional cache misses that such incorrect last touch predictions trigger also corrupt the global reuse distance history, inserting CA values into the history instead of the actual reuse distances. This can cause additional mispredictions and cache misses down the road.

To address this problem, we augment the cache with shadow tags, as shown in Figure 3. In particular, we implement a shadow tag array containing SA shadow tags. When a cache block is evicted, we remove its data from the cache, but retain its tag in a shadow tag entry. We maintain LRU ordering between *all* tags (normal and shadow), thus extending the cache's LRU stack depth by SA. It requires at least $CA - 1$ shadow tags to track the true reuse distances of recently evicted cache blocks, including those evicted early due to last touch predictions, for as long as they remain in the shadow tag array (*i.e.*, until they become the least recently used among both normal and shadow blocks). The extended reuse distance visibility provided by the shadow tags allows us to identify premature evictions caused by last touch mispredictions, and hence, avoid corruption of global reuse distance history and improve cache performance.

3.4. LNO vs OPT Last Touches

Like all previous LTPs, RD-LTPs predict the last touches observed under an LRU policy, a natural consequence of the fact that the underlying cache management policy is itself LRU. However, many LRU last touches are not last touches under OPT. These references are referred to as LRU non-OPT, or LNO, last touches [5]. LNO last touches typically have a reuse distance that is only slightly larger than CA, so they can be converted into cache hits if the referenced blocks are kept in cache a bit longer. In particular, when multiple cache blocks are marked as LRU last touches simultaneously, evicting those blocks with larger reuse distances in favor of those with shorter reuse distances can keep the soon-to-be-referenced blocks in cache longer than an LRU policy would, perhaps long enough to convert what would be cache misses under LRU into cache hits.

In this paper, we propose two methods for distinguishing LNO and OPT last touches. The first method works with our RD-LTP, and employs a very simple heuristic: when multiple cache blocks are marked by the RD-LTP as LRU last touches, pick the most-

recently-used block that has been marked. We find the MRU marked block is often a good choice. In Section 5, we will present results that validate this claim, and provide more intuition behind why it works. The second method takes a more direct approach: predict the actual reuse distance for each marked LRU last touch block. Then, we can simply pick the block with the largest predicted reuse distance. One challenge of the second method is the reuse distances we need are necessarily larger than CA since the cache blocks of interest are guaranteed to be LRU last touches. Observing such long reuse distances requires LRU stacks larger than CA. Recall from Section 3.3. that RD-LTPs already extend the LRU stack using shadow tags to improve prediction accuracy. Such shadow tags can also track reuse distances beyond CA for distinguishing LNO and OPT last touches.

To enable the second method, we propose reuse distance predictors, or RDPs. Our RDPs predict the exact reuse distance up to depth $CA + SA$. They are very similar to RD-LTPs. In particular, they use signatures based on global reuse distance history, so all the mechanisms in Figure 3 for creating signatures and indexing into the predictor table remain the same. The main difference is RDPs store actual reuse distance values in the predictor table instead of saturating counters. Notice, on a cache hit, we actually know the exact reuse distance value by observing the position of the referenced cache block in the LRU stack. An RD-LTP uses this information to form signatures (label "3" in Figure 3), but ignores it when updating the predictor (label "9"). An RDP updates its predictor with this exact reuse distance value. Specifically, the predictor entry is set to the observed LRU stack position of a cache block on a hit to the cache tags (including both normal and shadow tags); otherwise, it is set to $CA + SA$, signifying a miss in all the tags. Another (more minor) difference is the LT field must be replaced with a reuse distance value field to store predicted reuse distances. It is important to emphasize that RDPs can only provide limited reuse distance information (between 0 and $CA + SA$). However, as we will see in Sections 5 and 6, this is an important range.

4. Experimental Methodology

The remainder of this paper conducts an in-depth evaluation of our predictors from Section 3, applying them for cache management and quantifying the resulting cache performance. Our evaluation focuses on managing the L2 cache since this is an especially critical part of the memory hierarchy for modern high-performance CPUs. As part of our evaluation, we also compare our approach against existing LTP and insertion policy techniques.

4.1. Benchmarks

We considered the 24 SPEC CPU2000 benchmarks shown in Table 1 for driving our simulations. For all the benchmarks, we use the pre-compiled Alpha binaries provided with the SimpleScalar tools [13] which have been built using the highest level of compiler optimization.[1] All of our benchmarks use the reference input set. To acquire our memory traces on M5, we first fast-forward each benchmark to its representative region (the columns labeled "Skip Ins" in Table 1 report the number of fast forwarded instructions). Then, we turn on tracing

1. The binaries we used are available at http://www.simplescalar.com/benchmarks.html.

High Potential				Low Potential			
App	Skip Ins	MPKI	Type	App	Skip Ins	MPKI	Type
ammp	4.75B	3.27	FP	perlbmk	1.7B	0.01	Int
art	2.0B	100.70	FP	eon	7.8B	0.00	Int
bzip2	1.8B	1.07	Int	gzip	4.2B	0.15	Int
facerec	69.3B	3.00	FP	gap	8.3B	0.98	Int
galgel	14B	1.41	FP	apsi	2.3B	2.15	FP
gcc	2.1B	3.73	Int	fma3d	2.6B	0.00	FP
mcf	14.75B	70.04	Int	equake	4.8B	13.58	FP
mesa	2.1B	0.08	FP	lucas	1.5B	9.84	FP
parser	13.1B	1.26	Int	swim	5.7B	17.56	FP
sixtrack	3.8B	0.12	FP	applu	1.5B	14.30	FP
twolf	2.0B	3.01	Int				
vortex	2.5B	0.49	Int				
vpr	7.6B	4.77	Int				
wupwise	3.4B	2.05	FP				
AVG		13.93		AVG		5.33	

Table 1: SPEC CPU2000 benchmarks used to drive our cache simulations (B = Billion).

of L2 references, and simulate for 2 billion instructions. The amount of fast-forwarding for each benchmark was selected using SimPoint [14] by consulting the SimPoint website.[2]

Out of the 24 benchmarks we considered, we found several benchmarks exhibit a very small difference between the LRU and OPT policies for a baseline 1MB L2 cache. Since OPT is theoretically optimal, a small LRU-OPT difference implies there isn't much opportunity for improvement. We divide our benchmarks into two categories based on this observation. Benchmarks with an LRU-OPT difference less than 10% are placed in the "Low Potential" category, while benchmarks with an LRU-OPT difference of 10% or greater are placed in the "High Potential" category. Table 1 shows our suite contains 11 low potential and 14 high potential benchmarks. We verified that our techniques provide very little gain for the low potential benchmarks (less than 1% improvement or degradation over the LRU policy). Since there's no room for improvement for the "Low Potential" category, we will focus on the "High Potential" category in the remainder of this paper.

In addition to simulating our own techniques, we also compared against the AIP [2], LvP [2], and DIP [7] techniques described in Section 2. AIP and LvP represent the state-of-the-art for LTP-driven cache management, both in terms of performance and hardware cost. They have been shown to outperform several other existing LTPs [2]. DIP also represents the state-of-the-art, but for cache insertion policies. We are the first to compare DIP against LTP techniques. To enable our comparison, we implemented the AIP and LvP last touch predictors, and integrated them into our cache simulator. For DIP, we used the simulator provided on the authors' web site.[3]

2. Simulation regions for the Alpha binaries we use are published at http://www-cse.ucsd.edu/~calder/simpoint/multiple-standard-simpoints.htm.
3. The DIP simulator is available at http://users.ece.utexas.edu/~qk/dip/.

Cache Parameters			
L1 I-cache		16 Kbyte, 2-way set associative, 64 byte blocks	
L1 D-cache		16 Kbyte, 2-way set associative, 64 byte blocks	
L2 U-cache		1 Mbyte, 8-way set associative, 64 byte blocks	
Predictor Parameters			
History Length	2	RD-LTP Shadow Tags	7 per cache set, 7 bits each
Reuse Distance Values	3 bits	RDP Shadow Tags	8 per cache set, 7 bits each
Predictor Table	1024 entries	RD-LTP Table Entries	2 bits
Signature Size	10 bits	RDP Table Entries	4 bits

Table 2: Cache and predictor parameter settings.

4.2. Baseline Configuration

Our evaluation employs trace-driven simulation. We use the in-order processor model from the M5 simulator [15] configured with baseline L1 and L2 cache to simulate several uniprocessor benchmarks. The M5 simulator was instrumented to record the post-L1 memory address trace, along with the PC of each referencing instruction, seen at the input to the L2 cache during execution-driven simulation. After running the M5 simulations, we replayed the L2 memory traces on a trace-driven cache simulator. The top portion of Table 2 reports the parameters for the L1 and L2 caches in the M5 simulations.

The cache simulator includes architectural models for an RD-LTP and an RDP. In the bottom portion of Table 2, we report the baseline configuration parameters for the predictors. We use a global reuse distance history of length 2. Since the simulated cache is 8-way set associative, each reuse distance value in the history is between 0–8 (0–7 encode the 8 positions in the LRU stack, while 8 encodes references not found in the LRU stack). We encode reuse distances of 0 and 1 using the same value, thus enabling a compact 3-bit encoding. (We did not notice any performance degradation due to this simplification). This results in a 6-bit global reuse distance history. For the predictor table, we assume 1024 entries. To form the 10-bit signature needed to index the table, we pad the 6-bit history with 4 leading 0s, and XOR the result with the 10 least significant bits of the memory reference's PC.[4] For each set in the cache, we augment the normal tags with 7 shadow tags in RD-LTP, and 8 in RDP. Puzak's thesis [9] shows many references have reuse distances slightly beyond the cache associativity. Using $SA = CA$ enables RDP to track these important short reuses, while $SA = CA - 1$ is enough for RD-LTP to track true LRU reuse distance. Finally, the predictor table entries differ in size depending on the type of predictor. RD-LTP table entries contain a 2-bit saturating counter while RDP entries contain a 4-bit reuse distance value. (The 4-bit RDP table entry encodes reuse distance values between 0–16 using the same compact encoding trick described earlier; 0–15 encode the 16 positions in the LRU stack including shadow tags, and 16 encodes references not found in the LRU stack).

Component	Cost
GRDH array (2 * 3-bit/set * 2048 sets)	1.5 kB
data blocks signature (10-bit/block * 8-block/set * 2048 sets)	20 kB
RD-LTP LT prediction bit (1-bit/block * 8-block/set * 2048 sets)	2 kB
RD-LTP predictor table (1024 * 2-bit)	0.25 kB
RD-LTP shadow tags ((10 + 7) bit * 7 entry/set * 2048 sets)	29.75 kB
RDP RD prediction (4-bit/block * 8-block/set * 2048 sets)	8 kB
RDP predictor table (1024 * 4-bit)	0.5 kB
RDP shadow tags ((10 + 7) bit * 8 entry/set * 2048 sets)	34 kB
RD-LTP total cost	53.5 kB
RDP total cost	64 kB

Table 3: Storage Cost of RD-LTP and RDP for 1MB cache.

4.3. Implementation Cost

We analyze the cost of our technique in terms of area, power, and cycle time. Given the baseline configuration parameters in Table 2, our RD-LTP and RDP incur 53.5 and 64 Kbyte of additional storage, respectively. The breakdown of where this additional storage is incurred is shown in Table 3. This extra hardware is needed to implement the per-block prediction and signature fields, shadow tags, GRDH array, and predictor tables illustrated in Figure 3. Compared to the 1 Mbyte L2 cache these predictors are used to manage, this represents at most a 6% area overhead for both predictors. In terms of power consumption, we anticipate the additional storage will increase the L2 power in proportion to its area overhead. Given that the L2's activity factor is normally low, the overall CPU power impact, however, should be much less than 6%. And in terms of cycle time, the two predictor table accesses (predict and update) are the most expensive because they occur in series with each L2 access. However, the latency (which we estimate to be at most 3 or 4 cycles per table access) is off the CPU's critical path as the predictor operations can be performed *after* the cache data is returned to the CPU. Given the low frequency of L2 references, there should be ample time to completely hide the predictor's latency before the next L2 reference.

In comparison, Kharbutli and Solihin report 61 and 57 Kbyte of additional storage for AIP and LvP, respectively, assuming a 512 Kbyte L2 cache [2]. For a 1 Mbyte L2 cache, this overhead increases to 82 and 73 Kbytes. So, our hardware overhead is very similar to AIP and LvP. Unfortunately, the predictor table evaluated in Kharbutli and Solihin's previous study achieved poor performance for several of our benchmarks. Hence, in our study, we use infinite predictor tables for LvP and AIP. (With infinite tables, our LvP and AIP performance is similar to what is reported in [2]). In contrast, there is negligible hardware overhead associated with DIP [7].

4. Since M5 instructions are 4 bytes wide, we divide the PC by 4 prior to truncating to 10 bits. This removes the two least significant 0 bits in all instruction PCs.

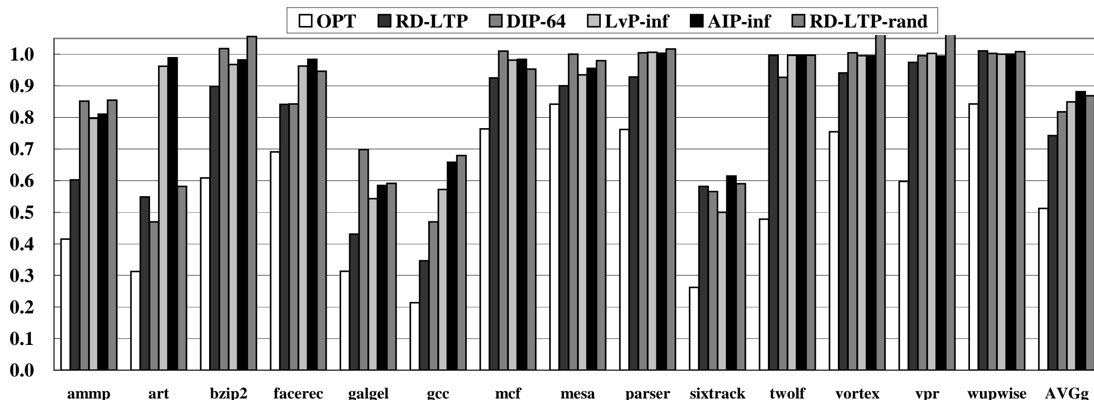

Figure 4: Cache miss rates for the high potential benchmarks achieved by OPT, RD-LTP, DIP, LvP, AIP, and RD-LTP-Rand. All of the miss rates are normalized to the LRU miss rate.

5. Last Touch Prediction Results

We begin our evaluation by studying the performance achieved when driving cache management decisions using our RD-LTP predictor. Later, in Section 6, we will evaluate the RDP predictor.

5.1. LTP and Insertion Policy Evaluation

Figure 4 presents the cache miss rates achieved by RD-LTP, and compares them against LvP, AIP, DIP, and OPT for high potential benchmarks. (The bars labeled "RD-LTP-Rand" will be explained later in Section 5.4.). All of the miss rates in Figure 4 have been normalized to the miss rate achieved by an LRU policy for the same benchmark, and the bars labeled "AVGg" report the geometric mean across all the benchmarks.

In Figure 4, we see there is a significant opportunity for performance gains in these high potential benchmarks as the LRU-OPT difference is nearly 50% on average. As the AVGg bars show, RD-LTP capitalizes on this opportunity, reducing the miss rate over LRU by 25.8%. In addition, compared to existing LTPs, RD-LTP achieves a miss rate that is 12.6% and 15.8% lower than LvP and AIP, respectively. In particular, RD-LTP outperforms both LvP and AIP in 11 out of the 14 benchmarks, outperforms AIP alone in 1 benchmark, and matches the performance of LvP and AIP in 1 benchmark. These improvements reduce by 30% the performance gap separating LvP/AIP from OPT. Compared to existing insertion policies, RD-LTP achieves a miss rate that is 9.3% lower than DIP. In particular, RD-LTP outperforms DIP in 9 out of 14 benchmarks, and matches the performance of DIP in 1 benchmark. These improvements reduce by 23.7% the performance gap separating DIP from OPT.

RD-LTP outperforms LvP and AIP because our predictor achieves a higher prediction rate and selects victims effectively, two issues that Sections 5.3. and 5.4. will study in greater depth. RD-LTP outperforms DIP because it more effectively addresses a wide range of memory use patterns. Insertion policies such as DIP specifically address thrashing due to

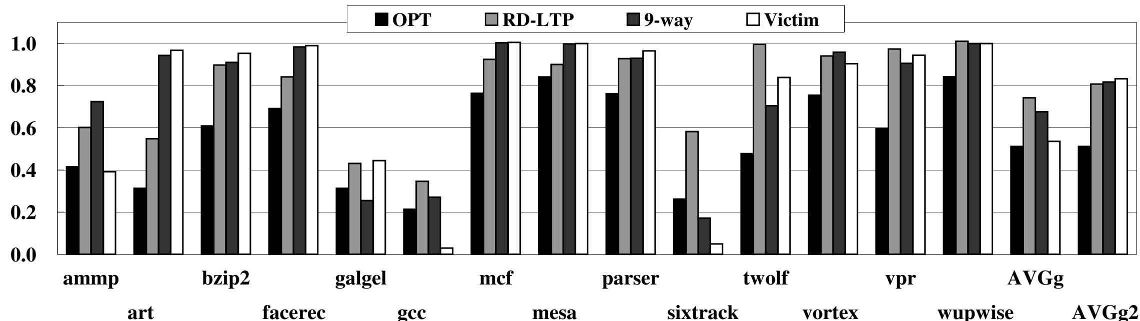

Figure 5: Cache miss rates for the high potential benchmarks achieved by OPT, RD-LTP, 9-way cache, and victim cache. All of the miss rates are normalized to the LRU miss rate.

working sets that are larger than the cache capacity. They allocate cache blocks at the LRU position rather than the MRU position to retain a portion of the working set and increase reuse on that portion. However, for other memory use patterns, LRU insertion can perform poorly; hence, DIP reverts back to MRU insertion. In these cases, RD-LTP can outperform DIP by identifying dead blocks and performing early eviction. Figure 4 shows DIP is quite competitive, especially considering that it requires almost no additional hardware support. Nonetheless, RD-LTP still provides higher performance.

5.2. Cache Organization Evaluation

As described in Section 4.3., our hardware costs about 60 Kbyte storage. To justify the hardware cost invested in implementing an LT predictor rather than other cache organization improvements such as higher associativity and victim caches, we evaluate a 9-way set associative LRU cache that has an additional 128 Kbyte way compared to the baseline 1 MB cache, as well as a 1 Mbyte LRU cache with a 64 Kbyte victim cache [16](1024 cache blocks, random replacement policy). Figure 5 compares the performance of these techniques against RD-LTP and OPT assuming the baseline 1 MB cache. Among the 14 benchmarks evaluated, RD-LTP outperforms the 9-way cache in 8 benchmarks, and outperforms the victim cache in 7 benchmarks. For the remaining benchmarks, RD-LTP does not perform as well as the 9-way cache or the victim cache because the increased cache capacity accommodates the majority of the benchmarks' working sets. In fact, the 9-way cache reduces miss rates in *galgel* and *sixtrack* even more than OPT. And for *gcc* the victim cache reduces miss rate by 97%, while OPT can only reach an 88% reduction. The victim cache also performs best in *ammp* and *sixtrack*, which helps the victim cache to reach similar cache performance as OPT on average.

While the 9-way cache and victim cache perform well on average, this is primarily due to a small number of benchmarks that receive a huge performance boost from the extra cache capacity because their working sets are just slightly larger than 1 MB. When these special cases are eliminated, the comparison becomes more favorable for RD-LTP. In particular, we compute another average, labeled AVGg2 in Figure 5, that excludes *gcc* and *sixtrack*. In this case, RD-LTP outperforms both 9-way and victim cache on average for the smaller group of

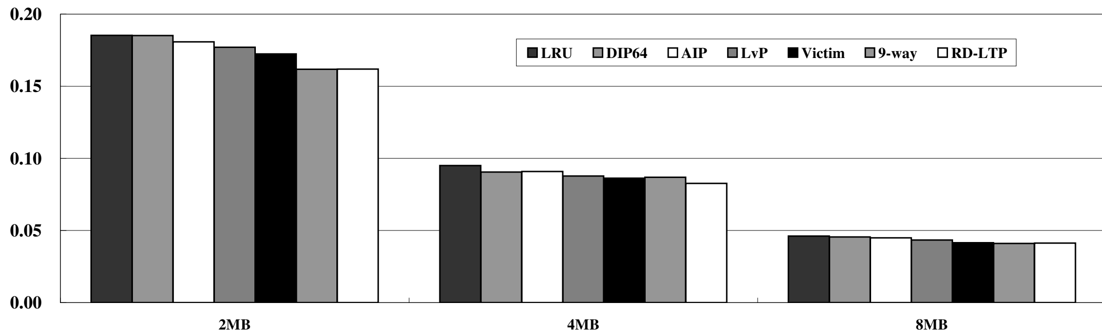

Figure 6: Average cache miss rates for different cache sizes achieved by LRU, 9-way, victim cache, LvP, AIP, DIP, and RD-LTP. All of the miss rates are normalized to the LRU miss rate for 1 MB cache.

benchmarks. Based on these results, we conclude that when benchmarks' working sets are very close to the cache size, using the extra hardware to increase cache capacity performs better than implementing an RD-LTP. However, if benchmarks' working sets are noticeably larger than the baseline cache capacity, which is the more general case, the investment of additional hardware in an RD-LTP is more beneficial.

To measure the impact of different baseline cache capacities on various policies and organizations, we also evaluate LRU, 9-way, victim cache, LvP, AIP, DIP and RD-LTP for larger cache sizes from 2 MB to 8 MB while keeping the associativity at 8-way(except for the 9-way cache). Figure 6 shows the average miss rates achieved by the techniques under evaluation, with their hardware scaled up accordingly. RD-LTP continues to outperform or match other techniques up to 8MB as more working sets fit into the larger caches.

5.3. Prediction Rate

To provide insight into how RD-LTP achieves its performance gains, Figure 7 shows the accuracy of the predictions performed by RD-LTP, and compares it against LvP and AIP. Each bar in Figure 7 breaks down the last touch outcomes for each predictor into 3 categories. Components labeled "Correct Prediction" report predictions that correctly identify LRU last touch references; components labeled "Not Predicted" report LRU last touch references that are not identified by the predictor; and components labeled "Wrong Prediction" report the predictions that incorrectly identify LRU last touch references (*i.e.* these references are not LRU last touches). All bars are normalized to the total number of LRU last touch references in the corresponding benchmark, with the last group of bars reporting the average across the 12 benchmarks.

As Figure 7 shows, RD-LTP correctly predicts a much larger fraction of the LRU last touch references than either LvP or AIP. On average, RD-LTP identifies 71.2% of the LRU last touches compared to only 19.2% and 15.6% for LvP and AIP, respectively. Because RD-LTP correctly identifies a greater number of LRU last touches, it has the *potential* to perform a larger number of beneficial early evictions. (In a moment, we will discuss how RD-LTP capitalizes on this potential). Unfortunately, the higher prediction rate is also accompanied by a larger number of mispredictions. As the "Wrong Prediction" components in Figure 7

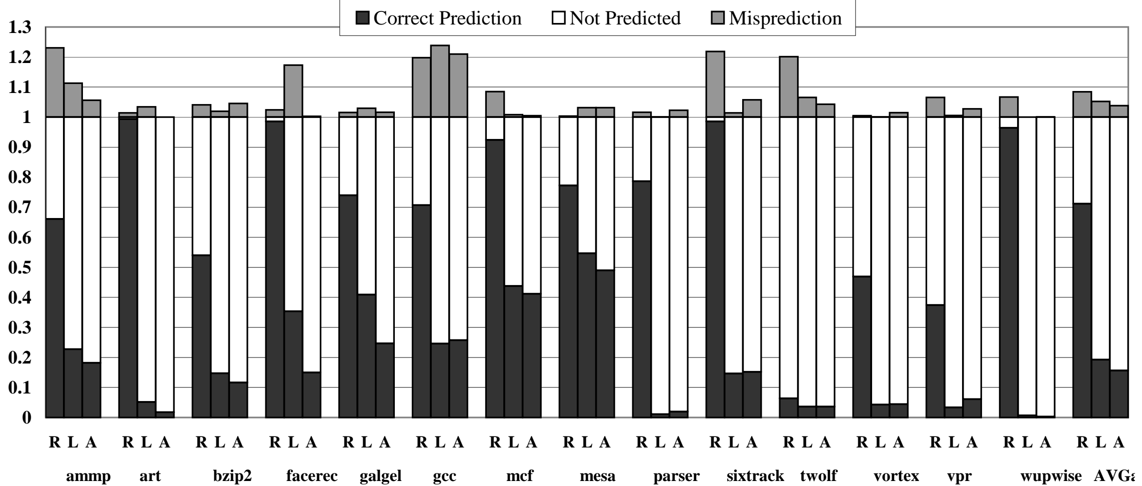

Figure 7: Prediction accuracy of RD-LTP, LvP, and AIP.

show, RD-LTP incurs 8.5% mispredictions whereas LvP and AIP incur only 5.2% and 3.8%, respectively. Such mispredictions can lead to premature evictions, converting some LRU cache hits into cache misses. However, RD-LTP's higher prediction rate far outweighs the negative consequences of its mispredictions.

These prediction accuracy results demonstrate RD-LTP is a more effective predictor than LvP and AIP. We credit three factors. First, last touch events are highly correlated to global reuse distance history. Our signatures simply identify more last touches. Second, RD-LTP's shadow tags improve predictor training. As discussed in Section 3.3., once the cache management hardware begins acting on predictions and performing early evictions, the LRU last touch outcomes of blocks that leave the cache early cannot be observed. Our shadow tags allow us to continue tracking recently evicted blocks, thus permitting us to observe LRU last touches even when replacement deviates from a strict LRU order. Finally, because RD-LTP's accuracy is inherently higher than LvP and AIP, it can be applied more aggressively. RD-LTP predicts *all* memory references; in contrast, LvP and AIP avoid predicting memory references with low accuracy (both predictors employ confidence mechanisms). The smaller pool of predicted memory references in LvP and AIP further reduces their total correct predictions.

5.4. Victim Selection

When an eviction occurs under RD-LTP, there are usually *multiple* cache blocks marked as last touches. To illustrate this point, the column labeled "RDMrk" in Table 4 reports the number of marked blocks encountered on average during an eviction, while the column labeled "RD$\geq$2" reports the percentage of evictions when 2 or more marked blocks are encountered under RD-LTP. As Table 4 shows, 4.7 blocks are found marked on average during each eviction, and 77.9% of the evictions encounter 2 or more marked blocks. So, most of the time, RD-LTP must choose between multiple LRU last touch blocks for eviction. This is the *victim selection problem*, and arises because RD-LTP predicts a large number of LRU last touches, as discussed in Section 5.3.. Note, victim selection is not an issue for

	RDMrk	RD≥2	LPMrk	LP≥2	CRD
ammp	4.3	89.8	0.6	12.1	12
art	7.6	100.0	0.07	1.6	12
bzip2	3.3	82.0	0.3	3.8	14
facerec	7.4	99.8	1.5	31.8	20
galgel	4.3	83.4	1.6	45.7	9
gcc	5.0	92.1	0.9	1.6	9
mcf	7.3	100.0	0.5	4.4	30
mesa	6.2	86.5	1.1	28.1	44
parser	5.2	91.5	0.07	0.9	14
sixtrack	0.9	21.8	1.1	19.2	9
twolf	0.3	3.7	0.1	0.7	10
vortex	3.4	72.1	0.2	2.4	20
vpr	2.4	68.5	0.09	0.4	16
wupwise	7.6	100.0	0.03	0.6	44
AVG	4.7	77.9	0.6	11.0	18.8

Table 4: Number of marked blocks and percentage of evictions with at least 2 marked blocks for RD-LTP and LvP. The last column reports CRD for each benchmark.

LvP/AIP. The columns labeled "LPMrk" and "LP≥2" in Table 4 report the same data, but for the LvP technique (the results for AIP are similar). As Table 4 shows, only 0.6 blocks are found marked on average during each eviction, and only 11% of the evictions encounter 2 or more marked blocks under LvP. Most of the time, there are either 0 or 1 marked blocks because LvP predicts much fewer last touches compared to RD-LTP, so there's no choice.

While RD-LTP identifies a large number of LRU last touches, unfortunately, not all of them are profitable to evict, as discussed in Section 3.4.. In particular, LNO last touches can be converted into cache hits by retaining them in cache a bit longer. Since OPT evictions are a subset of LRU last touches, predicting a large number of LRU last touches and blindly evicting them does not guarantee high performance. It is also important to select the *best* last touch candidates for eviction.

To provide insight into which LRU last touches are most profitable to evict, let us examine the evictions made by LRU and OPT. Figure 8 shows a histogram of last touch references under the LRU and OPT cache management policies for the AMMP benchmark. For different reuse distances (X-axis), the histogram plots the number of last touch references exhibiting that reuse distance (Y-axis). The histograms for LRU and OPT include *actual* last touch references (*i.e.*, all of these lead to evictions). The rightmost point in the histogram reports the cumulative count for all reuse distances beyond the end of the X-axis.

Notice the number of OPT last touches in Figure 8 is always smaller than the number of LRU last touches. The difference between the LRU and OPT histograms constitutes the LNO last touches. Most importantly, notice that beyond some reuse distance, practically all OPT last touches are also LRU last touches, *i.e.* there are very few LNO last touches. For example, in AMMP, beyond a *critical reuse distance* (CRD) of 12, 90% or more of the LRU last touches are also OPT last touches. This makes sense: LRU last touches with large reuse distances have no hope of becoming cache hits, so they are also likely to be OPT

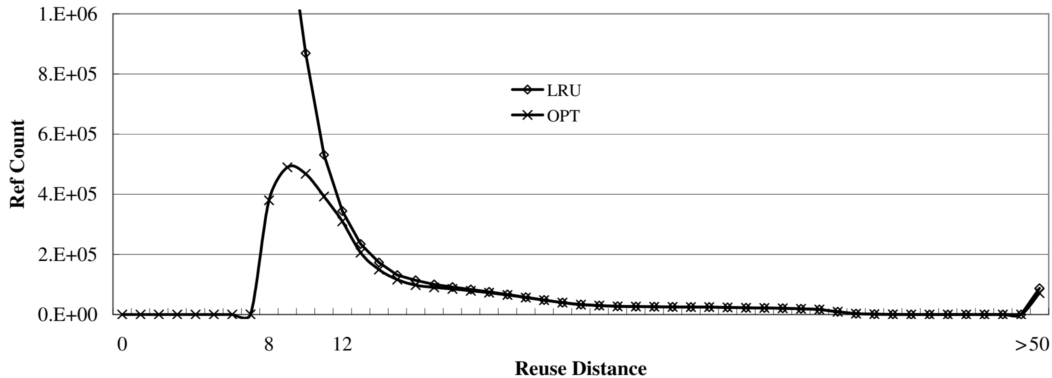

Figure 8: Last touch reference histograms under LRU and OPT cache management for the AMMP benchmark. CRD = 12.

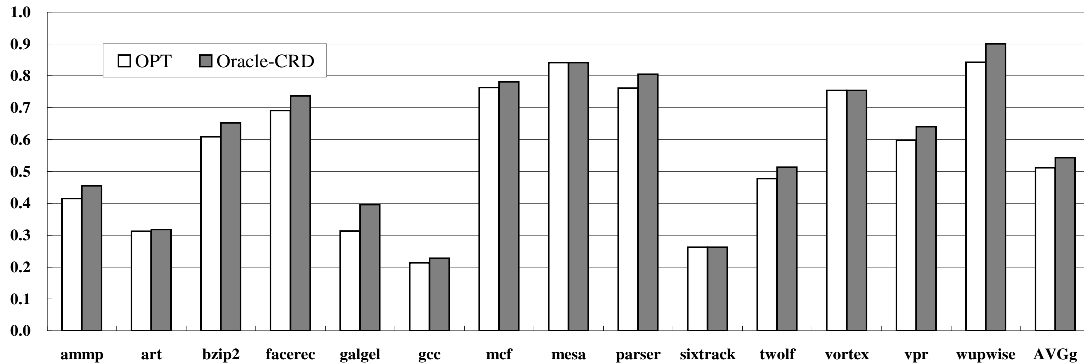

Figure 9: Cache misses under the OPT and Oracle-CRD policies.

last touches. This implies that beyond CRD, *it doesn't matter which LRU last touches we evict–all of them are profitable*. However, below CRD, we must be careful which blocks we evict since there is a mixture of OPT and LNO last touches. We find that all benchmarks exhibit this property, though the exact CRD value is application dependent. In Table 4, the column labeled "CRD" reports the CRDs for all our benchmarks. As the last row in Table 4 shows, on average, there are very few LNO last touches beyond a reuse distance of 18.8.

Figure 9 demonstrates the profitability of evicting LRU last touches beyond CRD. In Figure 9, we show for each benchmark the miss rate achieved by OPT and an ideal eviction policy, called Oracle-CRD. Oracle-CRD employs a perfect last touch predictor that marks all the LRU last touches correctly with no wrong predictions–*i.e.*, it has 100% coverage and 0% wrong predictions (this factors out the effects of a non-ideal predictor, and allows us to just focus on the victim selection policy). In addition, amongst all LRU last touches, Oracle-CRD also knows which ones occur beyond CRD (*i.e.*, the long-reuse blocks). On an eviction, Oracle-CRD always evicts the long-reuse blocks first. If no long-reuse block exists–*i.e.*, there are only short-reuse blocks–it selects one of the short-reuse blocks randomly. As

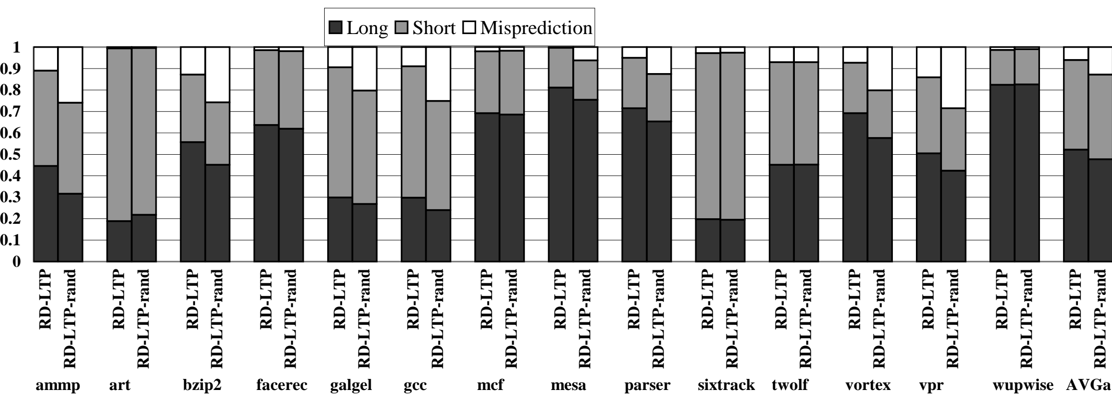

Figure 10: Breakdown of the evictions performed by the RD-LTP and RD-LTP-Rand techniques.

Figure 9 shows, Oracle-CRD comes within 3.1% of OPT. This demonstrates that evicting the long-reuse blocks practically achieves the optimal performance.

Although RD-LTP cannot identify long-reuse blocks (it only predicts LRU last touches), we find the likelihood of evicting long-reuse blocks increases when selecting the MRU marked block for eviction. To illustrate this point, Figure 10 shows a breakdown of the evictions performed by RD-LTP. The components labeled "Long" and "Short" indicate the fraction of evictions that involve a long-reuse and short-reuse block, respectively. In addition, the components labeled "Misprediction" indicate the fraction of evictions that involve a block that was incorrectly marked as an LRU last touch. Breakdowns are given for RD-LTP which always evicts the MRU marked block, and an alternate version, called RD-LTP-Rand, which evicts a randomly selected marked block. As the AVGa bars in Figure 10 show, the MRU policy finds a long-reuse block 52% of the time, while a random policy finds a long-reuse block only 48% of the time.

Not only does the MRU policy select long-reuse blocks more frequently, it also more effectively avoids evicting mispredicted blocks. In Figure 10, we see that the random policy evicts roughly twice as many mispredicted LRU last touches compared to the MRU policy (12.8% versus 6%). Since RD-LTP makes very few mispredictions (see Figure 7), the inter-misprediction time is fairly large. Hence, when a cache miss occurs, it is unlikely for a mispredicted block to be the MRU marked block as several other (correctly) predicted LRU last touches have likely occurred since the last misprediction. Hence, selecting the MRU block often avoids mispredictions.

The advantages of the MRU policy illustrated in Figure 10 translate into performance benefits. The bars labeled "RD-LTP-Rand" in Figure 4 report the miss rate achieved by RD-LTP-Rand. Comparing the RD-LTP and RD-LTP-Rand bars in Figure 4, we see the MRU policy outperforms a random policy by 14.5%. Based on these results, we conclude that MRU victim selection is a good policy for our RD-LTP technique.

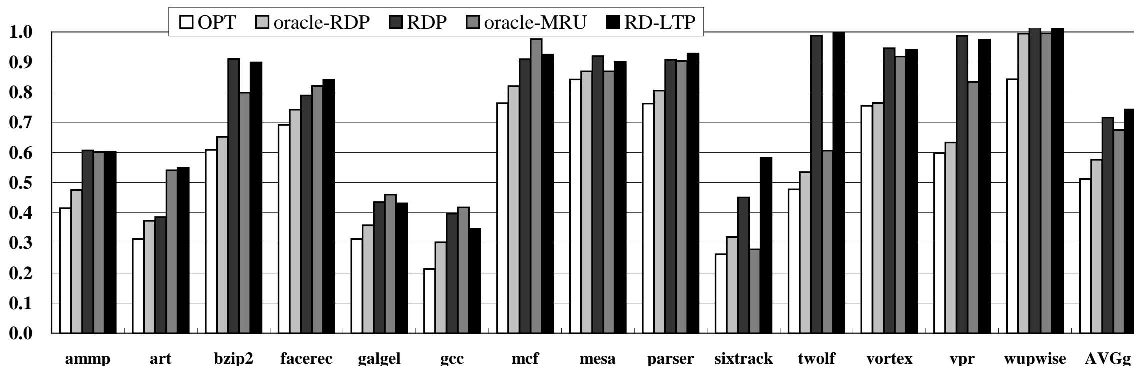

Figure 11: Cache miss rates achieved by OPT, Oracle-RDP, RDP, Oracle-MRU, and RD-LTP for the high potential benchmarks.

6. Reuse Distance Prediction Results

Figure 11 presents our RDP results. In Figure 11, the bars labeled "RDP" report the miss rates achieved when driving cache management decisions using an RDP across our high potential benchmarks. For comparison, the miss rates achieved by RD-LTP and OPT have been included from Figure 4. All bars are normalized to the LRU miss rate for each benchmark, and the group of bars labeled "AVGg" report the geometric mean across all the benchmarks. Comparing the RDP and RD-LTP bars, we see RDP provides an additional 2.7% miss rate reduction over RD-LTP on average. And comparing RDP to the LvP and AIP results from Figure 4, we see RDP improves the miss rate by 14.9% and 17.8%, respectively, over existing LTP techniques.

The additional benefit achieved by RDPs is due to the detailed reuse distance information provided by the predictor. As discussed in Section 5.4., while some predictions identify LRU last touches beyond CRD which are likely to be OPT last touches, many predictions identify LRU last touches below CRD that may possibly be LNO last touches. In the latter case, RDPs help by providing the exact reuse distance for marked blocks. Hence, when multiple blocks are marked, the block referenced farthest in the future can be identified directly, instead of using the MRU heuristic discussed in Sections 3.4. and 5.4..

To further understand our RDP result, Figure 11 also reports two ideal cache management algorithms, Oracle-MRU and Oracle-RDP. Oracle-MRU is RD-LTP with perfect last touch information. In Oracle-MRU, LRU last touch blocks are always marked perfectly (like Oracle-CRD in Figure 9). But like RD-LTPs, Oracle-MRU still uses the MRU policy to select a marked block for eviction.[5] Oracle-RDP is RDP with perfect reuse distance information. In Oracle-RDP, LRU last touch blocks are always labeled with their actual reuse distances perfectly. As Figure 11 shows, Oracle-MRU improves upon RD-LTP by 12.5%. This represents the performance lost by RD-LTP due to predictor inaccuracy (*i.e.*, the "Not Predicted" and "Wrong Prediction" components in Figure 7). In addition, Figure 11 also

5. Although Oracle-MRU has perfect last touch information, the MRU policy may still mistakenly evict LNO last touches over OPT last touches. In fact, Oracle-MRU may perform worse than RD-LTP if the additional correct last touch predictions expose more LNO last touches for eviction.

shows Oracle-RDP improves upon Oracle-MRU by 13.8%. This performance difference represents the actual potential benefit of exact reuse distance information. Unfortunately, RDP does not fully achieve this potential, as demonstrated by its 2.7% performance gain over RD-LTP. RDP's inability to achieve its full potential is due to inaccuracies in predicting the exact reuse distance.

7. Conclusion

This paper advances the state-of-the-art in LTP-driven cache management by investigating three novel mechanisms. First, we propose a new signature-based LTP that correlates last touch outcomes with global reuse distance history and the memory instruction's PC. To determine reuse distances, we observe a cache block's position in the LRU stack at reference time. By augmenting the cache with shadow tags, we can also monitor the reuse distances of recently evicted cache blocks. Second, for LTPs, we also advocate selecting the most-recently-used LRU last touch block for eviction. We find an MRU victim selection policy evicts fewer LNO last touches and mispredicted LRU last touches. Our RD-LTP technique employs both of these mechanisms. Our results show that for an 8-way 1 MB L2 cache, a 70 KB RD-LTP can reduce the cache miss rate by 12.6% and 15.8% compared to LvP and AIP, respectively, and by 2.7% compared to DIP. We find RD-LTPs exhibit a much higher prediction rate, predicting 71.2% of the LRU last touches compared to only about 19% for LvP and AIP. We also find RD-LTP's MRU victim selection policy selects a good choice for early eviction.

Finally, we also propose RDPs, a new technique that predicts actual reuse distance values. An RDP is very similar to an RD-LTP except its predictor table stores exact reuse distance values instead of last touch outcomes. Because RDPs predict reuse distances, we can better determine which cache blocks are used farthest in the future, thus distinguishing LNO and OPT last touches more precisely. Our results show an 64 KB RDP can improve the miss rate compared to an RD-LTP by an additional 2.7%.

Acknowledgements

The authors would like to thank Xuanhua Li, Hameed Badawy, Steve Crago, Vida Kianzad, Seungryul Choi, Inseok Choi, Aamer Jaleel, Janice McMahon, Priyanka Rajkhowa, and Meng-Ju Wu for helpful discussions. This research was supported in part by NSF CAREER Award #CCR-0093110, and in part by the Defense Advanced Research Projects Agency (DARPA) through the Department of the Interior National Business Center under grant #NBCH104009. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), or the U.S. Government.

References

[1] L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[2] M. Kharbutli and Y. Solihin, “Counter-Based Cache Replacement Algorithms,” in *Proceedings of the International Conference on Computer Design*, pp. 61–68, October 2005.

[3] A.-C. Lai and B. Falsafi, “Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction,” in *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 139–148, June 2000.

[4] A.-C. Lai, C. Fide, and B. Falsafi, “Dead-Block Prediction and Dead-Block Correlating Prefetchers,” in *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 144–154, June 2001.

[5] W.-F. Lin and S. K. Reinhardt, “Predicting Last-Touch References under Optimal Replacement,” CSE-TR 447-02, University of Michigan, January 2002.

[6] S. Kaxiras, Z. Hu, and M. Martonosi, “Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power,” in *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 240–251, June 2001.

[7] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, “Adaptive insertion policies for high performance caching,” in *Proceedings of the 34th International Symposium on Computer Architecture*, pp. 381–391, June 2007.

[8] W. A. Wong and J.-L. Baer, “Modified LRU Policies for Improving Second-Level Cache Behavior,” in *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pp. 49–60, January 2000.

[9] T. R. Puzak, “Analysis of cache replacement algorithms,” *PhD Thesis*, February 1985.

[10] V. Phalke and B. Gopinath, “An inter-reference gap model for temporal locality in program behavior,” in *SIGMETRICS ’95/PERFORMANCE*, pp. 291–300, May 1995.

[11] H. Zhou, M. C. Toburen, E. Rotenberg, and T. M. Conte, “Adaptive Mode Control: A Static-Power-Efficient Cache Design,” *ACM Transactions on Embedded Computing Systems*, vol. 2, pp. 347–372, August 2003.

[12] G. Chen, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, and M. Wolczko, “Tracking Object Life Cycle for Leakage Energy Optimization,” in *Proceedings of the ISSS/CODES joint conference*, pp. 213–218, October 2003.

[13] D. Burger and T. M. Austin, “The SimpleScalar Tool Set, Version 2.0,” CS TR 1342, University of Wisconsin-Madison, June 1997.

[14] T. Sherwood, E. Perelman, and B. Calder, “Basic block distribution analysis to find periodic behavior and simulation points in applications,” in *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, pp. 3–14, September 2001.

[15] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, “The M5 Simulator: Modeling Networked Systems,” *IEEE Micro*, vol. 26, pp. 52–60, July/August 2006.

[16] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *SIGARCH Comput. Archit. News*, vol. 18, no. 3a, pp. 364–373, 1990.