Last-Touch Correlated Data Streaming

Michael Ferdman and Babak Falsafi

Computer Architecture Laboratory (CALCM)
Carnegie Mellon University, Pittsburgh, PA 15213
{mferdman,babak}@ece.cmu.edu

Abstract

Recent research advocates address-correlating predictors to identify cache block addresses for prefetch. Unfortunately, address-correlating predictors require correlation data storage proportional in size to a program's active memory footprint. As a result, current proposals for this class of predictor are either limited in coverage due to constrained on-chip storage requirements or limited in prediction lookahead due to long off-chip correlation data lookup.

In this paper, we propose Last-Touch Correlated Data Streaming (LT-cords), a practical address-correlating predictor. The key idea of LT-cords is to record correlation data off chip in the order they will be used and stream them into a practically-sized on-chip table shortly before they are needed, thereby obviating the need for scalable on-chip tables and enabling low-latency lookup. We use cycle-accurate simulation of an 8-way out-of-order superscalar processor to show that: (1) LT-cords with 214KB of on-chip storage can achieve the same coverage as a last-touch predictor with unlimited storage, without sacrificing predictor lookahead, and (2) LT-cords improves performance by 60% on average and 385% at best in the benchmarks studied.

1 Introduction

Advances in semiconductor fabrication, along with circuit and microarchitectural innovation, have led to an unprecedented performance gap between microprocessors and memory. Today's processors attempt to reduce this performance gap by incorporating a hierarchy of cache memories. Further expansion of these hierarchies has reached the point of diminishing returns because accesses to distant (e.g., off-chip) hierarchy levels require hundreds of processor cycles, an order of magnitude higher than the lowest on-chip level. Today's wide-issue out-of-order superscalar processors with non-blocking caches can at best tolerate access latencies to nearby cache levels, and incur performance penalties upon long-latency memory accesses.

There is a myriad of techniques proposed to address the memory-access performance bottleneck. A number of proposals [7,14,16,21,22] advocate microarchitectural enhancements to the speculation window and allow instructions to flow speculatively beyond long-latency memory accesses. These techniques rely primarily on accurate control flow and load-value speculation in the presence of pending memory accesses to avoid stalling. However, these enhancements do not benefit pointer-chasing applications with little or no memory-level parallelism.

Some researchers and vendors have explored prefetching to mitigate the processor-memory performance gap. Many proposals are inherently limited in scope, targeting only specific access patterns, such as strided accesses [1,19], pointer dereference [6,8], or accesses to linked data structures [17]. Although frequently accurate for the subset of memory accesses they target, these prefetchers have inherently low coverage overall. More recently, researchers have proposed generalizing stride predictors to target miss sequences that exhibit recurring patterns of (non-constant) strides, an approach called delta correlation. The delta-correlating Global History Buffer (GHB) prefetcher [15] was recently shown to outperform a variety of other hardware prefetching schemes [9]. Although delta correlation subsumes many previous approaches, it is not effective for data structures with irregular access patterns. Furthermore, current delta-correlating prefetchers cannot track repetitive deltas if several individually-correlated access sequences are interleaved.

Another proposed class of prefetchers utilizes address correlation [3,4,10,11,15,20], which promises wider applicability across a diverse spectrum of workloads because they target generalized memory access patterns. Rather than detecting patterns in data layout, these prefetchers correlate data addresses to predict future misses. Ideally, address-correlating prefetchers are able to predict sufficiently early to tolerate long off-chip access latency while predicting a large fraction of cache misses. Unfortunately, no prior design is able to achieve both long lookahead and high predictor coverage. To achieve long lookahead, a predictor requires high-bandwidth on-chip access to correlation data, which precludes large storage. To achieve high coverage, address-correlating predictors require storage proportional in size to a program's memory footprint, which precludes fast prediction.

A recent proposal, the Dead-Block Correlating Prefetcher (DBCP) [12], achieves maximal prefetch lookahead through correlation to "last touch" accesses—the last access to each cache block prior to eviction. However, capacity constraints of DBCP's on-chip correlation table limit its coverage. Conversely, the designs of Solihin et al. [20] and Wenisch et al. [24] record address correlation data in off-chip DRAM. Although these mechanisms have abundant correlation data storage, lookup is performed off chip, drastically increasing prediction latency.

We propose Last-Touch Correlated Data Streaming (LT-cords), the first address-correlating predictor design to combine the prediction timeliness of last-touch on-chip lookup with the high coverage enabled by off-chip predictor storage. The key idea of LT-cords is to record correlation data off chip in the order they will be used, and stream them into a practically-sized on-chip

table shortly before they are needed. Just as prior predictors rely on a repetitive sequence of cache misses, LT-cords depends on a repetitive sequence of predictions. In this paper, we demonstrate that the order in which DBCP-like prediction data are used (the last-touch order) closely matches the order these entries are discovered (the blocks' eviction order).

We use trace-driven and cycle-accurate simulation of an 8-way out-of-order superscalar processor running SPEC CPU2000 and Olden benchmarks to show:

- **Last-touch order is correlated to block eviction order**. The observed sequence of block evictions exactly matches the sequence of last touch accesses for 21% of evictions. Allowing for local reorderings, over 98% of last touch accesses match eviction order, enabling a hardware mechanism that uses block eviction order to approximate last-touch order.

- **High coverage and timely prediction with practical on-chip storage**. LT-cords eliminates 69% of L1D misses using only 214KB of on-chip storage, while increasing off-chip traffic for pin-bandwidth-hungry applications by at most 15% and less than 4% on average. In contrast, DBCP requires on average 80MB of on-chip storage to eliminate the same fraction of L1D misses.

- **Speedup**. LT-cords achieves a performance improvement of 60%, compared to 123% achieved by a perfect L1D. In contrast, the delta-correlating GHB prefetcher achieves only 31% performance improvement. DBCP with 2MB on-chip storage [12] achieves only 17% performance improvement. Finally, increasing L2 cache size by a factor of four achieves only 16% performance improvement.

The rest of this paper is organized as follows. Section 2 provides background on dead-block correlated prefetching. Section 3 investigates the temporal correlation of last touches and how it can be exploited to create sequences of last-touch signatures. Section 4 describes a LT-cords design based on last-touch signature sequences. Section 5 details our evaluation methodology and results. We conclude in Section 6.

2 Background: DBCP Prefetching

Lai et al. proposed Dead-Block Correlated Prefetching [12] to predict the last touch to a cache block prior to that block's eviction, and replace the block by prefetching a subsequently accessed block. By triggering the prefetch at the last touch, DBCP provides maximal lookahead among address-correlating prefetchers.

DBCP correlates each last touch of a cache block to the address of the block that replaces it. Figure 1 depicts an example
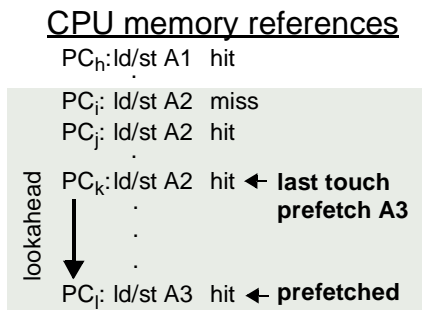

FIGURE 1. Example of dead-block correlated prefetching; last access to A2 triggers a request for its replacement with A3.
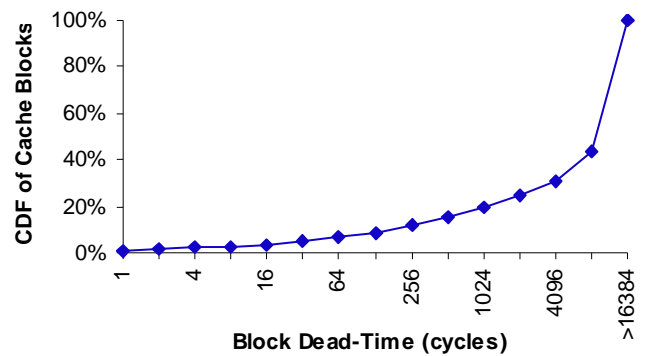

FIGURE 2. Cumulative distribution of dead-times (cycles between last access to a block and its eviction).

of prefetching using a DBCP. We assume a direct-mapped L1D for this example. Cache blocks A1, A2, and A3 all map to the same cache index. The predictor tracks all instructions {PC_i, PC_j, PC_k} accessing block A2 from the miss until A2 is evicted. Upon a miss to block A3, block A2 is evicted, and the predictor records the trace that led to the eviction and replacement address as a fixed-size last-touch signature; the history trace is represented by a hash of the address history (e.g., {A1, A2}) with the instruction trace (e.g., {PC_i, PC_j, PC_k}). On subsequent encounters of the trace {PC_i, PC_j, PC_k} accessing A2, the predictor identifies the access at PC_k as a last touch and initiates a prefetch to retrieve A3 directly into L1D to replace A2.

Because cache blocks remain in the cache for long periods of time after the last access, DBCP attains increased lookahead as compared to conventional prefetcher proposals. Figure 2 shows the cumulative distribution of L1 block dead-times—the number of cycles between a last touch to a block until its eventual eviction.[1] The figure corroborates prior results [12,13,26], showing that over 85% of all cache-block dead-times are longer than the memory access latency. Therefore, prefetch requests issued at last touch can complete before the next access to the same cache index, eliminating the entire off-chip miss latency. The figure also indicates that potential remains in future generation systems, even if memory latencies increase.

The DBCP mechanism correlates data addresses to last-touch accesses, enabling DBCP to increase memory-level parallelism for dependent memory accesses. Modern out-of-order processors often stall, unable to overlap the latency of dependent L1D misses [5]. Correlating miss addresses with last-touch instruction traces enables the predictor to retrieve data-dependent blocks in parallel and enhances memory level parallelism for pointer-dependent traversals (e.g., linked lists or trees).

Unlike delta-correlating prefetchers, DBCP does not require a regular layout to repeat over many memory locations. Instead, like all address-correlating prefetchers, DBCP learns correlations between arbitrary address pairs. Moreover, DBCP can exploit these correlations even if several access sequences are interleaved when the sequences recur.

---

1. Simulation system parameters listed in Table 1. Presented dead-times are an average across benchmarks listed in Table 2.
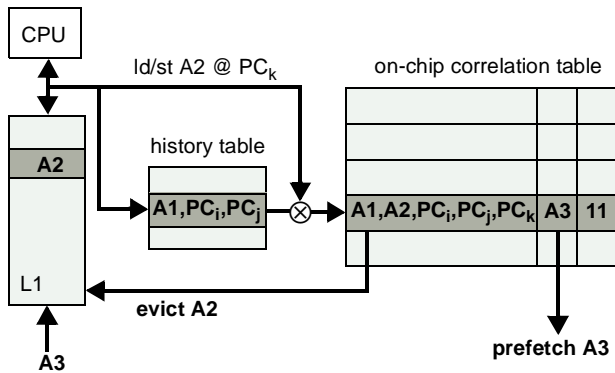
FIGURE 3. Previously-proposed DBCP hardware.

FIGURE 4. Sensitivity of DBCP to on-chip correlation table size, normalized to DBCP with unlimited storage.

Finally, DBCP prefetches blocks directly into the cache rather than into an auxiliary prefetch buffer. Many prefetchers deliver data into prefetch buffers to avoid cache pollution [19]. However, due to lookup complexity and size restrictions, prefetch buffers limit the amount of speculatively fetched data and can affect the cache's critical access path. By accurately predicting dead blocks in L1D, DBCP can place prefetched blocks directly in the cache without incurring cache pollution.

2.1 Why is DBCP Impractical?

Figure 3 depicts the anatomy of DBCP [12]. On every memory access, DBCP computes a last-touch history trace by combining the PC and address of that access with history table data. History traces for previously observed replacements are stored in the on-chip correlation table. If the history trace for an access is found in the correlation table, a request for the corresponding replacement block is issued.

Although address-correlating prefetchers like DBCP show great promise, they are impractical to implement because they require on-chip storage proportional in size to the number of blocks referenced by the application. To prefetch effectively, DBCP and similar address-correlating predictors must store correlation data across long (e.g., billions of instructions) recurring program phases [18] with little temporal reuse. Because last-touch signatures are calculated and looked up on every L1D access, the correlation table must reside on chip to provide high lookup bandwidth. Figure 4 plots the average prefetch coverage (fraction of L1D misses eliminated) of DBCP with a finite-size correlation table, normalized to coverage of DBCP with an unlimited-size correlation table; worst-case benchmark (wupwise) is shown in light gray. To realize its full potential, DBCP needs a 160MB correlation table and in the worst-case achieves negligible coverage with less than 80MB. Our results corroborate prior work showing that DBCP with practically-sized storage is ineffective [9].

3 LT-cords

In this paper, we propose Last-Touch Correlated Data Streaming (LT-cords), a practical address-correlating predictor. LT-cords exploits the inherent temporal address correlation of data accesses to enable accurate and timely streaming of data into the L1D cache. The key innovation of LT-cords is in using off-chip storage to record long repetitive sequences of consecutively-used last-touch signatures, while retaining only a single "head" signature per sequence on c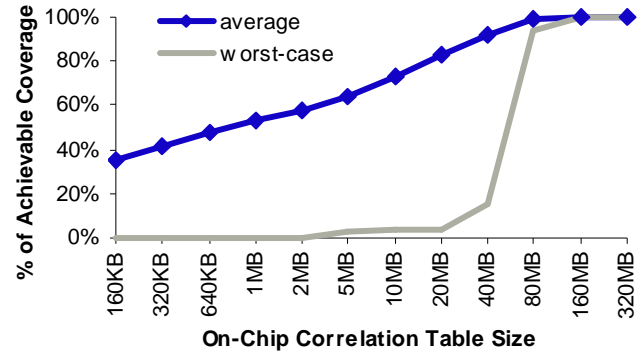hip, thus drastically reducing the on-chip storage requirements of the predictor. When an access sequence recurs, LT-cords streams the corresponding sequence of last-touch signatures from off chip into a practically-sized on-chip table. Large last-touch prediction lookahead enables timely signature retrieval from off-chip, while contiguous packing of signatures in off-chip memory enables bandwidth-efficient retrieval.

3.1 LT-cords Example

To illustrate the operation of LT-cords, we consider an application that executes an outer loop many times to perform computation on a large data set. Thousands of L1D cache misses are encountered during the execution of each loop iteration. Because data structures remain mostly static throughout program execution, similar access sequences repeat during each loop iteration. Hence, each iteration repeats the same sequence of cache misses.

When LT-cords first encounters a particular cache miss sequence during an early loop iteration, it learns the entire sequence of last-touch signatures that corresponds to these misses. LT-cords stores the head signature of the sequence on chip, and stores the remainder of the sequence in off-chip DRAM. When the processor issues a memory reference matching the on-chip head signature (corresponding to the start of another loop iteration), LT-cords retrieves the beginning of the signature sequence from off-chip memory into an on-chip table that functions like the correlation table in DBCP. As in DBCP, signatures that recur in the program trigger prefetches into the L1 cache. In addition, as signatures in the on-chip table are used, LT-cords replaces them with the subsequent signatures from off-chip memory. In this fashion, LT-cords keeps just the necessary portion of the signature sequence on chip.

3.2 Order Disparity

At the time of prediction, LT-cords requires last-touch signatures to be made available on chip in the order that the signatures are to be used by the predictor. However, the mechanism that discovers last-touch signatures and records sequences off chip must do so in cache-miss order. Although the last touch and cache miss order are generally similar, localized reordering is abundant in typical access patterns (see Section 5.1). For example, consider the sequence of references to addresses {A1,B1,B2,A2} made to cache indices A and B in a direct-mapped cache. Accesses A1 and B1 are last touches prior to those blocks being evicted from the cache, and A2 and B2 are the corresponding misses that cause the
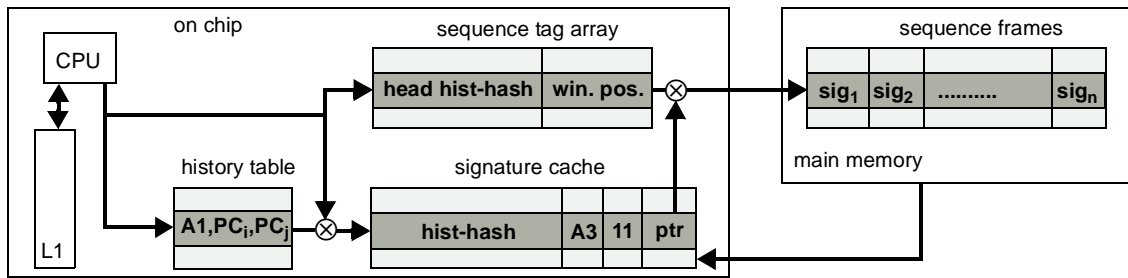
FIGURE 5. The anatomy of a LT-cords implementation.

two evictions. Although the cache miss to address B2 happens before the miss to A2, the corresponding last touches, A1 and B1, occur in the reverse order. Therefore, the sequence discovered by LT-cords {B2,A2} does not match the last-touch order {A1,B1} in which predictions must be made. LT-cords must tolerate order disparity between the recorded sequence and last-touch sequence to predict successfully.

Control-flow irregularities and data structure modification can cause last-touch signatures in the middle of previously-recorded sequences to become stale (no longer valid). Stale signatures in a sequence can prevent LT-cords from accurately tracking the sequence to its end. Avoiding coverage loss from stale signatures in the middle of a sequence requires a sequence tracking mechanism that supports skipping stale signatures.

To tolerate reordering and stale signatures, previous temporal streaming designs [24] temporarily buffer data in a small fully-associative structure (e.g., 32 entries). LT-cords requires a similar mechanism to allow on-chip random access to signatures to tolerate order disparity. However, unlike temporal streaming, LT-cords typically follows several signature sequences in parallel. Furthermore, the disparity between last-touch and cache miss order is larger than the reorderings typically observed in temporal streams. As such, LT-cords must buffer many signatures on chip (upwards of 1000 signatures, see Section 5.2) which precludes using a fully-associative structure. Instead, LT-cords places signatures from simultaneous sequences into a set-associative on-chip table. In Section 5.4, we present results indicating that, in practice, tolerating reorder with a moderately-sized (e.g., ~200KB) signature cache enables accurate tracking of last-touch sequences recorded in cache miss order.

3.3 Lookahead for Sequence Retrieval

Cache misses are frequently clustered [12], corresponding to bursts of last touches. During such bursts, last touches may remain undetected by LT-cords if their corresponding last-touch signatures have not yet arrived from off-chip memory. To avoid losing prediction opportunity due to long off-chip access latency, LT-cords must initiate the retrieval of signatures from off chip sufficiently early to bring the signatures on chip before their corresponding last touches take place. Furthermore, because predictions must take place during signature retrieval, a LT-cords implementation must maintain enough signatures in the on-chip table to overlap long off-chip signature retrieval latency.

4 LT-cords Implementation

Figure 5 depicts the anatomy of a LT-cords implementation. LT-cords comprises three on-chip hardware structures and a section of main memory for off-chip signature sequence storage. The on-chip structures include: (1) a history table, organized like the L1D tag array, that maintains last-touch history trace encodings and previous tags for each L1D cache index, (2) a sequence tag array, which tracks the contents of the off-chip sequence storage, and (3) a signature cache, a small set-associative correlation table that temporarily holds signatures on chip for prediction. The signature cache must store enough signatures to tolerate signature reordering and overlap signature retrieval latency for all active sequences. We assume that the sequence storage in main memory is managed by the operating system. Due to the long training time of the predictor (potentially billions of instructions), contents of LT-cords structures must persist across context switches

4.1 Recording Last-Touch Signatures

As in DBCP [12], LT-cords constructs signatures using the history table. Each history table entry maintains a PC trace of committed memory instructions that access the corresponding L1D set. The trace is incrementally constructed from program counter values of all instructions that access the set and is reset on every eviction from the set. The history table entry also contains the tags of the last two blocks evicted from the set. Upon an eviction, LT-cords constructs a last-touch signature from the PC trace hash, the previous and evicted tags, a 2-bit confidence counter, and the replacement address. Constructed signatures are stored off chip in eviction order. To optimize off-chip write bandwidth, LT-cords buffers a small number of consecutive signatures and transfers them as a single unit.

4.2 Indexing Off-Chip Sequence Storage

The signature sequences that LT-cords exploits are arbitrarily long. To manage main memory sequence storage, LT-cords divides sequences into fixed-length fragments, each holding a sub-sequence of consecutive last-touch signatures. As the application follows an access sequence, the signatures from corresponding fragments are streamed from off chip into the signature cache. To predict when a particular fragment will be needed, LT-cords associates each fragment with a signature, called the head signature, that precedes the fragment in the signature sequence. When the head signature recurs, LT-cords begins retrieving the corresponding fragment. Because of the long latency required to access off-chip signatures, the head signature must precede the fragment by several hundred signatures (see Section 3.3).

TABLE 1. System configuration.

Processor		Caches	
Clock rate	4 GHz	L1 D	64KB, 64-byte line
Issue/retire	8 instructions/cycle		2-way, 2-cycle
Reorder buffer	256 entries	L1 I	64KB, 64-byte line
Load/store queue	128 entries		4-way, 2-cycle
Branch predictor	8K/8K hybrid	L1 D ports	4
	12-cycle penalty	L1 D MSHRs	64
Functional Units	8 IALU (1)	L2 (unified)	1MB, 8-way
(latencies)	2 FALU (2)		20-cycle
	2 I-MUL/DIV (3/19)	L2 ports	1
	2 F-MUL/DIV (4/12)	L1/L2 bus	1-cycle request
	all pipelined except		32 byte per-cycle
	IDIV and FDIV	TLB	256 entry, 4-way
			600-cycle miss
Predictors		Memory	
DBCP	2MB correlation	Size	1GB (30-bit space)
	table, 18-cycle	Latency	200 cycles first 32B
GHB	PC/DC, 4-deep		3 cycles each 32B
	256-entry IT	Bus	32-byte wide
	256-entry GHB		1333 MHz

TABLE 2. Benchmarks, base miss rates and IPCs.

	L1miss%	L2miss%	IPC		L1miss%	L2miss%	IPC
ammp	15	24	1.07	**gcc**	38	3	2.71
applu	34	68	1.53	**gzip**	5	2	1.55
apsi	6	16	2.69	**lucas**	44	67	1.25
art	60	63	0.72	**mcf**	53	67	0.08
bh*	7	94	0.67	**mesa**	2	25	3.76
bzip2	4	21	1.56	**mgrid**	18	49	1.56
crafty	0	2	2.24	**parser**	6	17	1.14
em3d*	67	87	0.50	**perlbmk**	2	14	1.58
eon	0	0	1.94	**sixtrack**	1	74	4.29
equake	31	85	0.68	**swim**	49	59	1.18
facerec	22	42	2.04	**treeadd***	5	92	0.24
fma3d	11	62	1.74	**twolf**	15	12	0.84
galgel	17	16	3.13	**vortex**	4	16	3.11
gap	2	54	1.07	**wupwise**	9	72	2.66

Together, the sequence tag array and main-memory sequence storage act as a direct mapped cache of sequence fragments. LT-cords divides the main-memory sequence storage into frames that each store a sequence fragment. Fragments map to frames based on the low-order bits of the head signature. The sequence tag array stores the head hash of the fragment in each frame.

No explicit sequence start or stop criteria exists; LT-cords continues to record the sequence of last-touch signatures off chip as long as cache misses occur. LT-cords appends signatures to a fragment until the frame becomes full. Then, a new frame is allocated for the subsequent fragment. As in a direct mapped cache, an existing fragment is overwritten by a new fragment that maps to the same frame.

4.3 Last-Touch Prediction and Sequences

LT-cords uses the signature cache to predict addresses for prefetch. As memory accesses commit, signatures stored in the history table are updated. LT-cords looks for each updated signature in the on-chip signature cache. Presence of the corresponding signature with a high confidence count indicates that the memory access is a last touch. LT-cords retrieves the data at the predicted address; when it arrives, data are placed into the L1D.

LT-cords must buffer enough signatures in the signature cache to tolerate reordering (see Section 3.2). However, to efficiently utilize signature cache capacity and allow space for several simultaneously active fragments, LT-cords cannot load fragments in their entirety at once; fragment size is chosen to optimize storage efficiency and greatly exceeds the number of signatures required to tolerate reordering.

Instead, LT-cords keeps a sliding window of signatures from each active fragment in the signature cache. As signatures are used, the sliding window is advanced (win. pos. in the sequence tag array of Figure 5) and new signatures are streamed into the signature cache. The signature cache is organized as a set-associative structure with signatures replaced in FIFO order. Along with the signature, signature cache entries store a pointer to the location of the signature in the off-chip sequence storage. When a signature is accessed, LT-cords uses this pointer to identify the signature's frame, advance its corresponding fragment's sliding window, and load additional signatures. The location of a frag-

ment's sliding window is tracked in the sequence tag array entry for the fragment. To utilize bandwidth efficiently, LT-cords advances the window and transfers signatures in the same size units used to record sequences (see Section 4.1).

4.4 Updating the Signature Confidence

LT-cords uses 2-bit saturating confidence counters for each last-touch signature to avoid premature eviction of L1D cache blocks by signatures that become invalid. The pointer kept along with each signature in the signature cache provides an exact location of the signature in off-chip sequence storage, allowing for a direct update of the counter value on confidence changes. Confidence updates are not frequent, and incur minimal overhead by utilizing otherwise-unused bus cycles. Because most signatures are valid immediately after creation, confidence counters are initialized to the value "2" to expedite training.

5 Results

We use SimpleScalar 3.0/Alpha for both trace-driven and cycle-accurate simulation of an aggressive out-of-order processor and cache hierarchy. Table 1 specifies the simulated system configuration. We extend SimpleScalar to model MSHR contention and queuing accurately at both the L1/L2 and L2/memory busses. We model two channels between the L1 and L2, allowing for an L2 request to be issued while an L1 fill is in progress.

Results in Section 5.1 through Section 5.6 are derived from trace-driven simulation of each benchmark in its entirety. Speedup and bandwidth (Sections 5.7 and 5.8) are obtained by cycle-accurate simulation using SMARTS statistical sampling and checkpointing [25]. We use trace-driven simulation to create many evenly spaced checkpoints of the cache hierarchy and predictor state for each application. We launch cycle-accurate simulation from each checkpoint, and aggregate the results. For each checkpoint, we measure 10M instruction regions; prior to measurement of each region, we perform 10M instruction warm-up. Sample sizes are chosen to maintain a 95% confidence interval of ±3% on performance change (except for results with greater than 50% performance improvement, where the confidence interval is less than one tenth of the performance improvement).

For trace-driven results, we use 32-bit last-touch signatures to minimize the effects of hash collisions. In cycle-accurate tim-
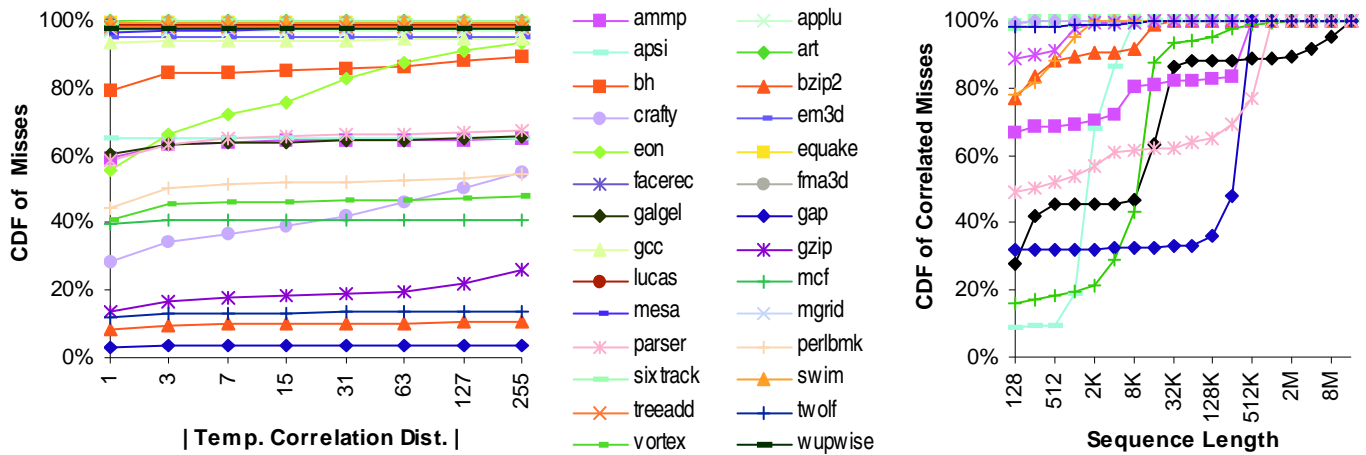
FIGURE 6. Absolute temporal correlation distance of all cache misses; correlation distance of 1 implies perfect repetition (left). Lengths of sequences of temporally correlated misses in applications that exhibit more than 5% uncorrelated misses (right).

ing simulations, all DBCP and LT-cords requests are placed into a 128-entry circular queue. When the request queue is full, new requests replace old (unissued) ones at the queue head. Requests are only issued when the L1/L2 bus is free.

Table 2 lists our benchmarks and their L1 and L2 miss rates and IPCs for our baseline processor configuration. We run all SPEC CPU2000 benchmarks except vpr; SimpleScalar 3.0/Alpha is unable to correctly execute vpr. The first reference input is used for all benchmarks except perl, for which the "splitmail" input is used. We include results for three pointer-intensive Olden [2] benchmarks (indicated by * in Table 2) because they represent memory intensive applications with access patterns that are not amenable to simple address predictors (e.g., stride predictors).

5.1 Temporal Correlation Opportunity

We begin our evaluation by quantifying the temporal correlation of L1D cache misses. We investigate the presence of correlation by determining the degree of reordering between each pair of consecutive cache misses and the previous occurrence of the same two misses. We employ a Temporal Correlation Distance metric similar to that introduced in [24]. Temporal correlation distance between two consecutive misses is the distance between the previous occurrence of the same two misses in the sequence of all cache misses.[1] A temporal correlation distance of +1 implies perfect correlation, where the two most recent occurrences of a pair of consecutive misses appeared in exactly the same order; a distance of -1 corresponds to a reversal of the two misses compared to their previous occurrence, as in the sequence {A,B,...,B,A}.

Figure 6 (left) shows absolute temporal correlation distances as a cumulative distribution of all cache misses. The figure indicates that many applications (15 out of 28) exhibit nearly perfect temporal correlation, with most cache misses repeating in exactly the same order. Of the remaining applications with imperfect correlation, many present significant opportunity for LT-cords.

---

1. We define a cache miss as a point in time when a cache replacement occurs. In our trace study, we label cache misses with the tuple (miss PC, miss block address, evicted block address). The previous occurrence of a miss is the nearest preceding miss with the same label.

Ammp, apsi, galgel, and parser contain approximately 60% perfectly correlated misses; mcf, perlbmk, and vortex contain over 40%. Not surprisingly, we note that gzip, bzip2, and twolf, applications that rely heavily on hashed or randomized memory accesses, exhibit little temporal correlation.

Figure 6 (left) shows strong pair-wise correlation of cache misses (very high percentage of miss pairs have +1 correlation distance), suggesting that repetitive sequences of cache misses exist. However, the temporal correlation distance metric says little about the length of the existing sequences. We measured the lengths of sequences with temporal correlation distances up to ±16. Our results show that benchmarks exhibiting strong temporal correlation have long sequences, ranging from over 2K misses for apsi to 16M misses for fma3d. For applications that exhibit imperfect correlation, Figure 6 (right) presents the cumulative distribution of correlated misses, grouped by sequence length.

The plot shows that even for a narrow temporal correlation distance range (±16), a large fraction of correlated cache misses belong to long sequences; for example, 80% of mcf's correlated cache misses belong to sequences longer than 2K, and over 30% of ammp's correlated misses are found in sequences greater than 4K in length. In practice, LT-cords tolerates a greater degree of reordering than ±16, potentially enabling even greater coverage.
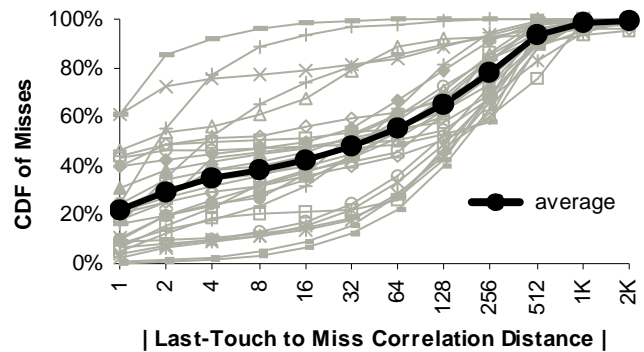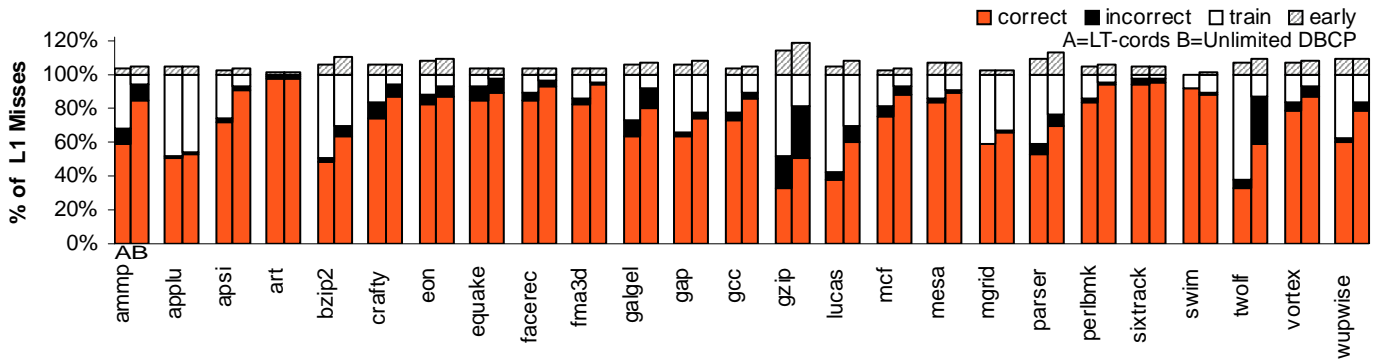

FIGURE 7. Last-Touch to Cache Miss correlation, plotted as a cumulative percentage of all misses up to a given absolute (negative or positive) correlation distance.

FIGURE 8. Comparing coverage and accuracy of LT-cords to DBCP with unlimited storage.

5.2 Cache Miss vs Last-Touch Ordering

LT-cords stores last-touch signatures in the order they are discovered, the order of cache misses. However, when predicting, LT-cords follows these sequences in last-touch order. Using the temporal correlation distance metric from Section 5.1, we present the degree of reordering that LT-cords must tolerate to remain effective. We report the difference between the sequences of last touches and corresponding cache misses. Correlation distance of +1 indicates that cache misses corresponding to consecutive last touches appear in the same order, with no intervening misses.

Figure 7 shows that, on average, only 21% of cache misses are perfectly correlated with the last touches that precede them. Although this is a significant fraction of misses, this result indicates that LT-cords must tolerate reordering to achieve maximum potential. Figure 7 shows that LT-cords must hold on chip up to 1K signatures per sequence to tolerate reordering in over 98% of cache misses.

5.3 LT-cords Coverage and Accuracy

In this section, we demonstrate the ability of LT-cords with realistic on-chip storage to approximate an "oracle" correlation table. We configure LT-cords parameters based on the sensitivity study presented in Section 5.4; LT-cords is compared to a Dead-Block Correlating Prefetcher with an unlimited-capacity correlation table.

Figure 8 expresses results as percentages of the prediction opportunity—the total number of all L1D cache misses that would occur without a predictor. *Correct* represents eliminated cache misses, *incorrect* represents mispredicted replacement addresses, and *train* represents the fraction of cache misses not predicted due to training or low confidence. Cache misses that form prediction opportunity are either eliminated (correct), or not (incorrect and train); these percentages add up to 100%. *Early* represents premature evictions induced by the predictor, expressed above 100% as a percentage of the base case misses.

We compare LT-cords to a DBCP with unlimited-capacity storage to present an upper bound for prediction opportunity. Figure 8 indicates that, for most of the applications studied, LT-cords exhibits coverage and accuracy that closely track an oracle predictor.

The most pronounced differences in coverage are in applications that have a large fraction of uncorrelated cache misses. Imperfect correlation in these applications (ammp, apsi, bzip2, gzip, parser, and twolf) diminishes the ability to bring last-touch

signatures on-chip. For example, apsi exhibits sequences of hundreds to thousands of last touches that do not recur. Non-repetitive signatures pollute sequences, allowing successful tracking of only short correlated sequences (Figure 6).

5.4 Storage Size Sensitivity

We first examine predictor sensitivity to signature cache size. We conduct an experiment with an unlimited number of 512-signature sequence fragments in off-chip memory. To reduce bias due to signature conflict at small signature cache sizes, we use an 8-way set associative signature cache. Figure 9 depicts normalized LT-cords coverage as a function of signature cache size.

Ideally, only one in-progress sequence is necessary. In practice, dynamic control flow and data structures result in multiple parallel sequences. Multiple sequences introduce a larger number of signatures that the cache must hold and require the cache size to account for conflicts. Based on results in Figure 9, we estimate that a signature cache with 32K signatures is sufficient for our benchmarks, enabling approximately 20 simultaneously active sequences with up to ±1024 reordering and necessary sequence retrieval lookahead. At this size, 2-way associativity is sufficient.

Figure 10 presents a breakdown of the necessary off-chip sequence storage size for the benchmarks with the largest storage requirements. The results show predictor coverage improvement up to 32M signatures for many applications. With 32M signatures, we found minimal sensitivity to fragment size up to 8K signatures (on average less than 2% decrease in LT-cords coverage). We therefore select this sequence fragment size to minimize the on-chip storage requirements of the sequence tag array.
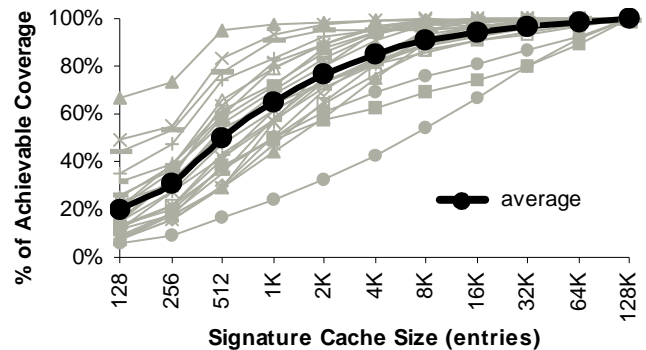

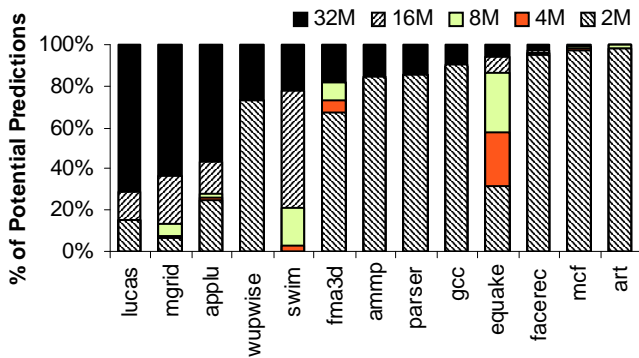**FIGURE 9. Coverage sensitivity to signature cache size.**

FIGURE 10. Off-chip sequence storage size (in millions of signatures) needed to achieve given coverage.
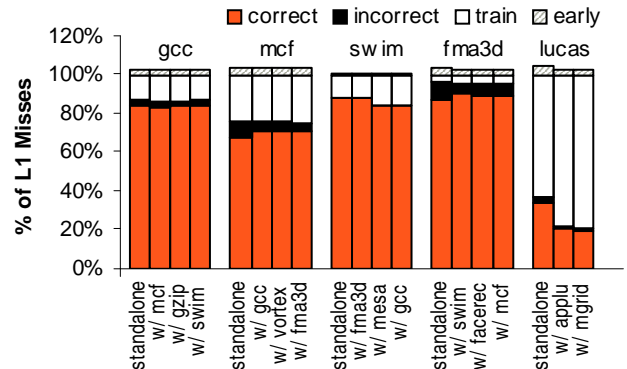

FIGURE 11. LT-cords coverage in a multi-programmed environment. The left bar shows standalone application; remaining bars show the application running with one other application.

The results of these experiments further stress that on-chip tables are insufficient for last-touch signature storage, even for applications with small active memory footprints. Applications such as facerec, mcf, and art, which have the smallest storage requirements, still require approximately 2M signatures (10MB of storage, 5 bytes per signature), exceeding reasonable on-chip structure sizes. For lucas, mgrid, and applu, less than half of the coverage is attainable with 16M sequence storage, and full predictor potential is only possible with 32M signatures (over 150MB of storage, comparable to application memory footprint). These widely-varying and workload-dependent storage requirements suggest that LT-cords storage is best managed by the operating system, to allow scaling predictor DRAM usage based on the application's needs.

5.5 Multi-Programmed Environments

LT-cords records miss sequences that span billions of instructions and therefore must be preserved across context switches. Consequently, both the on-chip and off-chip storage structures must be shared among multiple contexts. In this section, we investigate the effect of multiple programs utilizing shared LT-cords structures by simulating a multi-programmed environment. We alternate execution between pairs of benchmarks, mimicking context-switches. To estimate appropriate duration of execution, we assume IPC of 1.5 for integer applications, and 3.0 for floating point applications, resulting in respective simulated quantums of 60M and 120M instructions with a 4GHz clock. The addresses accessed by one application in each pair were shifted to simulate non-overlapping physical address ranges. We present results from simulating the first 60 context switches.

Figure 11 presents LT-cords coverage results for applications simulated on their own and in a multi-programmed environment with one other application. The benchmarks were selected as a representative subset of integer and floating point applications with comparatively high and low LT-cords coverages.

The results indicate that as long as predictor state is preserved across context switches, and off-chip sequence storage has ample space for recording sequences, the effect of multiple, simultaneously running programs on LT-cords coverage is negligible. On the other hand, pairings of lucas with applu and mgrid demonstrate the effect of insufficient combined sequence storage.

5.6 Cycle-Accurate Simulation Parameters

Based on our sensitivity analyses, for cycle-accurate timing results we simulate LT-cords with 160MB of off-chip sequence storage, a 204KB signature cache, and a 10KB sequence tag array. The sequence storage is partitioned into 4K frames, each frame holding a fragment of 8K last-touch signatures. Each last-touch signature consists of a 23-bit last-touch history trace, a 2-bit confidence counter, and 15-bit prediction address tag. The signature cache is organized as a 2-way set-associative structure, indexed by the low-order 14 bits and tagged by the 9 high-order bits of the last-touch signatures. Each signature cache entry is 42-bits, consisting of the 15-bit prediction address tag, 2-bit confidence counter, and 25-bit pointer to itself (addressing 32M signatures) in off-chip storage.

5.7 Speedup

We first evaluate LT-cords performance against a baseline processor augmented with a perfect L1D cache. Table 3 presents the percent performance improvement of all benchmarks over the baseline configuration of Table 1. On average, we see 123% performance improvement opportunity if all accesses hit in L1D. LT-cords is able to achieve a large fraction of this opportunity, attaining 60% average performance improvement across all applications. LT-cords achieves speedups that are roughly proportional to its coverage for the benchmarks with significant performance opportunity (e.g., 242% of the possible 338% for swim). However, in several cases, LT-cords speedup is lower than trace coverage suggests. For these benchmarks, a significant fraction of the speedup opportunity arises from block accesses that hit in L2 that are partially hidden by the out-of-order core. LT-cords correctly identifies and initiates fetches for these blocks, but only a small number of cycles ahead of the demand-fetch.

Table 3 compares LT-cords performance with the program counter / delta correlation variant of the Global History Buffer (GHB PC/DC, subsumes stride prefetching), a realistic DBCP implementation, and a baseline processor with a larger L2 cache. GHB uses 256-entry index and history tables, as recommended for SPEC applications [9,15]. The realistic DBCP is implemented with a 2MB on-chip correlation table as in [12]. For comparison with a larger L2, we quadruple the size of the L2 cache of the

TABLE 3. Performance comparison. Each value indicates percent performance improvement over the baseline processor configuration**.**

	bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	*SPECint mean*	bh	em3d	treeadd	*Olden mean*
Perfect L1	43	3	1	65	29	17	1637	67	31	89	54	*73*	262	439	266	*315*
LT-cords	4	1	0	0	22	0	385	15	3	0	3	*20*	206	247	224	*225*
GHB	6	0	0	46	5	0	143	22	7	-8	0	*15*	2	33	179	*56*
DBCP	0	0	0	0	6	0	465	2	4	0	3	*19*	153	0	0	*36*
4MB L2	22	0	0	1	7	0	245	28	5	56	1	*23*	8	12	0	*7*

	ammp	applu	apsi	art	equake	facerec	fma3d	galgel	lucas	mesa	mgrid	sixtrack	swim	wupwise	*SPECfp mean*	*overall*
Perfect L1	212	162	26	301	470	141	155	67	211	9	156	10	338	93	*139*	*123*
LT-cords	95	39	9	197	267	76	108	31	27	3	88	3	242	40	*71*	*60*
GHB	46	40	2	16	113	60	65	16	49	2	114	0	43	51	*40*	*31*
DBCP	100	0	0	24	0	58	0	16	0	1	0	7	0	0	*12*	*17*
4MB L2	22	4	0	91	2	56	0	47	0	0	1	1	0	0	*13*	*16*

baseline processor to 4MB, conservatively assuming the same access latency as the base 1MB cache.

We corroborate prior results [9,15] showing that GHB, an advanced stride/delta correlating predictor, can eliminate a large fraction of L2 cache misses in many applications, attaining on average 31% performance improvement across the applications we studied. Delta correlation is effective when the data layout is regular and accesses to distinct addresses follow a repeating pattern. For example, many SPECfp applications include array accesses that GHB can capture. Delta correlation can also be effective for pointer-intensive data structures if systematic heap allocation results in a regular layout (e.g., as in treeadd).

LT-cords can also predict the accesses captured by delta correlation if the same addresses are revisited more than once. In addition, LT-cords can predict repetitive accesses that do not follow a regular pattern, common in pointer-chasing accesses (e.g., in bh and em3d). Hence, LT-cords eliminates on average 59% of off-chip misses, while GHB eliminates 26%. However, delta correlation can outperform address correlation for applications with regular data layouts, but little data reuse (e.g., gap).

In addition to its off-chip coverage advantage, LT-cords further outperforms GHB by reducing the stalls associated with dependent chains of L1D misses that hit in L2. Unlike GHB, LT-cords is able to prefetch directly into L1D without pollution because last-touch prediction ensures that useful data is not replaced. Furthermore, to achieve substantial coverage, GHB must be highly aggressive, fetching many blocks that are never accessed. GHB's aggressiveness can lead to significant bandwidth and L2 contention, degrading performance (e.g., as in twolf).

LT-cords also outperforms address-correlating predictors that use on-chip predictor storage. DBCP coverage is restricted by its limited on-chip signature storage, achieving an average performance improvement of only 17%. DBCP is able to achieve speedup on benchmarks with small memory footprints (e.g., bh) and benchmarks with large memory footprints but small working sets (e.g. mcf). Although both techniques use the same prediction signatures, DBCP marginally outperforms LT-cords in mcf and ammp because DBCP's on-chip signature lookup is always timely. However, the majority of applications with substantial memory system improvement opportunity incur misses to many distinct addresses and actively utilize most allocated memory. The signature storage requirements for these applications therefore scale with the application footprint. LT-cords allows the entire set of signatures to fit into its off-chip sequence storage, enabling LT-cords to achieve four times the average DPCP speedup.

Finally, LT-cords offers superior speedup compared to quadrupling the base system's L2 cache size. A 4MB L2 cache cannot mitigate the impacts of: (1) an application's working set that is larger than 4MB, or (2) dependent chains of L1D misses, even if L2 accesses hit. The larger cache achieves a 16% performance improvement on average, mainly in applications with small working sets or non-repetitive (hash lookup) access patterns (e.g., bzip2, twolf). For most applications with substantial memory system opportunity, LT-cords is therefore significantly more effective in reducing the processor's memory stall cycles than a larger L2.

Many SPEC benchmarks exhibit little sensitivity to memory system improvements; they are included only for completeness. These benchmarks do not benefit significantly from any prediction technique. In these cases, LT-cords attains only slight performance improvements (3% of possible 10% for sixtrack, 3% of 9% for mesa, and 9% of 26% for apsi). LT-cords does not adversely affect performance of these benchmarks.

5.8 Memory Bandwidth Overhead

To be effective, LT-cords must operate with practical memory bandwidth demands. Intuitively, LT-cords bus overhead is inversely proportional to the program's off-chip data traffic. LT-cords transfers, on average, one signature from off-chip storage for every L1D miss. When the L2 miss rate is high, LT-cords overhead constitutes a small fraction of total bus utilization; the 5-byte LT-cords signature is small relative to the cache block transferred as a result of the cache miss. Conversely, when L2 miss rate is low, LT-cords traffic constitutes a significant fraction of bus bandwidth. However, in this case, LT-cords overhead is unimportant, because a low L2 miss rate implies an under-utilized bus.

Figure 12 presents the bus utilization of both the base system and LT-cords overhead for all benchmarks. Bus utilization is expressed as average bytes per instruction to ensure a meaningful comparison that is independent of the application runtimes. Per-
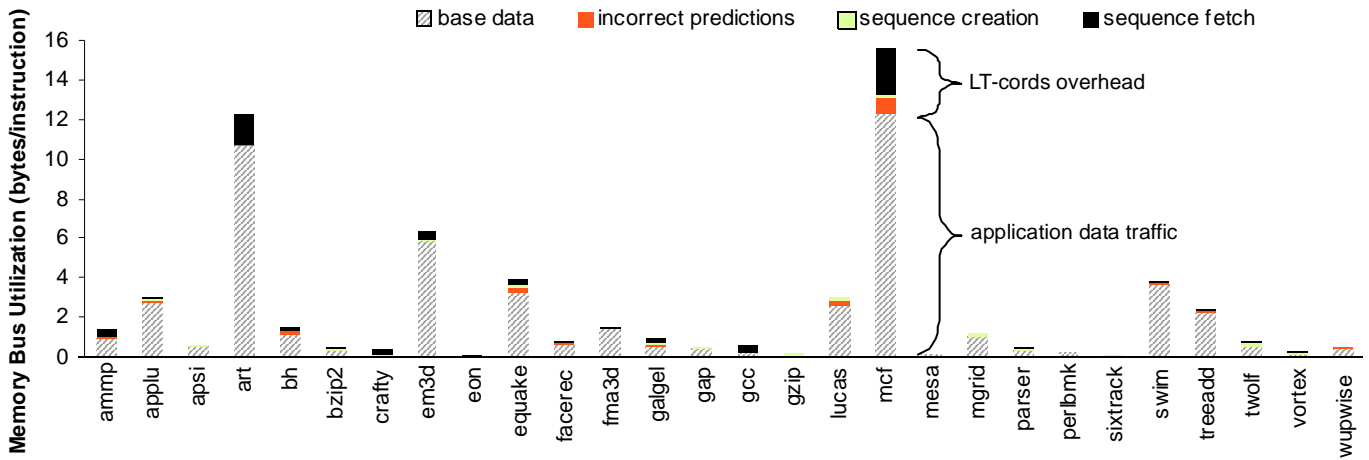
FIGURE 12. LT-cords memory system utilization, normalized to bytes/instruction to remove effect of application speedup.

application bus utilization is broken down into four key components. *Base data* are cache block transfers (including speculative loads) made by the base system without a predictor. *Incorrect predictions* are extraneous cache block transfers initiated by LT-cords as a result of misprediction. *Sequence creation* is the LT-cords memory system utilization for writing of off-chip signature sequences and confidence counter updates. *Sequence fetch* is the traffic overhead introduced by LT-cords retrieving last-touch signatures from off-chip sequence storage. The memory utilization results account for bus request and data transfer cycles.

With the exception of four benchmarks, the average memory bus bandwidth overhead of LT-cords is small: 17% for applications that exceed 1 byte per instruction off-chip traffic. Ammp, art, gcc, and mcf experience significant traffic overhead from incorrect predictions. For these applications, LT-cords incorrectly predicts up to 12.5% of the replacement addresses. Incorrect predictions do not directly degrade application performance (no new cache misses are introduced); however, an increase in memory system utilization is observed. Despite higher bus utilization, these applications experience over 22% performance improvement with LT-cords (385% in the case of mcf), indicating that the overhead of bringing signatures on chip is outweighed by LT-cords' ability to reduce CPU stalls from cache misses.

In many situations, LT-cords can parallelize dependent cache misses that must be incurred in series in a demand-fetch system. Consequently, some benchmarks (applu, art, swim) begin to approach the peak bandwidth available in the system, becoming memory bandwidth-bound and not latency-bound.

5.9 Power

Although the LT-cords on-chip storage structures are larger than the L1D and are accessed as frequently, LT-cords implementation should contribute significantly less to the chip power dissipation than the L1D. First, despite lookup on every L1D access, the signature cache and sequence tag array read out data less frequently than the L1D cache. LT-cords structures contain signatures only for cache misses. It therefore follows that the majority of accesses to LT-cords structures require only a tag check and not a data read operation. By employing a serial lookup policy in the implementation of the LT-cords structures, less energy must be spent for performing data reads compared to the L1 data cache.

Next, the actual data width of the LT-cords structures is substantially narrower than the L1D. To exploit spatial locality, a cache line will contain on the order of 512 bits, of which only a single word must be selected and read or written. Conversely, each sequence tag array entry contains only a single counter and each signature cache entry is only 42 bits. The narrower datapath of the LT-cords structures therefore allows for lower read and write energy compared to the L1D.

To get a sense for the relative contribution of LT-cords compared to the L1D of our architecture, we used CACTI 4.2 [23] to estimate the energy of the LT-cords storage structures and an L1D-like 64KB cache in a 70nm technology. CACTI estimates 18pJ dynamic energy to read a cache block from the data array of the L1D-like fast cache. Due to considerably narrower width of the data array and lack of selection logic, signature read energy is estimated at below 6pJ, despite the larger size of this structure.

The L1D tag and data array accesses must be performed in parallel to minimize access latency. As a result, total dynamic tag lookup and data read energy of a fast four-port L1D-like cache is approximately 73pJ, while using cache structures with serial lookup for the sequence tag array and signature cache results in a combined energy of 30pJ and only infrequently, once per L1D miss, incurs an additional 6.5pJ to read signature data. Conservatively estimating a 20% L1D cache miss rate, the average power dissipation of LT-cords structures is about 48% of L1D dissipation if implemented in the same technology.

CACTI estimates combined leakage power of the sequence tag array and signature cache at 800mW, while the L1D data cache will only leak approximately 230mW. This drastic difference arises because CACTI assumes that these structures will all be implemented using similar transistors. Lookup in the LT-cords structures does not require low latency and is not on the critical path, enabling a deeply pipelined design using high-Vt and/or long channel length transistors, which significantly reduce leakage compared to the highly latency-sensitive L1D cache.

Finally, LT-cords substantially raises chip performance on many workloads. By increasing the IPC, LT-cords may allow meeting target performance at a lower clock rate and lower power supply voltage, providing a way to offset the power dissipation overhead of LT-cords.

6 Conclusions

Technological advances in microarchitecture, circuits, and semiconductor fabrication all contribute to the long-term trend of a widening performance gap between microprocessors and memory. Larger on-chip caches can only partially mitigate the effect of growing memory latency. To facilitate efficient data transfer to the processor, recent research advocates using last-touch correlating prefetchers. Unlike delta-correlation-based prediction schemes, these prefetchers can eliminate most data cache misses despite interleaved access sequences and irregular memory layouts. However, existing designs are inefficient and impractical to implement, requiring prohibitive on-chip storage.

In this paper, we observed that last-touch signatures are temporally correlated and proposed Last-Touch Correlated Data Streaming (LT-cords), a practical design for accurate and timely streaming of data to the L1 cache. LT-cords leverages the temporal correlation of last-touch signatures to obviate the need for large, high-bandwidth, on-chip storage. LT-cords keeps last-touch signatures in off-chip DRAM, in the order they are discovered, and brings them on-chip when needed. Using only 214KB of on-chip storage and with little impact on memory bandwidth, LT-cords achieves coverage and lookahead similar to an "oracle" DBCP, a last-touch correlating prefetcher with unlimited on-chip resources. Our results show that LT-cords can eliminate 69% of all cache misses, achieving an average performance improvement of 60%, four times better than DBCP with 2MB on-chip storage, and two times better than a GHB PC/DC prefetcher.

Acknowledgements

References

[1] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proc. of Supercomputing '91*, Nov. 1991, 176–186.

[2] M. C. Carlisle, A. Rogers, J. H. Reppy, and L. J. Hendren. Early experiences with Olden. In *Proc. of Sixth Lang. and Compilers for Parallel Computing*. Springer-Verlag, 1994, 1–20.

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

[4] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proc. of SIGPLAN '02 Conf. on Program. Lang. Design and Impl. (PLDI)*, New York, NY, USA, 2002. ACM Press, 199–209.

[5] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proc. of 31st Annual Intl. Symp. on Comp. Arch. (ISCA-31)*, June 2004.

[6] J. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *Proc. of 35th Annual IEEE/ACM Intl. Symp. on Microarch. (MICRO 35)*, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press, 62–73.

[7] J. D. Collins, H. Wang, D. M. Tullsen, C. J. Hughes, Y. fong Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proc. of 28th Annual Intl. Symp. on Comp. Arch. (ISCA-28)*, July 2001, 14–25.

[8] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *Proc. of Tenth Intl. Conf. on Arch. Support for Program. Lang. and Op. Syst. (ASPLOS X)*, New York, NY, USA, 2002. ACM Press, 279–290.

[9] D. Gracia Perez, G. Mouchard, and O. Temam. Microlib: a case for the quantitative comparison of micro-architecture mechanisms. In *Proc. of Third Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD04)*, June 2004.

[10] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Proc. of 29th Annual Intl. Symp. on Comp. Arch. (ISCA-29)*, May 2002.

[11] D. Joseph and D. Grunwald. Prefetching using Markov Predictors. In *Proc. of 24th Annual Intl. Symp. on Comp. Arch. (ISCA-24)*, June 1997, 252–263.

[12] A.-C. Lai and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proc. of 28th Annual Intl. Symp. on Comp. Arch. (ISCA-28)*, July 2001.

[13] A. Mendelson, D. Thi'ebaut, and D. Pradhan. Modeling live and dead lines in cache memory systems. Technical Report TR-90-CSE-14, Department of Electrical and Computer Engineering, University of Massachusetts, 1990.

[14] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: an effective alternative to large instruction windows. *IEEE Micro*, 23(6):20–25, November/December 2003.

[15] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *Proc. of Tenth IEEE Symp. on High-Perf. Comp. Arch.*, February 2004.

[16] M. Pericas, A. Cristal, R. Gonzalez, D. A. Jimenez, and M. Valero. A decoupled kiloinstruction processor. In *Proc. of Twelfth IEEE Symp. on High-Perf. Comp. Arch.*, 2006, 52–63.

[17] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proc. of Eighth Intl. Conf. on Arch. Support for Program. Lang. and Op. Syst. (ASPLOS VIII)*, Oct. 1998.

[18] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proc. of Tenth Intl. Conf. on Arch. Support for Program. Lang. and Op. Syst. (ASPLOS X)*, Oct. 2002.

[19] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proc. of 33rd Annual IEEE/ACM Intl. Symp. on Microarch. (MICRO 33)*, December 2000, 42–53.

[20] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proc. of 29th Annual Intl. Symp. on Comp. Arch. (ISCA-29)*, May 2002.

[21] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proc. of Eleventh Intl. Conf. on Arch. Support for Program. Lang. and Op. Syst. (ASPLOS XI)*, Oct. 2004.

[22] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: improving both performance and fault tolerance. In *Proc. of Ninth Intl. Conf. on Arch. Support for Program. Lang. and Op. Syst. (ASPLOS IX)*, Nov. 2000.

[23] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. *HP Technical Report HPL-2006-86*, June 2006.

[24] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *Proc. of 33d Annual Intl. Symp. on Comp. Arch. (ISCA-33)*, June 2005.

[25] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Simulation sampling with live-points. In *Proc. of Intl. Symp. on the Perf. Analysis of Syst. and Software*, June 2006.

[26] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *Proc. of 1991 ACM SIGMETRICS Conf. on Measurement and Modeling of Comp. Syst.*, May 1991, 79–89.