Dead-Block Prediction & Dead-Block Correlating Prefetchers

An-Chow Lai
Electrical & Computer Engineering
Purdue University
West Lafayette, IN 47907
laia@ecn.purdue.edu

Cem Fide
Sun Microsystems
901 San Antonio Rd
Palo Alto, CA 94303
cem.fide@eng.sun.com

Babak Falsafi
Electrical & Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
babak@ece.cmu.edu

http://www.ece.cmu.edu/~impetus

Abstract

Effective data prefetching requires accurate mechanisms to predict both "which" cache blocks to prefetch and "when" to prefetch them. This paper proposes the Dead-Block Predictors (DBPs)*, trace-based predictors that accurately identify "when" an L1 data cache block becomes evictable or "dead". Predicting a dead block significantly enhances prefetching lookahead and opportunity, and enables placing data directly into L1, obviating the need for auxiliary prefetch buffers. This paper also proposes* Dead-Block Correlating Prefetchers (DBCPs)*, that use address correlation to predict "which" subsequent block to prefetch when a block becomes evictable. A DBCP enables effective data prefetching in a wide spectrum of pointer-intensive, integer, and floating-point applications.*

We use cycle-accurate simulation of an out-of-order superscalar processor and memory-intensive benchmarks to show that: (1) dead-block prediction enhances prefetching lookahead at least by an order of magnitude as compared to previous techniques, (2) a DBP can predict dead blocks on average with a coverage of 90% only mispredicting 4% of the time, (3) a DBCP offers an address prediction coverage of 86% only mispredicting 3% of the time, and (4) DBCPs improve performance by 62% on average and 282% at best in the benchmarks we studied.

1 Introduction

Increasing processor clock speeds along with microarchitectural innovation have led to a tremendous gap between processor and memory performance. Architects have primarily relied on deeper cache hierarchies, where each level trades off faster lookup speed for larger capacity, to reduce this performance gap. Conventional cache hierarchies employ a *demand-fetch* memory access model, in which data are fetched into higher levels upon processor requests. Unfortunately, the limited capacity in higher cache levels and the simple data placement mechanisms used in conventional hierarchies often result in high miss rates and reduce performance. While superscalar engines with non-blocking caches [19] allow overlapping the miss latency among the higher cache levels, limited available instruction-level parallelism and long access latencies to lower cache levels often expose the miss latency in many important classes of applications.

Many architects have additionally relied on the *prefetch* memory access model to mitigate the shortcomings of the demand-fetch model. Prefetching helps fetch data in advance to hide the memory latency by predicting future memory requests. While prefetching can be initiated in either hardware [17,5,10,3,15,4,6] or software [9,8,14,12], many researchers and vendors opt for hardware implementations for transparency and due to availability of runtime information which can significantly improve prefetching's effectiveness. Most previous proposals for hardware prefetchers target specific memory access patterns — such as strided accesses [15,4,6] and accesses to linked data structures [17]. While effective for the targeted access patterns, these prefetchers have limited general applicability across a wide spectrum of applications.

There are a number of prefetcher proposals in the literature that target generalized memory access patterns [5,3] — including strided accesses, and indirect accesses to linked data structures and arrays. These proposals primarily rely on *miss address correlation* [1] as a technique to predict and prefetch memory addresses. These prefetchers, which we refer to as Miss Correlating Prefetchers (MCPs), record a history of prior L1 cache miss addresses, and correlate the history to a subsequent miss to trigger a prefetch.

Unfortunately, MCPs suffer from several key shortcomings. First, L1 cache misses are often clustered, especially in out-of-order engines with high-bandwidth L1 caches, significantly limiting the lookahead and opportunity for timely prefetching. Second, rather than predicting block evictability, these prefetchers place the (prefetched) data in small associative buffers, and look them up either in parallel with L1 thereby increasing L1's critical access path or upon an L1 miss thereby increasing the prefetch hit latency. Finally, miss address correlation has not been shown to offer *both* high prediction accuracy (i.e., correct predictions as a fraction of all predictions) and high coverage (i.e., correct predictions as a fraction of all misses) [5].

This paper proposes the *Dead-Block Predictors (DBPs)* and the *Dead-Block Correlating Prefetchers (DBCPs)*. A DBP is a novel hardware mechanism that predicts "when" a block in a data cache becomes evictable. In a recent paper [7], we proposed trace-based predictors that record a trace of shared memory references to predict a last reference to a cache block prior to an invalidation in a multiprocessor. Similarly, a DBP records a trace of memory references that accurately predict the last reference to a block in an L1 data cache, prior to the block's eviction. A *DBCP* uses address correlation in conjunction with dead-block traces to predict a subsequent address upon a dead-block prediction. Accurate predicton of a block's evictability enables timely prefetching of data directly into an L1 data cache.

We use a cycle-accurate simulation of an aggressive out-of-order superscalar processor and a spectrum of memory-intensive benchmarks to show the following:

- For *critical* cache misses (that are not fully overlapped by computation and incur stalls), on average 92% of the intervals between a last reference to a block until its eviction from L1 are larger than L2 latency, indicating excellent lookahead opportunity for DBCP. In contrast, on average only 38% of the intervals between two subsequent cache misses are larger than L2 latency, indicating a much lower opportunity for MCPs.
- A *DBP* predicts a block's evictability on average for 90% of the time and only mispredicts 4% of the time.
- We show for the first time that correlating two prior addresses significantly enhances MCPs' accuracy and coverage.
- Using a 2M on-chip implementation, a DBCP offers timely prefetching and on average speeds up applications by 62% and at best by 282%. In contrast, ideal MCP implementations assuming unlimited storage increase performance on average only by 17% and at best by 51%.

The rest of the paper is organized as follows. Section 2 describes previous work on miss correlating prefetchers. In Section 3, we present the design details of our predictors and prefetchers. In Section 4, we present the methodology and results. Finally, we discuss the related work in Section 5 and conclude the paper in Section 6.

2 Miss correlating prefetchers

There is a myriad of proposals for hardware and software data prefetching. We will briefly describe these previous proposals in the related work in Section 5. In this section, we will focus on Miss Correlating Prefetchers (MCPs) [3], the most effective of current hardware prefetchers applicable to generalized memory access patterns. Prefetching relies on two key mechanisms to success-

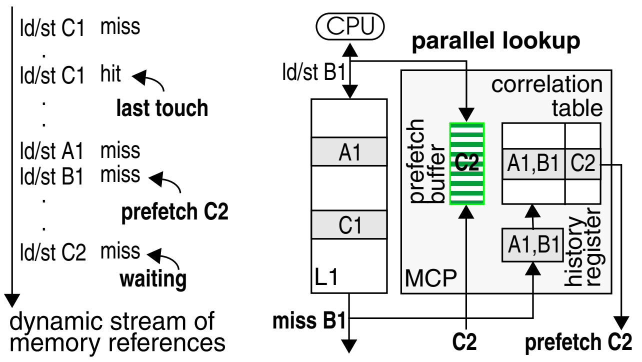

FIGURE 1. A Miss Correlating Prefetcher.

fully fetch and place data prior to a processor reference: (1) an accurate memory address predictor to predict "which" data to prefetch, and (2) and an accurate predictor of "when" to prefetch the data. MCPs rely on correlating cache miss addresses to predict both which data to prefetch and when to prefetch it.

Figure 1 depicts the anatomy of an MCP. An MCP uses a miss address predictor and a prefetch buffer. Much as two-level branch predictors, the miss address predictor consists of two storage levels. A history register maintains an encoding of the most recent miss addresses. A correlation table, organized as a cache, records a prediction for a subsequent address given a history encoding. Upon prediction, MCP prefetches a block and places it in the prefetch buffer for lookup by the processor. The lookup occurs either in parallel with L1, increasing L1's critical access path, or upon an L1 miss incurring high prefetch hit latency.

A recent study [5] evaluated MCPs in detail and concluded that miss address correlation *alone* results in low prediction accuracy and coverage independently of the number of addresses recorded in the history. They proposed Markov prefetchers that recorded and prefetched up to four subsequent missing addresses for every history entry. We present results in this paper that indicate that encoding two previous addresses results in *high prediction accuracy and coverage* in the spectrum of pointer-intensive, integer, and floating-point applications we studied.

Despite a high address prediction accuracy and coverage, prefetching using MCPs is often not timely. Figure 1 depicts an example of MCPs' shortcomings. A reference to block B1 results in a cache miss which triggers a prefetch to cache block C2. Cache blocks C1 and C2 are mapped to the same block frame in the cache. While the last reference (or "last touch") to C1 may occur well in advance of the reference to C2, the block frame in the cache holds C1 until C2 is moved from the prefetch buffer into the frame upon a processor reference. In contrast, predicting the last touch to C1 would allow replacing C1 by C2 in the corresponding block frame earlier.

Recent out-of-order superscalar engines further reduce lookahead in MCPs. These processors rely on non-blocking L1 caches and often issue multiple accesses in parallel, reducing the distance between two misses. Furthermore, these engines issue cache accesses out of program order, resulting in a re-ordering of miss addresses and address misprediction. The degree of re-ordering and its impact on prediction accuracy highly depends on the available parallelism in the application and a processor's issue-window size. While using the ordered (i.e., committed) miss address stream [5] would help increase accuracy, it significantly reduces the timeliness in prefetching because MCPs have limited prefetching lookahead.

3 Dead-block correlating prefetchers

In this paper, we propose the *Dead-Block Correlating Prefetchers (DBCPs)*, that predict a last reference to a block frame in a data cache, replace the contents of the block frame upon the last reference, and subsequently predict and prefetch a new cache block. Cache block frames alternate between two states: (a) a "live" state which begins with a miss and is followed by a sequence of hits to the frame, and (b) a "dead" state which begins after the last hit to the frame and ends with a subsequent miss. The key observation behind a DBCP is that the time during which a frame is "dead" is quite long [20,11] and well more than the time needed to fetch a cache block from the bottom of the hierarchy. Therefore, by accurately predicting both when a frame becomes "dead" and what cache block the processor will reference next, an DBCP can eliminate the miss and improve performance.

A DBCP uses a two-level predictor to predict both a cache block replacement and a subsequent address to prefetch for the corresponding block frame. In a recent paper [7], we proposed Last-Touch Predictors (LTPs) to predict memory *invalidations* for shared data in a multiprocessor. In this paper, we derive predictors from LTPs that predict the last reference to a cache block prior to its eviction (i.e., when the block "dies") in the L1 cache and correlate a subsequent address to prefetch with every last touch.

Much like MCPs, DBCPs rely on repetitive memory access behavior in programs to prefetch effectively. Unlike MCPs, DBCPs primarily capitalize on repetitive *instruction sequences* — rather than memory address sequences — to predict memory access behavior. Figure 2 depicts prefetching using a DBCP for our example of memory references. The predictor encodes the trace of memory references to A2 from the time it is fetched (by the reference at PCi) into L1. Upon a last reference to A2 by PCk, the DBCP predicts that A2 is "dead", replaces A2 and prefetches A3. Because the last reference to A2 often arrives much earlier than a subsequent reference to A3 (at PCl), DBCP hides the latency for fetching A3. In the rest of the section, we will describe in detail DBCP's prediction/prefetching mechanisms and their implementation.

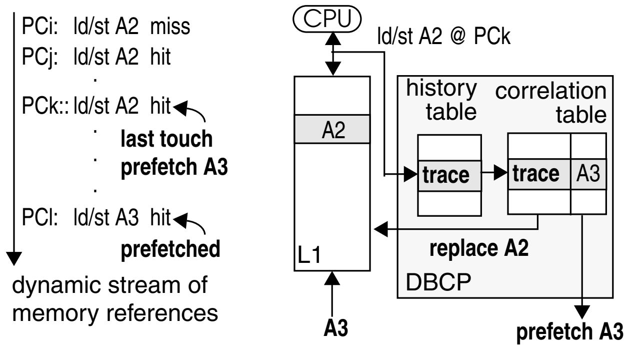

FIGURE 2. A Dead-Block Correlating Prefetcher.

3.1 A dead-block predictor

The first predictor we derive is a *Dead-Block Predictor (DBP)* for L1 data caches. Figure 3 (top) depicts the anatomy of a trace-based DBP. A history table duplicates the L1 tag array and stores a trace encoding associated with every tag. We use truncated addition (as before [7]) to maintain a fixed-size encoding for every instruction trace. In practice, truncated addition allows a compact encoding (i.e., ~12 bits) while offering high prediction accuracy and coverage. We also studied using *xor* (used in other predictors [13]) and found that repetition of PCs (due to iterative control flow) prevents *xor* from accurately encoding a trace. While other encoding functions are possible, a more detailed study of encoding is beyond the scope of this paper.

A dead-block table maintains encoded traces, called *signatures,* that end with a dead block. Upon a new history encoding, a DBP looks up in the dead-block table to match the trace against a signature. When learning, an L1 replacement places the block's history encoding as a signature in the dead-block table. To reduce misprediction frequency, a DBP uses two-bit saturating counters for every signature to estimate prediction confidence.

Due to control flow irregularities in applications, multiple cache blocks may have dead-block signatures that are proper subsequences of each other resulting in *subtrace aliasing* [7]. To prevent aliasing, DBP maintains dead-block signatures per cache block address. Because, the number of signatures highly varies across blocks (as data structure usage varies across application phases), we use a simple hash function to distribute the signatures across the table. Results in Section 4 indicate that *xor* works well as a hash function in practice.

The figure depicts prediction in DBP using our example of memory references. The trace in the history table for block A2 is {PCi,PCj}. A memory reference to A2 from

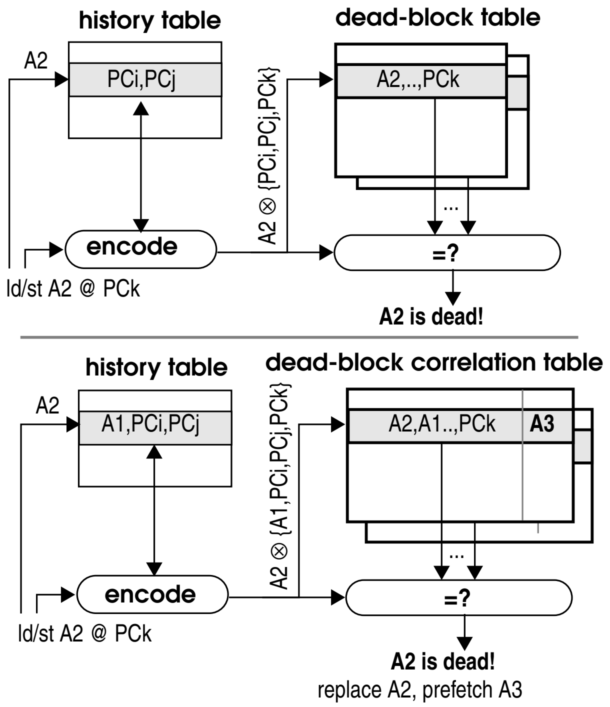

FIGURE 3. A dead-block predictor (top) and a dead-block correlating address predictor (bottom).

PCk updates the corresponding history table entry and looks it up in the dead-block table. Because the table indicates a match, the DBP predicts that block A2 is dead.

3.2 A dead-block correlating address predictor

We derive a *Dead-Block Correlating address Predictor,* from DBPs, that correlates the trace leading to a dead block to a subsequent memory address. In this paper, we use the abbreviation *DBCP* to refer to both the address predictor and the prefetcher we propose. In its minimal form, a DBCP is a DBP where each dead-block entry also includes a (prediction for) subsequent address. The dead-block table simply keeps track of the address referenced the last time a recorded trace resulted in a dead block. In general, a DBCP can also include prior address information in the dead-block traces for improved address prediction accuracy and coverage at the cost of higher storage overhead. In Section 4.3, however, we show that in practice, unlike MCPs, DBCPs exhibit high address prediction accuracy and coverage without prior address correlation.

Figure 3 (bottom) depicts the anatomy of a DBCP. A history table entry encodes a list of prior memory addresses (mapped to the block frame) along with the current PC trace. A dead-block correlation table records history entries that result in a dead block, and includes a prediction for a subsequent memory address. For instance, in our example

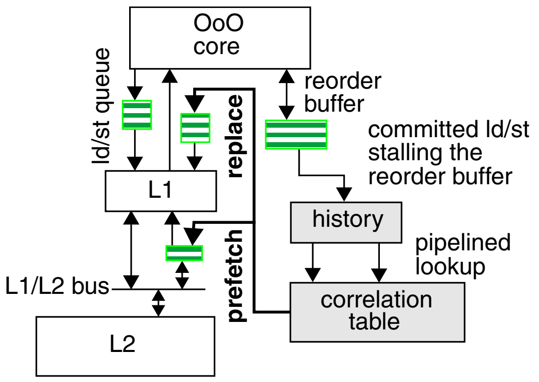

FIGURE 4. Using a DBCP in an out-of-order core.

in the figure, the predictor maintains a history of a prior memory address, A1, previously mapped to the same block frame as A2, with the trace {PCi,PCj}. Upon a reference to A2 by PCk, the dead-block correlation table predicts that A2 is dead and predicts A3 for prefetching.

A key difference between MCPs and DBCPs is that MCPs correlate and predict the miss address stream across block frames whereas DBCPs correlate and predict the miss address stream in a given block frame. Intuitively, neither address stream is fundamentally more predictable than the other because MCPs' cache address stream is just an interleaving of DBCP's block frame address stream. In practice, we present results in Section 4.3 that indicate that the best achievable address prediction accuracy and coverage for MCP and DBCP are comparable.

3.3 Predictor & prefetcher implementation

Unlike previous MCP studies [5,3] that evaluated the prefetcher's effectiveness for single-issue in-order processors, we incorporate and evaluate our prefetchers in a wide-issue out-of-order superscalar engine (Figure 4). Because cache block deadtimes are typically large (e.g., hundreds to thousands of cycles), a DBCP can tolerate high latencies in dead-block and prefetch address prediction without sacrificing timeliness. Moreover, the execution order of instructions and the instruction issue width have little impact on a DBCP's effectiveness because of the large prediction/prefetch lookahead. Therefore, unlike MCPs, DBCPs can monitor and record the *program ordered* (i.e., committed) memory reference stream. Because the history table maintains a copy of the L1 tags, the ordered memory reference stream in the history table also produces an ordered L1 miss address stream.

As in any table-based predictor, a key design parameter affecting a DBCP's accuracy is the size and organization of the correlation table. Due to DBCP's large tolerance for latency, the table can be built as a highly associative structure to minimize the number of conflicts among the signatures and increase accuracy and coverage. Moreover, the

Processor		Caches	
Clock rate	2 GHZ	L1 I/D	32K, 32-byte block
Issue/retire	8 instructions/cycle		direct-mapped d-cache
Branch predictor	8K/8K hybrid		4-way i-cache, 1-cycle
Reorder buffer	128 entries	L1 D ports	4
Load/store queue	128 entries	L1 D MSHRs	unlimited
Prefetcher		L2 I/D	each 1M 64-byte block
Prefetch MSHRs	16		4-way, 12-cycle
Prefetch requests	128 entries (FIFO)	L2 D ports	1
MCP buffer	128 entries	L2 D MSHRs	unlimited
MCP correlation	unlimited, 12-cycle	L1/L2 bus	32-byte wide, 2GHz
DBCP history	1K entries	Memory	
DBCP on-chip	2M 8-way, 18-cycle	Latency	70 processor cycles
DBCP off-chip	7.6M 16-way, 70-cycle	Bus	64-byte wide, 400 MHz
Table ports	4		

TABLE 1. System configuration.

Benchmarks	Inputs & parameters	L1 miss rate (%)	L2 miss rate (%)
bh	8192 bodies	2	70
em3d	2000 nodes, arity 2, 50 iter.	18	1
health	5 levels, 500 iter.	19	22
mst	1024 nodes	9	53
treeadd	1.6M nodes, 16 levels, 100 iter.	4	2
compress	train.in	4	4
perl	prime.pl	1	0
gcc	166.i -o 166.s	11	22
mcf	inp.in	24	49
ammp	ammp.in	14	74
art	reference input	46	40
equake	inp.in	6	49
mgrid	mgrid.in	4	25
swim	swim.in	5	30

TABLE 2. Benchmarks and input parameters.

table can be built either on chip to reduce interference with off-chip traffic, or off chip to optimize for size at the cost of a higher required bandwidth. Alternatively, the table can be built as a hierarchy with the on-chip storage holding the signatures for all the cache blocks in L1 backed up by main memory. Prefetching a block into L1 would simultaneously initiate bringing the block's dead-block signatures into the on-chip table. In this paper, we evaluate standalone (fast) on-chip and (slow) off-chip table implementations.

A DBCP can use reference *filters* to reduce storage and lookup bandwidth requirements for the tables. In out-of-order engines with non-blocking caches, cache misses in L1 that hit in L2 (e.g., conflict misses that are temporally close) are often overlapped and do not introduce memory stalls on the execution path. In this paper, we augment the processor's re-order buffer with a single bit that indicates whether a load/store instruction at the head of the buffer stalls (i.e., takes two or more cycles to retire). Such a load/store instruction incurs a *critical* miss. The correlation table only allocates entries for these critical misses. All signatures that do *not* result in a critical miss are therefore filtered and are not placed in the correlation table. In general, a DBCP can benefit from more elaborate hardware or software techniques to accurately identify the instructions responsible for a high fraction of memory stalls.

4 Results

We use SimpleScalar 3.0 to simulate an aggressive out-of-order processor with a cache hierarchy. Table 1 depicts the configuration parameters for the base system. We have augmented the simulator to accurately model contention at the L1/L2 and memory buses accurately. The buses always give priority to processor requests over prefetch requests.

To gauge the full potential of the predictors and the effectiveness of the entropy they encode independently of storage, we assume correlation tables with an unlimited number of entries in our predictor studies (Sections 4.2 and 4.3). For performance evaluation (Section 4.4), we consider practical implementations of DBCP with cache-like correlation tables. We consider an on-chip configuration with 2M (including tag and data) 8-way set-associative table with 18-cycle access latency (including ~64K entries), and an off-chip configuration with 7.6M (including tag and data) 16-way set-associative table with 70-cycle access latency (containing ~460K entries). Both configurations use a 1K-entry (as many entries as the tag array in the 32K-L1 D) history table and 12-bit signatures. The off-chip configuration eliminates all capacity and conflict misses in the correlation table and is the maximum size needed by all the applications we studied.

The correlation table entries consist of a 19-bit data (17-bit next cache block address equal to an L1 block tag and a two-bit saturating counter) and a 27-bit tag and index (including the current address and the dead-block signature). We use 27 bits from the L1 cache block's tag and index and *xor* it with the 12-bit signature from the history table. We use this resulting 27 bits as tag and index for the correlation table. We have found that this placement function in practice eliminates conflicts in the tables.

All prefetchers use a request queue with 128 entries. When the request queue is full, new entries in the queue replace the old (unissued) ones at the queue head. The MCPs also use a 128-entry (fully-associative) prefetch buffer with a 1-cycle access latency in parallel with L1. As in the request queue, new entries in the buffer remove old (unreferenced) entries. Prefetch requests are only issued at most one per cycle when the L1/L2 bus is free.

Table 2 describes the benchmarks we studied, their inputs, and the corresponding L1 and L2 miss ratios. These benchmarks include five pointer-intensive applications (i.e., *bh*, *em3d*, *health*, *mst*, and *treeadd*) from the Olden suite

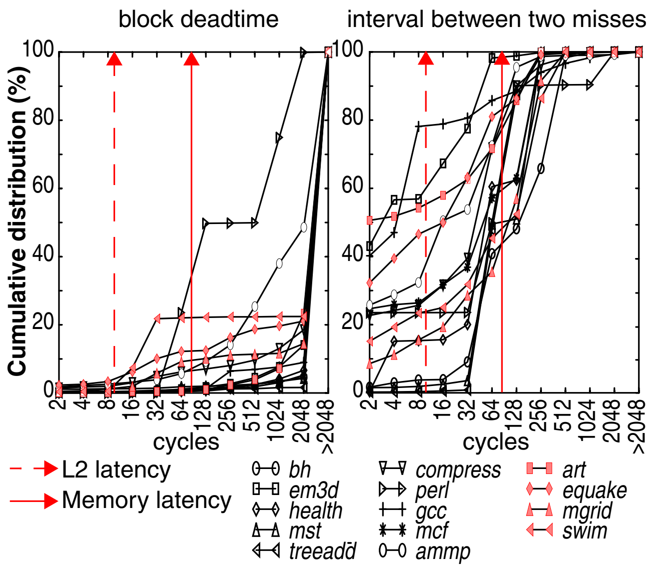

FIGURE 5. Lookahead analysis: cumulative distribution of distance in processor cycles from a last touch to a subsequent miss (left), and between two consecutive misses (right).

[2], and four integer and five floating-point applications from the SPEC2K (i.e., *gcc*, *mcf*, *ammp*, *art*, and *equake*) and SPEC95 (i.e., *compress*, *perl*, *mgrid*, and *swim*). All tested SPEC benchmarks use reference inputs except for *compress*. In the interest of reduced simulation time, we simulated the benchmarks for three billions cycles (six billion cycles for *equake*) after skipping an initialization period of one billion cycles. *Compress* requires simulating the entire input to benefit from repetitive memory access behavior. Instead we simulated its train input.

The benchmarks in this study all exhibit a large fraction of memory stall cycles in their execution with varying degrees of memory parallelism and overlap.[1] On one end of the spectrum, some of the Olden benchmarks primarily exhibit a high degree dependent memory accesses, exposing all the miss latencies on the critical path. On the other end, the floating-point benchmarks exhibit a high degree of memory parallelism, but are primarily limited by the processor's inability to hide long L2 miss latencies.

4.1 Lookahead opportunity

In this section, we compare the lookahead opportunity for DBCPs and MCPs. Figure 5 (left), illustrates the cumulative distribution of distances (in cycles) between a last reference to a cache block prior to the block's eviction due to a subsequent miss to another block. The graphs illustrate the deadtimes for *critical* misses, those misses which stall the reorder buffer because their latencies cannot be fully overlapped. The graphs indicate that except in *perl*, 80% of the deadtimes in all applications are 500 cycles are more, several times larger than the memory latency. *Perl* exhibits a high fraction of conflict misses that are clustered in time, reducing the fraction of deadtimes over 500 cycles to 50%. These results corroborate previous findings on cache block deadtimes [20,11], and show that the deadtimes offer excellent prefetch lookahead for DBCPs.

The results on block deadtimes have several implications. First, because deadtimes are on average several times longer than memory latency, as the gap between processor and memory speeds increases, lookahead opportunity remains high allowing effective data prefetching. Second, prediction and prefetching techniques relying on block deadtimes may trade off speed for higher accuracy; the latency of off-chip storage (or slow but large on-chip storage) for dead-block tables is not as critical if larger storage helps improve prediction accuracy and coverage.

Figure 5 (right) illustrates the cumulative distribution of distance between two critical misses in the cache. As we can see, in 9 out of 14 applications, 20%-80% of the miss intervals are smaller than L2 latency, allowing insufficient lookahead to prefetch and overlap these misses. Moreover, in 10 of the applications, 50% or more of the miss intervals are smaller than memory latency preventing a prefetcher from overlapping L2 misses. These results indicate that cache misses are highly clustered, therefore even if MCPs offer high address prediction accuracy and coverage, they may be limited significantly by prefetching lookahead.

The results on cache miss intervals also have several implications. First, as the gap between processors and memory speeds increases, lookahead opportunity using miss intervals relative to memory latency decreases. Second, aggressive wide-issue engines exacerbate the negative impact of miss clustering by issuing a larger number of memory references every cycle, thereby reducing the lookahead. Third, due to limited lookahead, MCPs require fast address prediction and prefetching mechanisms and cannot trade off accuracy for speed. Unlike DBCPs which can use the program ordered reference streams, MCPs must use the speculative reference stream which may be re-ordered and may reduce prediction accuracy and coverage.

4.2 Dead-block prediction accuracy & coverage

Figure 6 (bars labeled "A") presents the prediction accuracy and coverage of a DBP for a 32K L1 direct-mapped cache. The graphs plot the fraction of correct dead-block predictions (hits), the fraction of incorrect predictions

1. We evaluated DBP's and DBCP's prediction accuracy/coverage for *all* of SPEC95 and those SPEC2K benchmarks compiled for SimpleScalar. While the benchmarks we omit exhibit as high a prediction accuracy/coverage as those presented in this paper, they incur a small number of memory stalls and do not benefit from prefetching.

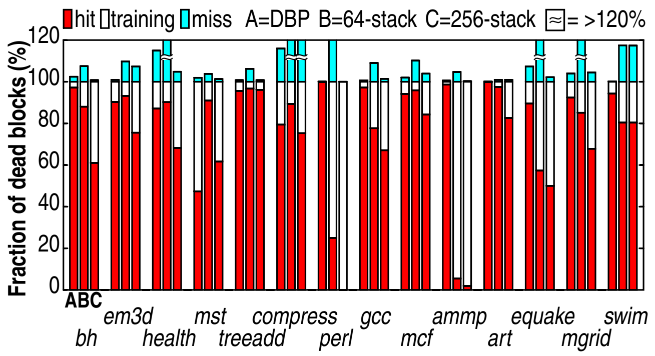

FIGURE 6. Accuracy and coverage of DBP, and 64- and 256-entry LRU stacks.
The graphs only present bars with up to 20% of mispredicted dead blocks. High misprediction values are not plotted to enhance clarity of the coverage numbers. The misprediction not plotted vary from 24% to 115%.

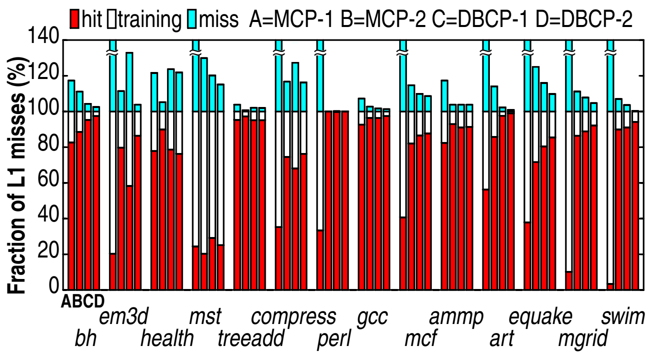

FIGURE 7. Accuracy and coverage of MCP and DBCP with history depths of one and two.
The graphs only present bars with up to 40% of mispredicted L1 misses. High misprediction values are not plotted to enhance clarity of the coverage numbers. The mispredictions not plotted vary from 44% to 96%.

(misses), and the fraction of dead blocks not predicted due to predictor training. The graphs indicate that on average a DBP predicts 90% of dead blocks and mispredicts (i.e., prematurely predicts live blocks as dead) only 4% of the dead blocks. We also evaluated DBPs for set-associative caches an LRU replacement policy and various sizes and found similar prediction accuracy and coverage. However, we omit these results in this paper in the interest of brevity.

These results indicate that the potential for trace-based predictors to predict memory system events is beyond just predicting memory invalidation and sharing for scientific applications in multiprocessors [7]. The results corroborate the intuition that because memory instructions drive the movement of data in the cache hierarchy, repetitive code fragments and the resulting instruction traces can help predict the movement.

Prediction coverage in *mst* is only 47% because *mst*'s primary heap-based data structure is constantly modified, resulting in many of the dead-block signatures to be obsolete upon creation. The two-bit saturating counters in DBP filter these signatures, preventing them from triggering a prediction. In *health*, DBP exhibits a subtrace aliasing problem because of the irregular control flow in the application's main two-level nested loop, giving rise to a large number of mispredicted dead blocks. *Compress* has a prediction accuracy and coverage of 80% because of low predictability in accesses to specific segments of the main data structure which is a compression hash table.

Figure 6 also compares the prediction accuracy and coverage of DBP against simple LRU stacks (bars labeled "B" and "C") proposed by Peir, et al. [16] to predict evictability of L1 blocks. The LRU stacks simply maintain a list of least-recently used addresses. The key idea is that an application's working set has a finite number of cache blocks. The stacks estimate the maximum size of the working set. When an address falls out of the stack, the stack predicts that the corresponding block is dead. Unfortunately, because working set sizes in a cache may largely vary both *within* and *across* applications [18], LRU stacks fail to predict dead blocks accurately. The graphs corroborate previous findings [16] on stacks and indicate that a 64-entry stack achieves a coverage of 77% while prematurely predicting dead blocks by over 100% (not shown). A 256-entry stack predicts dead blocks more conservatively and reduces the fraction of mispredicted dead blocks to 5%, only slightly over DBP's. However, the 256-entry stack's coverage is much lower than DBP, and is on average 60%. Moreover, the stack fails to cover any dead blocks for *perl* and *ammp* due to conflict misses.

4.3 Address prediction accuracy & coverage

While dead-block prediction offers high accuracy and coverage, the success of a prefetcher also relies on accurate address prediction. In general, either MCP or DBCP can use an arbitrary history depth — i.e., record a history of an arbitrary number of previous addresses to predict a next address. Larger history depth in MCP and DBCP increases accuracy by reducing subtrace aliasing — i.e., identical address sequences leading to different subsequent addresses — at the cost of higher storage. Too large a history depth, however, also reduces coverage by increasing the learning time for predictions that do not benefit from larger depth. Figure 7 illustrates prediction accuracy and coverage for MCP and DBCP with depths of one and two. We experimented with varying the depth and found little improvement in accuracy and a decrease in coverage with a higher depth.

MCP-1 and MCP-2 in the graphs correspond to MCPs with a history depth and one and two respectively. The graphs show that there is a significant improvement in prediction coverage from, 52% to 80%, for MCPs when the depth increases from one to two. Moreover, the fraction of mispredicted addresses on average drops from 47% to 13%, increasing the prediction accuracy. Previous studies evaluating MCPs for technical and commercial workloads [5] reported no added advantage to more history than a single address. Our results *clearly* indicate that for the wide spectrum of applications we study, the addition of another prior address enhances the predictor's effectiveness. Moreover, the previous study evaluated MCPs using a program-ordered stream of miss addresses from a single-issue in-order engine. The numbers we evaluate are for the addresses generated by our wide-issue out-of-order engine. We found that while the prediction coverage is slightly higher for the ordered stream, the ordered stream has minimal lookahead opportunity.

The key intuition behind why a larger history depth increases an MCP's accuracy and coverage is that while data structures are often referenced in multiple distinct program contexts or *phases* [7] (e.g., a given sequence procedure invocations), they are not always used in conjunction with the same other data structures in every phase. Moreover, a group of data structures referenced together are not always referenced in the same order. An increase in history depth helps correlate a reference (with a given address) to a program phase and consequently to a subsequent address.

In DBCP, dead-block signatures can precisely pinpoint which program phase a reference is from and therefore what subsequent address is following the given reference in that phase. DBCP-1 and DBCP-2 correspond to a DBCP with a history depth of one and two prior addresses respectively. The graphs show that DBCP-1 achieves a high coverage of 82% and with only 4% misprediction. However, the addition of a small number of extra bits increases coverage and decreases the fraction of mispredicted addresses in DBCP-2 to 86% and 3% respectively. We experimented with varying the number of bits used from the second prior address and found four bits to offer the best coverage while minimizing storage.

There is a single application in which the addition of bits from a second prior address helps significantly improve the predictor's coverage. In *em3d*, the entire program runs in a single phase, generating common dead-block signatures. Because the program marches down a bipartite graph in which one graph node shares both multiple incoming edges and outgoing edges with other nodes, a dead-block signature and the node's address alone cannot distinguish the subsequent addresses along the different edges.

On average, MCP-2 and DBCP-2 achieve roughly the same prediction accuracy and coverage, with DBCP-2 having only a slight advantage over MCP-2. On a closer look, we also found that the number of entries the predictors maintain are roughly the same. Therefore, DBCP-2's primary advantage is in prefetching lookahead opportunity. However, DBCP-2 effectively encodes the information as to when to prefetch and what to prefetch simultaneously, obviating the need for decoupling the predictors, and optimizing for storage.

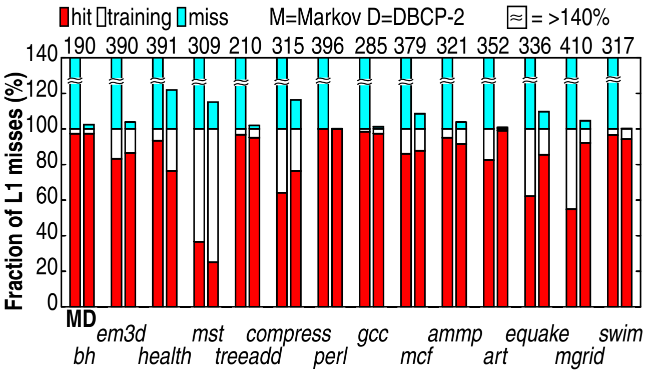

FIGURE 8. Accuracy and coverage of Markov and DBCP-2.
The graphs only present bars with up to 40% of mispredicted L1 misses. The numbers on top indicate the maximum values for each bar.

Figure 8 compares DBCP-2 with a Markov predictor — i.e., an MCP-1 predictor that predicts four LRU addresses rather than a single address. Markov predictors increase coverage at the cost of a much higher fraction of mispredicted addresses (only a single address out of the four predicted can be correct). On average, Markov achieves a coverage of 81% (slightly less than DBCP-2) but increases the fraction of mispredicted addresses to 229%. The mispredicted addresses may significantly increase traffic in the memory hierarchy and place a large demand on bandwidth. Fortunately, not all of the mispredicted addresses go to waste. By predicting multiple subsequent addresses and prefetching them into a prefetch buffer, Markov actually somewhat offsets the negative impact of miss address re-ordering due to either re-ordering of references to data structures across program phases or re-ordering of miss addresses in the out-of-order engine.

In *equake* and *mgrid*, the number of different miss addresses after the current miss is often larger than four, because these applications reference different sets of arrays in different program phases (e.g., different procedures). Hence, Markov is unable to capture a large fraction of the misses in these applications. Because DBCP-2 identifies where the program is executing, it exhibits high prediction accuracy and coverage in these applications. DBCP-2 performs worse than Markov in *health* and *mst* because these two benchmarks have dynamically alternating sequences of memory references to their main data structures. Consequently, DBCP-2 spends much time in correlating these references, resulting in a low prediction coverage. In contrast, Markov captures these changes in groups of four, and yields a higher prediction coverage.

4.4 Prefetching performance

In this section, we evaluate the prefetcher's effectiveness in improving performance. We compare DBCP against

	Ideal L1	MCP-2	Markov	Markov inf b/w	DBCP-2 on chip	DBCP-2 off chip
bh	61	28	38	38	59	59
em3d	60	6	6	13	35	42
health	135	26	47	50	88	89
mst	123	11	24	28	*18*	*18*
treeadd	36	17	16	16	33	33
compress	18	4	5	6	7	6
perl	10	5	8	8	*5*	*5*
gcc	46	2	9	11	24	27
mcf	185	15	38	53	125	131
ammp	303	62	63	69	282	283
art	155	4	1	3	52	54
equake	50	2	10	11	*2*	27
mgrid	56	4	21	30	27	48
swim	94	7	25	33	31	51

TABLE 3. Performance comparison of the prefetchers against an ideal demand-fetched system.
The table depicts percent speedup over our base demand-fetched system. Ideal L1 numbers correspond to an L1 with no capacity/conflict misses. MCP and DBCP use address history depth of 2 (i.e., MCP-2 and DBCP-2). The table also depicts Markov numbers with unlimited bandwidth to L2 and memory (Markov inf b/w). The DBCP numbers correspond to an on-chip implementation with 2M 8-way storage and 18-cycle latency and off-chip storage with 7.6M 16-way storage and 70-cycle latency. The numbers appear in italic are those which Markov outperforms DBCP. All results are normalized against the base system (no prefetch) using 12-cycle 1M L2.

MCP implementations (i.e., both MCP-2 and Markov). To gauge MCPs' best performance independently of storage size, we assume correlation tables for MCPs that are large enough to fit all of the generated address encoding with a fast lookup latency of 12 cycles, equal to L2's latency (we also evaluated Markov with an ideal lookup latency of one cycle and observed less than 5% improvement in speedups). Because Markov can potentially benefit from extra bandwidth, we also present Markov numbers with unlimited bandwidth to L2 and memory.

Table 3 compares a DBCP against an "ideal" L1 with no capacity/conflict misses. The comparison to "ideal" L1 helps determine what fraction of the memory stalls the prefetchers can eliminate. The table presents speedups over a demand-fetched system. Our first observation is that all prefetchers improve performance over the demand-fetched system even though our base system is quite aggressive. On average, DBCP eliminates 62% of the memory stalls in L1 and achieves a 62% speedup. In contrast, MCP and Markov only eliminate 22% and 30% of the memory stalls and achieve a speedup of only 14% and 17% respectively. While Markov benefits from extra L2 and memory bandwidth, the improvement is only on average an additional 4%, not enough to break even with DBCP.

Figure 9 breaks down the fraction of memory stalls removed and incurred in a prefetching system relative to a demand-fetched system. The incurred stalls are either stalls

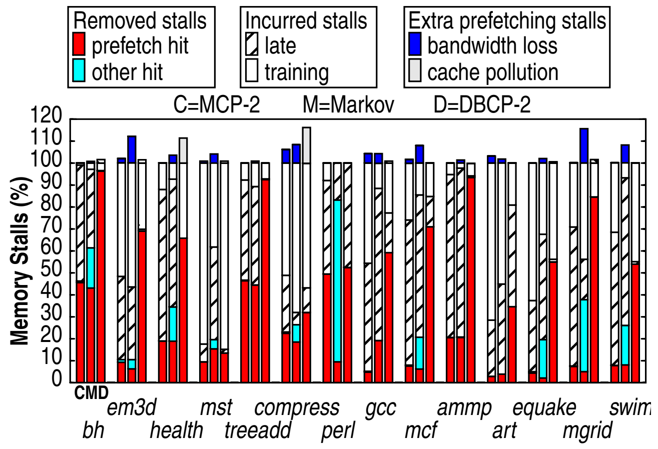

FIGURE 9. Breakdown of all memory stalls relative to a demand-fetched system.
The DBCP numbers correspond to an off-chip table implementation. Incurred stalls are stalls that the prefetchers are unable to remove. These stalls are due to prefetcher training and late (not timely) prefetching. The extra prefetching stalls are extra memory stalls incurred because of prefetching. These stalls are bandwidth loss due to misprediction and cache pollution (in DBCP) because of incorrect prefetching into L1.

originally present in the demand-fetched system or extra stalls due to incorrect or late prefetching. The removed stalls are either hits in the L1 (for the case of DBCP) or prefetch buffer due to a successful (accurate and timely) prefetch, or hits in the prefetch buffer due to an earlier mispredicted prefetch.

The figure indicates that DBCP is timely in most applications. DBCP prefetches late in *art, compress, gcc,* and *mcf* due to the bursty prefetch requests. These applications would benefit from multiple L2 cache ports and a higher bandwidth memory system. The request queue in these applications often becomes full and drops requests. Nevertheless, DBCP eliminates a significant fraction of the memory stalls in these applications even with a single L2 port.

DBCP is extremely effective in Olden benchmarks which exhibit a high degree of data dependence through memory in linked data structures; DBCP virtually eliminates the dependence bottleneck in two applications and significantly reduces memory stalls in another two. Despite low prediction accuracy and coverage in *mst*, DBCP still speeds up execution by 18%. Mispredictions somewhat offset the gains from prefetching in *health* and *compress*, due to cache pollution. *Mcf* and *ammp* have large footprints which do not fit in L2. DBCP, however, accurately predicts the references and successfully fetches the data into L1, reducing the memory stalls. *Equake* generates a large number of correlation signatures due to a large working set of data. While DBCP only allocates signatures for critical misses, there are still more signatures than can fit in a 2M on-chip table. The off-chip table, however, fits all the signatures and significantly improves speedup in *equake*.

	bh	*em3d*	*health*	*mst*	*treeadd*	*compress*	*perl*	*gcc*	*mcf*	*ammp*	*art*	*equake*	*mgrid*	*swim*
DBCP-2 on chip	59	35	88	18	33	7	5	24	125	282	52	2	27	31
3.1M L2 12-cycle	56	0	*96*	15	2	0	0	*31*	53	2	29	2	23	3
3.1M L2 18-cycle	47	-15	79	12	-15	-6	-2	*30*	44	1	25	-1	-1	-3

TABLE 4. Performance comparison of DBCP against demand-fetched systems with larger L2. The table presents percent speedups over our base demand-fetched system. The L2 numbers correspond to an ideal 12-cycle lookup latency and a more realistic 18-cycle latency. The numbers appear in italic are those which 3.1M L2 outperforms DBCP.

In contrast, MCPs are timely for many of the Olden benchmarks. The benchmarks exhibit a high degree of data dependence and large cache miss intervals (Figure 5) allowing for prefetching lookahead. In SPEC benchmarks, the misses are bursty and often independent allowing little prefetching lookahead. Markov always improves performance over MCP-2 even though the predictors achieve comparable coverage (Section 4.3). Markov's *effective* coverage is higher than MCP-2 because some of the mispredicted prefetches placed in the buffer actually hit for subsequent misses. The bandwidth loss in MCPs is not significant because most prefetches are either late but accurate or early but inaccurate. In the former case, prefetches do not increase traffic as misses are merged with outstanding prefetch requests. In the latter case, inaccurate prefetches which are buffered are useful for subsequent misses.

Besides *equake*, MCPs only improve performance over DBCP in *mst* and *perl*. In *mst*, Markov improves prediction coverage over DBCP (as discussed in Section 4.3) and removes relatively more stalls. *Perl* primarily incurs L1 conflict misses satisfied by L2 and therefore neither prefetcher is timely for *perl*. However, hits in the prefetch buffer due to mispredicted prefetches significantly improve performance in Markov over DBCP.

Table 4 compares the on-chip 2M DBCP with a base 1M L2 (with a 12-cycle hit latency) against an 3.1M 6-way L2 with approximately equal storage cost including the tag overhead. We evaluate both an aggressive large L2 implementation with the same 12-cycle hit latency as the base L2, and a slower but more realistic L2 hit latency of 18 cycles. The table indicates that the addition of an on-chip DBCP is much more cost-effective than increasing L2's size. Half of the applications do not benefit from a larger L2. A larger L2 slightly outperforms DBCP only in two applications, *health* and *gcc*, in which the larger L2 captures a significant fraction of their working sets. Moreover, in a realistic 3.1M L2 implementation with a longer access latency, seven of the applications actually exhibit slowdown because the out-of-order engine fails to overlap the long L2 latency.

5 Related work

There are a number of hardware-based data prefetching techniques, many of which customize hardware for specific memory reference patterns. Chen and Baer proposed stride prefetchers [4] that correlate non-unit data address strides with a memory instruction PC in a small table and prefetch based on the stride. Jouppi proposed stream buffers [6] to detect unit stride cache miss address sequences and correlated them with a "starting" address to prefetch them sequentially. Palacharla and Kessler [15] extended stream buffers to non-unit stride stream. Mehrorta and Harrison [10] proposed the indirect reference buffers to identify data address dependence in recursive or linked data structures. Roth, et al. [17], proposed prefetchers that capture memory reference dependence in linked data structures and associate them with instruction PCs to initiate a prefetch. Charney and Reeves [3] were first to use address correlation in hardware on the L1 miss stream to prefetch. Joseph and Grunwald [5] proposed Markov prefetchers that are miss correlating prefetchers associating multiple subsequent addresses with each correlation.

Many software prefetchers rely on accurate compile-time analysis of memory access patterns to detect both what memory addresses are subsequently referenced and when the data can be placed in the cache. Mowry, et al. [12], show that for numerical and scientific applications, software prefetchers can successfully hide the memory access latency. Luk and Mowry [9], Lipasti, et al. [8], and Ozawa, et al. [14], also evaluate the effectiveness of heuristics-based techniques which insert compile-time prefetch instructions in pointer-intensive applications and applications with recursive data structures.

6 Conclusions

In this paper, we proposed and evaluated *Dead-Block Correlating Prefetchers (DBCPs)*. These prefetchers, use a novel mechanism, *Dead-Block Predictors (DBPs)*, to predict when L1 data cache blocks become evictable. Previous techniques for data prefetching primarily relied on correlating the L1 data miss address stream to predict and trigger a prefetch. Predicting a dead block, however, significantly enhances prefetching lookahead over previous techniques. DBCPs predict and prefetch a subsequent block address upon predicting a block's eviction. DBCPs enable effective data prefetching in a wide spectrum of pointer-intensive, integer, and floating-point applications with arbitrary memory access patterns.

We used cycle-accurate simulation of an aggressive wide-issue out-of-order superscalar processor and memory-intensive benchmarks to show that: (1) dead-block prediction enhances prefetch lookahead by at least an order of magnitude over previous techniques, (2) a DBP can predict dead blocks on average with a coverage of 90% with only 4% misprediction, (3) a DBCP offers an address prediction coverage of 86% with only 3% misprediction, and (4) DBCPs offer timely prefetching of data directly into L1 and help improve performance by 62% on average and 282% at best. In contrast, the best current proposal for prefetching generalized memory access patterns achieves a speedup of only 17% and at best 51%.

7 Acknowledgements

This work is supported in part by a grant from Intel corporation and an NSF CAREER Award titled "Prediction and Speculation in High-Performance Memory Systems". We also thank Milo Martin, Se-Hyun Yang, and the anonymous reviewers for feedback on an earlier draft of this paper.

References

[1] Jean-Loup Baer and Tien-Fu Chen. Dynamic improvements of locality in virtual memory systems. *IEEE Transactions on Software Engineering*, March 1976.

[2] Martin C. Carlisle, Anne Rogers, John H. Reppy, and Laurie J. Hendren. Early experiences with Olden. In *Proceedings of the Sixth Languages and Compilers for Parallel Computing*, pages 1–20. Springer-Verlag, 1994.

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

[4] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 51–61, October 1992. Also available as U. Washington CS TR 92-06-03.

[5] Doug Joseph and Dirk Grunwald. Prefetching using Markov Predictors. *IEEE Transactions on Computers*, 48(2):121–133, February 1999.

[6] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[7] An-Chow Lai and Babak Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[8] Mikko H. Lipasti, William J. Schmidt, Steven R. Kunkel, and Robert R. Roediger. Spaid: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 28)*, pages 231–236, November 1995.

[9] Chi-Keung Luk and Todd C. Mowry. Compiler based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 222–233, October 1996.

[10] Sharad Mehrotra and Luddy Harrison. Examination of a memory access classification scheme for pointer-intensive and numeric programs. In *Proceedings of the 1996 International Conference on Supercomputing*, pages 133–139, May 1996.

[11] Abraham Mendelson, Dominique Thi'ebaut, and Dhiraj 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.

[12] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 62–73, October 1992.

[13] Ravi Nair. Dynamic path-based branch prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 29)*, pages 142–152, December 1996.

[14] Toshihiro Ozawa, Yasunori Kimura, and Shin'ichiro Nishizaki. Cache miss heuristics and preloading techniques for general-purpose programs. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 28)*, November 1995.

[15] Subbarao Palacharla and Richard E. Kessler. Evaluating stream buffers as a secondary cache placement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, April 1994.

[16] Jih-Kwon Peir, Yongjoon Lee, and Windsor W. Hsu. Capturing dynamic memory reference behavior with adaptive cache topology. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 240–250, October 1998.

[17] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, October 1998.

[18] Alan J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[19] Gurindar S. Sohi and Manoj Franklin. High-bandwidth data memory systems for superscalar processors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 53–62, April 1991.

[20] David A. Wood, Mark D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 79–89, May 1991.