2018 IEEE International Symposium on High Performance Computer Architecture

Domino Temporal Data Prefetcher

Mohammad Bakhshalipour[†1], Pejman Lotfi-Kamran[‡], and Hamid Sarbazi-Azad[†‡]

[†]*Department of Computer Engineering, Sharif University of Technology*
[‡]*School of Computer Science, Institute for Research in Fundamental Sciences (IPM)*

***Abstract*—Big-data server applications frequently encounter data misses, and hence, lose significant performance potential. One way to reduce the number of data misses or their effect is data prefetching. As data accesses have high temporal correlations, temporal prefetching techniques are promising for them. While state-of-the-art temporal prefetching techniques are effective at reducing the number of data misses, we observe that there is a significant gap between what they offer and the opportunity.**

This work aims to improve the effectiveness of temporal prefetching techniques. We identify the lookup mechanism of existing temporal prefetchers responsible for the large gap between what they offer and the opportunity. Existing lookup mechanisms either not choose the right stream in the history, or unnecessarily delay the stream selection, and hence, miss the opportunity at the beginning of every stream. In this work, we introduce *Domino* prefetching to address the limitations of existing temporal prefetchers. Domino prefetcher is a temporal data prefetching technique that logically looks up the history with both one and two last miss addresses to find a match for prefetching. We propose a practical design for Domino prefetcher that employs an *Enhanced Index Table* that is indexed by just a single miss address. We show that Domino prefetcher captures more than 90% of the temporal opportunity. Through a detailed evaluation targeting a quad-core processor and a set of server workloads, we show that Domino prefetcher improves system performance by 16% over the baseline with no data prefetcher and 6% over the state-of-the-art temporal data prefetcher.

I. Introduction

Server workloads have vast datasets beyond what can be captured by on-chip caches of modern processors [1], [2]. Consequently, server workloads encounter frequent cache misses during their execution. The cache misses prevent server processors from reaching their peak performance because cores are idle waiting for the data to arrive.

Data prefetching is a widely-used approach to eliminate cache misses or reduce their effect. While it has been shown that simple prefetching techniques, such as stride prefetching [3], [4], [5], are ineffective for server workloads [1], [6], more advanced data prefetchers may eliminate or reduce the negative effect of data misses. One of the promising prefetching techniques is temporal prefetching [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17].

[1]This work was done while the author was at Sharif University of Technology. He is currently affiliated with the Institute for Research in Fundamental Sciences (IPM).

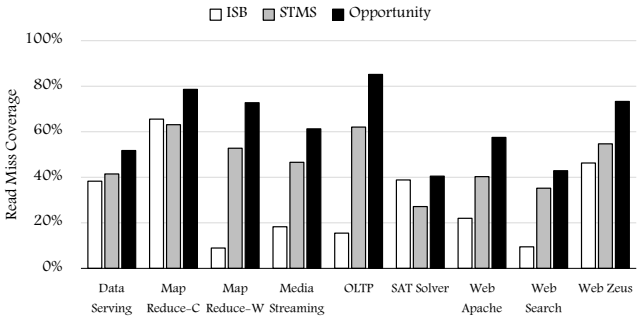

Figure 1. Read miss coverage of two state-of-the-art temporal data prefetchers with unlimited storage versus the opportunity.

Temporal data prefetchers record the sequence of past cache misses and use them to predict future cache misses. Temporal prefetching works because programs consist of loops, and hence, the sequence of addresses, and consequently, miss addresses repeat [18], [19]. Upon a miss, temporal prefetchers look up the history to find a match (usually the most recent match) and replay the sequence of misses after the match for eliminating future misses. Many pieces of prior work [20], [21], [22], [23] demonstrated the effectiveness of temporal prefetching in reducing data misses and boosting the performance of processors. A variant of temporal prefetching has been implemented in IBM Blue Gene/Q, where it is named *List Prefetching* [24].

Temporal prefetching is suitable for accelerating chains of *dependent* data misses, which are common in pointer-chasing applications [25], [26] (e.g., OLTP [27]). A dependent data miss refers to a data access that results in a cache miss while the access is dependent on a piece of data from a prior cache miss. These misses have a negative effect on the performance of processors, as they usually stall the core because both misses are fetched serially [28], [29]. The length of the chain of dependent misses varies across applications and across different chains in a particular application, ranging from a couple to hundreds of thousands of misses [18]. While stride [3], [4], [5] or spatial [30], [31], [32], [33], [34] prefetchers are usually incapable of prefetching dependent misses [22] due to the lack of stride/spatial access patterns among dependent misses, temporal prefetchers can capture such misses, and hence, boost performance through substantially increasing

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00021

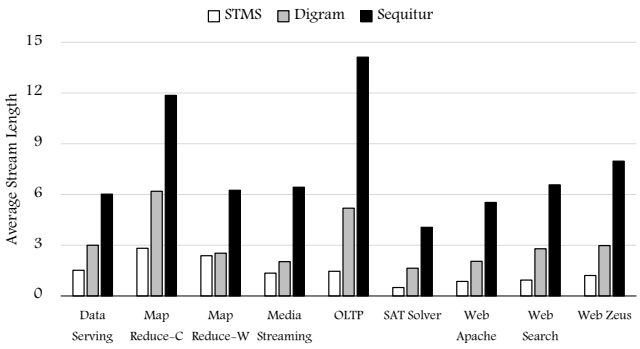

Figure 2. Average stream length with STMS, Digram, and Sequitur.

the memory-level parallelism (MLP).

While existing temporal data prefetchers are useful at eliminating cache misses, we observe that there is a significant gap between what they offer and what opportunity analysis shows for temporal prefetching. Figure 1 shows the opportunity of temporal prefetching and what STMS [10] and ISB [13], two state-of-the-art temporal data prefetchers, offer for several big-data server applications. Like prior studies of measuring repetitiveness in data misses [7], [18], [22], we use the Sequitur hierarchical data compression algorithm [35] to identify the opportunity of temporal prefetching. While STMS looks for temporal correlation in the global miss sequence, ISB applies temporal correlation to the PC-localized miss sequences. As shown, there is a significant gap between the opportunity and what existing state-of-the-art temporal prefetchers offer across all workloads. On average, the best-performing temporal prefetcher, i.e., STMS, captures less than 47% of data misses.

This work aims to bridge the gap between what temporal prefetching techniques offer and the opportunity. Corroborating prior work [21], our studies show that PC localization is not useful for temporal correlation in server workloads. This fact is evident in Figure 1: ISB, a PC-localized temporal prefetcher, offers lower coverage than STMS that benefits from the sequence of global misses for prefetching[1].

We observe that the lookup mechanism is responsible for the gap between what the state-of-the-art global-miss-sequence-based prefetcher (i.e., STMS) offers and the opportunity. STMS relies on a single miss address to identify a stream in the history. Unfortunately, a single miss address cannot distinguish two streams that begin with the same miss address. Consequently, STMS frequently picks a wrong stream for prefetching, as is evident from Figure 1. We observe that if instead of a single miss address, two consecutive miss addresses are used in the lookup mechanism, the chosen streams will be longer, and hence, more useful for prefetching. Figure 2 shows the average stream length for Sequitur, which picks the longest stream, STMS that selects a stream based on the last occurrence of the miss address, and Digram [21] that chooses a stream based on the last appearance of two previous misses for several server workloads. In this experiment, we refer to a stream as the sequence of consecutive correct prefetches. The figure shows that using two miss addresses in the lookup mechanism (i.e., Digram) instead of one (i.e., STMS) results in longer streams. While average stream length with Sequitur is 7.6, the stream length reduces to 1.4 for STMS. As a result of shorter stream length, STMS looks up 2.7× more streams than Sequitur and at the end of each stream inevitably encounters a cache miss.

The idea of using last two consecutive misses in the lookup mechanism is evaluated in Wenisch's Ph.D. thesis [21] and is discarded due to not being effective, and has never been pursued for publication. Prior work [21] evaluated the effect of lookups with the last two consecutive misses on temporal prefetching with a prefetcher named Digram [21] and concluded that using a single miss address is more reasonable. Lookups with the last two consecutive misses ensure that a temporal prefetcher cannot issue prefetch requests for the first two addresses of a stream. As the length of streams in server workloads is short (7.6 on average), the benefit of having longer streams is compensated with the fact that we issue one fewer prefetch request for every stream. As single-address lookup is simple, prior work discarded the idea of using two consecutive miss addresses in the lookup mechanism of temporal prefetchers.

To address the problem, we use a combination of one and two last misses in the lookup mechanism of the proposed temporal prefetcher, named *Domino*. When a new stream begins, we use a single miss address to prefetch the next miss, and when the next miss or prefetch hit occurs, we use the last two cache accesses to identify a stream. We propose a practical design for Domino that benefits from an *Enhanced Index Table (EIT)* that is indexed by a single miss address. Unlike a conventional Index Table [11] that solely stores a pointer for each address in the history, EIT, in addition, keeps the subsequent miss of each address. Having the next miss of every address in the EIT enables Domino to (1) find the correct stream in the history using the last two misses, and (2) issue the first prefetch request of a stream in one round-trip memory access latency. Therefore, unlike STMS that requires two accesses to the off-chip memory to issue the first prefetch request of a stream, Domino prefetcher issues the first prefetch of a stream after one access to the memory, as it stores the next miss in its EIT. This improves the timeliness of the proposed prefetcher.

In this paper, we make the following contributions:

- We observe that the lookup mechanism of the state-of-the-art temporal prefetcher is responsible for the large gap between its coverage and the opportunity.
- We corroborate prior work that a lookup mechanism

[1]Later in Section V, we discuss why PC localization is ineffective for server workloads in the context of temporal prefetching.

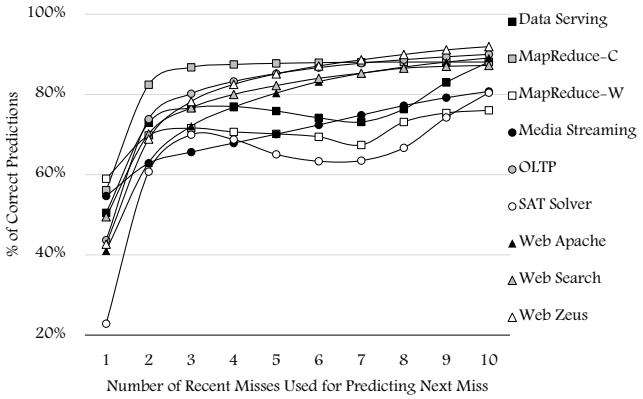

Figure 3. The fraction of lookups that result in a correct prediction over all lookups that find a match in the history, as a function of the number of addresses that they attempt to match.

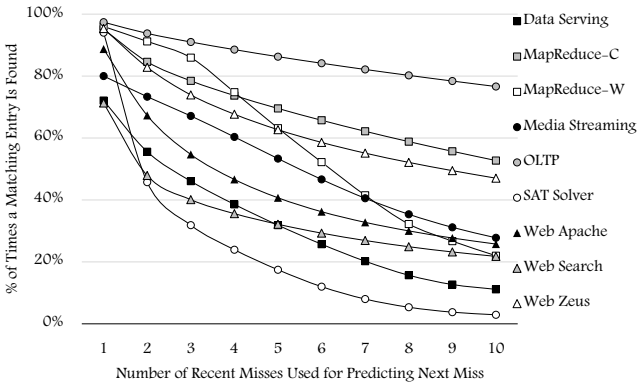

Figure 4. The fraction of lookups that find a match in the history over all lookups, as a function of the number of addresses that they attempt to match.

based on the last two miss addresses results in larger streams but at the cost of one fewer prefetch per stream, as compared to a single-address lookup. The net results show no significant difference between the two lookup mechanisms for server workloads.

- We propose a lookup mechanism that benefits from both one and two previous miss addresses to find larger streams without imposing fewer prefetch requests per stream.
- We incorporate the proposed lookup mechanism in a practical temporal prefetcher named Domino. The practical design has just one Index Table that is indexed by a single miss address (instead of two). Moreover, Domino prefetcher issues the first prefetch for a stream after one round-trip off-chip memory access latency (instead of two).
- We use a full-system simulation infrastructure to evaluate our proposal in the context of a four-core server processor on a set of server workloads. Our results show that our proposal offers, on average, 16% speedup over the baseline with no prefetcher, and 6% over the state-of-the-art temporal data prefetcher (i.e., STMS).

II. Motivation

Prefetching is a technique that aims to eliminate cache misses by bringing a piece of data to the cache before a processor's request for the piece of data. Temporal prefetching techniques rely on the repetitiveness of cache miss sequences. In the rest of this paper, we use the term *temporal prefetching* to refer to a technique that relies on the repetitiveness of the global cache misses for prefetching.

Upon a cache miss, a temporal prefetcher should look up the history to find a stream, i.e., a sequence of cache misses that tend to occur together, for the purpose of predicting future cache misses. Temporal prefetchers commonly locate the last occurrence of the missed address in the history and prefetch the addresses that follow the missed address in the history [6], [10]. As it is evident from Figure 1, the coverage of such a prefetcher is considerably less than that of an oracle that upon a miss, always picks the longest stream in the history.

In this section, we reduce the problem of temporal prefetching to identifying the *next* miss based on the previously-observed miss sequence. We offer justification for looking up the history with both last one and two miss addresses.

As the number of addresses that the lookup mechanism of a temporal prefetcher attempts to match increases, the prefetcher becomes more accurate. Figure 3 shows the fraction of lookups that lead to a correct prediction over all lookups that find a match in the history, as a function of the number of addresses that lookups attempt to match. As shown, the fraction of lookups that lead to a correct prediction is low if they match only a single address. Moreover, as the number of addresses that lookups match increases, the fraction of useful lookups also increases. However, increasing the number of addresses that a lookup matches beyond three yields only little improvements in the fraction of useful lookups. These results clearly show that temporal prefetchers that rely on just one miss address to look up the history (e.g., STMS [10]), frequently prefetch incorrectly.

Another interesting data point, as we increase the number of addresses that lookups match, is the fraction of lookups that find a match in the history. Only for lookups that find a match in the history, the temporal prefetcher issues a prefetch request. Figure 4 shows the fraction of lookups that find a match in the history as a function of the number of addresses that they match. As expected, the number of lookups that find a match reduces when the number of matched addresses increases. So temporal prefetchers that rely on a couple of misses to lookup the history (e.g., Digram [21]) miss a significant opportunity due to lack of finding a match in the previously-observed misses.

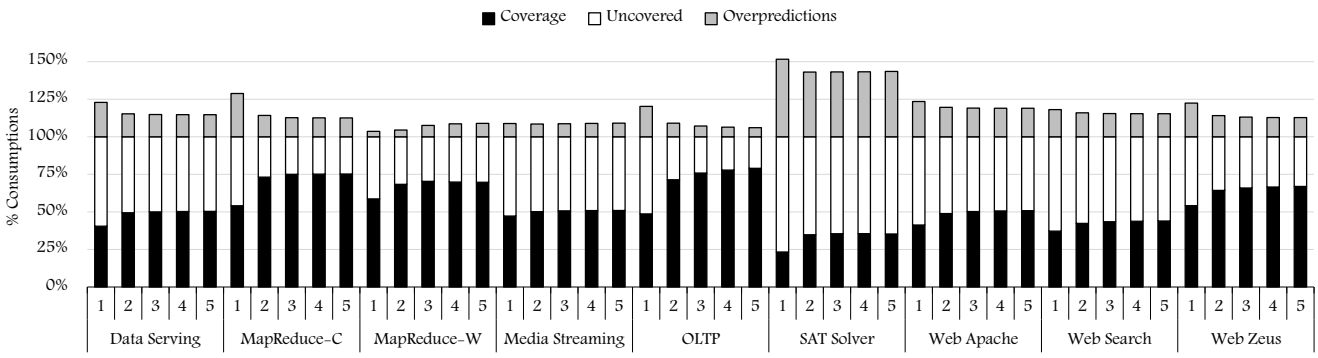

Figure 5. Coverage and overpredictions of a temporal prefetcher with unlimited storage for lookups with varying number of addresses.

This observation motivates using more than one miss address for lookup. A prefetcher can look up the history with the last N misses; if a match is found, the prefetcher issues a prefetch based on the match; otherwise, it looks up the history with one fewer miss in a recursive manner. This way, the prefetcher benefits from both high accuracy and high opportunity, overcoming the limitations of previously-proposed temporal prefetchers.

To show the importance of using more than one address for lookup, Figure 5 shows the coverage and overpredictions of a temporal prefetcher with lookups of varying number of addresses. In all cases, if a prefetcher uses N addresses for lookup, it attempts to find a match with $1, 2, 3, ..., N$ addresses and picks the match with the largest number of addresses. As shown, the coverage with a single-address lookup is low across all workloads while the overpredictions are high. As the number of looked up addresses increases, both coverage and overpredictions improve. However, only few workloads benefit from a temporal prefetcher that has a lookup mechanism that matches more than two addresses. Even on these workloads, the benefit of matching more than two addresses is insignificant.

We conclude that using two previous miss addresses is sufficient to correctly identify the right group of addresses in the history. As such, a temporal prefetcher should use both one and two last misses to have high coverage and accuracy.

III. The Proposal

Domino prefetcher is a temporal prefetching technique, and consequently, relies on past misses to predict and prefetch future memory references. Domino prefetcher performs its actions on cache misses and prefetch hits, which we refer to as triggering events.

Logically, every time a miss occurs, Domino prefetcher looks up the history with both the last two triggering events and the current triggering event to find a match. If the lookup using the last two triggering events finds a match, Domino uses the stream of misses that follows the match in the history for prefetching. Otherwise, if the lookup using the current triggering event finds a match, it just prefetches the first address after the match. Then, Domino waits for the next triggering event to occur. As looking up the history takes a long time, the following triggering event usually happens quickly, and Domino does not need to wait for a long time. If the triggering event is the hit of the prefetched cache block, Domino continues to prefetch from the already-found stream, and otherwise, Domino attempts to find a new stream. In case no match is located in the history for the two lookups, Domino does nothing and waits for the next triggering event to occur.

At any point in time, Domino has several active streams. If a miss occurs, Domino finds a new stream and replaces one of the old streams with it (round robin). In case a prefetch hit occurs, Domino continues to prefetch from the active stream that is responsible for the prefetch hit. Moreover, this stream becomes the most-recently-used stream in the LRU stack.

A. Background

STMS is the state-of-the-art temporal prefetcher, and Domino is built upon it. STMS has a dedicated per-core *History Table (HT)* that stores the sequence of misses observed by the core and a dedicated *Index Table (IT)* that for every observed miss address has a pointer to its last occurrence in the HT. The HT is a circular buffer, and the IT is a bucketized hash table [36] managed with LRU replacement policy. As both HT and IT require multi-megabyte storage for STMS to have a reasonable coverage, both tables are placed off chip in the main memory [10]. Consequently, every access to these tables (read or update) should be sent to the main memory, which is slow, and brings/updates a cache block worth of data.

Figure 6 shows the timing of events with STMS [10]. Upon a cache miss, a request is sent to the main memory to bring an entry of the IT that points to the last occurrence of the cache miss in the HT. Whenever the entry is received, STMS finds the pointer to the HT. Having the pointer, another request is sent to the main memory to bring a cache block worth of data from the HT. The data contains several

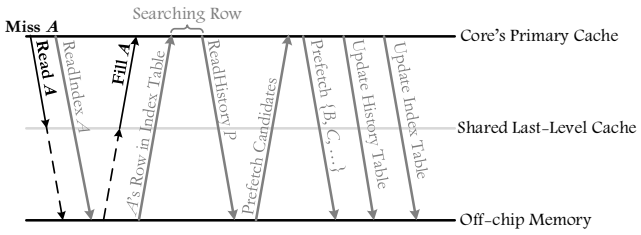

Figure 6. Timing of events in state-of-the-art temporal data prefetcher (i.e., STMS).

consecutive miss addresses that immediately followed the last occurrence of the missed address. Upon receiving the data, STMS sends prefetch requests for the addresses that follow the miss address.

With this implementation and for every stream, the temporal prefetcher needs to wait for two (serial) memory requests to be sent (one to read the IT and one to read the correct location of the HT) and their responses to come back to the prefetcher before issuing prefetch requests for the stream. The delay of the two off-chip memory accesses is compensated over several prefetch requests of a stream if the stream is long enough.

A row of the HT, which has a cache block worth of data, contains a sequence of consecutive data misses as observed by the core. As compared to the HT, IT is more complicated. IT is indexed by a hash of a single miss address (e.g., STMS) or two miss addresses (e.g., Digram). A row of the IT has some *tag-pointer* pairs. The *tag* along with the row number identifies a single miss address (or two miss addresses in Digram) and the *pointer* points to the last occurrence of the miss address(es) in the HT.

Every time a miss address is recorded in the end of the HT, its pointer in the IT needs to be updated to point to the last row of the HT. Unfortunately, the misses that are observed close to each other are usually mapped to different rows in the IT due to not being spatially correlated. As such, updating the pointers requires several accesses to the IT, which imposes off-chip bandwidth overhead and is time-consuming. To reduce the pressure on the off-chip memory, the state-of-the-art implementation of a temporal prefetcher benefits from statistical updates for the IT. Randomly, for every several index updates (e.g., eight), only one of them is recorded in the IT. It has been shown that this implementation offers the level of performance similar to that of the non-practical always-update implementation [10].

As both metadata tables are off-chip, the on-chip storage requirement of the prefetcher is negligible. The prefetcher needs only few kilobytes of per-core storage for recording misses and replaying prefetch candidates.

Domino uses both one and two last triggering events for prefetching. A naive implementation of Domino requires two ITs and one HT. One IT points to the last occurrence of every triggering event in the HT, and the other one points to the last appearance of every pair of consecutive triggering events in the HT. Compared to STMS and Digram, the naive implementation of Domino requires one more off-chip access per stream due to having two ITs, and as such, significantly wastes precious off-chip bandwidth. In this section, we detail a practical design for Domino that requires a single IT that is indexed by a single triggering event (and not two). Moreover, unlike STMS and Digram, the practical design can prefetch the first address of a stream in one round-trip memory access latency, which improves the timeliness of the prefetcher.

B. Practical Implementation

Domino prefetcher relies on an Enhanced Index Table (EIT) and an HT to record past triggering events and replay the sequence of future data misses. As the size of these two tables is very large (several megabytes), just like prior work [10], both tables are stored in the main memory.

Domino prefetcher allocates a contiguous portion of the physical address space for the two tables. Each core has a dedicated address space for the tables. The allocated address space is hidden from the operating system. The size of the allocated address space depends on the requirements of the workloads (it is usually several megabytes per core). As Domino prefetcher requires two tables, it statically divides the allocated address space into two parts. The start address of the EIT is recorded in a register named *EIT-Start*, and the start address of the HT is recorded in a register named *HT-Start*.

The memory system has a particular read request that, like an ordinary read request, fetches a block from memory but instead of placing the fetched block into the cache, puts the block into the prefetcher's on-chip metadata storage. There is no need to cache the content of the two tables in the cache hierarchy because metadata accesses exhibit neither spatial nor temporal locality.

Just like STMS and unlike Digram, the EIT in Domino is indexed by a single miss address. Associated with every tag, there are several address-pointer pairs, where the address is a miss experienced by the core and the pointer is a location in the HT. An *(a, p)* pair associated to a tag *t* indicates that the pointer to the last occurrence of miss address *t* followed by *a* is *p*. We refer to a tag and its associated address-pointer pairs as a *super-entry*, and to an address-pointer pair as an *entry*. Figure 7 shows the organization of Domino's EIT. Every row of the EIT has several super-entries, and each super-entry has several entries (three in our configuration). Domino keeps the LRU stack among both the super-entries and entries within each super-entry. To better understand the differences of Domino prefetcher's organization and those of the prior work, Figure 8 shows the structure of the metadata tables of STMS, Digram, and Domino.

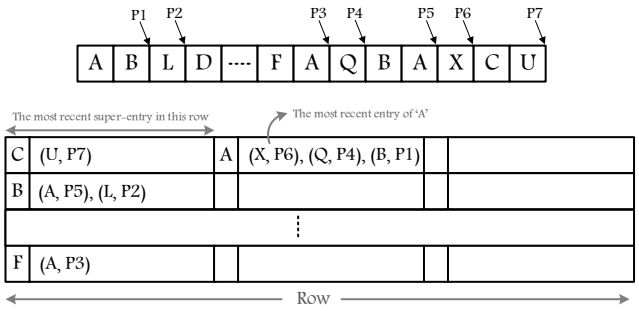

Figure 7. The details of the Enhanced Index Table in Domino prefetcher.

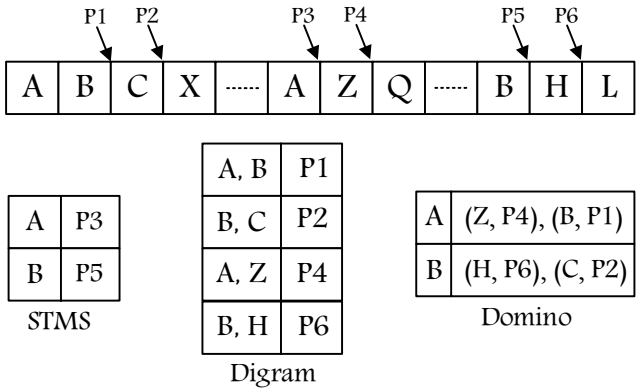

Figure 8. The details of the Index and History tables of STMS, Digram, and Domino.

Domino benefits from several storage elements next to each core. These elements are (1) a buffer to record the sequence of triggering events named *LogMiss*, (2) a buffer to keep the prefetched cache blocks named *Prefetch Buffer*, (3) a buffer that holds the sequence of addresses of an active stream named *PointBuf*, and (4) a buffer named *FetchBuf*.

Recording. Upon a triggering event, the address of the triggering event is appended to LogMiss. LogMiss has the capacity of two cache blocks. When one cache block worth of data is in LogMiss, Domino writes it to the end of the HT in the main memory and statistically updates the pointers of the written miss addresses in the EIT.

To update the EIT, for one out of every N triggering events (e.g., eight) written into the HT, the corresponding row of the EIT is fetched into FetchBuf. If a super-entry for the triggering event is not found in the fetched row, a new super-entry is allocated with the LRU policy. In the chosen super-entry, Domino attempts to find an entry for the address following the triggering event. If no match is found, a new entry is allocated with LRU policy. The pointer field of the entry is updated to point to the last row of the HT. Finally, Domino updates the LRU stack of entries and super-entries. Once Domino is done with the row, it is written back to the EIT.

Replaying. Upon a successful use of a prefetched block, Domino uses the active stream responsible for the prefetch hit and issues the next item of the stream (using PointBuf). Upon a cache miss, however, Domino prefetcher attempts to find a new stream and replaces the least-recently-used stream with it (which means discarding the contents of the prefetch buffer and PointBuf related to the replaced stream).

To find a new stream, Domino uses the missed address to fetch a row of the EIT. When the row is brought into PointBuf, Domino attempts to find the super-entry associated with the missed address. The delay of the search is tolerable because it is considerably smaller than the off-chip latency [10]. In case a match is not found, nothing will be done, and otherwise, a prefetch will be sent for the address field of the most recent entry in the found super-entry to be brought into the Prefetch Buffer.

When the next triggering event occurs (miss or prefetch hit), Domino searches the super-entry and picks the entry for which the address field matches the triggering event (might not be the most recent entry). In case no match is found, the stream is discarded and Domino uses the triggering event to bring another row from the EIT, and otherwise, Domino creates an active stream using the matched entry. It means that Domino sends a request to read the row of the HT pointed to by the pointer field of the matched entry to be brought into PointBuf. Once the sequence of miss addresses from the row of the HT arrives, Domino issues prefetch requests to be appended to the Prefetch Buffer.

As both recording and replaying may require accessing the tables, and since replaying is on the critical path but the recording is not, Domino prefetcher prioritizes replaying over recording. Only when replaying is done, Domino prefetcher attempts to follow the steps necessary for recording if it wants to access the tables.

Table I
EVALUATION PARAMETERS.

Parameter	Value
Chip	Four cores, 4 GHz
Core	SPARC v9 ISA, 8-stage pipeline, out-of-order execution, 4-wide issue, 128-entry ROB, 64-entry LSQ
I-Fetch Unit	64 KB, 2-way, 2-cycle load-to-use, next-line prefetcher, hybrid branch predictor, 16 K gShare & 16 K bimodal
L1-D Cache	64 KB, 2-way, 2-cycle load-to-use, 4 ports, 32 MSHRs
L2 Cache	4 MB, 16-way, 18-cycle hit latency, 64 MSHRs
Memory	45 ns delay, 37.5 GB/s peak bandwidth

IV. METHODOLOGY

Table I summarizes key elements of our methodology, with the following sections detailing the processor parameters, workloads, simulator, and evaluated designs.

A. Processor Parameters

The evaluated processor has four cores with 4 MB of last-level cache (LLC). The cache hierarchy of each core includes a 64 KB data and a 64 KB instruction cache. The 4 MB LLC

Table II
APPLICATION PARAMETERS.

CloudSuite	
Data Serving	Cassandra 0.7.3 Database 15GB Yahoo! Cloud Serving Benchmark
MapReduce-C	Hadoop 0.20.2 Bayesian Classification Algorithm
MapReduce-W	Hadoop 0.20.2 Mahout 0.4 Library
SAT Solver	Cloud9 Parallel Symbolic Execution Engine Four 5-byte and One 10-byte Arguments
Media Streaming	Darwin Streaming Server 6.0.3 7500 Clients, 60GB Dataset, High Bitrate
Web Search	Nutch 1.2/Lucene 3.0.1, 230 Clients 1.4 GB Index, 15 GB Data Segment
Web Server (SPECweb99)	
Web Apache	Apache HTTP Server v2.0, 16 K Connections FastCGI, Worker Threading Model
Web Zeus	Zeus Web Server v4.3 16 K Connections, FastCGI
Online Transaction Processing (TPC-C)	
OLTP	Oracle 10g Enterprise Database Server 100 Warehouses (10 GB), 1.4 GB SGA

is distributed among four slices. Cache line size is 64 bytes. The chip has two memory controllers that provide up to 37.5 GB/s of off-chip bandwidth.

B. Workloads

We simulate systems running *Solaris* and executing the workloads listed in Table II. We include a variety of server workloads from competing vendors, including online transaction processing, CloudSuite [37], and Web server benchmarks. Prior work [1] has shown that these workloads have characteristics representative of the broad class of server workloads.

C. Simulation Infrastructure

We use a combination of trace-based and timing full-system simulations to evaluate our proposal. Our trace-based analyses use traces collected from Flexus with in-order execution, no memory system stalls, and a fixed instruction-per-cycle (IPC) of 1.0.

We estimate the performance of various designs using Flexus full-system timing simulator [38]. Flexus timing simulator extends the Virtutech Simics functional simulator with timing models of cores, caches, on-chip protocol controllers, and interconnect. Flexus models the SPARC v9 ISA and is able to run unmodified operating systems and applications.

We use the SimFlex multiprocessor sampling methodology [39]. For each measurement, we launch simulations from checkpoints with warmed caches and branch predictors and run 300 K cycles to achieve a steady state of detailed cycle-accurate simulation before collecting measurements for the subsequent 150 K cycles. We use the ratio of the number of application instructions to the total number of cycles (including the cycles spent executing operating system code) to measure performance; this metric has been shown to accurately reflect overall system throughput of multiprocessors [39]. Performance measurements are computed with 95% confidence and an error of less than 4%.

D. Prefetchers' Configurations

We consider seven systems, as follows:

Baseline. As a recent study [1] showed that simple data prefetchers do not work for server workloads, the baseline has no data prefetcher. The baseline benefits from a next-line instruction prefetcher. All evaluated prefetchers are implemented on top of the baseline.

Variable Length Delta Prefetcher. We include VLDP [34] because it has similarities with the lookup mechanism of our proposal. VLDP is a prefetcher that relies on spatial correlation for prefetching and benefits from multiple previous deltas (the difference of two successive miss addresses in a page) for lookup. We equip VLDP with 16-entry DHB, 64-entry OPT, and three infinite-size DPTs. As this prefetcher works based on spatial correlation, it is orthogonal to our work and can be used together [22].

Irregular Stream Buffer. ISB [13] combines the use of PC localization and address correlation. We implement idealized PC/AC with an infinite-size history table. It has been shown that the idealized PC/AC has significantly better performance as compared to its practical implementation [13].

Sampled Temporal Memory Streaming. STMS [10] records miss sequences in a global per-core HT and locates streams through an IT. It benefits from a stream-end detection heuristic [10], [40] to reduce useless prefetches. We implement STMS with infinite-size metadata tables. Both HT and IT are located in the main memory. STMS can track four active streams at any given point in time. The sampling probability is set to 12.5% as suggested by the original proposal. Other parameters are taken from the original proposal.

Digram. Like STMS, Digram [21] stores misses in an HT and locates streams through an IT. We include Digram because, like Domino, it uses two misses for locating streams (but unlike Domino it does not look up the history with one miss address). We equip Digram with an infinite-size HT and IT. Metadata tables are located in the main memory. Digram can track four active streams at any given point in time. Digram also benefits from the stream-end detection mechanism. The sampling probability is set to 12.5%.

Domino. The EIT and HT are located off-chip in the main memory. We set the size of the metadata tables based on sensitivity analysis. LogMiss, Prefetch Buffer, PointBuf, and FetchBuf are 128 B, 2 KB, 256 B, and 64 B, respectively. Domino can track four active streams at any given point in time. Domino benefits from the stream-end detection mechanism. The sampling probability is set to 12.5%, similar to STMS and Digram.

Sequitur. Like prior studies of measuring repetitiveness of access sequences [7], [18], [22], we use the Sequitur

Figure 9. The coverage of Domino prefetcher as a function of HT size. Size of the EIT is unlimited.

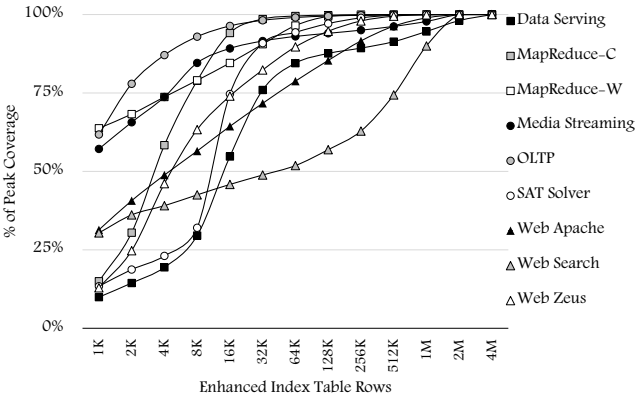

Figure 10. The coverage of Domino prefetcher as a function of EIT size. Size of the HT is 16 M entries.

hierarchical data compression algorithm [35] to identify temporal prefetching opportunity for data miss sequences. Sequitur is an algorithm that builds a grammar based on the input. The production rules of the grammar are formed in a way that capture repetitions in the input. Sequitur repeatedly reads a symbol from the input and extends its grammar accordingly. When a symbol is added, the grammar is adjusted in a way that captures new repetitions caused by the added symbol. We compare our prefetcher against Sequitur to see what fraction of the opportunity it has covered.

To have a fair evaluation, all prefetchers are trained using L1-D miss sequences, and all prefetchers prefetch into a small prefetch buffer near the L1-D cache with the capacity of 32 cache blocks. The degree of prefetching for all prefetchers is set to four.

V. Evaluation

A. Sensitivity Analysis

The storage requirement. The effectiveness of Domino prefetcher, or any other temporal prefetcher, directly depends on the size of the history on which the predictions are made. Figure 9 shows the coverage of Domino prefetcher for different numbers of entries (i.e., cache misses) in the HT for each workload. In this experiment, there is no limit on the size of the EIT. As the number of entries in the HT increases, the coverage of Domino prefetcher also increases because the prefetcher is able to make more correct predictions. Beyond 16 M entries, the coverage reaches its peak, effectively exploiting the opportunity. As HT is placed in the main memory, the space requirement is less important than the coverage of the prefetcher. So, we decide to have 16 M entries (85 MB) in the HT. Note that every 12 entries (addresses of triggering events) are placed into a row of the HT.

Having determined the size of the HT, Figure 10 shows the coverage of Domino prefetcher for different numbers of rows in the EIT for each workload. HT has 16 M entries. The coverage of Domino prefetcher reaches its peak when the size of the EIT becomes 2 M rows. So we decide to have an EIT with 2 M rows (128 MB). Note that both metadata tables are stored in the main memory, and as such, they do not impose space overhead to the processor.

B. Trace-based Evaluation

We compare Domino prefetcher against ISB [13], VLDP [34], STMS [10], and Digram [21]. As a point of reference, we also include the temporal opportunity measured using Sequitur [35]. VLDP is a spatial prefetching technique while ISB, STMS, Digram, and Domino are temporal prefetchers. For all temporal prefetchers except Domino, we assume unlimited-size storage for metadata. For Domino prefetcher, we limit the size of EIT to 2 M rows and HT to 16 M entries.

Figure 11 shows the coverage and overpredictions for the competing prefetching techniques when prefetching degree is one. Covered misses are the ones that are successfully eliminated by a prefetcher. Overpredictions are incorrectly prefetched cache blocks, which cause bandwidth overhead and potentially pollute the prefetch buffer. The incorrect prefetches are normalized against the number of cache misses in the baseline system.

On average, Domino offers 8% higher coverage as compared to the second-best prefetcher (i.e., STMS) and covers 56% of data misses. With respect to overpredictions, Domino is the second-best prefetcher (after Digram) across all workloads, except *SAT Solver*. Comparing Domino prefetcher against Sequitur, which identifies the temporal opportunity in the data miss sequence, Domino prefetcher captures slightly more than 90% of the opportunity.

While Figure 2 shows that Digram, which benefits from two-address lookups, has longer streams as compared to STMS, Figure 11 shows that STMS has slightly higher coverage as compared to Digram. To shed light on the reasons behind this phenomenon, Figure 12 shows the

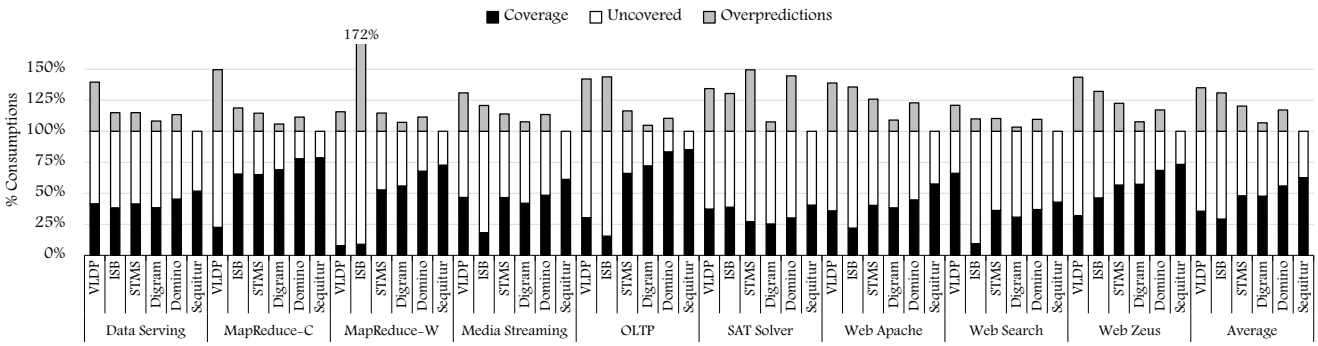

Figure 11. Domino prefetcher compared to other prefetchers. Prefetching degree of all prefetchers is one. For all temporal prefetchers except Domino, we assume unlimited-size history. For Domino prefetcher, we limit the size of EIT to 2 M rows and the HT to 16 M entries, respectively.

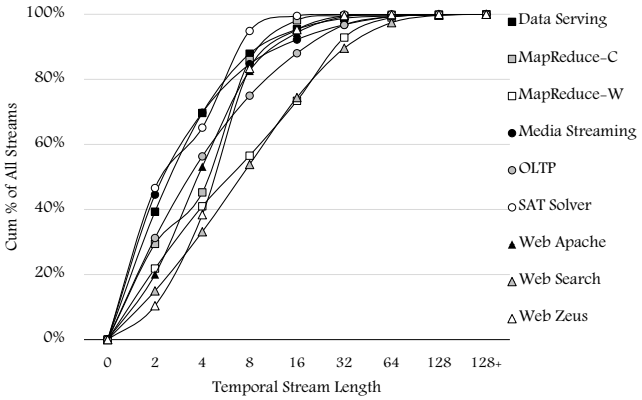

Figure 12. Histogram of stream length with Sequitur.

histogram of stream length for various workloads obtained using Sequitur analysis. Across all workloads, 10% to 47% of streams have a length of less than or equal to two, for which Digram cannot act. The significant majority of other streams are also short (have a length of less than eight). Unlike STMS, Digram consumes two accesses of a stream before issuing prefetch requests. As most of the streams are short, losing one prefetch request per stream has a significant negative effect on the coverage of the prefetcher. While the average stream length of Digram is larger than STMS, its coverage is slightly less than that of STMS. That is why prior work [21] picked STMS over Digram and concluded that two-address lookup has no practical advantage over single-address lookup.

Corroborating prior work [21], our results show that PC-localized temporal prefetchers (e.g., ISB) are not useful in the context of server workloads. PC-localized temporal prefetchers suffer from two fundamental obstacles: (1) PC localization breaks the strong temporal correlation among the global miss addresses (see the results of Sequitur), and (2) PC localization makes prefetchers predict the following misses of a memory instruction, which may not be the subsequent misses of the workload. Since server workloads have extensive instruction working sets [1], the re-execution of a specific memory instruction in the execution sequence may take a long time. Therefore, the prefetched blocks may be evicted before re-execution of the memory instruction.

VLDP performs poorly because, unlike LLCs, L1 caches cannot significantly exploit the spatial correlation of data accesses due to the low residency of data in the cache [41], [42]. Sharing the metadata of different pages in a unified history is another reason for the low accuracy of VLDP.

Prefetchers usually benefit from a prefetching degree greater than one to improve the timeliness of the prefetch requests (i.e., to have the prefetched blocks ready before the processor actually requests for them). Figure 13 shows the coverage and overpredictions of the competing prefetching techniques when the prefetching degree is four. As compared to Figure 11, overpredictions of many prefetchers have increased significantly.

Just like the prefetching degree of one, Domino prefetcher outperforms other prefetchers with respect to coverage. The second-best prefetcher is STMS. In all workloads, Domino either significantly outperforms STMS (e.g., 19% in *OLTP*) or with similar coverage substantially lowers the overpredictions. On average, Domino's overpredictions are one-third of those of STMS, and are close to those of Digram. The gap between the overpredictions of STMS and Domino-Digram grows with increasing the prefetching degree. It mainly comes from the lookup mechanism of STMS. STMS looks up history with a single miss address, and as such, frequently picks wrong streams. At the beginning of a wrongly-chosen stream, STMS prefetches as many incorrect cache blocks as the prefetching degree. Meanwhile, Domino and Digram often locate the correct stream using the two miss addresses and avoid the overpredictions.

Increasing the degree of prefetching significantly increases the possibility of early-eviction of prefetched cache blocks in ISB. Therefore, compared to degree one, both coverage and overpredictions have been worsen in this prefetcher.

With prefetching degree greater than one, upon predicting

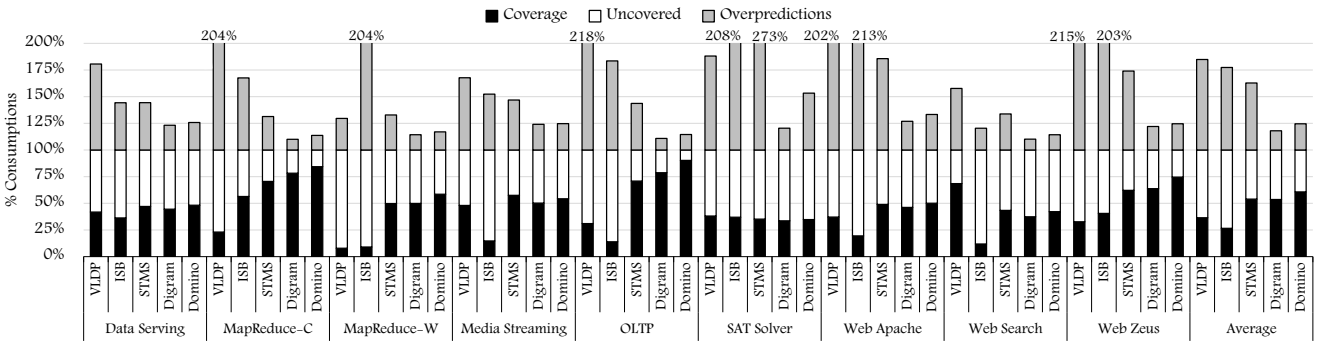

Figure 13. Domino prefetcher compared to other prefetchers. Prefetching degree of all prefetchers is four. For all temporal prefetchers except Domino, we assume unlimited-size history. For Domino prefetcher, we limit the size of EIT to 2 M rows and the HT to 16 M entries, respectively.

the next access in a page, VLDP uses the prediction as input to the metadata tables to make more predictions. We found that this approach is inaccurate for server workloads and the results become less accurate as the prefetching degree increases.

C. Cycle-Accurate Evaluation

Figure 14 shows the performance improvement of Domino prefetcher along with ISB, VLDP, STMS, and Digram, over a baseline with no prefetcher. All temporal prefetchers except Domino have unlimited storage for metadata in the main memory. For Domino prefetcher, we limit the size of EIT to 2 M rows and HT to 16 M entries. The prefetching degree for all prefetchers is four. The figure clearly shows the ability of Domino prefetcher in boosting performance.

In eight out of nine workloads, Domino outperforms other temporal prefetchers due to its higher coverage and/or better timeliness. The average performance improvement of Domino prefetcher over the baseline is 16%. The second-best prefetcher is STMS with the average performance improvement of 10%. As compared to VLDP, which is a recently-proposed spatial prefetcher, Domino offers 7% higher performance.

For most of the workloads, Domino provides a significant performance improvement. *Web Search* and *Media Streaming* have relatively high MLP, and hence, many of the misses that prefetchers capture, are already fetched in parallel with the out-of-order execution mechanism. Therefore, despite high coverage, prefetchers are unable to boost the performance of these workloads significantly. In *MapReduce-W*, temporal streams identified by the examined prefetching techniques are drastically short, and hence, the delay of fetching metadata from memory cannot be amortized over subsequent prefetches, resulting in less performance enhancement. *SAT Solver* produces its dataset on-the-fly during the execution, and thus, does not have a static and well-structured dataset [41]. Consequently, its memory accesses are hard-to-predict and all techniques manifest low coverage and high overpredictions, and accordingly, low performance improvement.

D. Off-chip Bandwidth Overhead

Figure 15 shows the off-chip bandwidth overhead of STMS, Digram, and Domino over the baseline with no prefetcher. Out of the three prefetchers, STMS has the highest and Digram and Domino have the lowest overhead. STMS has the highest off-chip traffic because of its high overprediction rate, as shown in Figure 13. Compared to STMS, Domino consumes less off-chip bandwidth due to (1) its low overprediction rate and (2) fewer metadata fetches, as Domino finds correct streams with fewer memory accesses than STMS. Compared to Digram, while Domino has slightly higher overprediction rate, it brings less metadata because more often lookups can find a match in its EIT (cf., Figure 4).

Compared to other prefetching techniques, the bandwidth requirement of temporal prefetchers is relatively high. Fortunately, server workloads consume only a small fraction of the available off-chip bandwidth offered by today's commercial processors [1]. Today's quad-core processors provide off-chip bandwidth, typically in the range of 37.5 GB/s [43] to 85 GB/s [44]. Meanwhile, the most bandwidth-hungry server workload (i.e., *Web Apache*) consumes only 8 GB/s of the available bandwidth.

The unused bandwidth can be utilized by a temporal prefetcher, like Domino, to improve the execution of server workloads. Using Domino, the bandwidth utilization ranges from 8.7% in *MapReduce-C* to 32.8% in *Web Apache*. We conclude that Domino offers the highest performance improvement with the lowest off-chip bandwidth overhead as compared to the state-of-the-art global-miss-based temporal prefetchers.

E. Spatio-Temporal Prefetching

VLDP relies on spatial correlation for prefetching while Domino captures temporal correlation of data accesses. As each technique targets different subset of misses, they can be used orthogonally [22]. VLDP uses patterns that fall into

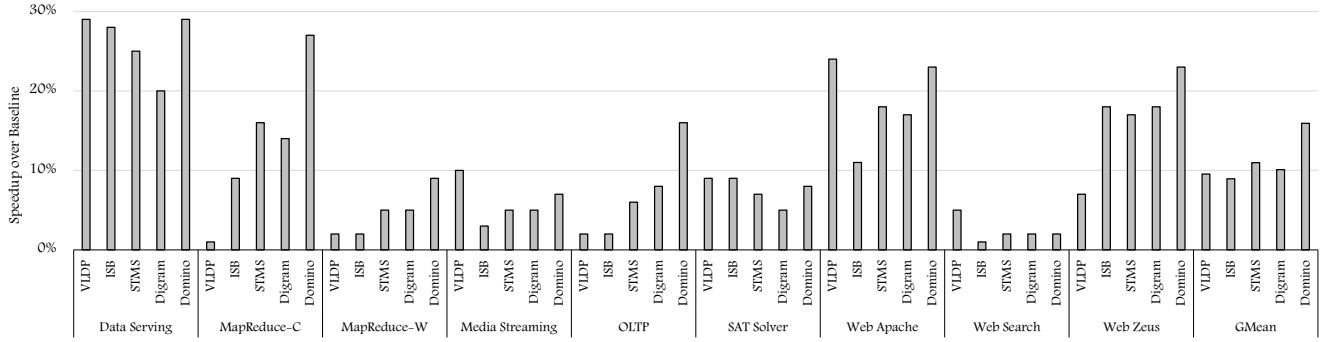

Figure 14. Performance improvement of Domino prefetcher compared with VLDP, ISB, STMS, and Digram. For temporal prefetchers except Domino, we assume unlimited-size storage for history in the main memory. For Domino prefetcher, we limit the size of EIT to 2 M rows and the HT to 16 M entries, respectively.

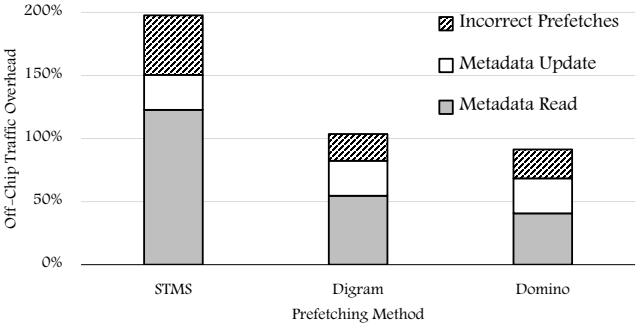

Figure 15. Bandwidth overhead of STMS, Digram and Domino over the baseline with no data prefetcher.

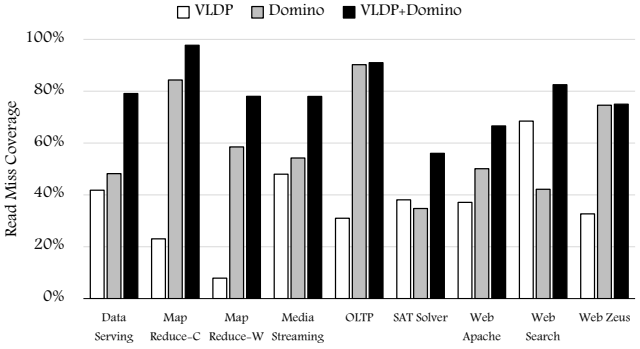

Figure 16. Spatio-Temporal prefetching.

a single page and is able to prefetch unobserved misses but is incapable of capturing consecutive misses that fall across pages. Domino replays previously-observed miss sequences, regardless of their spatial region in the memory, but is unable to prefetch cold misses. Each technique captures a particular type of misses, leaving the other type unpredicted.

To demonstrate the orthogonality of these techniques, we implement both of them in a single system. We stack Domino to a system that has VLDP. Domino trains and prefetches on misses that VLDP cannot capture. As Figure 16 shows, there is a large fraction of misses that are predictable solely by one of these techniques. On average, the combination of VLDP and Domino can cover 43%/20% more misses than VLDP/Domino alone.

The effectiveness of spatio-temporal prefetching drastically varies across workloads. In *Data Serving*, spatio-temporal prefetching efficiently increases the coverage of VLDP and Domino by 37% and 30%, respectively. Meanwhile, spatio-temporal prefetching provides almost no advantage over Domino in *OLTP*. An extreme behavior is observed in *MapReduce-W*, where the coverage of the combination of VLDP and Domino is higher than the arithmetic sum of the individual coverage of the two prefetchers. We found that, in this case, the remaining misses of a system with VLDP has higher temporal correlation than the original misses of the system without VLDP. With VLDP as the baseline prefetcher, the length of temporal streams extracted by Domino roughly doubles, which results in higher coverage and effectiveness of the temporal prefetcher.

VI. Related Work

Data prefetching is an active research area in computer architecture. Thread-based prefetching techniques [12], [45], [46], [47], [48], [49], [50], [51] exploit idle thread contexts to execute threads that prefetch for the main program thread. However, the extra resources the prefetcher threads need may not be available when the processor is fully utilized.

Software-based techniques [7], [52], [53], [54], [55], [56], [57], [58], [59] use compiler or programmer hints to issue prefetch operations. However, the complicated access patterns and fast changes in the dataset of big-data server applications make prefetching more difficult for such approaches.

Complex access patterns in the context of hardware prefetching are considered in many pieces of recent work [60], [61], [62], [63], [64], [65]. Most of prior work is either not temporal and/or is far from covering most of the temporal opportunity. In this work, we proposed a practical temporal data prefetcher and showed that it covers a significant fraction of the temporal opportunity.

VII. Conclusion

Data misses are a major source of performance degradation in server applications. Data prefetching is a technique for reducing the number of cache misses or their negative effect. Among data prefetching techniques, temporal prefetching has high potential in eliminating data misses due to existence of high temporal correlation among data accesses. Unfortunately, existing temporal prefetching techniques fall significantly short of efficiency and cannot capture the opportunity and minimize the number of data misses. In this paper, we proposed Domino, a temporal data prefetcher, and showed that it achieves more than 90% of the theoretical opportunity.

Acknowledgment

We thank the anonymous reviewers for their valuable comments. We appreciate the reviewers of IEEE CAL for the positive feedback on the preliminary version of this work [66]. The authors would like to thank members of the IPM HPC center, especially Armin Ahmadzadeh, for maintaining and managing the cluster that is used to conduct the experiments. This work was supported in part by a grant from Iran National Science Foundation (INSF). The research of the third author was partially supported by the research deputy of Sharif University of Technology.

References

[1] M. Ferdman *et al.*, "Clearing the Clouds: A Study of Emerging Scale-Out Workloads on Modern Hardware," in *ASPLOS*, 2012.
[2] P. Lotfi-Kamran *et al.*, "Scale-Out Processors," in *ISCA*, 2012.
[3] J.-L. Baer and T.-F. Chen, "An Effective On-chip Preloading Scheme to Reduce Data Access Penalty," in *Supercomputing*, 1991.
[4] T. Sherwood *et al.*, "Predictor-Directed Stream Buffers," in *MICRO*, 2000.
[5] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers," in *ISCA*, 1990.
[6] T. F. Wenisch *et al.*, "Temporal Streaming of Shared Memory," in *ISCA*, 2005.
[7] T. M. Chilimbi and M. Hirzel, "Dynamic Hot Data Stream Prefetching for General-purpose Programs," in *PLDI*, 2002.
[8] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," in *ISCA*, 1997.
[9] Y. Chou, "Low-Cost Epoch-Based Correlation Prefetching for Commercial Applications," in *MICRO*, 2007.
[10] T. F. Wenisch *et al.*, "Practical Off-chip Meta-data for Temporal Memory Streaming," in *HPCA*, 2009.
[11] K. J. Nesbit and J. E. Smith, "Data Cache Prefetching Using a Global History Buffer," in *HPCA*, 2004.
[12] Y. Solihin *et al.*, "Using a User-level Memory Thread for Correlation Prefetching," in *ISCA*, 2002.
[13] A. Jain and C. Lin, "Linearizing Irregular Memory Accesses for Improved Correlated Prefetching," in *MICRO*, 2013.
[14] M. Ferdman and B. Falsafi, "Last-Touch Correlated Data Streaming," in *ISPASS*, 2007.
[15] Z. Hu *et al.*, "TCP: Tag Correlating Prefetchers," in *HPCA*, 2003.
[16] Z. Hu *et al.*, "Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior," in *ISCA*, 2002.
[17] A.-C. Lai *et al.*, "Dead-block Prediction & Dead-block Correlating Prefetchers," in *ISCA*, 2001.
[18] T. F. Wenisch *et al.*, "Temporal Streams in Commercial Server Applications," in *IISWC*, 2008.
[19] T. M. Chilimbi, "On the Stability of Temporal Data Reference Profiles," in *PACT*, 2001.
[20] D. G. Perez *et al.*, "Microlib: A Case for The Quantitative Comparison of Micro-Architecture Mechanisms," in *MICRO*, 2004.
[21] T. F. Wenisch, *Temporal Memory Streaming*. PhD thesis, Carnegie Mellon University, 2007.
[22] S. Somogyi *et al.*, "Spatio-Temporal Memory Streaming," in *ISCA*, 2009.
[23] T. F. Wenisch *et al.*, "Making Address-Correlated Prefetching Practical," *IEEE Micro*, 2010.
[24] R. Haring *et al.*, "The IBM Blue Gene/Q Compute Chip," *IEEE Micro*, 2012.
[25] O. Mutlu *et al.*, "Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns," in *MICRO*, 2005.
[26] A. Roth *et al.*, "Dependence Based Prefetching for Linked Data Structures," in *ASPLOS*, 1998.
[27] P. Ranganathan *et al.*, "Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors," in *ASPLOS*, 1998.
[28] M. Hashemi *et al.*, "Accelerating Dependent Cache Misses with an Enhanced Memory Controller," in *ISCA*, 2016.
[29] Y. Chou *et al.*, "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism," in *ISCA*, 2004.
[30] J. F. Cantin *et al.*, "Stealth Prefetching," in *ASPLOS*, 2006.
[31] C. F. Chen *et al.*, "Accurate and Complexity-Effective Spatial Pattern Prediction," in *HPCA*, 2004.
[32] S. Kumar and C. Wilkerson, "Exploiting Spatial Locality in Data Caches Using Spatial Footprints," in *ISCA*, 1998.
[33] S. Somogyi *et al.*, "Spatial Memory Streaming," in *ISCA*, 2006.
[34] M. Shevgoor *et al.*, "Efficiently Prefetching Complex Address Patterns," in *MICRO*, 2015.
[35] C. G. Nevill-Manning and I. H. Witten, "Identifying Hierarchical Structure in Sequences: A Linear-time Algorithm," *Journal of Artificial Intelligence Research*, 1997.
[36] T. H. Cormen *et al.*, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
[37] CloudSuite. http://cloudsuite.ch.
[38] Flexus. http://parsa.epfl.ch/simflex/flexus.html.
[39] T. F. Wenisch *et al.*, "SimFlex: Statistical Sampling of Computer System Simulation," *IEEE Micro*, 2006.
[40] M. Ferdman *et al.*, "Temporal Instruction Fetch Streaming," in *MICRO*, 2008.
[41] D. Jevdjic *et al.*, "Die-Stacked DRAM Caches for Servers," in *ISCA*, 2013.
[42] D. Jevdjic *et al.*, "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache," in *MICRO*, 2014.
[43] INTEL® XEON® PROCESSOR E3-1220 V6. https://www.intel.com/content/www/us/en/products/processors/xeon/e3-processors/e3-1220-v6.html.
[44] INTEL® XEON® PROCESSOR E7-8893 V4. https://www.intel.com/content/www/us/en/products/processors/xeon/e7-processors/e7-8893-v4.html.
[45] C.-K. Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *ISCA*, 2001.
[46] M. Annavaram *et al.*, "Data Prefetching by Dependence Graph Precomputation," in *ISCA*, 2001.
[47] J. D. Collins *et al.*, "Dynamic Speculative Precomputation," in *MICRO*, 2001.
[48] J. D. Collins *et al.*, "Speculative Precomputation: Long-range Prefetching of Delinquent Loads," in *ISCA*, 2001.
[49] M. Kamruzzaman *et al.*, "Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads," in *ASPLOS*, 2011.
[50] C. Zilles and G. Sohi, "Execution-based Prediction Using Speculative Slices," in *ISCA*, 2001.
[51] I. Atta *et al.*, "Self-contained, accurate precomputation prefetching," in *MICRO*, 2015.
[52] T. C. Mowry *et al.*, "Design and Evaluation of a Compiler Algorithm for Prefetching," in *ASPLOS*, 1992.
[53] C.-K. Luk and T. C. Mowry, "Compiler-Based Prefetching for Recursive Data Structures," in *ASPLOS*, 1996.
[54] A. Roth and G. S. Sohi, "Effective Jump-pointer Prefetching for Linked Data Structures," in *ISCA*, 1999.
[55] E. Ebrahimi *et al.*, "Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems," in *HPCA*, 2009.
[56] M. H. Lipasti *et al.*, "SPAID: Software Prefetching in Pointer- and Call-Intensive Environments," in *MICRO*, 1995.
[57] A. Fuchs *et al.*, "Loop-aware memory prefetching using code block working sets," in *MICRO*, 2014.
[58] J. Lu *et al.*, "The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System," in *MICRO*, 2003.
[59] Z. Wang *et al.*, "Guided Region Prefetching: A Cooperative Hardware/Software Approach," in *ISCA*, 2003.
[60] J. Kim *et al.*, "Path Confidence Based Lookahead Prefetching," in *MICRO*, 2016.
[61] D. Kadjo *et al.*, "B-Fetch: Branch Prediction Directed Prefetching for Chip-Multiprocessors," in *MICRO*, 2014.
[62] P. Michaud, "Best-Offset Hardware Prefetching," in *HPCA*, 2016.
[63] S. H. Pugsley *et al.*, "Sandbox Prefetching: Safe Run-time Evaluation of Aggressive Prefetchers," in *HPCA*, 2014.
[64] X. Yu *et al.*, "IMP: Indirect Memory Prefetcher," in *MICRO*, 2015.
[65] Y. Ishii *et al.*, "Access Map Pattern Matching for Data Cache Prefetch," in *Supercomputing*, 2009.
[66] M. Bakhshalipour *et al.*, "An Efficient Temporal Data Prefetcher for L1 Caches," *IEEE CAL*, 2017.