Practical Off-chip Meta-data for Temporal Memory Streaming

Thomas F. Wenisch[*], Michael Ferdman[‡], Anastasia Ailamaki[‡], Babak Falsafi[‡] and Andreas Moshovos[†]

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

[*]*University of Michigan* [†]*University of Toronto*

[‡]*Ecole Polytechnique Fédérale de Lausanne and Carnegie Mellon University*

Abstract

Prior research demonstrates that temporal memory streaming and related address-correlating prefetchers improve performance of commercial server workloads though increased memory level parallelism. Unfortunately, these prefetchers require large on-chip meta-data storage, making previously-proposed designs impractical. Hence, to improve practicality, researchers have sought ways to enable timely prefetch while locating meta-data entirely off-chip. Unfortunately, current solutions for off-chip meta-data increase memory traffic by over a factor of three.

We observe three requirements to store meta-data off chip: minimal off-chip lookup latency, bandwidth-efficient meta-data updates, and off-chip lookup amortized over many prefetches. In this work, we show: (1) minimal off-chip meta-data lookup latency can be achieved through a hardware-managed main memory hash table, (2) bandwidth-efficient updates can be performed through probabilistic sampling of meta-data updates, and (3) off-chip lookup costs can be amortized by organizing meta-data to allow a single lookup to yield long prefetch sequences. Using these techniques, we develop Sampled Temporal Memory Streaming (STMS), a practical address-correlating prefetcher that keeps predictor meta-data in main memory while achieving 90% of the performance potential of idealized on-chip meta-data storage.

1. Introduction

Memory access latency continues to pose a crucial performance bottleneck for commercial server workloads [11]. System designers employ a variety of strategies to bridge the processor-memory performance gap. On the software side, efforts are under way to restructure server workloads to increase on-chip data sharing and reuse [15,19]. Although these efforts reduce off-chip accesses, application working sets continue to exceed available cache capacity. On the hardware side, multi-threading is effective in improving throughput when abundant software threads are available, but does not improve response time [11,20].

Prefetching improves both throughput and response time by increasing memory level parallelism [6,7,24,27] and remains an essential strategy to address the processor-memory performance gap. Today's systems employ spatial/stride based prefetchers [22,29] because they are practical to implement. These prefetchers require simple hardware additions and minimal on-chip area [17]. However, the effectiveness of these prefetchers is limited in commercial workloads (e.g., online transaction processing), which are dominated by pointer-chasing access patterns [1,25,27].

In contrast to stride-based approaches, address-correlating prefetchers [3,5,6,9,10,13,16,18,21,23,27] are effective for repetitive, yet arbitrarily-irregular access patterns, such as the pointer-chasing access patterns of commercial workloads [5,6,26,27]. Address-correlating prefetchers associate a miss address with a set of possible successor misses, or, in Temporal Memory Streaming (TMS) [10,26,27] and similar recent proposals [6,9,21,23], a sequence of successors. For modern commercial workloads, early address-correlating prefetcher designs are impractical because the on-chip meta-data size required to capture correlations is proportional to an application's data set and requires megabytes of storage [13,16,18].

To improve practicality, recent address-correlating prefetchers store meta-data off chip in main memory [6,9,23,27]. Shifting correlation tables to main memory eliminates on-chip storage costs, but creates two new challenges. First, correlation table lookups incur one or more main memory access latencies, which delay prefetches. The correlation algorithm must be designed to tolerate this long lookup latency by targeting prefetches several misses ahead in the anticipated future miss sequence [6,27]. Second, the extra memory traffic used to lookup and maintain meta-data increases memory bandwidth pressure. Existing designs require correlation table lookups and updates on nearly every cache miss, incurring overhead traffic three times larger than the base system's read traffic.

We observe three key requirements to make off-chip meta-data practical: (1) minimal off-chip meta-data lookup latency, (2) bandwidth-efficient meta-data updates, and (3) off-chip lookup amortized over many accurate prefetches. We propose *hash-based lookup* to achieve the first requirement. In hash-based lookup, we use a hardware-managed hash table to index previously-recorded miss-address sequences within a log of prior misses. Hash-based lookup enables retrieval of the corresponding miss addresses with only two main-memory round-trips (one for the hashed index table lookup, and the second for the address sequence). We propose *probabilistic update* to achieve the second requirement. Probabilistic update applies only a randomly-selected subset of updates to the hashed index table, reducing index table maintenance bandwidth to practical levels. Because recurring miss-address sequences either tend to be long or repeat frequently, stale or missing index table entries do not sacrifice significant coverage. We address the third requirement by applying these two mechanisms to a previously-proposed prefetch meta-data organization where misses are logged continuously in a circular buffer [21,27]. By separating indexing and logging, this prefetcher organization allows a single lookup to predict long miss-address sequences of up to hundreds of misses [26,27]. We evaluate our practical design, Sampled Temporal Memory Streaming (STMS), through cycle-accurate full-system simulation of scientific and commercial multiprocessor workloads, to demonstrate:

- **Performance potential.** Ideal on-chip lookup would enable TMS to eliminate 19-99% of off-chip misses (40-60% in online transaction processing and web workloads), improving performance by up to 80% (5-18% for OLTP and Web).
- **Storage efficiency.** Because meta-data is located off chip, STMS requires only 2KB of on-chip prefetch buffers per core. For maximum effectiveness, STMS needs 64MB of meta-data in main memory, a small fraction of memory in servers.
- **Latency efficiency.** By using hash-based lookup to prefetch sequences of tens of misses, STMS mitigates main-memory meta-data access latency. A practical lookup mechanism achieves 90% of the performance potential of idealized lookup.
- **Bandwidth efficiency.** We show that probabilistic update reduces the memory traffic overhead of meta-data updates by a geometric mean factor of 3.4 with a maximum coverage loss of 6%.

2. Background

An address-correlating prefetcher learns temporal relationships between accesses to specific addresses. For instance, if address *B* tends to be accessed shortly after address *A*, an address-correlating prefetcher can learn this relationship and use the occurrence of *A* to trigger a prefetch of *B*. Address-correlating prefetchers succinctly capture pointer-chasing relationships, and thus substantially improve the performance of pointer-intensive commercial workloads [5,6,27].

Pair-wise–correlating prefetchers. The Markov prefetcher [16] is the simplest prefetcher design for predicting pair-wise correlation between an address and its successor (i.e., two addresses that tend to cause consecutive cache misses). The Markov prefetcher hardware is organized as a set-associative table that maps an address to several recently-observed possibilities for the succeeding miss. On each miss, the table is searched for the miss address, and if an entry is found, the likely successors are prefetched. Several pair-wise–correlating prefetchers build upon this simple design to optimize correlation table storage [13], or trigger prefetchers earlier to improve lookahead [12,18].

The key limitation of pairwise-correlating prefetchers is that they attempt to predict correctly only a single miss per prediction, limiting memory level parallelism and prefetch lookahead. More recent address-correlating prefetchers use a single correlation to predict a sequence of successor misses [6,9,10,21,23,27]. We adopt the terminology of [26] and refer to these successor sequences as *temporal streams*—extended sequences of memory accesses that recur over the course of program execution.

Temporal streaming. The observation that memory access sequences recur was first quantified in memory trace analysis by Chilimbi and Hirzel [4]. To exploit this observation, researchers initially proposed correlation tables that store a temporal stream (i.e., a short sequence of successors) rather than only a single future access in each set-associative correlation table entry [6,23]. The primary shortcoming of this set-associative organization is that temporal stream length is fixed to the size of the table entry, typically three to six successor addresses. However, offline analyses of miss repetition [4,9,26] have shown that temporal streams vary drastically in length, from two to hundreds of misses. Fixing stream length in the prefetcher design leads either to inefficient use of correlation table storage (if the entries are too large) or sacrifices lookahead and prefetch coverage (if entries are too small).

To support variable length temporal streams while maintaining storage efficiency, several designs separate the storage of address sequences and correlation data [10,21,27]. A *history buffer* records the application's recent miss-address sequence, typically in a circular buffer. An *index table* correlates a particular miss address (or other lookup criteria) to a location in the history buffer. The split-table approach allows a single index-table entry to point to a stream of arbitrary length, allowing maximal lookahead and prefetch coverage without substantial storage overheads.

In this study, our goal is not to improve the prediction accuracy of state-of-the-art address-correlating

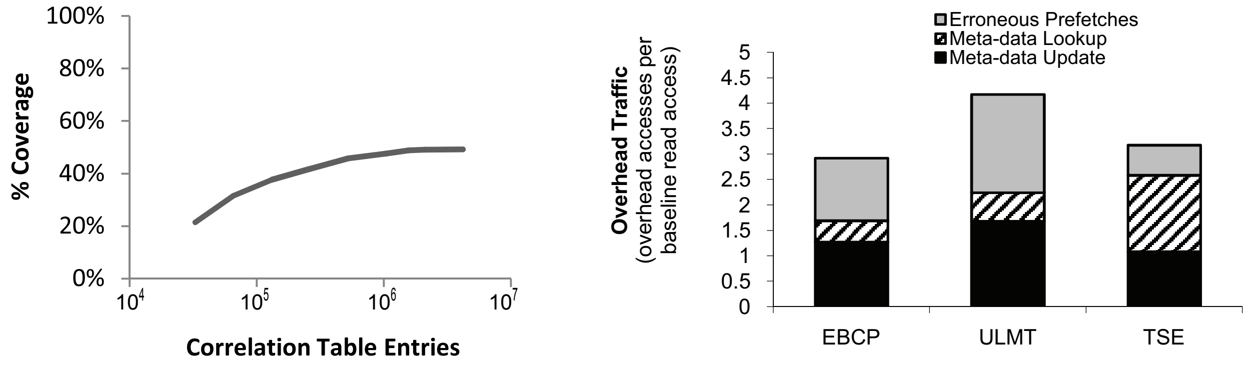


FIGURE 1: Practicality challenges. The left graph shows the number of correlation table entries required for a given coverage in commercial server workloads. One million correlation table entries can require up to 64MB of storage [6]. The right graph shows the memory traffic overheads of existing designs based on their published results [6,23,27]

prefetchers. Instead, we seek to identify and solve the key implementation barriers that make these prefetchers unattractive for practical deployment.

3. Practicality Challenges

More than a decade of research has repeatedly shown that address-correlating prefetchers can eliminate about half of all off-chip misses in pointer-intensive commercial server workloads, whereas stride prefetchers provide only minimal benefit [4,6,26,27]. Nevertheless, stride prefetchers are widely implemented, while, to date, no commercial design has implemented an address-correlating prefetcher. In this section, we enumerate the major practicality challenges of prior address-correlating prefetcher designs.

On-chip storage requirements. Initial address-correlating prefetcher designs located correlation tables entirely on-chip [13,16,18,21]. However, correlation table storage requirements are proportional to the application's working set. Hence, for commercial workloads, even the most storage-efficient design [13] requires megabytes of correlation table storage to be effective [6,27]. Figure 1 (left) shows the number of correlation table entries required for a given average coverage across commercial workloads for the idealized address-correlating prefetcher we analyze in detail in Section 5.2. Our result corroborates prior work [6]: to achieve maximum coverage in commercial workloads, correlation tables must store more than one million entries, which can require as much as 64MB of storage. High storage requirements make on-chip correlation tables impractical.

More recent prefetchers locate correlation meta-data in main memory [6,9,23,27], where multi-megabyte tables can easily be accommodated. However, off-chip tables lead to two new challenges: high lookup latency and increased memory bandwidth pressure.

High lookup latency. When correlation tables are off chip, each lookup requires at least one main memory access before prefetching can proceed. Unlike other predictors that can make use of the on-chip cache hierarchy to provide “virtual” on-chip lookup [2], address correlation tables are substantially larger than the on-chip caches and correlation entries exhibit minimal temporal locality. A correlation table entry is not reused until the address it corresponds to is evicted from on-chip caches. By that time, the correlation entry is also likely to be evicted.

Instead, the prefetching mechanism must be designed to account for long correlation table lookup latency. Epoch-based correlation prefetching (EBCP) [6] explicitly accounts for off-chip lookup latency and the memory level parallelism already obtained by out-of-order processing in choosing prefetch addresses. Rather than correlate an address to its immediate successors, EBCP skips over successor addresses that will be requested while the correlation table lookup is in progress. However, EBCP employs a set-associative correlation table, which, as noted in Section 2, bounds maximum stream length, limiting memory level parallelism, lookahead, and bandwidth efficiency.

With STMS, we instead mitigate lookup latency by following arbitrarily long streams to maximize the number of prefetches per lookup operation—a single lookup may lead to tens or hundreds of prefetches. The split-table meta-data organization and hash-based lookup mechanism are key to this strategy.

Memory bandwidth requirements. Address-correlating prefetchers with off-chip meta data substantially increase pressure on memory bandwidth. First, as with any prefetching mechanism, erroneous prefetches (cache blocks that are prefetched but never accessed) consume memory bandwidth, as prefetchers inherently trade increased memory bandwidth requirements to reduce effective access latency. However, off-chip correlation tables make matters worse: both lookups and updates require off-chip memory accesses.

Figure 1 (right) shows the average memory traffic overheads for three existing address-correlating prefetchers that store meta-data in main memory, the User Level Memory Thread (ULMT [23]), the Epoch-Based Correlation Prefetcher (EBCP [6]), and the Temporal Streaming Engine (TSE [27]), based on their published results. Overhead traffic is normalized to the number of memory reads without a prefetcher. “Erroneous Prefetches” are calculated directly from published accuracy and coverage. ULMT and TSE incur “Meta-data Lookup” traffic on each off-chip read miss (i.e., the remaining misses after prefetching), requiring one and three memory accesses per lookup, respectively. EBCP performs a single memory access to lookup its meta-data at the start of each off-chip miss epoch—that is, when the number of outstanding (non-prefetch) off-chip misses transitions from zero to one. ULMT and EBCP perform “Meta-data Update” following each lookup, both requiring three memory accesses per update. TSE updates its correlation tables on both off-chip misses and prefetcher hits, requiring slightly over one memory access per update on average.

As the figure shows, overhead traffic is triple the baseline read traffic without a prefetcher. Several factors mitigate the performance impact of massive traffic overheads in the existing designs. All three designs issue correlation table lookups and updates as low-priority traffic, prioritizing processor-initiated requests. ULMT collocates its prefetcher with an off-chip memory controller, and, hence, its meta-data traffic does not cross the processor’s pins. TSE’s meta-data lookups are embedded in existing cache coherence traffic. When unused memory bandwidth is abundant, overhead traffic can be absorbed by the memory system with minimal performance impact. However, as available memory bandwidth must be shared among cores in a multi-core system, memory bandwidth utilization is growing rapidly with chip multiprocessor scaling. Hence, the high traffic overhead of current main-memory correlation table designs limits their applicability.

If it were possible to store correlation tables on chip, lookup and update traffic would be of little concern—though large relative to off-chip traffic, required bandwidth can be easily sustained in dedicated on-chip structures. However, storage requirements preclude on-chip meta-data, requiring new solutions to improve bandwidth-efficiency for off-chip meta-data storage.

4. STMS Design

We leverage prior work in temporal streaming to reuse mechanisms and terminology whenever possible. We begin by enumerating the three key requirements for effective temporal streaming in Section 4.1. We then provide an overview of STMS in Section 4.2, constructing a generalized temporal streaming prefetcher that draws heavily from the stream-following mechanisms of the Temporal Streaming Engine (TSE) [27] and the predictor organization of the Global History Buffer (GHB) [21]. After we describe the basic hardware operation, the following sections provide details of how the proposed STMS mechanisms meet the requirements for efficient temporal streaming.

4.1. Requirements for Effective Temporal Streaming

Minimize Lookup Latency. Temporal-streaming prefetchers initiate predictor lookup on a trigger event (typically a cache miss). However, prefetches cannot be issued until an address sequence is located and retrieved. Cache misses incurred during this time result in lost prefetch opportunity, even if they comprise a predictable temporal stream [6]. Prior prefetchers targeting desktop/engineering applications rely on long temporal streams to overcome the startup cost [9]. However, unlike those applications, prior research [27] and our results (see Section 5.4) indicate that half of the temporal streams in commercial workloads are shorter than ten cache blocks. Therefore, effective streaming for commercial workloads requires minimal lookup latency to ensure timely prefetch.

Bandwidth-efficient Index Table Updates. Maintaining predictor meta-data in off-chip storage induces additional traffic across the memory interface. Spatial locality allows amortization of history buffer updates by storing multiple consecutive addresses with one off-chip write. Conversely, any index table updates are directed to randomized addresses and exhibit neither spatial nor temporal locality. Furthermore, each index table update incurs both a read and a write operation. Performing all updates on an un-optimized main-memory index table will therefore triple memory bandwidth consumption over a base system. Efficient bandwidth utilization is growing even more important in chip multiprocessor, where scaling in the number of cores per die will continue increasing strain on off-chip bandwidth [14].

Amortized Lookups. Another key requirement for effective temporal streaming is to amortize off-chip lookups over many successful prefetches, thereby keeping off-chip bandwidth low by reducing the number of index table lookups and history buffer reads. The static stream lengths imposed in previous single-table prefetcher designs fragment long temporal streams into short prefetch sequences (see Section 3), limiting prefetcher effectiveness, as half of the temporal streams

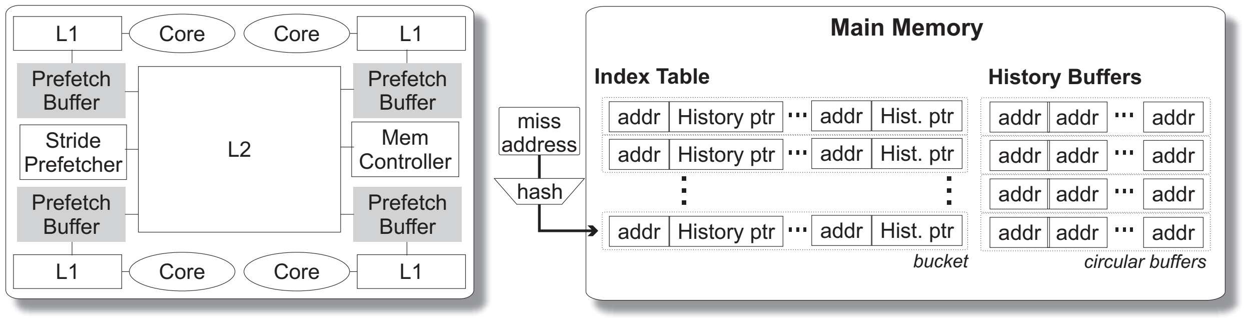


FIGURE 2: STMS block diagram.

found in commercial workloads are long (in excess of 10 misses). Furthermore, the bandwidth and congestion overhead of accessing the history structures for each short sub-sequence can match or exceed the bandwidth required to retrieve data. Widening correlation table entries to target long streams in a single-table prefetcher is also bandwidth-inefficient. Large correlation table entries that can contain many miss addresses result in considerable bandwidth overhead when useless addresses or empty space is retrieved from the history structures in the case of shorter streams.

4.2. Design Overview

Figure 2 shows a block diagram of a four-core single-chip processor equipped with STMS. STMS comprises on-chip prefetch buffers and queues and off-chip *index table* [21] and *history buffers* [21][1]. The prefetch buffers and queues, located along side the L1 victim caches [17], orchestrate the streaming process and act as temporary holding space for a small number of blocks that have been prefetched, but not yet accessed by the core. The off-chip structures are allocated in a private region of main memory, each core receives its own history buffer but all cores share a unified index table. The index table contains a mapping from physical addresses to pointers within the history buffer, facilitating lookup.

Recording temporal streams. STMS records correct-path off-chip misses and prefetched hits in the corresponding core's history buffer. To avoid polluting the history buffer with wrong-path accesses, instructions observing prefetched hits and off-chip loads are marked in the load-store queue [17]. Later, when a marked instruction retires, the effective physical address is appended to the history buffer. To minimize pin-bandwidth overhead, a cache-block-sized buffer accumulates entries which are then written to main memory as a group as proposed in [9]. As history buffer entries are created, the index table entry for the corresponding address may be updated to point to the new history buffer entry. As we discuss in Section 4.4, the majority of index table updates are skipped by probabilistic update to conserve memory bandwidth.

Each core's miss-address sequence is logged in a separate history buffer. Whereas each core's misses individually form temporal streams, when accesses from multiple cores are interleaved, repetitive sequences are obscured. In the case of multi-threaded cores, each thread requires its own history buffer. The index table is shared by all cores.

Lookup. Upon an off-chip read miss, STMS searches for a previously-recorded temporal stream that begins with the miss address. STMS performs a pointer lookup in the index table. If a pointer is found, the address sequence is read from the history buffer, starting at the pointer location. It is important to note that the STMS shared index table can locate a temporal stream from another core's history buffer.

Streaming cache blocks. Miss addresses read from the history buffer are held in a FIFO address queue. STMS prefetches addresses from the queue in order. Prefetched data are stored in the small, fully-associative prefetch buffer, avoiding cache pollution on erroneous prefetches. On correct prefetches, L1 misses are satisfied directly from the prefetch buffer while STMS continues to populate the address queue with addresses from the off-chip history buffer.

4.3. Latency-Efficient Hash-based Index Lookup

Any associative lookup structure can be used to implement an index table. We examined many possible structures (e.g., red-black trees, open address hash tables, direct-mapped tables), however these structures have unacceptable latency, bandwidth, or storage characteristics. To achieve low lookup latency, we elect to implement the index table as a bucketized probabilistic hash table [8] pictured in Figure 2 (right).

1. Although prior work in uniprocessors calls these structures *global history buffers*, we avoid the word *global* because *history buffers* in the CMP design are per-core.

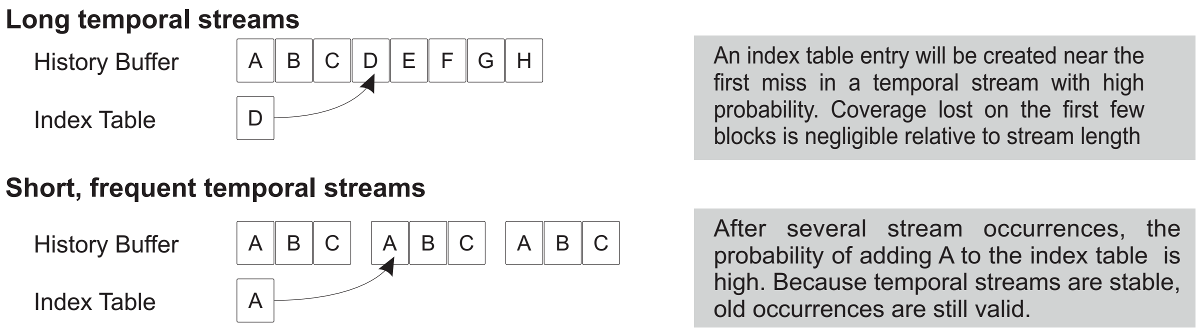

Long temporal streams
History Buffer
A B C D E F G H
Index Table
D
An index table entry will be created near the first miss in a temporal stream with high probability. Coverage lost on the first few blocks is negligible relative to stream length
Short, frequent temporal streams
History Buffer
A B C A B C A B C
Index Table
A
After several stream occurrences, the probability of adding A to the index table is high. Because temporal streams are stable, old occurrences are still valid.

FIGURE 3: Cases where probabilistic update is effective.

Physical addresses are hashed to select a bucket. Each bucket contains *n* {physical address, history buffer pointer} pairs, with *n* sized to the memory system interface width (i.e., one cache line). On lookup, the entire bucket is retrieved with one memory access, and searched linearly (linear search is negligible relative to the off-chip read latency). On update, the bucket is similarly retrieved and searched. If the trigger address is found in the bucket, the entry's history buffer pointer is updated. If the trigger address is missing, the least recently used entry of the bucket is replaced. Before being written back to memory, the elements are reshuffled to maintain LRU order. We find that assigning a low priority to predictor memory traffic is essential to minimize queueing-related stalls. To facilitate index table updates and to delay writeback until memory bandwidth is available, STMS employs a small (8 KB) bucket buffer.

The key advantage of our hash table design is low-latency index table lookup. In addition, the design results in high storage density (i.e., it can be fully loaded) and supports an arbitrary number of parallel reads and updates without synchronization, enabling independent parallel access from multiple cores.

4.4. Bandwidth-Efficient Probabilistic Index Update

Although only a small fraction of index table entries yields the vast majority of prefetch coverage, to date, no known property allows *a priori* distinction of useful index table entries at the time they are recorded. Rather than filtering index table updates, we propose *probabilistic update*. For every potential index table update, a coin flip, biased to a predetermined sampling probability, determines whether the update will or will not be performed. Index table update bandwidth is directly proportional to the sampling probability. For example, a 50% sampling probability halves bandwidth requirements. Probabilistic update is highly effective in reducing index table update bandwidth, while leading to only a small coverage loss.

Intuitively, sampling does not significantly reduce coverage for most cases: long temporal streams and short frequent temporal streams. Figure 3 illustrates both cases. For long temporal streams, probabilistic update likely skips several addresses before creating an index table entry pointing into the body of the stream. However, coverage loss on the first few blocks is negligible relative to the stream's length. For a sampling probability of one eighth, the probability of inserting at least one address into the index table reaches 50% within the first five blocks. For frequent temporal streams, the probability of inserting the first address into the index table grows with the number of stream recurrences. For short temporal streams, the index table may remain without a pointer for the first few occurrences, however a high appearance frequency results in an index update within a small number of occurrences.

4.5. Variable-Length Streams via Split Tables

The drawbacks of statically pre-determined stream length motivate our desire to support variable-length temporal streams. Variable-length temporal streams are made possible by splitting the history buffer and index table into separate structures, as outlined in Section 2. Short streams are accommodated with minimal storage overhead, while longer streams are accommodated by reading consecutive sections of the history buffer. A key result of this design is that long streams require only a single index lookup. We quantitatively contrast the latency- and bandwidth-efficiency of single-table and split-table organizations in Section 5.4.

To avoid streaming erroneous blocks past the end of a temporal stream within the history buffer, STMS annotates the history buffer entry following the last contiguous successfully-prefetched address. Whenever STMS encounters a marked entry, it pauses streaming, continuing only if the annotated address is explicitly requested by the core. In contrast to TSE [27], the STMS stream-end detection is highly bandwidth-efficient, requiring to read only one location from the history buffer to determine the end of stream.

Table 1: Application and system model parameters.

Online Transaction Processing (TPC-C)	
Oracle	Oracle 10g, 100 warehouse (10 GB), 16 clients, 1.4 GB SGA
DB2	DB2 v8, 100 warehouse (10 GB), 64 clients, 2 GB buffer pool
Decision Support (TPC-H on DB2 v8 ESE)	
Qry 2	Join-dominated, 480 MB buffer pool
Qry 17	Balanced scan-join, 480 MB buffer pool
Web Server (SPECweb99)	
Apache	Apache 2.0, 4K connect, FastCGI, worker threading model
Zeus	Zeus v4.3, 4K connections, FastCGI
Scientific	
em3d	768K nodes, degree 2, span 5, 15% remote
ocean	258x258 grid, 9600s relaxations, 20K res., err tol 1e-07
moldyn	19652 molecules, boxsize 17, 2.56M max interactions

Cores	UltraSPARC III ISA 4 GHz 8-stage pipeline; out-of-order 4-wide dispatch / retirement 96-entry ROB, LSQ
L1 D-Cache	64KB 2-way, 2-cycle load-to-use 3 ports, 32 MSHRs
Instruction Fetch Unit	64KB 2-way L1 I-cache 16-entry pre-dispatch queue Hybrid 16K gShare/bimodal branch pred. Next line prefetcher
Shared L2 Cache	8MB 16-way, 20-cycle access latency 16 banks, 64 MSHRs
Main Memory	3 GB total memory, 45 ns access latency 28.4 GB/s peak bandwidth, 64-byte transfers
Stride Prefetcher	32-entry buffer, max 16 distinct strides

5. Evaluation

The goal of our evaluation is to demonstrate that STMS (1) matches the coverage and performance of idealized temporal memory streaming while storing meta-data in off-chip memory, and (2) that it does not reduce performance of workloads that derive no benefit from temporal streaming.

5.1. Methodology

We evaluate STMS using a combination of trace-based and cycle-accurate full-system simulation using the FLEXUS infrastructure. FLEXUS builds on the *Virtutech Simics* functional simulator. We include four workload classes in our study: online transaction processing (OLTP), decision support (DSS), web server (Web), and scientific (Sci) applications. Table 1 (left) details our workload suite.

We model a four-core chip multiprocessor with private L1 caches and a shared L2 cache configured to represent an aggressive near-future high-performance processor. The minimum L2 hit latency is 20 cycles, but accesses may take longer due to bank conflicts or queueing delays. A total of at most 64 L2 accesses and off-chip misses may be in flight at any time. Further configuration details appear in Table 1.

We include a stride-based prefetcher in our base system [22,29]. All results report only coverage in excess of that provided by the stride prefetcher.

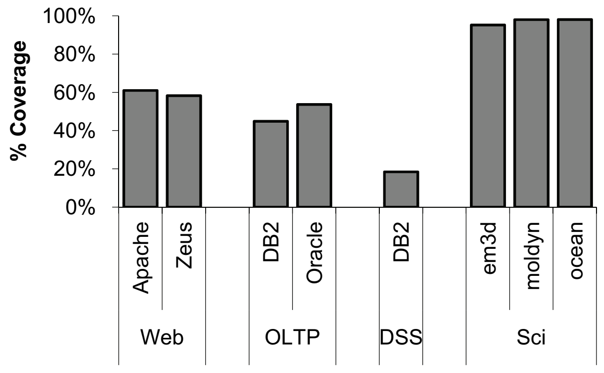
We measure performance using the SIMFLEX sampling methodology [28]. We report confidence intervals on change in performance using matched-pair sample comparison. Our samples are drawn over an interval of from 10s to 30s of simulated time for OLTP and web server workloads, over the complete query execution for DSS, and over a single iteration for scientific applications. We launch measurements from checkpoints with warmed caches, branch predictors, history buffer, and index table state, then warm queue and interconnect state for 100,000 cycles prior to measuring 200,000 cycles. We use the aggregate number of user instructions committed per cycle (i.e., committed user instructions summed over the 4 cores divided by total elapsed cycles) as our performance metric, which is proportional to overall system throughput [28].

5.2. Performance Potential of Idealized Prefetcher

We begin by demonstrating the performance potential of an idealized version of TMS with on-chip meta-data. The idealized prefetcher records a sequence of cache miss addresses in a "magic" on-chip history buffer that has impractically large storage capacity and zero-latency infinite lookup bandwidth.

Figure 4 presents the coverage (left) and speedup (right) achieved by the idealized prefetcher over our baseline system. Coverage is defined as the fraction of L2 cache misses eliminated by the prefetcher.

We corroborate prior work, showing that temporal memory streaming is an effective mechanism for eliminating cache misses in OLTP, web serving, and scientific computing workloads, and that temporal memory streaming is ineffective for DSS workloads because they exhibit non-repetitive access sequences where data is visited only once throughout execution. We also observe that, despite achieving high predictor coverage, minimal speedup opportunity is available in workloads whose dominant bottlenecks are not main memory accesses (in the case of OLTP Oracle, the primary bottlenecks are L1 instruction and data misses that hit in L2 and on-chip core-to-core coherence traffic). However, prefetch coverage indicates that even for such workloads, once other (on-chip) bottlenecks in the system are eliminated, temporal memory streaming offers a benefit.



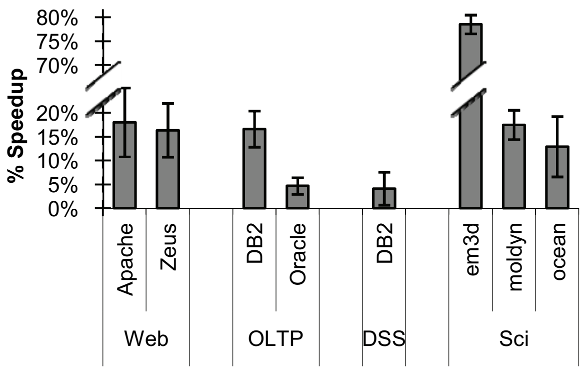


FIGURE 4: Prefetching potential. The left graph shows the prefetch coverage achieved by an idealized temporal streaming prefetcher. The right graph shows the corresponding performance impact.

Although the potential gains are evident in Figure 4, on-chip storage required to implement TMS is impractical (see Figure 1). We recognize that advances in the field will improve the coverage of stream-based address-correlating prefetchers beyond our idealized implementation of the proposed prior techniques [21,27]. However, improved predictors can leverage the basic mechanisms we propose in this work to facilitate off-chip lookup. Our aim is therefore not to improve over the idealized prior design that we describe, but instead to match the performance of idealized TMS. To this end, our goal is two-fold: (1) maintain performance improvement for the workloads where streaming is effective, (2) avoid adverse performance impact in workloads where streaming is ineffective. Accomplishing this goal requires bandwidth-, latency-, and storage- efficiency.

5.3. Achieving Storage Efficiency

On-chip structures. As demonstrated in Section 3, on-chip correlation meta-data are impractical, and, hence, STMS locates its history buffer and index table in main memory. On chip, STMS requires a prefetch buffer and address queue collocated with each core to track pending prefetch addresses and buffer prefetched data. Storage requirements for the address queue are negligible (under 128 bytes), while prefetch buffers each require 2KB. We do not report sensitivity to prefetch buffer size, as it has been studied extensively in prior work [6,27]. In addition, STMS uses a shared 8KB bucket buffer to store index table entries between lookup, update, and write back.

History buffer. The history buffer's main-memory storage requirements are driven by the prefetcher's meta-data reuse distance. The history buffer must be at least large enough to fit all intervening miss addresses between a recorded temporal stream and its previous occurrence. For commercial workloads, the reuse distance depends on the frequency at which data structures are revisited, resulting in a spectrum of distances, and giving rise to a smooth improvement in coverage as the size of the history buffer is increased. Conversely, scientific computing workloads typically exhibit a reuse distance proportional to the length of a single computational iteration and varies only with the size of the application's dataset.

Figure 5 (left) plots predictor coverage as a function of history buffer size. For our commercial workloads, the history buffer must be on the order of 32MB to achieve maximal coverage. For scientific workloads, coverage is bimodal: if the history buffer is sufficiently large to capture an entire iteration, coverage is nearly perfect; if the history buffer is insufficiently large, coverage is negligible. For both classes of workloads, the history buffer storage requirements are at least an order of magnitude greater than can be allocated on chip. However, relative to a server's main memory, history buffer footprint is small.

Index table. An ideal index table can locate the most recent occurrence of any miss address in the history buffers. Hence, in the worst case, if all addresses in the history buffer are distinct, then there must exist one index table entry per history buffer entry. In practice, far fewer index table entries are required.

To optimize index-table storage efficiency, we propose hash-based lookup. Hash-based lookup spreads miss addresses over buckets, applying the LRU replacement policy within each bucket. The LRU policy brings useful history buffer pointers to the top, naturally aging out unneeded entries. Figure 5 (right) plots predictor coverage as a function of the hash-table size for an ideal (unbounded) history buffer. Hash-based lookup achieves maximum coverage with a 16MB main-memory index table, retaining only a fraction of the index entries of an idealized prefetcher.

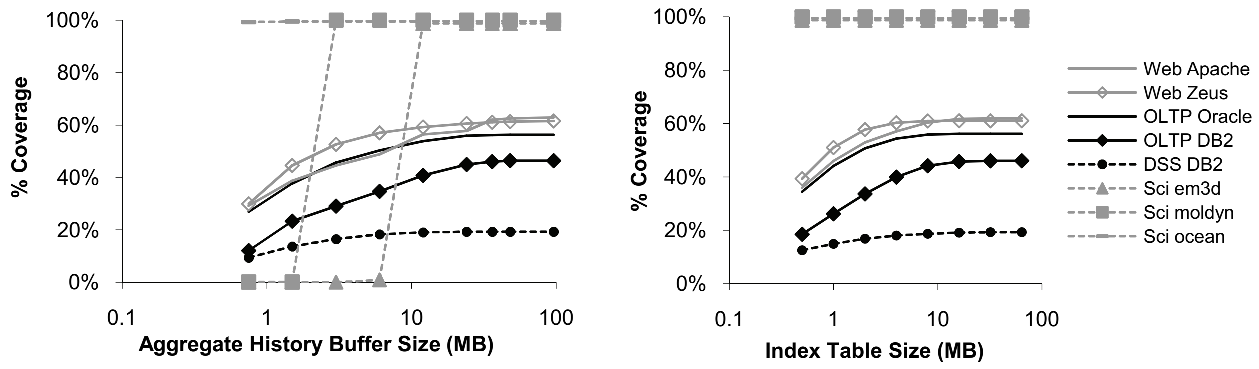


FIGURE 5: Storage requirements. The left graph shows the storage requirements for the history buffer. The right graph shows the storage requirements for the index table.

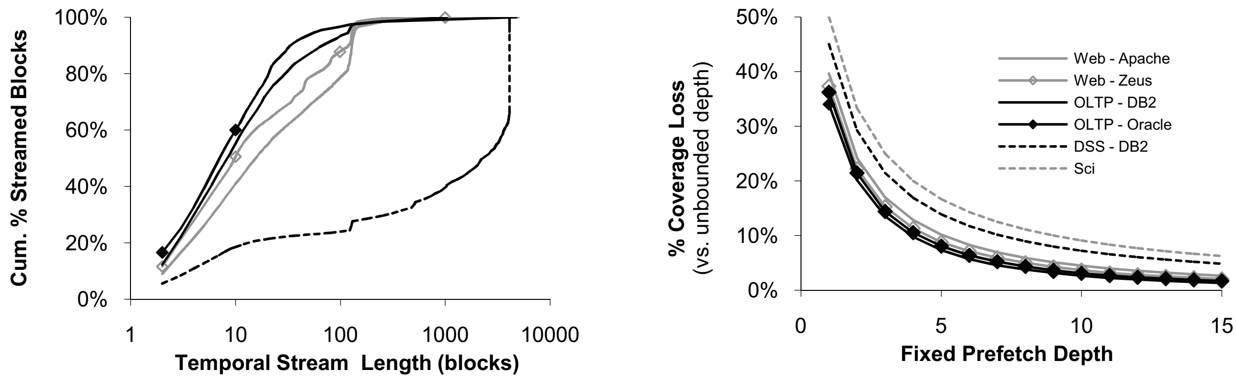


FIGURE 6: Amortizing lookup. The left graph shows the fraction of coverage arising from streams of a particular length (commercial workloads only). The right graph shows the prefetch coverage loss of restricted prefetch depth.

5.4. Achieving Latency Efficiency

An idealized prefetcher would have instant lookup, experiencing no latency between the cache miss that triggers a prediction and the predicted prefetch. However, a realistic implementation may lose prefetch opportunity on each lookup because time must be spent to locate and retrieve the miss address sequence [6,27].

We estimate lost prefetch opportunity by examining workload and lookup mechanism characteristics. A latency-efficient predictor will perform a lookup once per temporal stream recurrence, losing prediction opportunity only during retrieval of the initial addresses within the stream. Naturally, longer streams reduce opportunity loss, however temporal stream length is workload dependent and cannot be controlled. Figure 6 (left) shows a cumulative distribution of prefetches arising from streams of various lengths in our commercial workloads. In the scientific applications, the length of the single temporal stream depends on iteration length. For our configurations, this length is approximately 400,000 misses in em3d, 21,000 in ocean, and 81,000 in moldyn.

Lookup opportunity loss also depends on a workload's memory level parallelism (MLP [7]), the average number of off-chip loads issued while at least one such load is outstanding. As discussed in prior work [6], a predictor that retrieves meta-data from main memory loses coverage for each followed temporal stream, proportional to the number of round-trip memory accesses needed for a single lookup, multiplied by the workload's MLP (shown in Table 2). Like stream length, MLP is an inherent property of a workload, and is typically low in pointer-chasing applications such as the studied commercial workloads [1,27].

Table 2: Memory-level parallelism of off-chip reads (without STMS).

Benchmark		MLP	Benchmark		MLP
Web	Apache	1.5	DSS	DB2	1.6
	Zeus	1.5	Sci	em3d	1.7
OLTP	DB2	1.3		moldyn	1.0
	Oracle	1.3		ocean	1.2

As discussed in Section 3, when temporal streams are stored in a single set-associative correlation table (as in EBCP [6] and ULMT [23]), lookups require only a single memory access before prefetching can proceed. However, the set-associative table design limits maximum prefetch sequence length (referred to as the prefetch *depth* [21]) based on the size of a table entry. Because storage cost, complexity, and update-band-

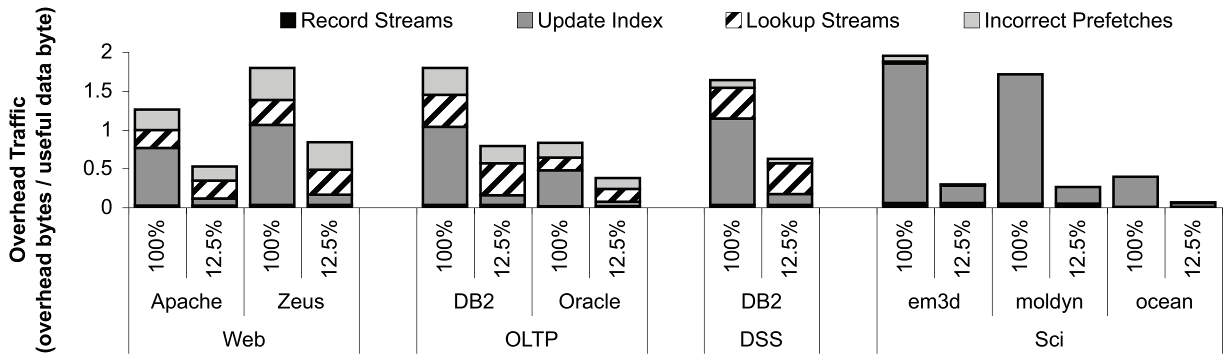


FIGURE 7: Overhead traffic. The left bar in each pair shows memory traffic overheads of off-chip index lookup without probabilistic update (sampling probability = 100%). The right bar shows overheads with a 12.5% probability.

width requirements grow with prefetch depth, it must remain small (three to six addresses [6,21,23]). As Figure 6 (left) shows, a small fixed depth falls far short of temporal stream lengths inherent in commercial workload. Restricted prefetch depth fragments long streams, forcing multiple lookups and sacrificing opportunity with each lookup. Figure 6 (right) shows the lost opportunity as a function of prefetch depth.

STMS instead employs a split-table meta-data organization, with separate history buffer and index table, allowing it to follow temporal streams for tens or hundreds of misses. However, the split-table approach implies a minimum of two round-trip memory accesses per lookup (one to each table). The expected coverage loss STMS incurs due to the additional memory round trip (equal to MLP reported in Table 2) is less than the fragmentation losses incurred by single-table designs.

We propose an efficient organization for index table entries by packing a hash bucket into a single memory block (64 bytes). Our organization limits the maximum number of index entries per bucket (to 12 in our design), but ensures that the index table can be searched with a single access. We performed an extensive analysis of alternative organizations for the index table (e.g., open address hashing, larger hash bucket chains, tree structures), and found that these organizations were either less storage efficient or sacrificed additional coverage due to increased lookup latency.

5.5. Achieving Bandwidth Efficiency

Placing predictor meta-data off chip introduces several sources of pin-bandwidth overheads: recording miss addresses in the history buffer, index table updates, lookup operations (index table and history buffer accesses), and prefetch of erroneously-predicted addresses. Figure 7 shows the relative importance of each overhead source normalized to the base system off-chip traffic (which includes demand-triggered cache block fetches and writebacks).

The largest bandwidth overhead in an un-optimized system arises due to index table maintenance (see the bars labelled 100%). Each index table update calls for a read and write operations on the corresponding index table entry, resulting in bandwidth utilization that exceeds the base system traffic for many workloads. The second largest contributor is index table lookups, performed on each demand-read-miss from the L2, searching for a temporal stream to follow. Although lookup traffic is substantial, it decreases as the system makes more correct predictions and eliminates demand misses. Accurate detection of the end-of-stream curtails the bandwidth consumed by erroneous prefetches. Finally, the bandwidth utilized for recording off-chip sequences is negligible (not visible in the graphs), as a single densely-packed history buffer write is performed for every twelve off-chip read misses.

The high traffic overheads shown in Figure 7 demonstrate the need for probabilistic update. We compare an un-optimized system (100% sampling probability) to one with a probabilistic update sampling probability of 1/8th (12.5%). Probabilistic update drastically reduces the off-chip traffic required for index updates.

Probabilistic update reduces traffic of index-table maintenance at the cost of predictor coverage. Figure 8 shows the traffic overhead of streaming (left) and predictor coverage (right) as function of the sampling probability. Index update traffic is proportional to sampling probability. Hence, Figure 7 shows that reducing sampling probability from 100% to 12.5% reduces total traffic. Below 12.5%, other sources of overhead (particularly lookup traffic) dominate.

Whereas traffic overhead decreases rapidly, predictor coverage decreases logarithmically with sampling probability. Omitting index table updates has only a small effect on coverage because temporal streams either tend to be long (in which case, a later address in the stream can be used to locate it) or to recur frequently (in which case an index entry from a

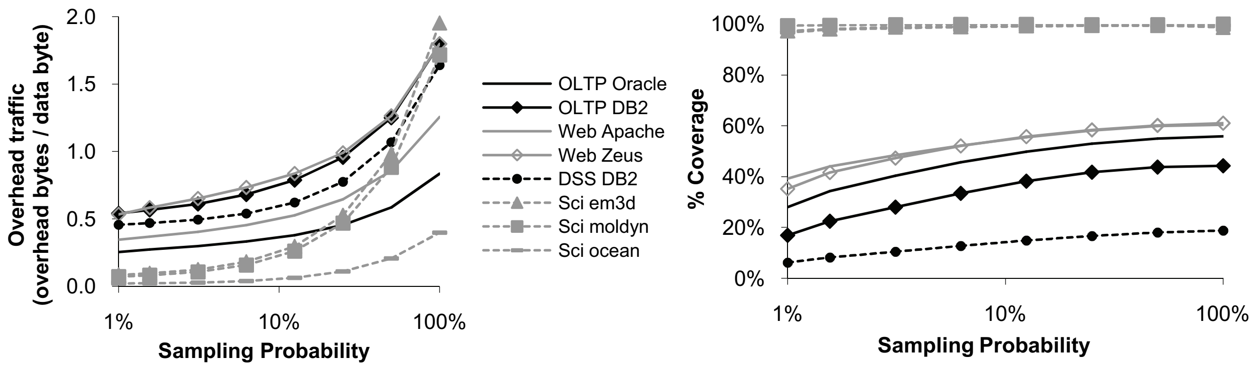


FIGURE 8: Probabilistic update sampling sensitivity. The left graph shows the traffic overhead impact of probabilistic update, while the right graph shows its coverage impact.

prior occurrence likely exists). Hence, probabilistic update is highly effective at reducing off-chip traffic for index table management.

5.6. Performance Impact of Practical Streaming

We conclude our evaluation by comparing the performance impact of STMS to idealized TMS. We employ a 12.5% update sampling probability, as it offers the best balance between bandwidth-efficiency and coverage for our workloads. Our mechanisms allow STMS to obtain on average 90% of idealized TMS coverage. Figure 9 (left) compares coverage of STMS and idealized TMS. For STMS, we subdivide coverage into fully-covered misses (off-chip latency is fully-hidden) and partially-covered misses (a core requested the block before the prefetch completed).

Because STMS maintains high coverage, it achieves 90% of the performance improvement possible with idealized lookup. Figure 9 (right) compares the performance improvement of STMS and idealized TMS. We conclude that our proposed mechanisms—hash-based lookup and probabilistic update—enable a bandwidth-, latency-, and storage-efficient design, meeting our stated goals of (1) matching the coverage and performance of an idealized prefetcher while storing meta-data in off-chip memory, and doing so (2) without penalizing the performance of workloads that derive no benefit from temporal streaming.

6. Conclusions

Address-correlating prefetchers are known to improve performance of commercial server workloads. To be effective, these prefetchers must maintain meta-data that cannot fit on chip. In this work, we identified the key requirements for implementing address-correlating prefetchers with off-chip meta-data storage. To satisfy these requirements, we proposed two techniques: hash-based lookup and probabilistic sampling of meta-data updates, and applied these techniques to an address-correlating prefetcher design with split history and index tables. Hash-based lookup achieves low off-chip meta-data lookup latency. Probabilistic sampling achieves bandwidth-efficient meta-data updates. Split index and history tables amortize each meta-data lookup over multiple prefetches. Our evaluation demonstrates that these techniques yield a bandwidth-, latency-, and storage- efficient temporal memory streaming design that keeps predictor meta-data in main memory while achieving 90% of the performance potential of idealized on-chip meta-data storage.

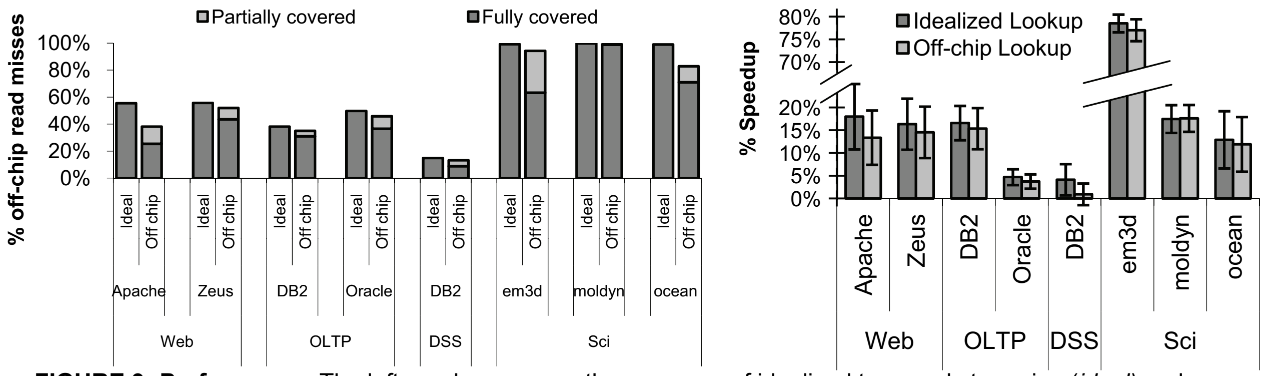


FIGURE 9: Performance. The left graph compares the coverage of idealized temporal streaming (*ideal)* and a practical STMS design with off-chip lookup *(off-chip)*. The right graph compares the performance of the two designs, relative to a design with stride prefetching only.

Acknowledgements

The authors would like to thank Brian Gold and the anonymous reviewers for their feedback. This work was supported by grants from Intel, two Sloan research fellowships, an NSERC Discovery Grant, an IBM faculty partnership award, and NSF grant CCR-0509356.

References

[1] Luiz Andre Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *Proc. of the 25th International Symposium on Computer Architecture*, 1998.

[2] Ioana Burcea, Stephen Somogyi, Andreas Moshovos, and Babak Falsafi. Predictor virtualization. In *Proc. of the 13th International Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

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

[4] Trishul M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proc. of the SIGPLAN '01 Conf. on Programming Language Design and Implementation*, 2001.

[5] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proc. of the ACM SIGPLAN 2002 Conf. on Programming language design and implementation*, 2002.

[6] Yuan Chou. Low-cost epoch-based correlation prefetching for commercial applications. In *Proc. of the 40th IEEE/ACM International Symposium on Microarchitecture*, 2007.

[7] Yuan Chou, Brian Fahs, and Santosh Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proc. of the 31st International Symposium on Computer Architecture*, 2004.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[9] Michael Ferdman and Babak Falsafi. Last-touch correlated data streaming. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2007.

[10] Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal instruction fetch streaming. In *Proc. of the 41st IEEE/ACM International Symposium on Microarchitecture*, 2008.

[11] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastasia Ailamaki, and Babak Falsafi. Database servers on Chip Multiprocessors: Limitations and Opportunities. In *Third Biennial Conf. on Innovative Data Systems Research,* 2007.

[12] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Proc. of the 29th International Symposium on Computer Architecture*, 2002.

[13] Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. Tcp: Tag correlating prefetchers. In *Proc. of the 9th IEEE Symposium on High-Performance Computer Architecture*, 2003.

[14] Jaehyuk Huh, Doug Burger, and Stephen W. Keckler. Exploring the design space of future cmps. In *Proc. of the 2001 International Conf. on Parallel Architectures and Compilation Techniques*, 2001.

[15] Ryan Johnson, Nikos Hardavellas, Ippokratis Pandis, Naju Mancheril, Stavros Harizopoulos, Kivanc Sabirli, Anastassia Ailamaki, and Babak Falsafi. To share or not to share? In *33rd Very Large Data Bases Conference*, 2007.

[16] Doug Joseph and Dirk Grunwald. Prefetching using Markov Predictors. In *Proc. of the 24th International Symposium on Computer Architecture*, 1997.

[17] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of the 17th International Symposium on Computer Architecture*, 1990.

[18] An-Chow Lai and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proc. of the 28th International Symposium on Computer Architecture*, 2001.

[19] James E. Larus and M. Parkes. Using cohort scheduling to enhance server performance. In *Proc. of the USENIX Technical Conference*, 2002.

[20] Jack L. Lo, Luiz Andre Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proc. of the 25th International Symposium on Computer Architecture*, 1998.

[21] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. In *Proc. of the Tenth IEEE Symposium on High-Performance Computer Architecture*, 2004.

[22] Timothy Sherwood, Suleyman Sair, and Brad Calder. Predictor-directed stream buffers. In *Proc. of the 33rd IEEE/ACM International Symposium on Microarchitecture (MICRO 33)*, 2000.

[23] Yan Solihin, Jaejin Lee, and Josep Torrellas. Using a user-level memory thread for correlation prefetching. In *Proc. of the 29th International Symposium on Computer Architecture*, 2002.

[24] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *Proc. of the 33rd International Symposium on Computer Architecture*, 2006.

[25] Pedro Trancoso, Josep-L. Larriba-Pey, Zheng Zhang, and Josep Torellas. The memory performance of DSS commercial workloads in shared-memory multiprocessors. In *Proc. of the Third IEEE Symposium on High-Performance Computer Architecture*, 1997.

[26] Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal streams in commercial server applications. In *IEEE International Symposium on Workload Characterization*, 2008.

[27] Thomas F. Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Temporal streaming of shared memory. In *Proc. of the 32nd International Symposium on Computer Architecture*, 2005.

[28] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. SimFlex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006.

[29] Chengqiang Zhang and Sally A. McKee. Hardware-only stream prefetching and dynamic access ordering. In *Proc. of the 14th International Conf. on Supercomputing*, 2000.