Temporal Streaming of Shared Memory

Thomas F. Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim,
Anastassia Ailamaki and Babak Falsafi
Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
http://www.ece.cmu.edu/~puma2

Abstract

Coherent read misses in shared-memory multiprocessors account for a substantial fraction of execution time in many important scientific and commercial workloads. We propose Temporal Streaming*, to eliminate coherent read misses by streaming data to a processor in advance of the corresponding memory accesses. Temporal streaming dynamically identifies address sequences to be streamed by exploiting two common phenomena in shared-memory access patterns: (1) temporal address correlation—groups of shared addresses tend to be accessed together and in the same order, and (2) temporal stream locality—recently-accessed address streams are likely to recur.*

We present a practical design for temporal streaming. We evaluate our design using a combination of trace-driven and cycle-accurate full-system simulation of a cache-coherent distributed shared-memory system. We show that temporal streaming can eliminate 98% of coherent read misses in scientific applications, and between 43% and 60% in database and web server workloads. Our design yields speedups of 1.07 to 3.29 in scientific applications, and 1.06 to 1.21 in commercial workloads.

1. Introduction

Technological advancements in semiconductor fabrication along with microarchitectural and circuit innovation have led to phenomenal increases in processor speed over the past decades. During the same period, memory (and interconnect) speed has not kept pace with the rapid acceleration of processors, resulting in an ever-growing processor/memory performance gap. This gap is exacerbated in scalable shared-memory multiprocessors, where a cache-coherent access often requires traversing multiple cache hierarchies and incurs several network round-trip delays.

There are a myriad of proposals for reducing or hiding the coherence miss latency. Techniques to relax memory order [1,10] have been shown to hide virtually all of the coherent write miss latency. In contrast, prior proposals to mitigate the impact of coherent read misses have fallen short of effectively hiding the read miss latency. Techniques targeting coherence optimization (e.g., [13,15,18,19,21,22,29]) can only hide part of the read latency.

Prefetching [26] or forwarding [17] techniques seek to hide the entire cache (read) miss latency. These techniques have been shown to be effective for workloads with regular (e.g., strided) memory access patterns. Unfortunately, memory access patterns in many important commercial [3] and scientific [23] workloads are often highly irregular and not amenable to simple predictive and prefetching schemes. As such, coherent read misses remain a key performance-limiting bottleneck in these workloads [2,23].

Recent research [3] advocates fetching data in the form of *streams*—i.e., sequences of cache blocks that occur together—rather than individual blocks. Streaming not only enables accurate data fetching through correlating a recurring sequence of addresses, but also significantly enhances fetch lookahead commensurately to the sequence length. These results indicate that streaming can hide the read miss latency even in workloads with long chains of dependent cache misses (e.g., online transaction processing, OLTP). Unfortunately, the prior proposal [3] for generalized streaming requires a sophisticated hierarchical compression algorithm to analyze whole program memory address traces, which may only be practical when run offline and is prohibitively complex to implement in hardware.

In this paper, we propose *Temporal Streaming*, a technique to hide coherent read miss latency in shared-memory multiprocessors. Temporal streaming is based on the observation that *recent* sequences of shared data accesses often recur in the same precise order. Temporal streaming uses the miss history from recent sharers to extract *temporal streams* and move data to a subsequent sharer in advance of data requests, at a transfer rate that matches the consumption rate. Unlike prior proposals for streaming [3] that require persistent stream behavior throughout program execution to enable offline analysis, temporal streaming can exploit streams with temporal (but not necessarily persistent) behavior by identifying streams on the fly directly in hardware.

Through a combination of memory trace analysis and cycle-accurate full-system simulation [12] of a cache-coherent distributed shared-memory system (DSM) running scientific, OLTP (TPC-C on DB2 and Oracle) and web server (SPECweb on Apache and Zeus) workloads, we contribute the following.

- **Temporal address correlation & stream locality:** We investigate the inherent properties of our workload suite, and show that (1) shared addresses are accessed in repetitive sequences, and (2) recently followed sequences are likely to recur system-wide. More than 93% of coherent read misses in scientific applications and 40% to 65% in commercial workloads follow precisely a recent sequence.
- **Temporal streaming engine:** We propose a design for temporal streaming with practical hardware mechanisms to record and follow streams. Our design yields speedups of 1.07 to

Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)
1063-6897/05 $20.00 © 2005 IEEE

3.29 in scientific applications, 1.11 to 1.21 in online transaction processing workloads, and 1.06 in web server workloads.

The rest of this paper is organized as follows. We introduce temporal streaming in Section 2, and show how to exploit it to hide coherent read latency. Section 3 presents the Temporal Streaming Engine, our hardware realization of temporal streaming. We describe our evaluation methodology in Section 4, and quantitatively evaluate the temporal streaming phenomena and our hardware design in Section 5. We present related work in Section 6 and conclude in Section 7.

2. Temporal Streaming

In this paper, we propose *Temporal Streaming*, a technique to identify and communicate streams of shared data dynamically in DSM multiprocessors. The objective of temporal streaming is to hide communication latency by streaming data to consuming nodes in advance of processor requests for the data. Unlike conventional DSM systems, where shared data are communicated throughout the system individually, temporal streaming exploits the correlation between recurring access sequences to communicate data in streams. While temporal streaming applies to generalized address streams, in this paper we focus on coherent read misses because they present a performance-limiting bottleneck in many workloads and their detrimental effect is aggravated as cache sizes increase [2].

Temporal streaming exploits two properties common in shared memory access patterns: (1) *temporal address correlation,* where groups of shared addresses tend to be accessed together and in the same order, and (2) *temporal stream locality,* where recently-accessed address streams are likely to recur. In this paper, we use the term *temporal correlation* to encompass both properties.

Temporal address correlation arises primarily from shared data access patterns. When data structures are stable (although their contents may be changing), access patterns repeat, and coherence miss sequences exhibit temporal address correlation. Thus, temporal address correlation can be found in accesses to generalized data structures such as linked-data structures (e.g., lists and trees) and arrays. In contrast, spatial or stride locality, commonly exploited by conventional prefetching techniques, rely on a data structures' layout in memory which is only characteristic of array-based data structures.

Temporal stream locality arises because recently accessed data structures are likely to be accessed again; therefore address sequences that were recently followed are likely to recur. In applications with migratory sharing patterns—most commercial and some scientific applications—this type of locality occurs system-wide as the migratory data are accessed in the same way by all nodes.

Figure 1 illustrates an example of temporal streaming in a DSM. Node i incurs coherent read misses and records the sequence of misses $\{A,B,C,D,E\}$, which we refer to as its coherence miss *order*. We define a *stream*[1] to be a sub-sequence of addresses in a node's order. Node j later misses on address *B*, and requests the data from the directory node. The directory node responds to this request through the baseline coherence mechanism, and additionally requests a stream (following *B*) from the most recent consumer, Node i. We call the initial miss address, *B*, a *stream head*. Node i looks up address *B* in its order and assumes that requests to the subsequent addresses $\{C,D,E\}$ are likely to follow. Thus, it forwards the stream $\{C,D,E\}$ to Node j. Upon receipt of the stream, Node j retrieves the data for each block. Subsequent accesses to these addresses hit locally and avoid long-latency coherence misses.

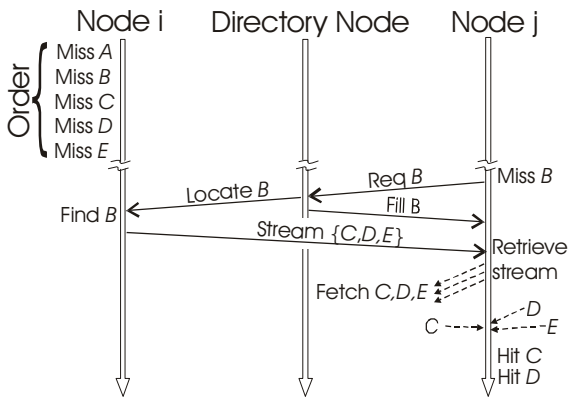

FIGURE 1: Temporal streaming.

Temporal streaming requires three capabilities: (1) recording the order of a node's coherent read misses, (2) locating a stream in a node's order and (3) streaming data to the requesting processor at a rate that matches its consumption rate.

3. The Temporal Streaming Engine

We propose the *Temporal Streaming Engine (TSE)*, a hardware realization of temporal streaming, to stream cache blocks to consuming nodes in advance of processor requests. TSE exploits temporal correlation in coherent read misses to reduce or eliminate processor stalls that result from long-latency coherent reads.

Figure 2 shows a diagram of a DSM node enhanced with TSE. The components marked with a grayscale gradient are added or modified by TSE to furnish the baseline node with the three capabilities required for temporal streaming.

To record a node's order, each node stores the sequence of coherent read miss addresses in a circular buffer, called the *coherence miss order buffer* (CMOB). Because the order may grow too large to reside on chip, the CMOB is placed in main memory. To locate streams in a node's order, TSE maintains a CMOB pointer corresponding to the most recent miss for each cache block in the block's directory entry. The *stream engine* fetches and manages both stream addresses and data. The *streamed value buffer (SVB)* is a small fully-associative buffer that stores streamed cache blocks. On an L1 cache miss, the SVB is examined in parallel with the L2 cache to locate data.

The following subsections present the TSE components in detail. In Section 3.1, we present the process for recording the orders. Section 3.2 describes the process of looking up and

1. Throughout this paper, we use "stream" as a noun to refer to a sequence of addresses, and "stream" as a verb to refer to moving a sequence of either addresses or data.

Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)
1063-6897/05 $20.00 © 2005 **IEEE**

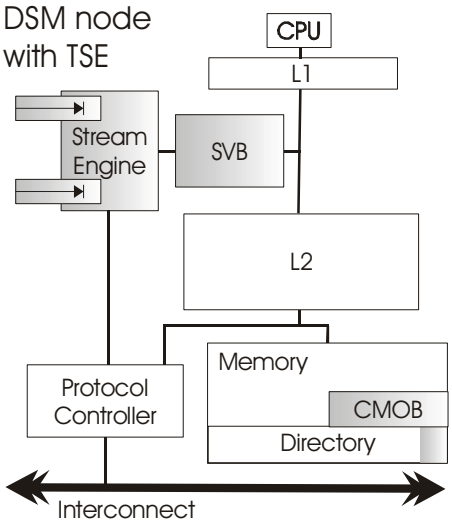

FIGURE 2: The TSE hardware.

forwarding streams upon a coherent read miss. Finally, we detail the operation of the stream engine in Section 3.3.

3.1 Recording the Order

To record the coherent read miss order, each node continuously appends the miss addresses, in program order, in its CMOB. Useful streamed blocks (i.e., resulting in accesses that hit in the SVB) are also recorded in the CMOB, as they replace coherent read misses that would have occurred without TSE. Much like prior proposals for recording on-chip generated metadata in memory (e.g., [9]), TSE packetizes the miss addresses in the form of cache blocks and ships them off chip to the CMOB. In Section 5.4, we present results indicating that because the CMOB entries are small relative to cache block sizes and CMOBs only record coherent read misses, this approach has a negligible impact on traffic through a node.

As misses are recorded, the recording node sends the corresponding CMOB pointer to the directory node for the block. The CMOB pointers stored in the directory allow TSE to find the correct CMOB locations efficiently given a stream head. While basic temporal streaming requires that only one CMOB pointer is recorded for each block, the TSE may choose to record pointers from the CMOBs of a few recent consumer nodes to enhance streaming accuracy (see Section 3.3).

Figure 3 illustrates the recording process. (1) The processor at the recording node issues an off-chip read for address *X*. (2) When the read request arrives at the protocol controller on the directory node, the directory identifies the miss as a coherent read miss. The directory node annotates the fill reply to indicate that the miss is a coherent read miss. (3) When the load instruction that incurred the coherence miss retires, the recording node appends the miss address to its CMOB. TSE appends addresses only upon retirement to ensure that the CMOB is properly ordered and does not contain addresses for wrong-path speculative reads. (4) Finally, the recording node informs the directory of the CMOB location of the newly appended address. This pointer update requires a separate message (as opposed to piggybacking on the original read request) because the recording node

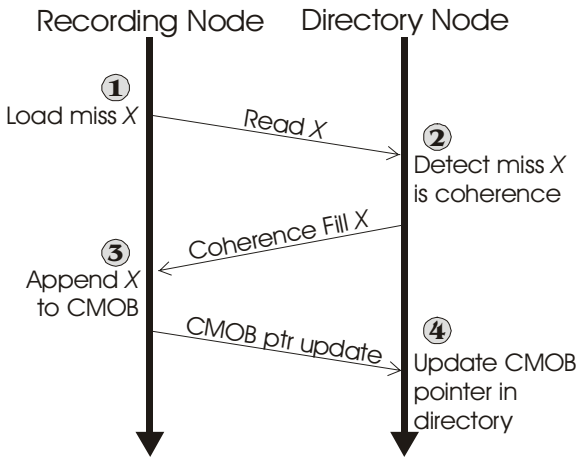

FIGURE 3: Recording the order.

does not know if or where each address will be appended until the load instruction retires.

The required CMOB capacity depends on the size of the application's active shared data working set, and may be quite large. Therefore, we place the CMOB in a private region of main memory which also allows us to tailor its capacity to fit an application's requirements. TSE can tolerate the resulting high access latency to CMOB in memory because write accesses (to append the packetized blocks of addresses to the order) occur in the background and are off the processor's critical path and read accesses (to locate or follow streams) are either amortized (on the initial miss) or overlapped through streaming lookahead. We report CMOB capacity requirements for our application suite in Section 5.4.

3.2 Finding and Forwarding Streams

TSE uses the information in each node's CMOB to identify candidate addresses for streaming. When a node incurs a coherent read miss, TSE locates one or more streams on CMOBs across the system, and forwards them to the stream engine at the requesting node.

Figure 4 illustrates the procedure to find and forward a stream. (1) A load to address *X* causes Node i to request the corresponding cache block from the directory node. (2) When the read request message arrives, the directory node detects that the miss is a coherent read miss, and retrieves the CMOB pointer for *X* from the directory. The CMOB pointer identifies that Node j recently appended *X* to its CMOB, and where on the CMOB *X* was appended. The directory node sends a stream request, including the corresponding CMOB pointer, to the streaming Node j indicated by the directory. (3) The protocol controller at Node j reads a stream of subsequent addresses from its CMOB starting at the entry following *X* (the contents of cache block *X* have already been sent to Node i by the baseline coherence mechanism), and forwards this stream to Node i. (4) When Node i receives the stream, the addresses are delivered to the stream engine.

There are several advantages to sending streams of addresses across nodes, rather than streaming data blocks directly. First, TSE does not require race-prone modifications to the baseline

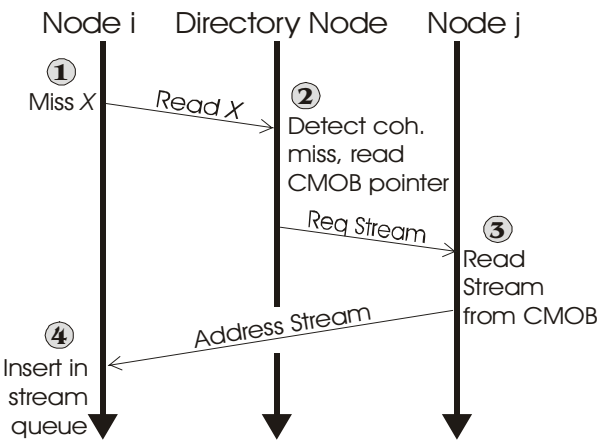

FIGURE 4: Locating and forwarding address streams.

cache coherence protocol. Second, streams of addresses do not incur any coherence overhead, whereas erroneously-streamed data blocks incur additional invalidation messages. Finally, sending streams of addresses allows the stream engine to identify temporal streams (i.e., consisting of temporally-correlated addresses) which are likely to result in hits.

The directory management mechanisms in DSM offer a natural solution for CMOB pointer storage and lookup. By extending each directory entry with one or more CMOB pointers, TSE enables random-access lookups within a CMOB; each CMOB pointer in the directory includes a node ID and an offset within the CMOB where the address is located, with the storage overhead of (number of CMOB pointers) × ($\log_2$(nodes) + $\log_2$(CMOB size)) bits. As such, CMOBs can be relatively large structures (e.g., millions of entries) residing in main memory. In contrast, prior proposals for prefetching based on recording address sequences in uniprocessors (e.g., [25]) resort to complex on-chip address hashing schemes and limited address history buffers.

3.3 The Stream Engine

The stream engine manages and follows the streams that arrive in response to coherent read misses. The stream engine plays a role similar to stream buffers in prior proposals (e.g., [28]). Unlike these proposals, however, TSE's stream engine locates, compares and follows more than one stream (i.e., from multiple recent consumers of the same addresses) for a given stream head simultaneously. Comparing multiple streams helps significantly to enhance streaming accuracy.

Figure 5 (left) depicts the anatomy of the stream engine. The stream engine contains groups of FIFO queues that store streams (with a common stream head), and comparators for checking if FIFO heads within a group match. We call each group of FIFOs a *stream queue*. Each stream queue also tracks the CMOB pointers for the streams it stores to facilitate requesting additional addresses when following a stream.

The stream engine continuously compares the FIFO heads in each group. In the common case, the FIFO heads will match, indicating high temporal correlation (i.e., the stream is likely to recur), in which case the stream engine proceeds to retrieve blocks. Upon retrieving the blocks, the corresponding address

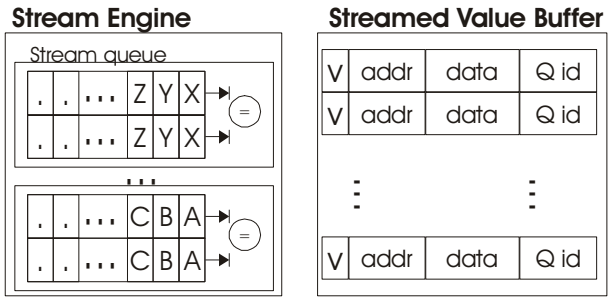

FIGURE 5: Stream engine and streamed value buffer.

entries in the FIFO queues are removed. When the FIFO heads disagree, indicating low temporal correlation, the stream engine stalls further data requests to avoid wasting bandwidth. However, the engine continues to monitor all off-chip memory requests to check for matches against the stalled FIFO heads. Upon a match, the processor is likely repeating the miss sequence recorded in the matching FIFO. Therefore, the stream engine discards the contents of all other (disagreeing) FIFOs and resumes fetching data using only the selected stream. We have investigated complex schemes that examine more than just the FIFO heads, but found they provide no advantage.

When a stream queue is half empty, the stream engine requests additional addresses from the source CMOB. The ability to follow long streams by periodically requesting additional addresses distinguishes TSE from prefetching approaches that only retrieve a constant number of blocks in response to a miss [25]. Without this ability, the system will incur one miss for each group of fetched blocks, even if the entire miss sequence exhibits temporal address correlation.

Figure 5 (right) depicts the anatomy of the SVB, a small fully-associative buffer for storing streamed data. Each SVB entry includes a valid bit, address, data, and the identity of the queue from which it was streamed. When a processor access hits in the SVB, the entry is moved to the L1 data cache, and the stream engine is notified to retrieve a subsequent cache block from the corresponding stream queue. The SVB entries contain only clean data, and are invalidated upon a write to the corresponding block by any (including the local) processor. SVB entries are replaced using an LRU policy.

The SVB serves a number of purposes. First, it serves as custom storage for stream data to avoid direct storage in, and inadvertent pollution of, the cache hierarchy when the addresses are not temporally correlated. Second, it allows for direct bookkeeping and management of streamed data and obviates the need for modifications to the baseline cache hierarchy. Finally, it serves as a window to mitigate small (e.g., a few cache blocks) deviations in the sequence of stream accesses (e.g., due to control flow irregularities in programs) by the processor. By presenting multiple blocks simultaneously from a stream in a fully-associative buffer, SVB allows the processor to skip or request cache blocks slightly out of stream order.

The SVB size dictates the maximum allowable *stream lookahead*—i.e., a constant number of blocks outstanding in the SVB—for each active stream. Ideally, the stream engine retrieves blocks such that they arrive immediately in advance of consumption by the processor. Therefore, effective streaming requires that

Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)
1063-6897/05 $20.00 © 2005 **IEEE**

the SVB holds enough blocks (i.e., allows for enough lookahead) to satisfy a burst of coherent read requests by the processor while subsequent blocks are being retrieved. We explore the issues involved in choosing the lookahead throughout Section 5. We show that in practice a small (e.g., tens of entries) SVB allows for enough lookahead to achieve near-optimal coverage while enabling quick lookup.

4. Methodology

We quantify temporal address correlation and stream locality, and evaluate our proposed hardware design across a range of scientific and commercial applications. We collect our results using a combination of trace-driven and cycle-accurate full-system simulation of a distributed shared-memory multiprocessor using *SIMFLEX* [12]. SIMFLEX is a simulation framework that uses modular component-based design and rigorous statistical sampling to enable the development of complex models and ensure representative measurement results with fast simulation turnaround. SIMFLEX builds on *Virtutech Simics* [20], a full system simulator that allows functional emulation of unmodified commercial applications and operating systems. SIMFLEX furnishes Simics with cycle-accurate models of an out-of-order processor core, cache hierarchy, microcoded coherence protocol engine, multi-banked distributed memory, and 2D torus interconnect. We implement a low-occupancy directory-based NACK-free cache-coherence protocol.

We simulate a 16-processor distributed shared-memory system with 3 GB of memory running *Solaris 8*. We implement an aggressive version of the total store order memory consistency model [1]. We perform speculative load and store prefetching as described by Gharachorloo et al. [8], and speculatively relax memory ordering constraints at memory barrier and atomic read-modify-write memory operations [10]. We list other relevant parameters of our system model in Table 1.

Table 2 describes the applications and parameters we use in this study. We target our study at commercial workloads, but include a representative group of scientific applications for comparison. We choose scientific applications which are (1) scalable to large data sets, and (2) maintain a high sensitivity to memory system performance when scaled. We include *em3d* [6], an electromagnetic force simulation, *moldyn* [23], a molecular dynamics simulation and *ocean* [30] current simulation.

We evaluate two database management systems, *IBM DB2 v7.2 EEE*, and *Oracle 10g Enterprise Database Server*, running the TPC-C v3.0 online transaction processing workload.[1] We use an optimized TPC-C toolkit provided by IBM for *DB2*. For *Oracle*, we developed and optimized our own toolkit. We tuned the number of client processes and other database parameters in our detailed timing model and chose the client and database configuration that maximized baseline system performance for each database management system. Client processes are configured with no think time, and database data and log files are striped across multiple disks to eliminate I/O bottlenecks.

We evaluate the performance of WWW servers running the SPECweb99 benchmark on *Apache HTTP Server v2.0* and *Zeus Web Server v4.3*. We simulate an 8-processor client system that sustains 16,000 simultaneous web connections to our 16-processor server via a simulated ethernet network. We run the client processors at a fixed IPC of 8.0 with a 4 GHz clock and provide sufficient bandwidth on the ethernet link to ensure that neither client performance nor available network bandwidth limit server performance. We collect memory traces and performance results on the server system only.

Our trace-based analyses use memory access traces collected from SIMFLEX with in-order execution, no memory system stalls, and a fixed IPC of 1.0. We analyze traces of at least ten iterations for scientific applications. We warm commercial applications for at least 5,000 transactions (or completed web requests) prior to starting traces, and then trace at least 500 transactions. We use the first iteration of each scientific and the first 100 million instructions (per processor) of each commercial application to warm trace-based simulations prior to measurement.

Our timing results for the scientific applications are derived from measurements of a single iteration started with warmed cache, branch predictor, and CMOB state. We use iteration runtime as our measure of performance.

Table 1. DSM system parameters.

Processing Nodes	UltraSPARC III ISA 4 GHz 8-stage pipeline; out-of-order execution 8-wide dispatch / retirement 256-entry ROB, LSQ and store buffer
L1 Caches	Split I/D, 64KB 2-way, 2-cycle load-to-use 4 ports, 32 MSHRs
L2 Cache	Unified, 8MB 8-way, 25-cycle hit latency 1 port, 32 MSHRs
Main Memory	60 ns access latency 64 banks per node 64-byte coherence unit
Protocol Controller	1 GHz microcoded controller 64 transaction contexts
Interconnect	4x4 2D torus 25 ns latency per hop 128 GB/s peak bisection bandwidth

Table 2. Applications and parameters.

Scientific Applications	
em3d	400K nodes, degree 2, span 5, 15% remote
moldyn	19652 molecules, boxsize 17, 2.56M max interactions
ocean	514x514 grid, 9600s relaxations, 20K res., err. tol. 1e-07
Commercial Applications	
Apache	16K connections, fastCGI, worker threading model
DB2	100 warehouses (10 GB), 64 clients, 450 MB buffer pool
Oracle	100 warehouses (10 GB), 16 clients, 1.4 GB SGA
Zeus	16K connections, fastCGI

1. "Solaris", "TPC", "Oracle", "Zeus", "DB2" and other trademarks are the property of their respective owners. None of the results presented in this paper should be construed to indicate the absolute or relative performance of any of the commercial systems used.

For the commercial applications, we use a systematic sampling approach developed in accordance with SMARTS [31]. SMARTS is a rigorous statistical sampling methodology, which prescribes a procedure for determining sample sizes, warm-up, and measurement periods based on an analysis of the variance of target metrics (e.g., IPC), to obtain the best statistical confidence in results with minimal simulation. We collect approximately 100 brief measurements of 400,000 cycles each. We launch measurements from checkpoints with warmed caches, branch predictors, and CMOBs, then run for 200,000 cycles to warm queue and interconnect state prior to collecting statistics.

We use the aggregate number of user instructions committed per cycle (i.e., user IPC summed over the 16 processors) as our performance metric. We exclude system commits from this metric because we cannot distinguish system commits that represent forward progress from those that do not (e.g., the idle loop). We have independently corroborated Hankins et al.'s [11] results that the number of user instructions per transaction in the TPC-C workload remains constant over a wide range of database configurations (whereas system commits per transaction do not). Thus, aggregate user IPC is proportional to database throughput.

5. Results

In this section, we investigate the opportunity for temporal streaming and the effectiveness of the Temporal Streaming Engine. Throughout our results, we report the effectiveness of TSE at eliminating *consumptions*, which we define as read requests that incur a coherence miss but are not a spin on a contended lock or barrier variable. We exclude coherent read misses that occur during spins because there is no performance advantage to predicting or streaming them.

5.1 Opportunity to Exploit Temporal Correlation

Temporal streaming relies on temporal address correlation and temporal stream locality to build and locate repetitive streams. We begin our evaluation by quantifying the fraction of consumptions that exhibit these phenomena.

When a stream of consumptions starting with address *X* precisely matches the sequence of consumptions at the most recent occurrence of *X,* there is perfect temporal address correlation and stream locality. In practice, because the stream lookahead keeps the streaming engine several blocks ahead of the processor's requests, TSE can also exploit imperfect correlation, where there is a small reordering of addresses between the current stream and the preceding order.

In this section, we investigate the fraction of consumptions that occur in temporally-correlated streams as a function of the degree of reordering between the processor's consumption order and that of the most recent sharer. We express reordering in terms of temporal correlation distance, which we define as the distance along the most recent sharer's order between consecutive processor consumptions. For example, if an order is {*A*,*B*,*C*,*D*} and a node has incurred miss *C*, then a subsequent miss to *D* yields a temporal correlation distance of +1 (i.e., perfect correlation), whereas a miss to *A* would correspond to a distance of -2.

Figure 6 shows the fraction of consumptions that exhibit temporal correlation, for temporal correlation distances (which

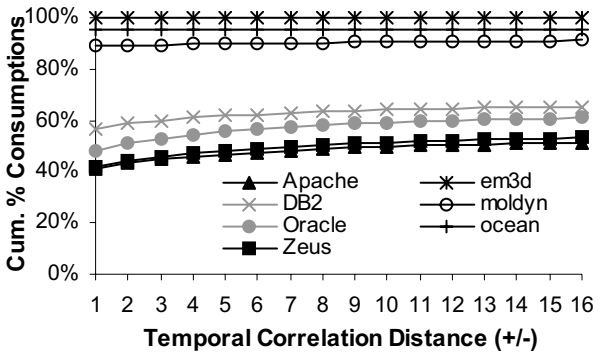

FIGURE 6: Opportunity to exploit temporal correlation.

corresponds roughly to stream lookahead) of up to ±16. All scientific applications in our suite exhibit near-perfect correlation, as they repeat the same data access pattern across all iterations. The commercial applications access data structures that change over time. Nevertheless, more than 40% of all consumptions in commercial applications are perfectly correlated, indicating that a significant portion of data structures and access patterns remain stable. Allowing for reordering of up to eight blocks increases the fraction to 49%–63% of consumptions. These results indicate that temporal streaming has the potential to eliminate nearly all coherent read misses in scientific applications, and almost half in commercial workloads.

5.2 Streaming Accuracy

Whereas accurate streaming improves performance by eliminating consumptions, inaccurate streaming may degrade performance, as a large proportion of erroneously streamed blocks can saturate available memory or interconnect bandwidth. TSE enhances stream accuracy by comparing several recent streams with the same stream head. When the streams match, TSE streams the corresponding blocks, whereas when they diverge, TSE conservatively awaits an additional consumption to select among the stream alternatives.

Figure 7 demonstrates the effectiveness of this approach for a stream lookahead of eight cache blocks and no TSE hardware restrictions (unlimited SVB storage, unlimited number of stream queues, near-infinite CMOB capacity). Coverage is the fraction of all consumptions that TSE correctly predicts and eliminates. Discards are cache blocks erroneously forwarded, also presented as a fraction of all consumptions. When TSE uses only a single stream, and therefore has no mechanism to gauge stream accuracy, commercial applications suffer very high discard rates. Although the commercial workload results in Figure 6 show that the majority of consumptions exhibit temporal address correlation, there remains a fraction that does not. Streaming on these non-correlated addresses produces many discards, but yields little coverage.

When TSE uses multiple streams, discards drop drastically to 40%–50% of total consumptions with minimal reduction in coverage. Further increasing the number of compared streams does not yield significant additional improvements, and does not warrant the increase in complexity. We configure TSE to compare two streams throughout the remainder of our results.

Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)
1063-6897/05 $20.00 © 2005 **IEEE**

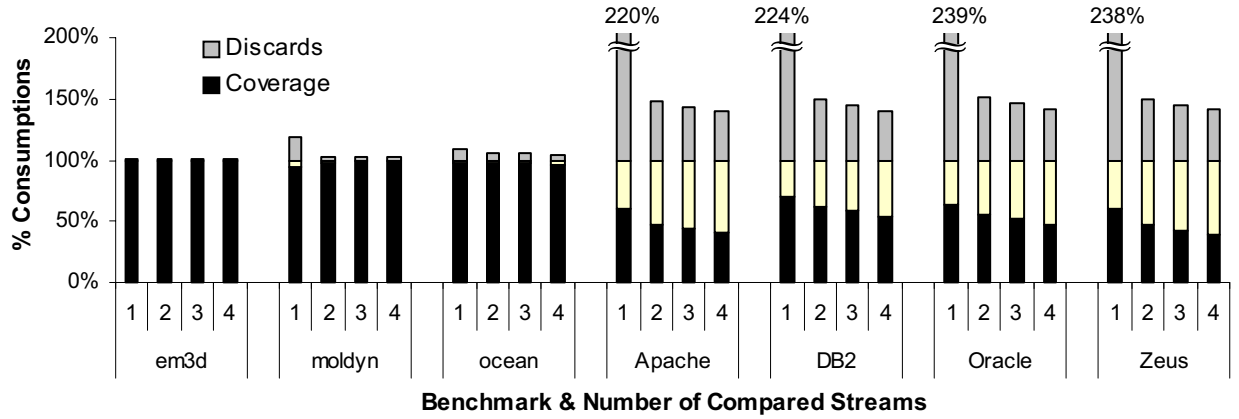

FIGURE 7: TSE sensitivity to the number of compared streams.

Effective streaming requires a stream lookahead sufficiently high to enable the SVB to satisfy consumption bursts by the processor. However, a stream lookahead higher than the required for effective streaming may erroneously stream too many blocks (i.e., discards) and degrade streaming accuracy. Figure 8 shows the effect of the stream lookahead on discards. For the scientific applications, which all exhibit near-perfect temporal correlation, even a high stream lookahead results in few discards. For the commercial applications, discards grow linearly with lookahead. In contrast, TSE coverage grows only slightly with increasing stream lookahead, as Figure 6 suggests. Thus, the ideal stream lookahead is the minimum sufficient to satisfy consumption bursts by the processor. We describe how to determine the value for the stream lookahead in Section 5.6.

5.3 Sensitivity to SVB Size and Stream Queues

Figure 6 suggests that an application typically follows only a single stream at a time. Were an application to interleave consumptions from two different streams, our temporal correlation measurement would classify them as uncorrelated accesses. Intuitively, we do not expect interleaved streams, as they imply the current consumer is interleaving the data access patterns of two previous consumers, or from two moments in time. We tested our intuition experimentally, and found no sensitivity to the number of stream queues.

Nevertheless, providing multiple stream queues in a TSE implementation compensates for the delays and event reorderings that occur in a real system. Most importantly, additional stream queues are necessary to avoid stream thrashing [28], where potentially useful streams are overwritten with useless streams from a non-correlated miss.

Our results show that applications typically follow one perfectly correlated stream at a time. Thus, the required SVB capacity in number of blocks is equal to the stream lookahead. For a stream lookahead of eight, the required SVB capacity is 512 bytes. Figure 9 confirms that there is little increase in coverage when moving from a 512-byte to an infinite SVB. The small increase in coverage results from the rare case of blocks that are accessed long after they are retrieved. We choose a 32-entry (2 KB) SVB because it offers near-optimal performance and is easy to implement a low-latency fully-associative buffer of this size.

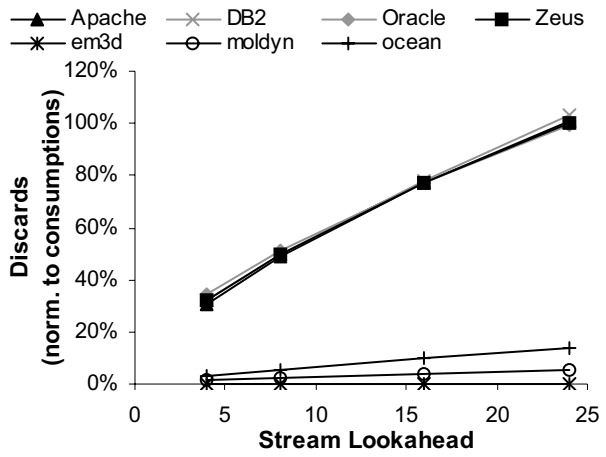

FIGURE 8: Effect of stream lookahead on discards.
Discards are normalized to true consumptions.

5.4 CMOB Storage and Bandwidth Requirements

Effective streaming requires the CMOB on each node to be large enough to record all the consumptions incurred by that node until a subsequent sharer begins following the sequence. In the worst case, for a system with 64-byte cache blocks and 6-byte physical address entries in the CMOB, the CMOB storage overhead is 11% of the aggregate shared data accessed by a node before the sequence repeats. The directory overhead for CMOB pointers grows logarithmically with CMOB size.

Figure 10 explores the CMOB storage requirements of our applications. The figure shows the fraction of maximum coverage attained as the CMOB ranges in size up to 6 MB. TSE achieves low coverage for the scientific applications until the CMOB capacity matches the shared data active working set for the problem sizes we simulate. For the commercial applications, TSE coverage improves smoothly with increasing CMOB capacity, reaching its peak at 1.5 MB. We also quantify the additional processor pin bandwidth due to recording the order off chip to be 4%-7% for the scientific and less than 1% for the commercial workloads.

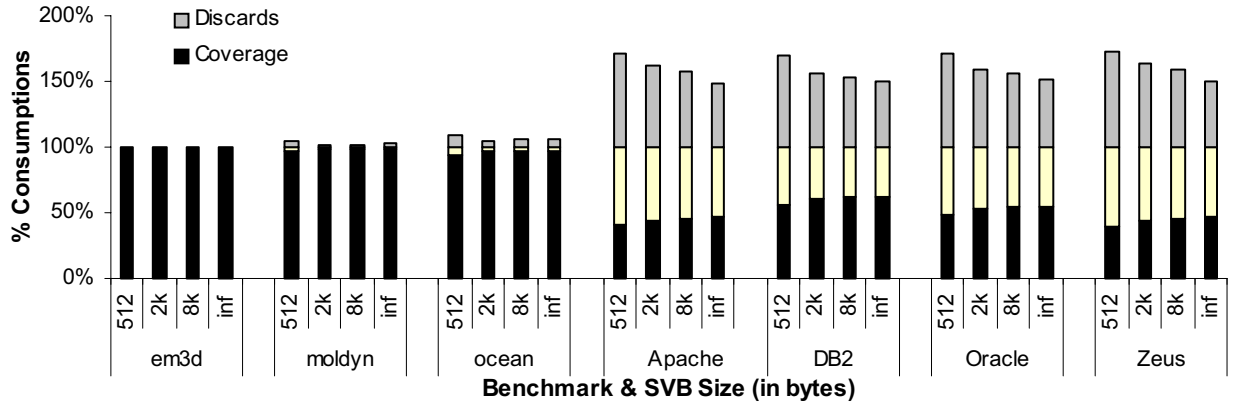

FIGURE 9: Sensitivity to SVB size. 'inf' indicates infinite storage.

Figure 11 shows the interconnect bisection bandwidth overhead associated with TSE. Each bar represents the bandwidth consumed by TSE overhead traffic (correctly streamed cache blocks replace processor coherent read misses in the baseline system one-for-one). The annotation above each bar indicates the ratio of overhead traffic to traffic in the base system. The dominant component of TSE's bandwidth overhead arises from streaming addresses between nodes.

The bandwidth overhead of TSE is a small fraction of the available bandwidth in current multiprocessor systems. The HP GS1280 multiprocessor system provides 49.6 GB/s interconnect bisection bandwidth in a 16-processor 2D-torus configuration [7]. Thus, the interconnect bandwidth overhead of TSE is less than 7% of available bandwidth in current technology, and less than 3% of bandwidth available in our DSM timing model.

5.5 Competitive Comparison

We compare TSE's effectiveness in eliminating consumptions against two previously-proposed prefetching techniques. We compare TSE against a stride-based stream buffer [28], as stride prefetchers are common in commercial microprocessors available today (e.g., AMD Opteron, Intel Xeon, Sun UltraSPARC III). We implement an adaptive stride predictor that detects strided access patterns if two consecutive consumption addresses are separated by the same stride, and prefetches eight blocks in advance of a processor request. Prefetched blocks are stored in a small cache identical to TSE's SVB. We also compare against the Global History Buffer (GHB) prefetcher proposed by Nesbit and Smith [25]. GHB was recently shown to outperform a wide variety of other prefetching mechanisms on SPEC applications [26]. In GHB, consumption misses are recorded in an on-chip circular buffer similar to the CMOB, and are located using an on-chip fully-associative index table. GHB supports several indexing options. We evaluate global distance-correlation (G/DC) as advocated by [26], and global address correlation (G/AC), as this is more similar to TSE. We use a 512-entry history buffer and fetch eight blocks per prefetch operation. We compare to TSE with a 1.5 MB CMOB and other parameters as previously described. Because TSE targets only consumptions, we configure the other prediction mechanisms to train and predict only for consumptions.

Figure 12 shows that TSE outperforms the other techniques by eliminating 43%-100% of consumptions. Because none of the applications exhibit significant strided access patterns, the stride prefetcher rarely prefetches, resulting in both low coverage and low discards. Address-correlating GHB (G/AC) outperforms distance correlation (G/DC) in terms of discards across commercial applications, but falls short of TSE coverage because its 512-entry consumption history is too small to capture repetitive consumption sequences.

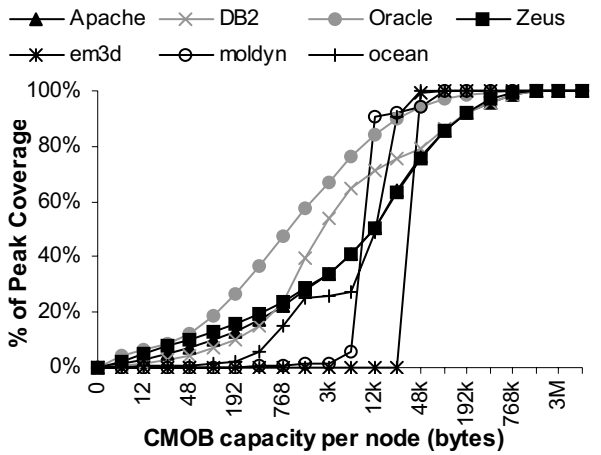

FIGURE 10: CMOB storage requirements.

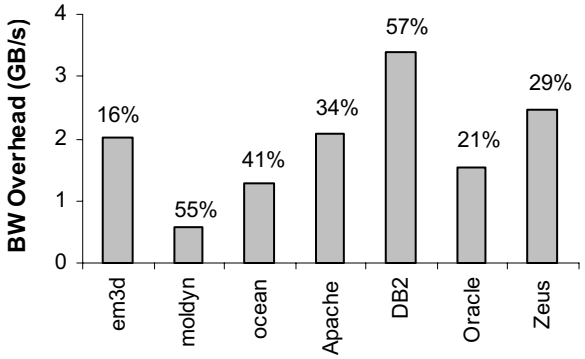

FIGURE 11: Interconnect bisection bandwidth overhead. The annotation above each bar indicates the ratio of overhead traffic to traffic in the base system.

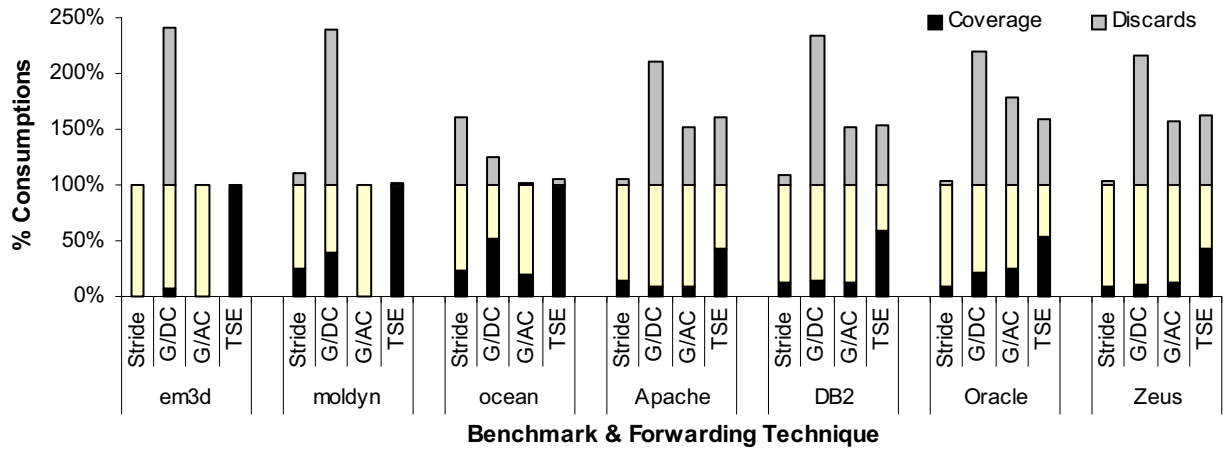

FIGURE 12: TSE compared to recent prefetchers. G/DC refers to distance-correlating Global History Buffer, G/AC refers to address-correlating Global History Buffer.

5.6 Streaming Timeliness

To eliminate consumptions effectively, streaming must both achieve high coverage—to stream the needed blocks—and be timely—so that blocks arrive in advance of processor requests. Timeliness depends on the stream lookahead, the streaming rate and the delay between initiating streaming and receiving the first data. TSE matches the consumption rate to the streaming rate simply by retrieving an additional block upon an SVB hit. Thus, in this section we focus on the effects of the streamed data delay and the stream lookahead.

Long temporally-correlated streams are insensitive to the delay of retrieving their first few blocks, as TSE can still eliminate most consumptions. Figure 13 shows the prevalence of streams of various lengths for our applications. The scientific applications are dominated by very long streams, hundreds to thousands of blocks each. Timely streaming for scientific applications requires configuring a sufficiently high stream lookahead. As Figure 8 shows, scientific applications exhibit low discard rates, allowing us to configure very high lookaheads without detrimental effects.

The commercial workloads obtain 30%-45% of their coverage from streams shorter than 8 blocks. Thus, the timely retrieval of the beginning of streams may impact significantly the overall performance. However, the data-dependent nature of the commercial workloads [27] and instruction window constraints may restrict the processor's ability to issue multiple outstanding consumptions. Whereas the processor may quickly stall, TSE can retrieve all blocks within a stream in parallel, thereby eliminating consumptions despite short stream lengths.

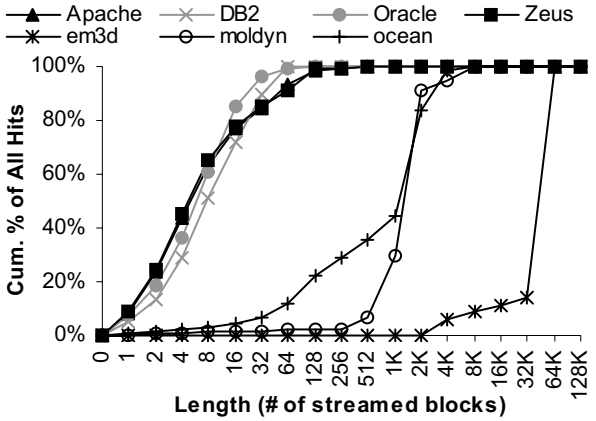

FIGURE 13: Stream length.

To verify our hypothesis, we measure the consumption memory level parallelism (MLP) [4]—the average number of coherent read misses outstanding when at least one is outstanding—in our baseline timing model, and report the results in Table 3. Our results show that, in general, the commercial applications issue consumptions serially. The latency to fill the consumption miss that triggers the stream lookup is approximately the same as the latency to retrieve streams and initiate streaming. Thus, streaming can begin at the time the processor requests the first block on the stream without sacrificing timeliness.

We determine the appropriate stream lookaheads for em3d and moldyn by first calculating the rate at which consumption misses would be issued in our base system if all coherent read latency was removed. We then divide the stream retrieval round-trip latency (i.e., 3-hop coherence miss latency) by the no-wait consumption rate. For ocean, this simple approach fails because all coherence activity occurs in bursts, as evidenced by its high consumption MLP in the baseline system. To improve cache locality, ocean blocks its computation, which, as a side effect, groups consumptions into bursts. We set the stream lookahead to a maximal reasonable value of 24 for ocean based on the number of available L2 MSHRs in our system model.

There is relatively little sensitivity to stream lookahead in commercial applications because of their low consumption MLP. We found that a lookahead of eight works well across these applications.

Table 3 shows the effect of streaming timeliness on TSE coverage using both trace analysis and cycle-accurate simulation. *Trace Cov.* indicates consumptions eliminated by TSE as reported by our trace analysis. *Full Cov.* indicates consumptions eliminated completely by TSE in the cycle-accurate simulation. *Partial Cov.* indicates consumptions whose latency was partially

Table 3. Streaming timeliness.

Benchmark	Trace Cov.	Cycle-accurate Simulation			
		MLP	Lookahead	Full Cov.	Partial Cov.
em3d	100%	2.0	18	94%	5%
moldyn	98%	1.6	16	83%	14%
ocean	98%	6.6	24	27%	57%
Apache	43%	1.3	8	26%	16%
DB2	60%	1.3	8	36%	11%
Oracle	53%	1.2	8	34%	9%
Zeus	43%	1.3	8	29%	14%

covered by TSE—the processor issued a request while a streamed value was still in flight.

TSE on the cycle-accurate simulator attains lower coverage relative to the trace analysis because streams may arrive late—after the processor has issued requests for the addresses in the stream. With the exception of ocean, most of the trace-measured coverage is timely (the consumptions are fully covered) in the cycle-accurate simulation of TSE, while the remaining consumptions are partially covered. We measured that partially covered consumptions hide on average 40% of the consumption latency in commercial workloads, and 60%-75% in scientific applications. In the case of ocean, partial coverage is particularly high. Even a stream lookahead of 24 blocks is insufficient to fully hide all coherent read misses, as the communication bursts in ocean are bandwidth bound.

5.7 Performance

We measure the performance impact of TSE using our cycle-accurate full-system timing model of a DSM multiprocessor. Figure 14 (left) illustrates the opportunity and effectiveness of TSE at eliminating stalls caused by coherent read misses. The *base* and *TSE* time breakdowns are normalized to represent the same amount of completed work. Figure 14 (right) reports the speedup achieved by TSE, with 95% confidence intervals for the sample-derived commercial application results.

TSE eliminates nearly all coherent read stalls in em3d and moldyn. TSE provides a drastic speedup of nearly 3.3 in communication-bound em3d. Despite high coverage, TSE eliminates only ~40% of coherent read stalls in ocean, as the majority of coherent read misses are only partially hidden. Although partially covered consumptions in ocean hide on average 60% of the consumption latency, much of the miss latency is overlapped in the baseline case as well because of the high MLP.

The commercial applications spend between 30%-35% of overall execution time on coherent read stalls. The TSE's performance impact is particularly large in DB2 because coherent read stalls are more prevalent in user (as opposed to OS) code than in the other commercial applications. User coherent read stalls have a disproportionately large impact on database throughput because misses in database code form long dependence chains [27], and are thus on the critical execution path. DB2 spends 43% of user execution time on coherent read stalls. TSE is particularly effective on these misses, eliminating 53% of user coherent read stalls.

As cache sizes continue to increase in future processors, coherence misses will become a larger fraction of long-latency off-chip accesses [2], and the performance impact of TSE and similar techniques will grow.

6. Related Work

Prior correlation-based prefetching approaches (e.g., Markov predictors [14] and Global History Buffer [25]) only considered locality and address correlation local to one node. In contrast, temporal streaming finds candidate streams by locating the most recent occurrence of a stream head across all nodes in the system.

Thread-based prefetching techniques [5] use idle contexts on a multithreaded processor to run helper threads that overlap misses with speculative execution. However, the spare resources the helper threads require (e.g., idle thread contexts, fetch and execution bandwidth) may not be available when the processor executes an application exhibiting high thread-level parallelism (e.g., OLTP). TSE, on the contrary, does not occupy processor resources.

Huh et al., [13] split a traditional cache coherence protocol into a fast protocol that addresses performance, and a backing protocol that ensures correctness. Unlike their scheme, which relies on detecting a tag-match to an invalidated cache line, TSE directly identifies coherent read misses using directory informa-

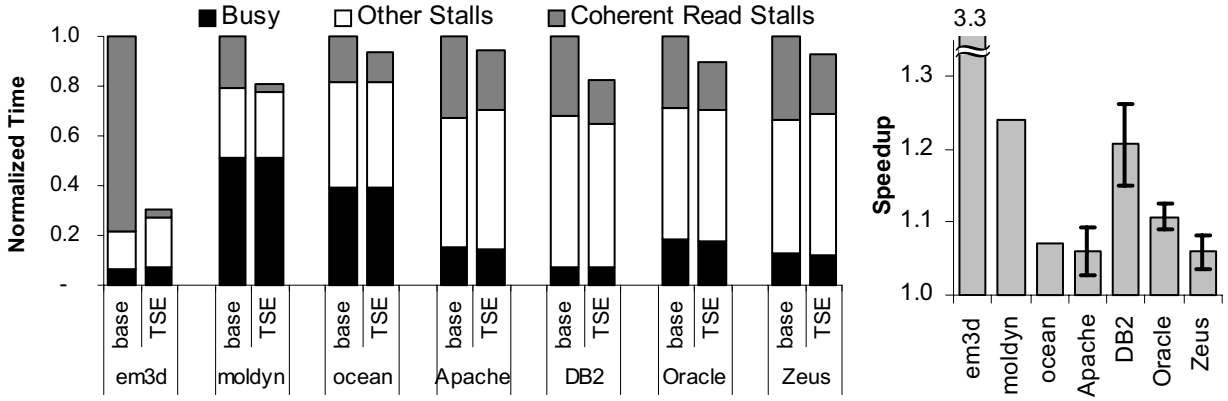

FIGURE 14: Performance improvement from TSE. The left figure shows an execution time breakdown. The right figure shows the speedup of TSE over the base system, with 95% confidence intervals for commercial application speedups.

tion, thus ensuring independence from the employed cache size. Moreover, coherent reads in [13] are still speculative for the entire length of a long-latency coherence miss and therefore stress the ROB, while our scheme allows coherent read references that hit in the SVB to retire immediately.

Keleher [16] describes the design and use of Tapeworm, a mechanism implemented as a software library that records updates to shared data within a critical section, and pushes those updates to the next acquirer of the lock. While tapeworm can be efficiently implemented in software distributed shared-memory systems, a hardware-only realization requires either the introduction of a race-prone speculative data push operation in the coherence protocol, or a split performance/correctness protocol as in [13]. Instead, our technique relies on streaming to communicate shared data to consumers, without changes to the coherence protocol or application modifications.

Recent research has also aimed at making processors more tolerant of long-latency misses. Mutlu et al. [24] allow MLP to break past ROB limits, by speculatively ignoring dependencies and continuing execution of the thread upon a miss to issue prefetches. However, their method is constrained by branch prediction accuracy and hides only part of the latency, as the runahead thread may not be able to execute far enough in advance during the time it takes to satisfy a miss. Techniques seeking to exceed the dataflow limit through value prediction or to increase MLP at the processor (e.g., SMT) or the chip level (e.g., CMP) are complementary to our work.

7. Conclusion

In this paper, we presented temporal streaming, a novel approach to eliminate coherent read misses in distributed shared-memory systems. Temporal streaming exploits two phenomena common in the shared memory access patterns of scientific and commercial multiprocessor workloads: temporal address correlation, that sequences of shared addresses are repetitively accessed together and in the same order; and temporal stream locality, that recently-accessed streams are likely to recur. We showed that temporal streaming has the potential to eliminate 98% of coherent read misses in scientific applications, and 43% to 60% in OLTP and web server applications. Through cycle-accurate full-system simulation of a cache-coherent distributed shared-memory multiprocessor, we demonstrated that our hardware realization of temporal streaming yields speedups of 1.07 to 3.29 in scientific applications, and 1.06 to 1.21 in commercial workloads, while incurring overhead of less than 7% of available bandwidth in current technology.

Acknowledgements

The authors would like to thank Sumanta Chatterjee and Karl Haas for their assistance with Oracle, and the members of the Carnegie Mellon Impetus group and the anonymous reviewers for their feedback on earlier drafts of this paper. This work was partially supported by grants and equipment from IBM and Intel corporations, the DARPA PAC/C contract F336150214004-AF, an NSF CAREER award, an IBM faculty partnership award, a Sloan research fellowship, and NSF grants CCR-0113660, IIS-0133686, and CCR-0205544.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.

[2] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, June 1998.

[3] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the SIGPLAN '02 Conference on Programming Language Design and Implementation (PLDI)*, June 2002.

[4] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[5] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 34)*, December 2001.

[6] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, Nov. 1993.

[7] Z. Cvetanovic. Performance analysis of the alpha 21364-based hp gs1280 multiprocessor. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 218–229, June 2003.

[8] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing (Vol. I Architecture)*, pages I–355–364, Aug. 1991.

[9] C. Gniady and B. Falsafi. Speculative sequential consistency with little custom storage. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2002.

[10] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 162–171, May 1999.

[11] R. Hankins, T. Diep, M. Annavaram, B. Hirano, H. Eri, H. Nueckel, and J. P. Shen. Scaling and characterizing database workloads: Bridging the gap between research and practice. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*, Dec. 2003.

[12] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Performance Evaluation Review*, 31(4):31–35, April 2004.

[13] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: making use of incoherence. In *Proceedings of the*

11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI), October 2004.

[14] D. Joseph and D. Grunwald. Prefetching using Markov Predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, June 1997.

[15] S. Kaxiras and C. Young. Coherence communication prediction in shared memory multiprocessors. In *Proceedings of the 6th IEEE Symposium on High-Performance Computer Architecture*, January 2000.

[16] P. Keleher. Tapeworm: High-level abstractions of shared accesses. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[17] D. A. Koufaty, X. Chen, D. K. Poulsena, and J. Torrellas. Data forwarding in scalable shared-memory multiprocessors. In *Proceedings of the 1995 International Conference on Supercomputing*, July 1995.

[18] A.-C. Lai and B. Falsafi. Memory sharing predictor: The key to a speculative coherent DSM. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

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

[20] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.

[21] M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[22] S. S. Mukherjee and M. D. Hill. Using prediction to accelerate coherence protocols. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[23] S. S. Mukherjee, S. D. Sharma, M. D. Hill, J. R. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed-memory machines. In *5th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 68–79, July 1995.

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

[25] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the 10th IEEE Symposium on High-Performance Computer Architecture*, Feb. 2004.

[26] D. G. Perez, G. Mouchard, and O. Temam. Microlib: a case for the quantitative comparison of micro-architecture mechanisms. In *Proceedings of the 3rd Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD04)*, June 2004.

[27] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 307–318, Oct. 1998.

[28] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 33)*, pages 42–53, December 2000.

[29] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Memory coherence activity prediction in commercial workloads. In *3rd Workshop on Memory Performance Issues*, June 2004.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, July 1995.

[31] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.