Store-Ordered Streaming of Shared Memory

Thomas F. Wenisch, Stephen Somogyi, Nikolaos Hardavellas,
Jangwoo Kim, Chris Gniady[†], Anastassia Ailamaki and Babak Falsafi

Computer Architecture Laboratory (CALCM)
Carnegie Mellon University

[†] Computer Science Dept.
University of Arizona

Abstract

Coherence misses in shared-memory multiprocessors account for a substantial fraction of execution time in many important scientific and commercial workloads. Memory streaming provides a promising solution to the coherence miss bottleneck because it improves memory level parallelism and lookahead while using on-chip resources efficiently.

We observe that the order in which shared data are consumed by one processor is correlated to the order in which they were produced by another. We investigate this phenomenon and demonstrate that it can be exploited to send Store-ORDered Streams (SORDS) of shared data from producers to consumers, thereby eliminating coherent read misses. Using a trace-driven analysis of all user and OS memory references in a cache-coherent distributed shared-memory multiprocessor, we show that SORDS-based memory streaming can eliminate between 36% and 100% of all coherent read misses in scientific workloads and between 23% and 48% in online transaction processing workloads.

1. Introduction

Long-latency cache-coherent accesses in scalable shared-memory multiprocessors pose a performance-limiting bottleneck in important commercial [3,18,31] and scientific [7,28,39] workloads. Advances in semiconductor fabrication technology and innovations in chip design promise to continue increasing both the number of transistors per die and transistor switching speeds. These trends suggest that processing speed and on-chip storage capacity will continue to grow. While processor speeds increase rapidly, communication latency between chips improves more slowly, magnifying the performance penalty of coherence misses. Furthermore, the trend towards larger on-chip cache hierarchies further exacerbates this shared memory wall, as larger caches increase the fraction of off-chip memory stalls due to sharing [3].

To alleviate the shared memory bottleneck, future architectures must hide the latency of coherence-induced read misses. Although out-of-order execution can effectively overlap on-chip accesses, it cannot hide long-latency coherent read misses because of limited instruction window size. To eliminate coherence miss latency completely, coherence transfers must be initiated well ahead of demand misses by the processor. Furthermore, approaches that transfer only a single block at a time will not match the processor's consumption rate. Instead, techniques targeting long-latency misses must increase the memory level parallelism (MLP) [5] as well as the lookahead of off-chip coherence transfers. Memory streaming approaches [15,32,38], which throttle the data transfer rate to match the consumption rate, provide a promising solution to the shared memory bottleneck because they improve MLP and lookahead while using on-chip resources efficiently.

The primary challenge of memory streaming lies in identifying the sequence of addresses to stream. Although stride-based prediction allows for easy implementation [15,32], memory access patterns in many important commercial [4] and scientific [28] workloads are often highly irregular and not amenable to simple predictive schemes. Recent research has shown that memory access patterns, although arbitrarily complex, often repeat over the course of program execution [4], a phenomenon called *temporal address correlation* [38]. Temporal streaming exploits this phenomenon to locate streams for arbitrarily complex patterns of shared read accesses within a history of recent read misses [38]. However, a past miss sequence may be a poor indicator of future accesses when a data structure's layout is changing. Furthermore, in some applications, the distance between recurring miss sequences, and thus temporal streaming storage requirements, grow with data set size.

In this paper, we propose *Store-ORDered Streaming (SORDS)*, a new memory streaming technique that addresses changing data structures and is independent of the distance between recurring data structure traversals. SORDS exploits the phenomenon that, in scientific and OLTP workloads, shared values are consumed in approximately the same order that they were produced. We call this phenomenon *producer-consumer temporal address correlation*. SORDS takes advantage of existing prediction technology to identify when shared values are produced [21,34] and which nodes will consume those data [17,20,34]. SORDS employs new hardware mechanisms to record the order shared values are produced and stream shared data from producers to consumers just before they are accessed. Unlike temporal streams, store-ordered streams reflect changes to the data structure layout made by the recorded stores. Furthermore, store-ordered streams need only be buffered for the interval between the production and consumption of a shared value, which can be far shorter than the interval between recurring consumption sequences.

By analyzing memory access traces from full-system simulation [12] of cache-coherent distributed shared-memory multiprocessors running OLTP workloads with IBM DB2 and scientific applications, we demonstrate:

- **Producer-Consumer Temporal Address Correlation:** We show that the order in which shared values are consumed is sim-

Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)
1089-795X/05 $20.00 © 2005 IEEE

ilar to the order in which they were produced. Across the applications we study, 28%-72% of coherent read misses precisely follow production order. This fraction increases to 66%-98% when allowing for slight reorderings within a four block window.

- **Practical SORDS Design:** We propose a design for store-ordered streaming with practical hardware mechanisms. Our design can eliminate 36%-100% of coherent read misses in scientific applications and 23%-48% in OLTP workloads.

The rest of this paper is organized as follows. In Section 2, we introduce store-ordered streaming and justify our approach from an analysis of the properties of shared data access sequences. In Section 3, we present our design for a practical hardware implementation of SORDS. In Section 4, we evaluate our SORDS design through trace-based simulation. We discuss related work in Section 5. Finally, we conclude in Section 6.

2. Store-Ordered Streaming

In this paper we propose *Store-ORDered Streaming (SORDS)*, a design for throttled streaming of data from producers to consumers to hide memory read latency in a distributed shared-memory (DSM) multiprocessor. SORDS is based on the key observation that there is temporal correlation between data production and subsequent consumption in shared memory: the order in which shared values are consumed is similar to the order in which they were produced. By capturing the production order, SORDS enables throttling of the stream of shared data into small buffers residing at the consumers just-in-time for consumption, thereby converting coherent read misses into hits. We call the similarity of the production and consumption orders *producer-consumer temporal address correlation*, or P-C correlation.

2.1. SORDS Overview

A node in a DSM system must obtain exclusive access to a cache block before writing it. Subsequently, the node continues to access the block until another node in the system issues a read to it, which causes a downgrade (exclusive-to-shared transition) at the writer. The last store to a block prior to downgrade is called a *production*. The first read of this newly-produced value by any node is a *consumption* by that node. If a consumption requires a coherence request to obtain the data, it is a *consumption miss*. In a baseline DSM system, all consumptions incur consumption misses. The goal of SORDS is to eliminate those misses.

Designs that forward memory values from one DSM node to another ahead of CPU requests must include mechanisms to determine which values to forward, when, and to which nodes. Figure 1 illustrates an example of how such mechanisms function in a DSM equipped with SORDS. Existing predictor technology [34] allows each node to identify which stores constitute productions of a shared cache block, and write the block back to the directory node (1). SORDS records the sequence of addresses that arrive at the directory, in production order, in a large circular buffer called a stream queue (2). When a request for an address arrives at the directory, SORDS fills the request, locates the requested cache block in the stream queue, and forwards several subsequent blocks to the consumer (3). As the consumer hits on forwarded blocks, it signals the directory to forward additional blocks (4).

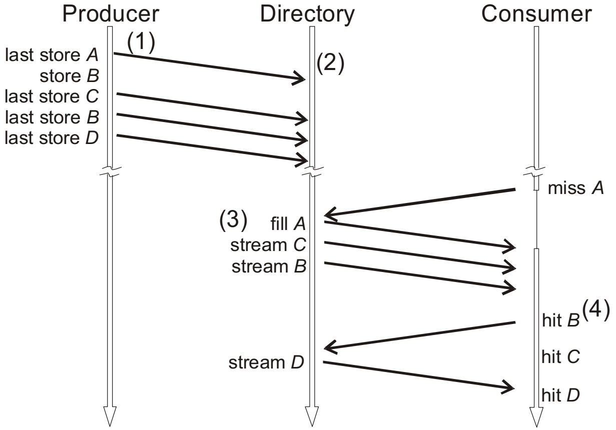


Figure 1. Eliminating coherent read misses in SORDS.

Successful forwarding depends upon a high degree of producer-consumer temporal address correlation. As long as the consumer continues to access blocks roughly in "store" (i.e., production) order, SORDS can eliminate the read misses. Intuitively such P-C correlation does exist (1) in general, for both data items within and across data structures [4] (e.g., parent and child nodes in a B-Tree), and (2) in shared memory in particular, because synchronization primitives guard against concurrent accesses to a shared data structure. In the remainder of this section, we show empirically that there is a high degree of P-C correlation in scientific and OLTP workloads, and justify the major design decisions of SORDS based on the nature of P-C correlation.

2.2. Methodology & Benchmarks

We demonstrate P-C correlation and evaluate our proposed SORDS design across a range of scientific and OLTP applications. We base our results on analysis of full-system memory traces of a distributed shared-memory multiprocessor using *FLEXUS* [12]. FLEXUS 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. FLEXUS builds on *Virtutech Simics* [24], a full system simulator that allows functional emulation of unmodified commercial applications and operating systems. The simulation models all memory accesses that occur in a real system, including all OS references. We configure Simics to run the scientific applications on a simulated 16-node distributed shared-memory multiprocessor running Solaris 8. The processing nodes implement the SPARC III ISA. We evaluate SORDS with OLTP workloads on Solaris 8 on SPARC and Red Hat Linux 7.3 on x86. We study DB2 on two platforms because OS code has a significant impact on database management system performance. We simulate a 16-node SPARC system and an 8-node x86 system (Simics uses a BIOS that supports only up to 8 CPUs for x86).

Table 1 describes the applications and inputs we use in this study. We select a representative group of pointer-intensive and array-based scientific applications that are (1) scalable to large

Scientific benchmarks	
barnes	*64K particles., 2.0 subdiv. tol., 10.0 fleaves*
em3d	*400K nodes, 15% remote, degree 2, span 5*
moldyn	*19652 molecules, max interactions 2560000*
ocean	*514x514 grid, 9600 sec*
OLTP benchmarks	
DB2 Solaris	*100 warehouses (10 GB), 96 clients, 450 MB buffer pool, 16 CPUs*
DB2 Linux	*100 warehouses (10 GB), 96 clients, 360 MB buffer pool, 8 CPUs*

TABLE 1. Applications and input parameters.

data sets, and (2) maintain a high sensitivity to memory system performance when scaled. These include barnes [39], a hierarchical N-body simulation; em3d [7], an electromagnetic force simulation; moldyn [28], a CHARMM-like molecular dynamics simulation; and ocean [39], current simulation.

We run DB2 7.2 with the TPC-C workload [23], an on-line transaction processing workload. We use a highly optimized toolkit, provided by IBM, to build the TPC-C database and run the benchmark. This toolkit provides a tuned implementation of the TPC-C specified queries and ensures that correct indices exist for optimal transaction execution. Prior to trace collection, we warm the database until the transaction completion rate reaches steady state. We analyze traces of at least 5,000 transactions.

2.3. Stream Properties

In this section, we explore the consumption sequence properties of multiprocessor applications, and identify the streaming mechanisms required to eliminate consumption misses. To gauge the full potential of streaming, we study it in the context of “oracle” knowledge of which stores are productions, and which nodes will subsequently consume these produced values. We present practical prediction techniques that approximate these oracles in Section 3.1.

Just-in-time streaming. Given perfect predictions, the simplest streaming approach is to forward each shared value immediately upon production to its precise set of consumers. Such eager forwarding guarantees that each value arrives at each consumer as early as possible, thereby minimizing the likelihood of incurring a miss penalty.

This aggressive approach often performs poorly because a producer often produces many values before consumers begin consuming them. For some applications, buffering the produced values at the consumer may require prohibitively large storage. Moreover, the storage requirement is highly dependent on the application’s sharing behavior. Figure 2 plots the fraction of consumption misses eliminated by aggressive streaming as a function of available (fully-associative) storage at the consumers. For em3d, moldyn, and DB2 Solaris, hundreds to thousands of cache blocks must be buffered to cover a significant fraction of consumption misses. This result shows that forwarding data into the conventional cache hierarchy would be counterproductive because: (1) forwarding into the L1 cache would displace many useful blocks, significantly reducing overall performance, and (2) forwarding into lower-level caches or the local DRAM memory [9] would incur a high (local) cache miss penalty, reducing the gains from forwarding. Similarly, custom storage would be too expensive both from an implementation cost and lookup time perspective. Finally, these results are conservative in that they assume perfect predictors. In practice, with real predictors, storage requirements may be even higher because of forwarding of unwanted data.

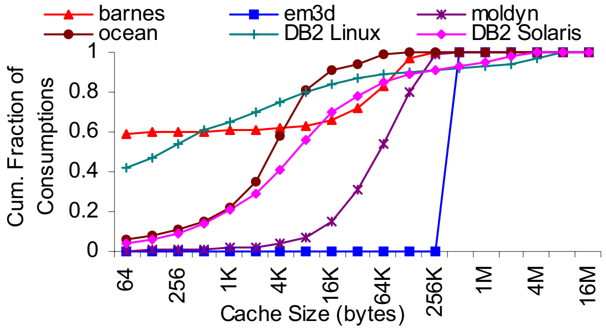
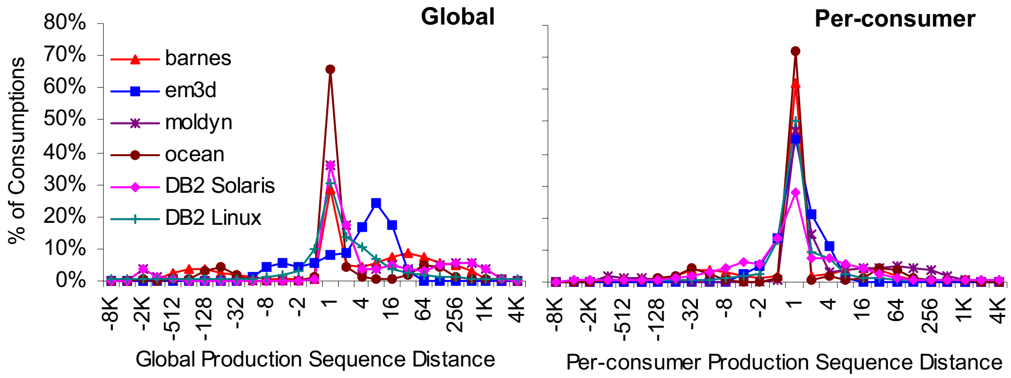


Figure 2. Cumulative fraction of consumptions eliminated as a function of storage size.

Aggressive streaming upon production requires too much storage at the consumer to be successful. Instead, we propose storing values in main memory upon production, and throttling the forwarding rate to match the consumption rate. Throttling will allow data to be streamed successfully into a small (e.g., 32-entry) buffer. SORDS throttles the rate by forwarding streams in *chunks* (i.e., small groups of blocks). When the consumer first accesses any block in a chunk, it signals SORDS to forward the next chunk. Thus, at steady state, only two chunks from each simultaneously live stream need to be stored at the consumer. The chunk size should be large enough to: (1) capture small reorderings between the production and consumption sequence, and (2) overlap consumptions of one chunk with the forwarding of the subsequent chunk. We address (1) in the following section and (2) in Section 4.2.

Producer-consumer temporal address correlation. Our goal with SORDS is to exploit strong temporal correlation between the production and consumption sequences to forward blocks in production order. We quantify P-C correlation by calculating the distance (in number of productions) on the production sequence between the productions that create values for two consecutive consumptions. For example, if the production order is {*A*,*B*,*C*,*D*} and the consumption order is {*A*,*B*,*D*,*C*} then the production sequence distance between A and B is +1 (i.e., perfect correlation; *B* follows *A* in the production sequence), whereas the distance between *D* and *C* is -1 (i.e., the production of *C* immediately precedes the production of *D*). Larger positive or negative distances indicate that the consumer has “jumped” from one part of the production sequence to another.

We first evaluate the production distances of consumptions relative to the total order of productions at each producer, labelled “global” in Figure 3 (left). These results indicate that production and consumption orders frequently match. An average of 31% of all consumptions precisely follow global



Per-consumer P-C Correlation (± 4)	
barnes	72%
em3d	98%
moldyn	66%
ocean	76%
DB2 Solaris	68%
DB2 Linux	84%

Figure 3. Producer-consumer temporal address correlation. The left graph shows distances between consecutive consumptions measured along the global production sequence. The right graph shows distance measured along per-consumer sequences. The table lists the total percentage of consumptions with per-consumer production distance ≤ ± 4.

production order, indicating that there is significant opportunity for production-ordered streaming.

It is not unusual for an application to interleave the production of shared values for multiple consumers. From the point of view of each consumer, productions destined for other consumers pollute the global production sequence, decreasing P-C correlation. Splitting the global production sequence into "per-consumer" sequences using perfect knowledge of future consumers extracts significantly more P-C correlation. Figure 3 (right) depicts the P-C correlation between each consumption sequence, and its corresponding per-consumer production sequence. An average of 51% of all consumptions precisely follow the split per-consumer production orders.

The figure also indicates that a significant fraction of consumptions are only slightly out-of-order with respect to the global and per-consumer production sequences. The table in Figure 3 sums the fraction of consumptions that follow the per-consumer production sequence with a production distance of up to four (i.e., the consumptions are out-of-order with respect to the production sequence by at most four blocks). By forwarding blocks in chunks, SORDS can tolerate these small reorderings and has the potential to cover these consumptions. With a chunk size of four blocks, SORDS can capture between 66% and 98% of all consumptions. Larger chunk sizes provide diminishing improvements.

In practice, SORDS can exploit both global and per-consumer P-C correlation. In applications where sharing patterns repeat and the set of consumers for each production can be predicted (e.g., em3d), SORDS can take advantage of per-consumer P-C correlation. When future consumers are less predictable (as in the lock-based applications barnes and DB2), SORDS can still exploit global P-C correlation. In contrast, eager forwarding approaches rely solely on accurate consumer-set prediction and have no recourse when consumer sets are not predictable, as they have no other mechanism to identify which nodes should receive forwarded data.

Stream on demand. The graphs in Figure 3 also indicate that whereas the majority of production distances are small, the distance distribution is fat-tailed in both directions. The tails of the distribution arise from cases where the consumer jumps from one portion of the production sequence to another. Together, the distributions' peak at +1 and significant tails indicate that the production sequence is composed of a number of distinct streams (i.e., consumption subsequences) that are ordered arbitrarily far apart from each other; the consumer often jumps between streams on the production sequence. This result has two important implications. First, simple FIFO throttling schemes can not be effective in streaming data from the production sequence, because they enforce a strict total order and thus can not support stream jumps. Second, to identify the start of the stream (i.e., stream head), to forward data just-in-time, and to avoid sending unwanted data, streams should be initiated on demand, with a miss to a cache block in the production sequence indicating a new stream head. Thus, to supply each consumer with the appropriate segment of the production sequence, SORDS must provide random access to the stream queue (which contains the production sequence).

Figure 4 shows a cumulative breakdown of the fraction of consumptions belonging to streams of a particular length, assuming a forwarding chunk size of four. As the graph shows, most streams are longer than 16 blocks. Although initiating streams on demand incurs one consumption miss per stream (to the stream head), SORDS sacrifices less than 1/16 of its potential coverage to these misses.

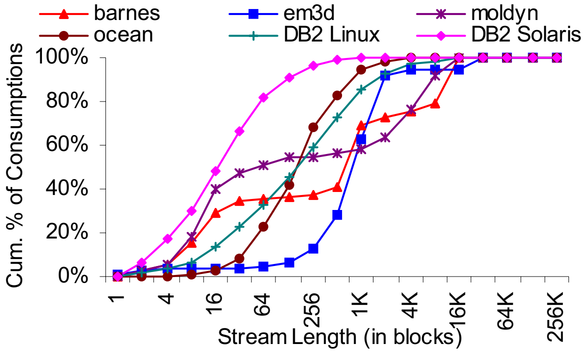


Figure 4. Cumulative percentage of consumptions on streams of a given length.

Figure 4 also shows that some streams are hundreds of cache blocks long. Prefetching or forwarding approaches that transfer only a fixed number of blocks per miss [30,33] will sacrifice a larger fraction of potential coverage than the single miss per stream that SORDS incurs.

Summary. We showed that: (1) to stream effectively, forwarding must be throttled, (2) SORDS can throttle forwarding by exploiting the strong producer-consumer temporal address correlation, and (3) SORDS must provide random access to the production sequence to allow for initiating streams on demand. Based on these observations, we now present a design for SORDS.

3. A Design for Store-Ordered Streaming

In Section 2, we presented an overview of how SORDS eliminates coherent read misses and analyzed the phenomenon of producer-consumer temporal address correlation on which SORDS relies and its implications for a SORDS design. In this section, we present our design for a practical hardware implementation of SORDS.

To support scalable systems, the SORDS functionality must be distributed across all DSM nodes, much like a distributed directory scheme. The SORDS hardware at each node records the production order for shared values and forwards streams of these values to consumers. SORDS's operation comprises five steps:

1. Predict which stores produce shared values and forward these values to the directory.
2. Predict the set of consumers for each production.
3. Append the block's address to the end of stream queues for each predicted consumer.
4. Upon a demand miss, locate the missing address in the stream queue and forward a chunk starting at this location.
5. Upon a hit in a consumer's streamed value buffer, notify the stream engine to forward the next chunk.

Figure 5 depicts the hardware components that SORDS adds to a base DSM node. The numbers in the figure indicate which of the above steps each component participates in. A *DownGrade Predictor (DGP)* at each processor approximates the production oracle discussed in Section 2.3. It predicts the last store to a cache block prior to a subsequent consumption miss, self-downgrades the cache block, and writes the produced data back to main memory (1). A *Consumer Set Predictor (CSP)* located in the directory approximates the consumer-set oracle discussed in Section 2.3. When a self-downgraded block arrives at main memory, CSP predicts which nodes will request shared copies of it (2). The operation of DGP and CSP is described in Section 3.1.

Once CSP has predicted a set of consumers, the *Stream Engine (SE)* records the address of the produced block on one or more stream queues (3) located in main memory. When a consumer later requests this block, the SE accesses the stream queue and begins forwarding the stream from that location (4). Note that the stream queue contains only a list of addresses—the data for each block are read from memory as the blocks are streamed. At the consumer node, forwarded data are stored in a *Streamed Value Buffer (SVB)* that is accessed in parallel with the data cache (5). When a load hits in the SVB, the data are transferred to the L1 data cache, and, if necessary, a hit notification is sent to the producer's SE requesting more data from the stream. Section 3.2 details the structure and operation of the SE and SVB.

3.1. Predicting Productions & Consumer Sets

SORDS uses two predictor components to identify when shared values are produced, and which nodes will subsequently consume those values. Computer architecture literature contains extensive studies of hardware mechanisms to make these predictions [17,20,21,34]. The choice of particular predictor designs is orthogonal to the streaming mechanisms in SORDS.

We study SORDS with DGP [34] as the production predictor because it has been shown to be effective in commercial workloads. We evaluate SORDS with two alternative consumer set predictors: CSP [24], a history-based predictor that can identify complex sharing patterns; and LastMask, a simple sharing predictor that predicts the consumer set for a new production of a block will match the final consumer set of the previous production of the block. We briefly summarize the operation of DGP and CSP here, but refer readers to [24] for a thorough discussion of the implementation, hardware costs, and design parameter sensitivity of these predictors.

The goal of DGP is to identify productions. DGP associates the downgrade event for a production with the sequence of store instructions accessing the block, from the time the block is first modified until the last store prior to its downgrade. As store instructions are processed, the DGP hardware encodes the PCs into a trace for each block present in the cache. A block's current trace is entered into a signature table when the block is downgraded. If the new on-line trace calculated for a block upon a store is present in the signature table, the DGP triggers a self-downgrade of the block. Thus, DGP captures program behaviors that repetitively lead to productions.

The goal of CSP is to predict the consumers of each production. The intuition underlying CSP is that the pattern by which values move between nodes, although arbitrarily complex, is repetitious. CSP maintains a history of the most recent sharing pattern for each block in the directory. Each history entry records the producer and consumers of the last few productions of the block. CSP associates the set of consumers of a production with the history that led to the production, and stores this association

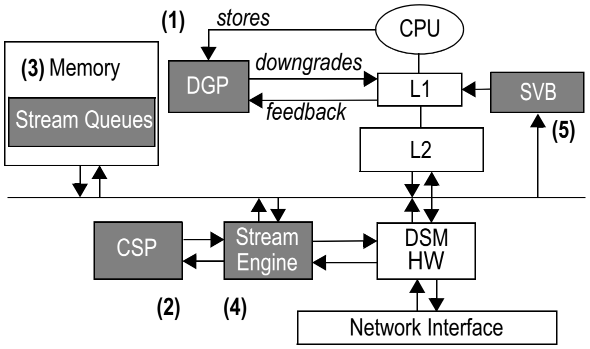


Figure 5. Anatomy of a SORDS-based DSM node.

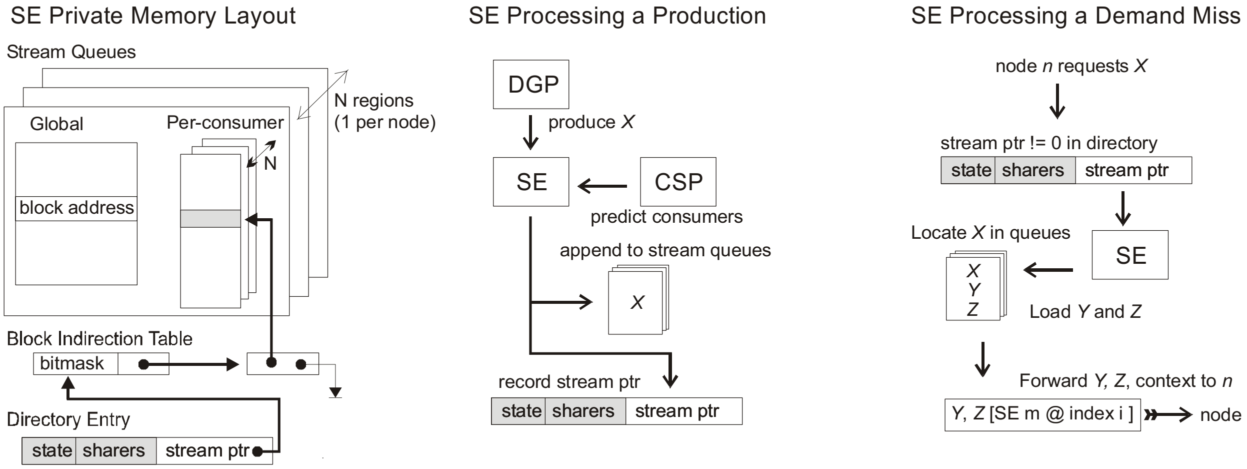


Figure 6. Stream Engine data structures and processing. The left-most figure depicts the data structures the SE stores in memory. The center and right-most figure depict the SE processing a production and demand miss, respectively.

in a signature table. Upon a production, CSP uses the current history for the block to obtain a predicted set of consumers from the table. Thus, CSP accurately predicts consumers when sharing patterns repeat.

3.2. Mechanisms for Streaming

The SORDS Stream Engine (SE) is designed to provide the functionality identified as necessary in Section 2.3 to exploit both global and per-consumer P-C correlation. This section details the functionality of the SE.

The SE records the sequence in which DGP-downgraded blocks arrive at the directory. Potentially thousands of values may be produced before any are consumed, resulting in large stream queues. Thus, the data structures pertaining to stream queues are stored in a private region of DRAM at each node, with a small cache in the SE used to accelerate accesses [26].

Figure 6 (left) depicts the layout of the SE's private memory space. The space is divided into two main structures: a set of *stream queues* (the majority of storage), and a *block indirection table*. The stream queues are circular queues that store lists of cache block addresses in production order, while the block indirection table enables lookup of an address across stream queues. The stream queue storage is divided into separate *regions* that each record productions by one node. Within each producer's region, there are *per-consumer* stream queues for each consumer node, and one additional *global* queue. In a 16-node system, there are 17 stream queues within each of the 16 producer regions.

Figure 6 (center) depicts the operation of the SE when a DGP-triggered self-downgrade arrives. The SE obtains a CSP prediction for the produced block. If a consumer set is not predicted (e.g., because of low confidence or because the sharing history has never been encountered before), the production address is appended to the global stream queue. To facilitate fast stream lookup, the SE also records the index of the stream queue entry in a *stream pointer* field stored with the block's directory entry. If CSP predicts a consumer set, the production address is appended to each of the indicated per-consumer stream queues. To support rapid lookup for all occurrences of the address, the SE creates a linked list within the block indirection table pointing to all private stream queue locations of the block. The head of this linked list records a bit mask indicating which private stream queues contain the address. The stream pointer in the directory points to the head of this linked list. The directory overhead of the stream pointer is $\log_2$(max entries on queue) + 1 bits. We analyze the storage requirement of stream queues in Section 4.3.

Figure 6 (right) depicts the operation of the SE upon receipt of a read miss at the directory. If the stream pointer for the block is initialized, the coherence engine passes the requested address, identity of the requesting node, and the stream pointer to the SE for processing. The SE uses the stream pointer to quickly determine which stream queues contain the block. If the block's address is present on a stream queue for this consumer, the stream engine initiates streaming from the indicated stream queue location. The number of blocks to forward is determined by the chunk size parameter of the SORDS design (see Section 4.3).

Each consumer stores streamed blocks in its Streamed Value Buffer (SVB), a small fully-associative buffer with LRU replacement. The buffer stores block addresses, values, and the *stream context*. The stream context is composed of the identity of the forwarding SE, an identifier for the associated stream queue, and a stream queue pointer indicating from where forwarding should continue. The SVB contains only clean data, and SVB entries are discarded upon a write by any node (including the local node) to maintain coherence. Upon a hit in the SVB, the streamed block is transferred to the L1 data cache and a hit notification containing the stream context is sent to the SE indicating where the stream should be continued. The advantage of tracking stream context through the SVB is that the SE does not need to track live streams—each consumer supplies the necessary state with each hit notification. Thus, the number of parallel streams is limited only by storage constraints at the consumer. Upon a hit, other blocks in the SVB from the same chunk are flagged to avoid duplicate hit notifications.

	DGP		*CSP*		*LastMask*	
	Cov	*Misp*	*Cov*	*Misp*	*Cov*	*Misp*
barnes	88%	4%	38%	4%	40%	56%
em3d	100%	0%	100%	0%	100%	0%
moldyn	97%	0%	99%	0%	31%	59%
ocean	82%	7%	89%	4%	80%	20%
DB2 Solaris	67%	14%	9%	6%	23%	75%
DB2 Linux	71%	8%	45%	17%	11%	87%

TABLE 2. Production and sharing prediction results.

4. Results

We first report the effectiveness of the predictor mechanisms we use to identify productions and predict consumer sets as input to the SORDS streaming mechanisms. We then analyze the design parameters of the SORDS streaming hardware. Finally, we evaluate the effectiveness of our proposed SORDS hardware at eliminating consumption misses.

4.1. Predictor Results

SORDS depends upon accurate prediction of productions, and benefits greatly from accurate prediction of the consumer set for each production. Table 2 presents the coverage and misprediction rate of our production predictor (DGP), and the two alternative sharing prediction techniques, CSP and LastMask, as described in Section 3.1. Coverage is the fraction of productions or consumers correctly identified by a prediction mechanism. Mispredictions represent over-predictions—stores incorrectly identified as productions or predicted consumers which do not read a produced value.

Our DGP results corroborate previously published results [34] for both scientific and commercial applications. The trace-based DGP exhibits near-perfect coverage with low discards on the scientific applications, which are generally repetitive across program iterations. OLTP workloads exhibit data-dependent behavior, and therefore productions are less predictable. The higher rate of DGP mispredictions for OLTP applications will not degrade performance if a relaxed memory system [1,10] is employed, because the additional write misses from DGP mispredictions can be fully overlapped. The high DGP coverage across applications ensures that there is significant opportunity for SORDS to eliminate consumption misses, as SORDS cannot stream producted values that the predictor does not identify.

The history-based CSP sharing predictor equals or outperforms simple last mask prediction across applications. For the scientific applications with stable and highly repetitive sharing patterns (em3d, moldyn, ocean), CSP predicts nearly all sharers correctly, with virtually no mispredictions. In the lock-based applications (barnes, DB2) where sharing patterns change frequently, CSP predicts conservatively, while last mask often predicts an incorrect sharing list. CSP's confidence mechanism gives it an advantage over last mask for these applications. Accurate CSP predictions, where possible, allow SORDS to exploit per-consumer P-C correlation for more accurate streaming.

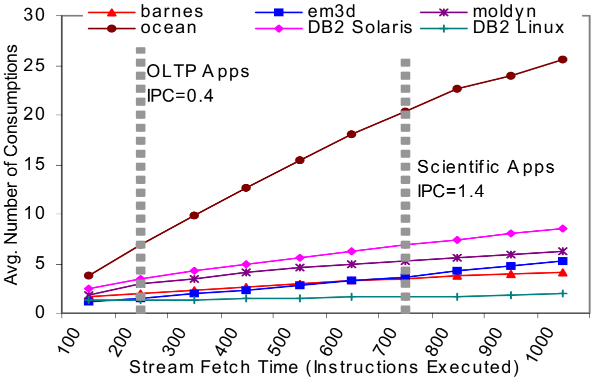


Figure 7. Required chunk size as a function of stream round-trip fetch time.

4.2. Chunk Size & Forwarding Lookahead

The primary role of the SORDS chunk size parameter is to ensure that the consumer node does not stall waiting for forwarded data while consuming a long stream. SORDS incurs a full network round-trip latency each time the consumer requests forwarding of the next stream chunk. When successive consumptions are clustered together in bursts, there is insufficient time to forward each block individually. For SORDS to be effective, we must select a chunk size that is sufficiently large to supply enough data to satisfy typical bursts of consumptions. However, if we choose too large a chunk size, storage at the consumer is wasted and fewer streams can be followed in parallel. Thus, selecting a chunk size involves balancing storage requirements at the consumer and overlapping the round-trip messaging delay of forwarding during consumption bursts.

We analyze each of our workloads to find the typical bursts of consumptions that must be overlapped for various forwarding delays. We measure forwarding delay in instructions executed at the consumer to remain independent of microarchitecture and cache configuration. For each forwarding delay, we measure how many consumptions on average occur within one forwarding window for all windows containing a burst of more than one consumption. We consider only consumptions that occur in clustered bursts because consumptions that are further apart than the forwarding delay can be successfully streamed at any chunk size.

Figure 7 shows the results of our chunk size analysis. The required chunk size for an application depends on its IPC and the round-trip network latency. For a 2-hop round-trip network latency of 500 cycles and an IPC of 0.4 for the OLTP workloads [2], a round-trip corresponds to 200 instructions. For this design point, Figure 7 shows that a chunk size of four will fully overlap the consumption bursts. For a typical scientific benchmark IPC of 1.4 [8], Figure 7 shows that a chunk size of four to six will fully overlap the consumption bursts for all scientific applications except for ocean.

The version of ocean we study (taken from [39]) is an enhanced version of the original benchmark that uses sub-blocking to improve the communication to computation ratio. Sub-blocking has the effect of grouping all the consumptions of a sub-block into a single burst. This optimization is counterproduc-

1089-795X/05 $20.00 © 2005 IEEE

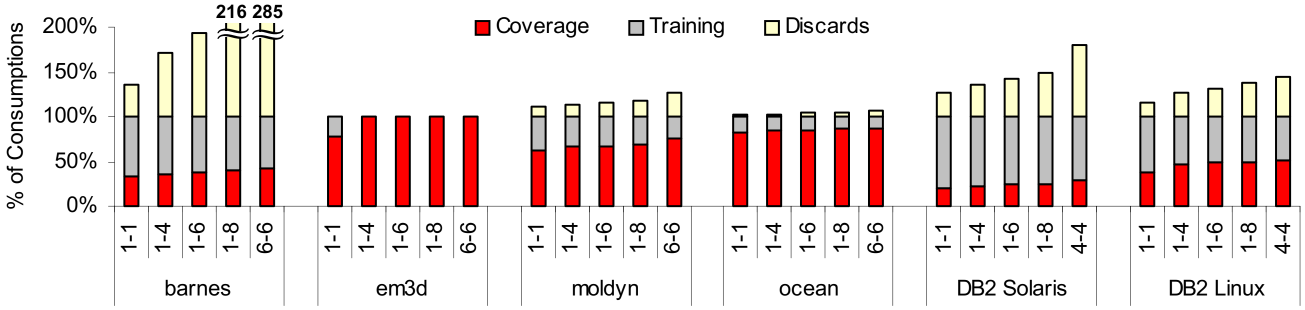


Figure 8. SORDS sensitivity to forwarding chunk size. Each forwarding chunk design is listed as *x-y. x* refers to the size of the *head* chunk sent upon a demand miss, *y* refers to the body chunk size sent in reply to a hit notification.

tive with SORDS, because SORDS will ensure a steady stream of blocks even if consumptions are evenly spaced. However, even if SORDS cannot fully overlap all consumptions for ocean, it will still improve performance by reducing the number of misses to one per chunk.

4.3. SORDS Design Space

We performed an analysis of the storage requirements for SORDS stream queues, and found that increasing storage beyond 2048 entries per stream queue had little effect on any application. With fewer entries, coverage drops off rapidly. With 2048 entries, the total storage required at each node for a 16-node system is roughly 5.5 MB (17 stream queues for each of 16 producers; up to 10 bytes per entry). The 5.5 MB storage requirement is large enough to prevent SORDS from using on-chip SRAM for stream queues, but is a negligible fraction of main memory.

Section 4.2 investigated the SORDS chunk size and determined that between four and six blocks are required to overlap the round-trip latency of forwarding. Chunk size also affects SORDS coverage. Increasing chunk size with fixed storage at the consumer reduces the number of streams that can be followed in parallel, which increases the likelihood of replacing useful but as yet unconsumed blocks. To avoid this effect, we have found that sending only a single head block upon creation of a new stream is effective at reducing the number of replaced blocks, without sacrificing much coverage. When the head block is consumed, we forward the remainder of long streams using the chunk size derived in Section 4.2.

Figure 8 presents SORDS results for a variety of forwarding chunk designs, demonstrating the effect of this optimization. These results use CSP as the sharing predictor. “Coverage” is the fraction of all consumptions that SORDS eliminates. “Training” are consumptions that SORDS cannot eliminate, and are instead used to train the prediction mechanisms. “Discards” are blocks that were forwarded to a consumer but never used—either the SVB evicted the block or it was invalidated because of a write by another processor. First, the graph shows that SORDS is very effective at eliminating nearly all consumptions for the applications where CSP is highly effective (em3d, moldyn, ocean) and SORDS can exploit per-consumer P-C correlation. In moldyn, there is a phase of execution that is characterized by many parallel, short streams. This phase causes the ~20% gap between SORDS coverage and moldyn’s perfect CSP coverage. For the lock-based applications, where CSP is less effective, SORDS still eliminates 25% to 50% of coherence misses. Second, Figure 8 shows that our head block optimization is effective at reducing discards. Only moldyn suffers from the optimization, again because of its frequent short streams.

Figure 9 evaluates SORDS across sharing predictors. Mask refers to the last sharing mask prediction technique. For applications where sharing prediction is effective, SORDS sees considerable advantage from being able to exploit per-consumer rather than global P-C correlation. In barnes, where consumers are generally unpredictable, the high discard rate for the None category shows that forwarding from the global stream queue causes many discards. Global P-C correlation is relatively poor for barnes (see Figure 3). The last mask prediction technique never places blocks on the global stream queue because it always predicts a set of consumers. CSP, however, will not predict sharers if prediction confidence is low. Thus, CSP exhibits a similar, though smaller, discard effect as seen without a sharing predictor. In DB2 Linux, coverage without a sharing predictor is slightly higher than CSP, as non-predicted consumers are able to find long streams on the global stream queue. However, removing the sharing predictor doubles the discards.

4.4. Comparison to Alternative Techniques

Figure 10 compares our final SORDS design with two other techniques for eliminating coherent read misses. *Eager* shares DGP and CSP with SORDS, but forwards produced blocks to predicted consumers immediately upon production. *Stride* is an adaptive stride prefetcher that examines memory patterns at the

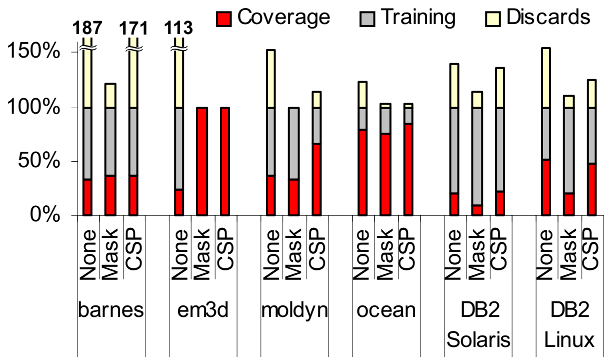


Figure 9. SORDS with various sharing predictors.

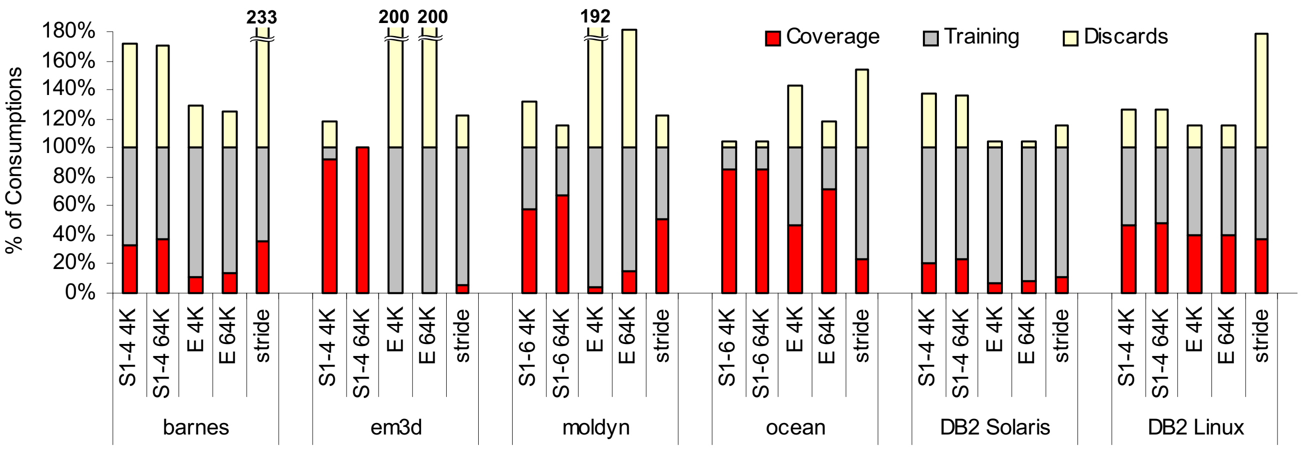


Figure 10. SORDS compared to alternative techniques for eliminating consumption misses. The "S *x-y z*" bars represent SORDS with a head chunk of *x,* a body chunk of *y,* and a forward buffer of *z* bytes. *The "*E *z*" bars refer to eager forwarding with a forward buffer of *z* bytes. The "stride" bar refers to stride-based prediction.

directory for strided accesses. When a stride is located, the prefetcher sends the next four blocks along the stride. SORDS and Eager are each considered for two different SVB sizes (4KB and 16KB).

SORDS is clearly superior to eager forwarding. Eager forwards data prematurely, replacing useful data in the consumer's small SVB, leading to high discard rates in em3d, moldyn, and ocean. In cases where CSP makes few predictions, eager forwarding sends few blocks, resulting in both low coverage and low discards.

SORDS is also superior to stride-based prefetching, in both coverage and discards. Stride results in many discards because it is incapable of throttling. The access patterns in em3d and ocean are not strided, resulting in much lower coverage for stride. SORDS provides about 10% more coverage for DB2 Solaris, and 20% more for moldyn. Coverage is similar for barnes. For DB2 Linux and barnes, SORDS coverage is limited by the difficulty of sharing prediction, which does not limit the stride prefetcher.

SORDS is relatively insensitive to the size of the consumer's SVB. Because there are few streams followed in parallel, and throttling limits occupancy at the buffer, 4KB of storage is sufficient. Moldyn is the exception, because its many parallel streams put significant pressure on the SVB during bursts. Eager forwarding is more sensitive to buffer size, because the SVB must contain all produced but unconsumed values.

5. Related Work

Prior studies have shown that coherent write miss latency can be hidden through relaxed consistency models [1], or by speculatively relaxing ordering constraints under sequential consistency [10]. Coherence optimizations directly reduce the latency of coherent read misses through optimizing the coherence protocol for particular access patterns [16,36], predicting coherence activity and initiating it in advance of explicit requests [17,20,21,27], or speculatively using incoherent values [13]. Token coherence [25] eases the implementation of these optimizations by splitting coherence protocols into separate performance and correctness protocols, reducing the protocol verification burden. However, these proposals either target only specific sharing patterns (e.g., migratory or false sharing) or hide only part of the coherence latency (e.g., one hop of a coherence transaction). Furthermore, none of these proposals increase coherence MLP.

Prefetching techniques [11,14,22,30,33] can initiate coherence transfers in advance of processor requests, and thus have the potential to fully eliminate the latency of a coherence miss. However, many prefetchers limit their maximum effectiveness by targeting only one miss at a time [11,14,22] or transferring a fixed number of blocks per miss [30,33]. At the other extreme, forwarding [19] places no restriction on block transfers, potentially forwarding too many blocks ahead of consumer demand and increasing pressure on limited on-chip storage. In contrast, memory streaming approaches throttle the transfer of arbitrary length access sequences and thereby avoid sacrificing opportunity while using limited on-chip storage efficiently.

Other proposals advocate increasing the effective MLP by simulating the effects of larger instruction windows through runahead execution [29] or by decoupling the computation and memory-access slices of program execution [6,35,37]. However, in contrast to history-based streaming techniques like SORDS, these approaches do not increase memory-level parallelism when memory accesses are dependent—for example, when chasing pointers in linked-data structures, one memory access must complete before the subsequent access can proceed. 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.

6. Conclusion

In this paper, we presented SORDS, a memory streaming technique for eliminating coherent read misses in distributed shared-memory systems. We demonstrated the phenomenon of producer-consumer temporal address correlation—that production and consumption orders are highly similar—and showed how to exploit this to improve performance. We demonstrated that throttled streaming is essential for eliminating a large fraction of coherence misses with minimal storage. We introduced a first design for SORDS comprising: DGP to identify downgrades; CSP to predict subsequent consumers; and a Stream Engine to stream data at the rate of consumption. We evaluated this design and showed that SORDS can eliminate 36%-100% of coherent read misses in scientific applications and 23%-48% in OLTP workloads.

References

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

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *The VLDB Journal*, pages 266–277, Sept. 1999.

[3] 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.

[4] 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.

[5] 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.

[6] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[7] 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*, Nov. 1993.

[8] 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.

[9] B. Falsafi and D. A. Wood. Reactive NUMA: A design for unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, June 1997.

[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] D. Gracia Pérez, G. Mouchard, and O. Temam. MicroLib: A case for the quantitative comparison of micro-architecture mechanisms. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 37)*, Dec. 2004.

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

[16] A. Kägi, N. Aboulenein, D. C. Burger, and J. R. Goodman. Techniques for reducing overheads of shared-memory multiprocessing. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 11–20, July 1995.

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

[18] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance characterization of a quad Pentium Pro SMP using OLTP workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[19] D. Koufaty, X. Chen, D. Poulsen, and J. Torrellas. Data forwarding in scalable shared-memory multiprocessors. In *Proceedings of the 1995 International Conference on Supercomputing*, page ?, 1995.

[20] 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.

[21] 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.

[22] A.-C. Lai and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[23] C. Levine. TPC-C: The OLTP benchmark. In *TPC Technical Report Article at www.tpc.org*.

[24] 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, Feb. 2002.

[25] 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.

[26] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent network interfaces for fine-grain communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, May 1996.

[27] 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.

[28] 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 *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, July 1995.

[29] 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.

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

[31] 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 Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, Oct. 1998.

[32] 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)*, December 2000.

[33] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[34] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Memory coherence activity prediction in commercial workloads. In *Proceedings of the Third Workshop on Memory Performance Issues (WMPI-2004)*, June 2004.

[35] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Oct. 2004.

[36] P. Stenstrom, M. Brorsson, and L. Sandberg. Adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[37] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, November 2000.

[38] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[39] 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*, pages 24–36, July 1995.