Computer Architecture Lab at Carnegie Mellon (CALCM) Technical Report 2004-2

SORDS: Just-In-Time Streaming of Temporally-Correlated Shared Data

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

Abstract

Coherence misses in shared-memory multiprocessors account for a substantial fraction of execution time in many important scientific and commercial workloads. While store miss latency can be effectively tolerated using relaxed memory ordering, load latency to shared data remains a bottleneck. Current proposals for mitigating coherence misses either reduce the latency by optimizing the coherence activity (e.g., self-invalidation) or prefetch specific memory access patterns (e.g., strides) but fall short of eliminating the miss latency for generalized memory access patterns.

This paper presents the novel observation that the order in which shared data is consumed by one processor is correlated to the order it was produced by another. We investigate this phenomenon, called temporal correlation, 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. We present a practical design that uses a set of cooperating hardware predictors to extract temporal correlation from shared data, and mechanisms for timely forwarding of this data. We present results using trace-driven analysis of full-system cache-coherent distributed shared memory simulation to show that our SORDS design can eliminate between 36% and 100% of all coherent read misses in scientific workloads and between 23% and 48% in OLTP workloads.

1. Introduction

Technological advancements in semiconductor fabrication along with microarchitectural and circuit innovation have led to phenomenal processor speed increases over the past decades. Over 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 sustains several network round-trip times. Adverse memory access patterns and frequent sharing of data promote coherence misses to a performance-limiting bottleneck in important commercial [22,4,12] and scientific [6,24,21] workloads.

There are a myriad of proposals for reducing or hiding coherence miss latency. Techniques to relax memory order have been shown to hide virtually all of the coherent write latency [1] by allowing reads of shared data to bypass in-program-order writes. Unfortunately, prior proposals have fallen short of hiding coherent read latency for generalized memory access patterns. Instead, most proposals seek to reduce read latency through coherence optimizations [19,23,10] or can hide only part of the latency [15,11]. Proposals that attempt to hide all read latency through prefetching/streaming [3] or forwarding [13] are only effective for simple memory access patterns (i.e., strided accesses). Scientific [14,21] and commercial [5] workloads, however, often exhibit *irregular* yet repetitive memory access patterns that are not amenable to simple predictive schemes such as stride prediction.

To hide the coherent read miss latency effectively, a design must deliver newly produced shared data *just-in-time* to consuming nodes. Recent research proposes generalized hardware prediction mechanisms for identifying *when* new shared values are produced [15] and *which* nodes will consume that data [11,14]. Much as modern branch predictors rely on repetitive program behavior to predict branch outcomes accurately using prior branch history, these predictors rely on repetitive memory access patterns to predict subsequent coherence events. Unfortunately, while these mechanisms have been shown to accurately predict generalized memory access patterns, they have only been tested on scientific [15,14] and desktop/engineering workloads [16]. Moreover, these predictors fall short of predicting *when* to forward the data to a consumer, and thus they are prone to either thrashing the consumer cache hierarchy (if they forward data early) or failing to fully hide read latency (if they forward data late).

Chilimbi [5] recently demonstrated that memory addresses exhibiting temporal locality at one point in a program recur together in nearly identical order throughout the program. By identifying long data reference sequences that recur frequently, one can form *hot streams* of references. These streams can then be fetched in stream order when the stream is accessed, thereby hiding memory read latency. Unfortunately, extracting repetitive streams from integer and on-line transaction processing (OLTP) applications [5] requires a sophisticated hierarchical compression algorithm to analyze whole program memory address traces, which may be practical only when run offline and is prohibitively complex to implement in hardware.

In this paper we demonstrate for the first time that, in scientific and OLTP workloads, shared data are consumed in approximately the same order that they were produced. We call this phenomenon *temporal correlation* of shared data. Based on this observation, we present *Store-ORDered Streaming (SORDS)*, a novel memory system design for just-in-time forwarding of temporally correlated shared values from producers to consumers. SORDS builds on existing prediction technology to identify the order in which shared data are produced and which nodes will consume them, and proposes novel hardware mechanisms to record this order and stream shared data to consumers just before they are needed. By analyzing memory access traces from full-system simulation [18] of cache-coherent distributed shared-memory multiprocessors running OLTP workloads with IBM DB2 and scientific applications, we demonstrate:

- **Temporal Correlation:** We show for the first time that the order in which shared values are consumed is very similar to the

order in which they are produced, and we present how this observation can be exploited in memory system design.

- **Just-In-Time Streaming:** We demonstrate that throttled streaming of shared values to consumer nodes enables forwarding into a small, low-latency buffer, thereby maximizing the potential performance benefits of streaming.
- **Practical Design:** We propose a first design for store-ordered streaming with practical hardware mechanisms. Our design eliminates 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 the approach of store ordered streaming and justify our approach based on 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 a combination of analytic modeling and trace-based simulation of scientific and OLTP workloads. Finally, we conclude in Section 5.

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 produced and subsequently consumed in shared memory: the order shared values are consumed is similar to the order in which they were produced. By capturing the order in which data are produced, SORDS enables throttling the stream of shared data into small buffers residing at the consumers just-in-time for consumption, thereby hiding the read miss latency.

A node in a DSM system must obtain exclusive access to a cache block prior to performing writes. Subsequently, the node continues to read and write the block until another node in the system requests access, which causes a downgrade at the writer. The last store operation to a block prior to downgrade is a *production*. The first read of this newly-produced value by each node is a *consumption*. If a consumption requires a coherence request to obtain the data, it is a *consumption miss*. The goal of SORDS is to eliminate consumption misses.

Designs that forward memory values from one DSM node to another prior to a request 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 [15] allows each node to identify 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 a group of subsequent blocks to the consumer (3). As the consumer hits on forwarded blocks, it signals the directory to forward additional groups (4).

Successful forwarding depends upon a high degree of temporal correlation between the production and consumption sequences. As long as the consumer continues to access blocks

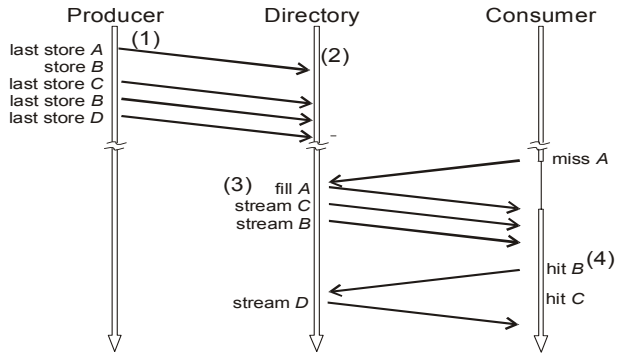


Figure 1. Eliminating coherent read misses in SORDS.

roughly in "store" (i.e., production) order, SORDS can eliminate the read misses. Intuitively such temporal correlation does exist: (1) in general, for both data items *within* and *across* data structures [5] — 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 given set of data items. In the remainder of this section, we show empirically that there is a high degree of temporal correlation in scientific and OLTP workloads, and justify the major design decisions of SORDS based on the nature of temporal correlation.

2.1. Methodology & Benchmarks

We demonstrate temporal 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 created using Virtutech *Simics* [18]. *Simics* is a full system simulator that allows functional simulation 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 multiprocessor system running *Solaris 8*. The processing nodes model SPARC v9 and the system employs 512MB of main memory. 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 (DBMS) performance. Moreover, DBMSs use different code bases across platforms, resulting in diversely varying synchronization and sharing behavior. We simulate a 16-node SPARC system and an 8-node x86 system (*Simics* uses a BIOS that does not support more than eight processors for x86).

Table 1 describes the applications we use in this study and their inputs. We select a representative group of pointer-intensive and array-based scientific applications: (1) that are scalable to large data sets, and (2) maintain a high sensitivity to memory system performance when scaled. These include *barnes* [24] a hierarchical N-body simulation, *em3d* [6] an electromagnetic force simulation, *moldyn* [21] a CHARMM-like molecular dynamics simulation, and *ocean* [24] current simulation.

We run version 7.2 of *DB2* with the TPC-C workload [17], an online 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

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.

exist for optimal transaction execution. Prior to measurement, we warm the database until the transaction completion rate in *Simics* reaches steady state. We analyze traces of at least 5,000 transactions.

2.2. Stream Properties

In this section, we explore consumption sequence properties of multiprocessor applications, and identify the streaming mechanisms required to eliminate the consumption misses. To gauge the full potential of streaming, we study it in the context of "oracle" knowledge of productions and their consumers. We present practical prediction techniques which 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. Such eager forwarding guarantees that each value arrives at consumers as early as possible, thereby minimizing the likelihood of incurring a miss penalty.

Unfortunately, this simple approach often fails because there is a large number of productions between two consumptions. For some applications, buffering these values at the consumer may require prohibitively large storage. Moreover, the worst-case storage requirement is highly dependent on application sharing behavior. Figure 2 plots the fraction of consumption misses eliminated as a function of available (fully-associative) storage at the consumers assuming our oracle model. For *em3d, moldyn,* and *DB2 Solaris*, hundreds to thousands of cache blocks must be buffered to cover a significant fraction of consumption misses. For *DB2 Linux*, similar size storage is required to capture the full opportunity for eliminating consumption misses. These results indicate that forwarding data into the conventional cache hierarchy would be counterproductive because: (1) forwarding into the L1 cache would thrash it, significantly reducing overall performance, and (2) forwarding into lower-level caches or the local DRAM memory [8] 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, worst-case size requirements may be higher due to forwarding unwanted data.

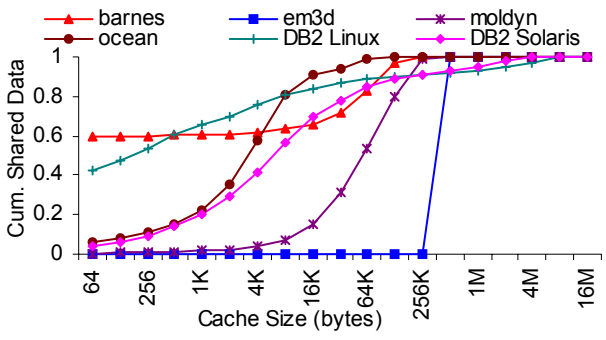


Figure 2. Cumulative fraction of consumptions eliminated as a function of storage size.

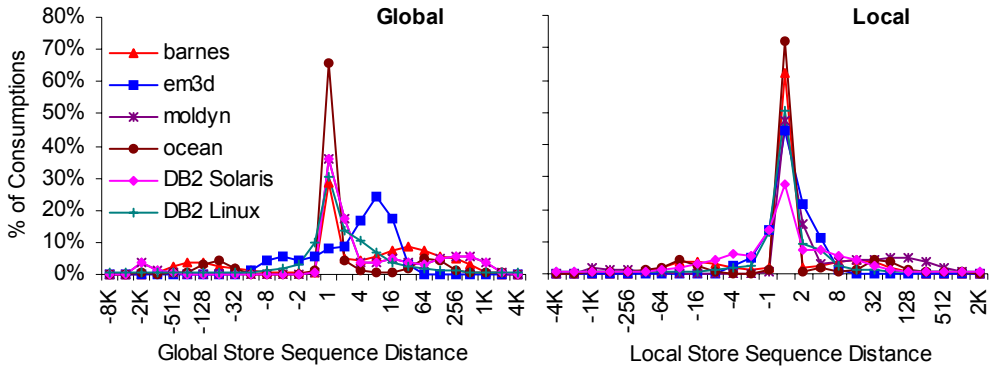
To stream data successfully into a small (e.g., 32-entry) buffer, the forwarding rate must be throttled to match the consumption rate. SORDS throttles the rate by forwarding streams in *chunks* (i.e., a small group of blocks). When the consumer first accesses any block in a chunk, it signals SORDS to 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 is selected so as 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.

Temporal correlation. To throttle forwarding, SORDS must record the order in which to forward. SORDS relies on strong temporal correlation between the production and consumption sequences to forward in production order. We measure temporal correlation by calculating the distance on the production sequence between two consecutive consumptions. Thus, a temporal correlation distance of +1 indicates that, for the two consumptions considered, they appear precisely in production order. Larger positive or negative distances indicate that the consumer has "jumped" from one part of the production sequence to another.

We first evaluate the temporal correlation distances of consumers on the "global" production sequence in Figure 3 (left). The global production sequence has no knowledge of future consumers and simply records the order in which productions arrive. These results indicate that an exact match between the global production and consumption orders is by far the most common case. An average of 31% of all consumptions precisely follow global production order. Therefore, there is much opportunity for throttled streaming even without predicting consumers.

It is not unusual for an application to interleave production of shared values for multiple consumers. Splitting the global production sequence into "local" (i.e., per consumer) sequences using perfect knowledge of future consumers extracts significantly more temporal correlation. Figure 3 (right) depicts the temporal correlation between each consumption sequence, and the per-consumer production sequence. A much higher average of 51% of all consumptions precisely follow the local production order (compared to global correlation).

The figure also indicates that there is a large fraction of consumptions that are only slightly out-of-order with respect to the global and local production sequences. These reorderings can



Total Local Temporal Correlation (± 4)	
barnes	72%
em3d	98%
moldyn	66%
ocean	76%
DB2 Solaris	68%
DB2 Linux	84%

Figure 3. Temporal correlation. The left graph shows distances between consecutive consumptions measured along the global production sequence. The right graph shows distance measured along the local sequence. The table lists the total percentage of consumptions for which the local distance is less than **± 4.**

be captured by simply using small chunk sizes. The table in Figure 3 sums up the coverage for distances that are within four cache blocks on the local production stream. The figure indicates that a chunk size of four has the potential to capture anywhere from 66% to 98% of all consumptions.

In practice, SORDS can exploit both types of temporal correlation. Upon accurate consumer-set prediction, SORDS can exploit local correlation where a production's consumers are repetitive (as in *em3d*), and fall back on global temporal correlation when future consumers are less predictable (as in lock-based applications *barnes* and *DB2*). In contrast, eager forwarding approaches that rely solely on consumer-set prediction have no recourse when consumer sets are not predictable.

Stream on demand. The graphs in Figure 3 also indicate that while the majority of the consumptions are covered within a small distance, the tail of the distance distribution is quite long in both directions. Therefore, the production sequence is made up 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 key implications. First, simple credit-based FIFO throttling schemes would not be effective in streaming data from the production sequence. To supply each consumer with the appropriate segment of the production sequence, SORDS must provide random access to the stream queue (containing the production sequence). Second, streams should be initiated on demand (upon a miss to a cache block in the production sequence) to identify the start of the stream (i.e., stream head), to forward data just-in-time, and to avoid sending unwanted data.

Figure 4 depicts a cumulative breakdown of the fraction of consumptions belonging to streams of a particular length, assuming a forwarding chunk size of four. A stream terminates when it intersects another stream. As the graph shows, streams are sufficiently long to render the miss to the head a negligible opportunity loss. *DB2 Solaris* generally has the shortest streams, with half of all consumptions on streams shorter than 16 blocks. *Em3d* is dominated by very long streams, with nearly 90% of consumptions on streams greater than 256 cache blocks.

Summary. We showed that, to stream effectively: (1) forwarding must be throttled, (2) SORDS can throttle data effectively due to the strong temporal correlation between the production and consumption orders, and (3) SORDS must provide random access to data on the production sequence to allow for initiating streams on demand. Based on these results, we now present a design for SORDS.

3. A Design For Store-Ordered Streaming

In Section 2 we present an overview of how SORDS eliminates coherent read misses, and analyze the temporal correlation property on which SORDS relies and its implications on the 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. Its function 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.

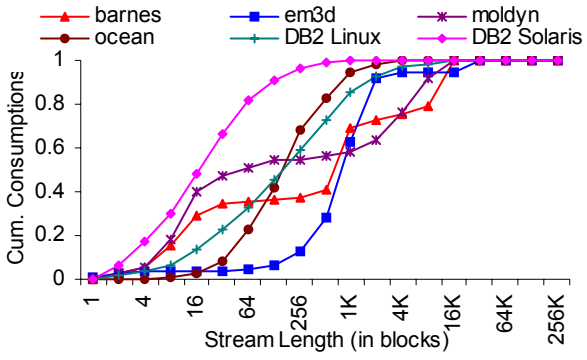


Figure 4. Cumulative fraction of consumptions on streams of a given length.

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 forward buffer, notify the stream engine to forward the next chunk.

Figure 5 depicts the hardware components which SORDS adds to a base DSM node. The numbers in the figure indicate roughly which of the steps above each component participates in. A *DownGrade Predictor (DGP)* at each processor approximates the production oracle discussed in Section 2.2. It predicts the last store to a cache block prior to a subsequent consumer miss, self-downgrades the cache block, and writes back the produced data to main memory (1). A *Consumer Set Predictor (CSP)* located in the directory approximates the consumer-set oracle discussed in Section 2.2. 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). At the consumer node, forwarded values are stored in a *Forward Buffer (FB)* that is accessed in parallel with the data caches of the CPU. When a load hits in the FB, the data is 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 data structures and operation of the SE and FB.

3.1. Predicting Productions & Consumer Sets

Computer architecture literature contains extensive studies for predicting when shared values are produced, and which nodes will subsequently consume those values [15,14,11,16]. We identified and tuned the most promising of these proposals to cooperate with our SORDS streaming mechanism.

The goal of DGP is to identify productions. Our DGP is based on Last-Touch Prediction (LTP) [15]. It reduces complexity and storage cost as compared to LTP because it only records stores (rather than both loads and stores), and predicts only downgrades (rather than both invalidations and downgrades). DGP associates the downgrade event for a production with the sequence of store instructions which access 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 in the cache. The current trace is entered into a signature table when a downgrade occurs. If the new trace signature calculated for a block upon a store is present in the table, the DGP triggers a self-downgrade of the block. Thus, DGP captures program behaviors which repetitively lead to productions.

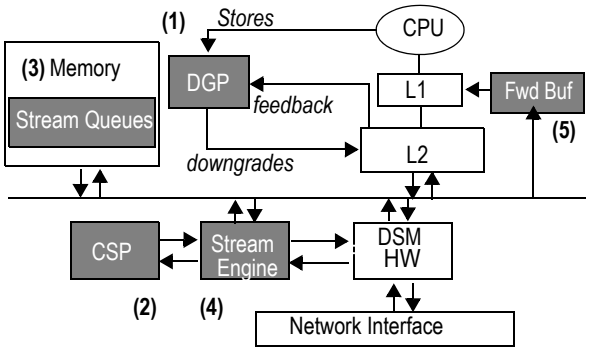


Figure 5. Anatomy of a SORDS-based DSM node.

The goal of CSP is to predict the consumers of each production. Our base CSP is derived from the Memory Sharing Predictor (MSP) [14]. It reduces complexity and storage cost as compared to MSP because it only predicts readers (rather than both readers and writes). 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 (producers and consumer sets) for each block in the directory. CSP associates the set of consumers of a production with the history that led to the production, and stores this association 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. CSP maintains a confidence for every signature and only predicts consumers if this confidence is high (if the signature and subsequent consumer set has recurred).

To gauge SORDS' coverage sensitivity to consumer-set prediction accuracy, in Section 4 we also evaluate a simple sharing predictor, *LastMask*, that uses the last consumer set at the directory as a prediction for future consumers.

3.2. Mechanisms for Streaming

The SORDS Stream Engine (SE) is designed to provide the functionality identified as necessary in Section 2.2 to exploit both global and local temporal correlation. This section details the functionality of the SE and its associated data structures.

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 and a cache is used to accelerate accesses [20].

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 which store lists of cache block addresses in production order. The stream queue storage is divided into separate *regions* for each producer node in the system. The stream queues within each region record productions by a single producer node, and they are comprised of one *private* stream queue for each consumer node, and one additional *global* queue. Each stream queue entry consists of a block address, and a consumer bit mask indicating if the block has been forwarded to that consumer. Thus, each entry is roughly the same size as a memory address. 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

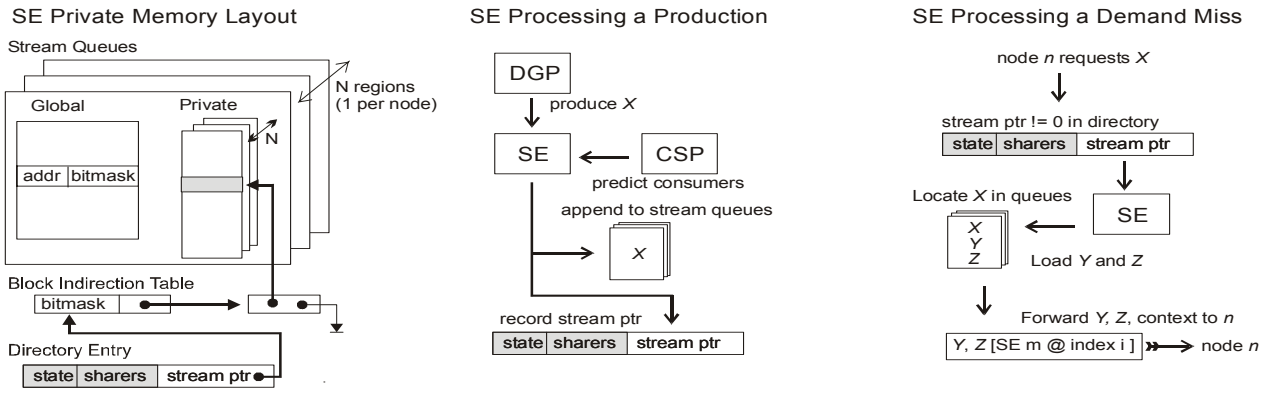


Figure 6. Stream Engine data structures and processing. The left-most figure depicts the data structures the SE stores in memory. The center and right-most figure depict the SE processing a production and demand miss, respectively.

entry in a *stream pointer* field stored with the block's directory information. If CSP predicts a consumer set, the production address is appended to each consumer's private stream queue. 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 the consumer stream queues in which the address appears. The SE also updates the stream pointer to point to the list head. The directory overhead of the stream pointer is $\log_2$(max entries on stream queue) + 1 bits. We analyze the storage requirement of stream queues in Section 4.4.

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 it is present on a stream queue for this consumer, it forwards data from the addresses consecutively following it in the stream, and marks all forwarded blocks as sent to the consumer. The number of blocks to forward is determined by the chunk size parameter of the SORDS design (see Section 4.4) Forwarding terminates at any block that has been invalidated or already forwarded to the consumer. When a node acquires write permission for a block, the SE invalidates existing stream queue entries for its address, because the value they represent is no longer the most recent for this address in the system.

Each consumer stores blocks in its Forward Buffer, 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. Upon a hit in the FB, a hit notification containing the stream context is sent to the SE which uses it to locate the next blocks to forward. The advantage of this approach 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. Note that upon a hit, other blocks in the FB from the same chunk are flagged to avoid duplicate hit notifications.

4. Results

4.1. Speedup Opportunity

In this section, we present a simple and intuitive model to gauge the performance improvement resulting from SORDS. In the interest of brevity and to gauge the maximum speedup opportunity using SORDS, we make a number of simplifying assumptions without loss of generality. First, we assume that all coherent write miss latency is hidden through a software-annotated [1] or hardware-speculative relaxed memory system [9]. Second, we assume that the applications are latency- and not bandwidth-bound. Prior research indicates that OLTP workloads' execution time is dominated by long chains of dependent memory accesses and therefore latency-bound [2,22]. Many pointer-intensive scientific applications (e.g., *barnes*, *em3d* [16] and *moldyn* [21]) exhibit similar memory dependence behavior and are also latency-bound. Consequently, we assume that coherent read misses are on the execution's critical path, and there is sufficient bandwidth in the interconnect and memory system for streaming with little impact on queueing and occupancy.

We model the performance of a base DSM system with a three-hop coherence protocol as the sum of an application-specific per-node *CPI* (accounting for execution stalls in the microarchitecture and the memory hierarchy), and the additional latency caused by coherent read misses (consumptions), as miss per instruction, *M*, multiplied by the round-trip latency per consumption, *L* as follows:

$$CPI_{base} = CPI + M \cdot L$$

We model a SORDS-based DSM accounting for the fraction of consumptions whose latency can not be entirely hidden by SORDS. There are three classes of such consumptions: (1) those whose productions are not successfully downgraded by the DGP, (2) those whose productions are downgraded but are not forwarded to the consumers, and (3) those that are forwarded but are not timely. In the first case, SORDS incurs a full round-trip latency. We conservatively assume that the latter two cases incur

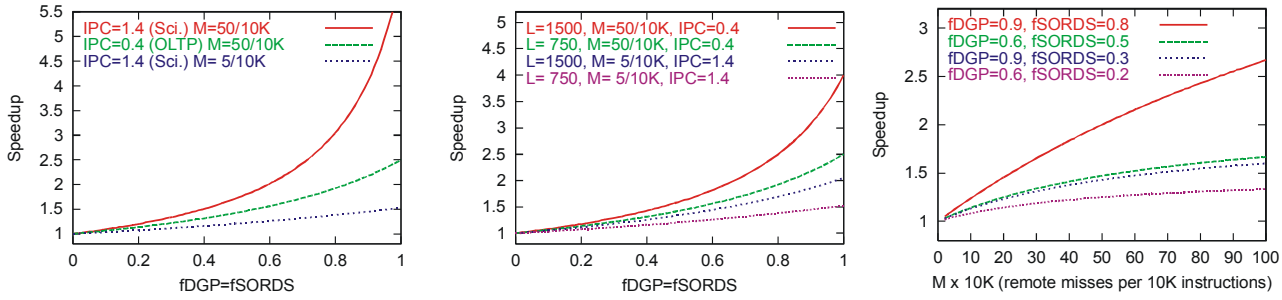


Figure 7. Model of SORDS performance impact. Each plot shows speedup from SORDS as we vary model parameters.

the full latency of fetching the production from the directory. This penalty is 2/3 of that of a full round-trip latency because the downgrade converts the 3-hop transaction for the read into a 2-hop transaction to the directory.

Let f_{SORDS} denote the fraction of consumptions that are successfully eliminated (i.e., correctly predicted and forwarded just-in-time). Let f_{DGP} denote the fraction of consumptions whose production is successfully predicted. Our model for the CPI of a SORDS-based system is:

$$CPI_{SORDS} = CPI + M \cdot L \cdot (1 - f_{DGP}) + \frac{2}{3} M \cdot L \cdot (f_{DGP} - f_{SORDS})$$

By dividing the two terms to obtain speedup, and rearranging we end up with:

$$\textbf{speedup} = \frac{CPI_{base}}{CPI_{SORDS}}$$

$$= \frac{1 + IPC \cdot M \cdot L}{1 + IPC \cdot M \cdot L \cdot (1 - f_{DGP}) + \frac{2}{3} IPC \cdot M \cdot L \cdot (f_{DGP} - f_{SORDS})}$$

We plug in IPC values based on published results for uniprocessor OLTP [2] and scientific workload [7] measurements on Pentium III and Alpha 21364 processors respectively. These results suggest an IPC of 0.4 for OLTP workloads and an IPC of 1.4 for scientific workloads. Table 2 lists the measured consumption miss rate (per 10000 instructions) for our applications. We conservatively assume an aggressive 3-hop round-trip latency for a coherent read miss of 150ns (a network traversal in an HP GS1280 Alphaserver with no memory access overhead), or 750 cycles for a 5 GHz processor.

Figure 7 (left) plots the base speedup results for the OLTP with M=50 misses per 10K instructions and scientific workloads with both M=5 and M=50 per 10K instructions. These results indicate that for scientific applications with high miss rates (e.g., *em3d*), there is a huge potential for performance improvement of over 5x using SORDS. However, the potential speedup significantly drops to only 1.5x at best with an order magnitude decrease in the miss rate (e.g., to values close to those of *barnes*, *moldyn*, and *ocean*). The OLTP workloads also show a huge speedup potential of up to 2.5x due to their high miss rate. Their speedup potential is lower as compared to *em3d* due to the overall low IPC (because of bottlenecks such as the local instruction and data cache misses, branch misprediction, and other pipeline dependence stalls [2]). Moreover, the speedup is exponential with respect to SORDS coverage and as such depends on the prediction accuracy and timeliness of the SORDS mechanisms.

Figure 7 (center) depicts performance sensitivity to round-trip latency for OLTP workloads with IPC=0.4 and M=50/10K and scientific workloads with IPC=1.4 and M=5/10K. These results indicate that because of high miss rates in OLTP workloads, an increase in system size and round-trip latency will have a much higher impact on their overall speedup opportunity using SORDS than it does on one of our typical scientific workload.

Finally, Figure 7 (right) compares performance sensitivity to DGP and SORDS coverage for the OLTP workloads (IPC=0.4) with varying miss rates and our base latency (L=750). The results indicate that while the maximum performance impact comes from high SORDS coverage, there is potential for moderate performance improvements (~1.5x) even when SORDS coverage is as low as 30% but DGP coverage is high (90%). However, speedup exhibits a logarithmic behavior with an increase in miss rate. Therefore, because downgraded productions that are not successfully forwarded incur 2/3 of the round-trip latency, with larger system sizes and higher miss rates, large speedup gains are not possible with high DGP coverage alone.

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. In order 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 analyzed 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

	barnes	*em3d*	*moldyn*	*ocean*	*DB2 Solaris*	*DB2 Linux*
M / 10K instructions	*2.1*	*84*	*3.7*	*4.1*	*57*	*36*

TABLE 2. Coherent read miss rates in applications.

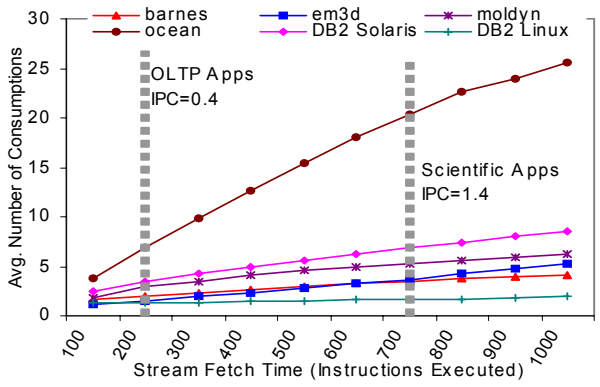


Figure 8. Required chunk size as a function of stream round-trip fetch time.

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. Figure 8 shows the results of this 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 8 shows that a chunk size of four will fully overlap the consumption bursts. For a typical scientific benchmark IPC of 1.4 [7], Figure 8 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 [24]) 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 counterproductive with SORDS, since 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. Predictor Results

SORDS depends upon accurate prediction of productions, and benefits greatly from accurate prediction of the consumer set for each production. Table 3 presents the coverage and misprediction rate of our production predictor (DGP), and two alternative sharing prediction techniques. CSP is the sophisticated sharing predictor described in Section 3.1. LastMask is a simple sharing predictor that predicts the consumer set for a new production will match the final consumer set of the previous production.

Our DGP results corroborate previously published results on instruction-trace based invalidation prediction for scientific applications [15]. Our study indicates that instruction-trace-based prediction is effective for OLTP applications as well. The higher rate of DGP mispredictions for OLTP applications will not degrade performance if a relaxed memory system [1,9] is employed, since the additional write misses from DGP mispredictions can be fully overlapped.

	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 3. Production and sharing prediction results.

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.

4.4. 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). This 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 examines SORDS chunk size and determines that between four to six blocks are required to overlap the round-trip latency of forwarding. Chunk size also affects SORDS coverage and discards. Increasing chunk size with a fixed storage at the consumer reduces the number of streams that can be stored in parallel, which increases the likelihood of replacing useful but as yet unconsumed blocks. Because of 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 9 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. Discards are blocks that were forwarded to a consumer but never used—either the forwarding buffer 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 local temporal correlation. In *moldyn*, there is a phase of execution that is char-

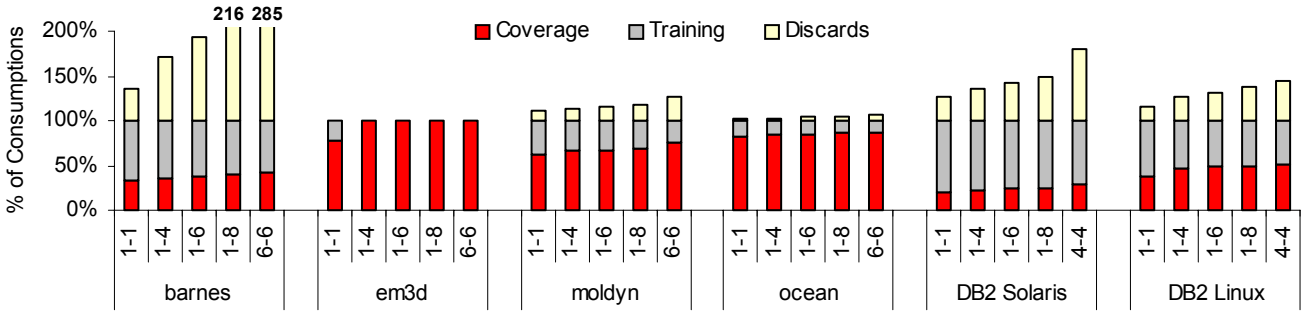


Figure 9. SORDS sensitivity to forwarding chunk size. Each forwarding chunk design is listed as *x-y*. *x* refers to the size of the *head* chunk sent upon a demand miss, *y* refers to the body chunk size sent in reply to a hit notification.

acterized 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 9 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 10 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 local rather than global temporal correlation. In *barnes*, where consumers are generally unpredictable, the high discard rate for the None result shows that forwarding from the global stream queue causes many discards. Global temporal correlation is relatively poor for *barnes* (see Figure 3). The last mask prediction technique never places blocks on the global stream queue, since last mask 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 discards.

Figure 11 presents a breakdown of SORDS hits and opportunity in user and OS code for our OLTP workloads. SORDS is more successful at predicting consumptions by the database than the OS. This graph also reveals a significant difference in the relative volume of OS and user sharing across platforms.

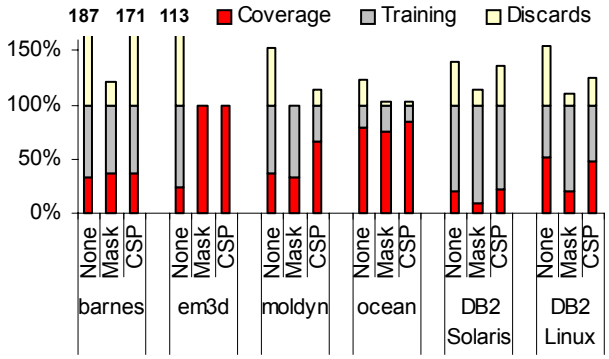


Figure 10. SORDS with various sharing predictors.

4.5. Comparison to Alternative Techniques

Figure 12 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 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 forward buffer sizes (4K and 16K).

SORDS is clearly superior to eager forwarding. Eager forwards data prematurely and thrashes the consumer's small forward buffer, leading to high discard rates in *em3d*, *moldyn*, and *ocean*. In cases where CSP makes very 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 always results in many discards since it is incapable of throttling. The access patterns in *em3d* and *ocean* are not strided. SORDS provides about 10% more coverage for *DB2,* 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 forward buffer. Since there are few streams followed in parallel, and throttling limits occupancy at the buffer, 4K of storage is sufficient. *Moldyn* is the exception, again because of its many parallel streams that put significant pressure on the forward buffer during bursts of activity. Eager forwarding is more sensitive to buffer size, since the forward buffer must contain all produced but unconsumed values.

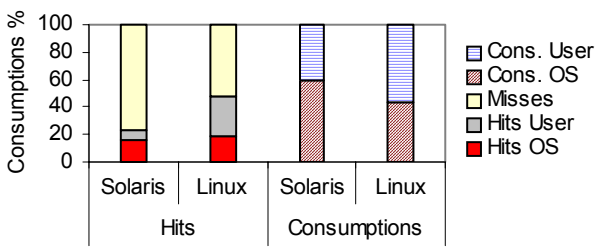


Figure 11. DB2 OS and user behavior.

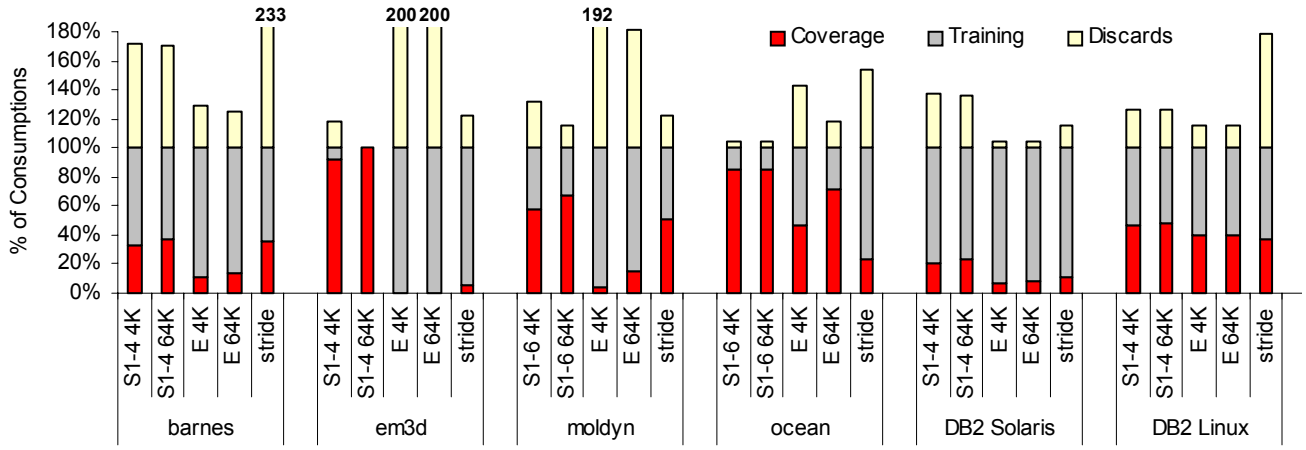


Figure 12. SORDS compared to alternative techniques for eliminating consumption misses. The "S *x-y z*" bars represent SORDS with a head chunk of *x,* a body chunk of *y,* and a forward buffer of *z* bytes. *The "*E *z*" bars refer to eager forwarding with a forward buffer of *z* bytes. The "stride" bar refers to stride-based prediction.

5. Conclusion.

In this paper, we presented SORDS, a novel hardware design for eliminating coherent read misses in distributed shared-memory machines. We demonstrated the phenomenon of temporal correlation—similarity between the production and consumption order—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 presented results that SORDS eliminates 36%-100% of coherent read misses in scientific, 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] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing'91*, pages 176–186, Nov. 1991.

[4] 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.

[5] 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.

[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] 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.

[9] 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.

[10] A. Ka"gi, 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.

[11] 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.

[12] 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*, pages 15–26, June 1998.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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 (PPOPP)*, pages 68–79, July 1995.

[22] 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)*, pages 307–318, Oct. 1998.

[23] 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.

[24] 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.