Temporal Streams in Commercial Server Applications

Thomas F. Wenisch[1 2], Michael Ferdman[2],
Anastasia Ailamaki[2 3], Babak Falsafi[2 3] and Andreas Moshovos[4]

[1]Advanced Computer Architecture Lab (ACAL), University of Michigan
[2]Computer Architecture Lab (CALCM), Carnegie Mellon University
[3]I&C School, Ecole Polytechnique Fédérale de Lausanne
[4]Dept. of ECE, University of Toronto

Abstract

Commercial server applications remain memory bound on modern multiprocessor systems because of their large data footprints, frequent sharing, complex non-strided access patterns, and long chains of dependant misses. To improve memory system performance despite these challenging access patterns, researchers have proposed prefetchers that exploit temporal streams*—recurring sequences of memory accesses. Although prior studies show substantial performance improvement from such schemes, they fail to explain why temporal streams arise; that is, they treat commercial applications as a black box and do not identify the specific behaviors that lead to recurring miss sequences. In this paper, we perform an information-theoretic analysis of miss traces from single-chip and multi-chip multiprocessors to identify recurring temporal streams in web serving, online transaction processing, and decision support workloads. Then, using function names embedded in the application binaries and Solaris kernel, we identify the code modules and behaviors that give rise to temporal streams.*

1. Introduction

Off-chip memory accesses continue to pose a critical performance bottleneck in commercial server applications [1, 3, 8, 22, 25]. Extensive research has shown that widely-deployed stride-based prefetchers provide only limited benefit for many server applications, such as online transaction processing and web serving, because these applications are dominated by pointer-based data structures with complex, non-strided access patterns [8, 22, 25].

To improve performance for these access patterns, over a decade of research has lead to the development of address-correlating prefetchers, which exploit correlation between consecutive memory accesses and are highly-effective for pointer-based structures [6, 7, 8, 9, 13, 14, 15, 19, 21, 25]. Building on this line of research, the latest proposals prefetch extended sequences of memory accesses that recur over the course of program execution [7, 9, 19, 21, 25]. We adopt the terminology of [25] and refer to these recurring memory access sequences as *temporal streams*.

Although prior studies present a variety of hardware designs and demonstrate substantial performance gains from temporal streams, they fail to explain the specific application behaviors that result in the underlying phenomenon of miss sequence repetition. Because of the difficulty in analyzing the behavior of large-scale closed-source applications, these studies have taken a "black-box" approach to all but the simplest of scientific applications.

Furthermore, prior studies have considered only a single system organization, focusing either on uniprocessors [7, 8, 9, 21] or multi-chip distributed-shared-memory systems [25]. As we will show, server applications' off-chip miss behavior changes drastically when all cores are located on a single chip, because coherence activity and contention are captured entirely within that chip. As multi-core system size and diversity increase, users will expect scalable performance whether or not all cores are collocated. Hence, it is critical to understand the impact of multiprocessor organization on memory access behavior.

In this paper, we present a hardware-independent study of temporal streams across single-chip and multi-chip multiprocessors. We identify temporal streams using a previously-proposed information-theoretic analysis that locates repetitive access sequences of arbitrary length without any assumptions about specific prefetching implementation [7]. Furthermore, by inspecting the call stack at each access, we build a profile of specific code modules within our commercial applications and the Solaris kernel that lead to cache misses and temporal streams. Through analysis of user and operating system cache miss traces generated through full-system simulation, we demonstrate:

- **Miss classification across system organizations.** We classify miss behavior using categories similar to the "4 C's" model [12]. Our results confirm prior observations that up to 80% of off-chip misses are coherence-induced in multi-chip multiprocessors [3, 25]; however, a single-chip multiprocessor captures this communication traffic on chip, and off-chip behavior is instead dominated by capacity and I/O-induced misses.
- **Information-theoretic analysis of temporal streams.** Our hardware-independent stream analysis reveals that 40-80% of off-chip misses are part of temporal streams; that streams are typically long, with a median length of eight misses; and that the reuse distance between consecutive occurrences of a stream varies drastically across coherence-induced (100's of intervening misses) and capacity-induced (10,000's of misses) streams.
- **Application-level origins of temporal streams**. Our analysis of the specific functions and modules that give rise to temporal streams reveals a great diversity of application

978-1-4244-2778-9/08/$25.00 ©2008 IEEE

and OS functionality that results in miss repetition. With the exception of frequent bulk memory copies in decision support workloads, no single source accounts for more than 25% of temporal streams. This result demonstrates the success of years of optimization and tuning effort expended on these commercial applications: no obvious, dominant memory bottlenecks remain.

2. Temporal Streams

A temporal stream is a sequence of two or more cache misses that occurs at least twice during program execution [25]. Temporal streams extend the notion of address correlation to sequences rather than pairs of misses. Although the term "temporal stream" was introduced in [25], a wide variety of recent prefetchers rely on the same underlying phenomenon, including hot data stream prefetching [7], the global history buffer [19], the user-level memory thread [21], epoch-based correlation prefetching [8], and last-touch correlated data streaming [9].

Prefetching mechanisms exploit temporal streams by recording miss-address sequences in tables or circular buffers, locating a previously-seen sequence upon a subsequent miss, and then prefetching the recorded addresses. In this study, we analyze temporal streams directly, ignoring the details of individual prefetching mechanisms.

2.1 Motivating Examples

To illustrate why memory accesses in commercial applications often occur in temporal streams, we present two motivating examples taken from actual behaviors observed in our commercial application suite. These examples demonstrate a variety of the characteristics we observe in temporal streams—that streams are long, repeat across cores, evolve over time, and are generally distinguishable based on their initial "head" address. Although these examples are real, we have chosen them for their usefulness in illustrating stream characteristics; they are not the largest sources of miss repetition (see Section 5).

Example one: B+-tree range scans. The B+-tree, one of the critical data structures used in database applications, enables the efficient insertion, removal, and search for database records [4]. A B+-tree maintains a sorted index of records according to a *key* constructed from one or more fields in the record. Each B+-tree node contains a sorted key list with pointers to children, such that the range of keys within a child's subtree is bounded by two adjacent keys in the parent. The leaves of the B+-tree point to tuple identifiers that indicate the location of a corresponding database record. The record(s) for a particular key can be located rapidly through a combination of binary search within each node and traversal from the root to the child containing the key.

A distinguishing feature of the B+-tree is the horizontal pointers that connect sibling leaves of the tree. These horizontal links enable fast in-order tree traversals, and are used to implement range scans. To scan over a set of records from some lower to some upper key, the database engine first locates the lower key. Then, it traverses horizontally along sibling links until it reaches the upper key.

Overlapping range scans result in temporal streams following these sibling links. The first range scan results in a miss sequence for leaves along the bottom of the tree. A second, overlapping range scan will access the same leaves in the same order. As leaves are typically not contiguous in memory, the leaf access sequence cannot be captured by stride prefetchers. Since the B+tree is a shared data structure, the temporal streams arising from scans recur across processors in a multiprocessor system.

Example two: Solaris thread scheduler. One of the key innovations of Solaris 2.3—introduced nearly 15 years ago, but still used in current Solaris releases—is per-processor dispatch queues [18]. In the earliest versions of Solaris, and older UNIX implementations, a single queue maintained pointers to all runnable threads waiting to be scheduled on a processor. To improve Solaris's multiprocessor scalability, this single dispatch queue was split into a real time queue and per-processor dispatch queues, each protected by separate locks, allowing multiple dispatch queues to be accessed or modified concurrently. A complicated set of thread prioritization and affinity algorithms determines to which queue and at which location a thread should be inserted when it becomes runnable.

In most cases, when the thread currently running on a processor blocks or exhausts its time quantum, the processor simply scans its own dispatch queue to identify which thread to run next. However, if its dispatch queue is empty, the processor tries to steal a runnable thread off another processor's queue. It scans other queues looking for the highest priority available thread (in the kernel functions `disp_getwork()` and `disp_getbest()`), removes the thread from the queue (via `dispdeq()`), and finally confirms that no higher priority thread has since become runnable (via `disp_ratify()`). These functions account for an astounding number of misses in commercial applications, as much as 12% of all off-chip misses.

These functions incur many coherence misses to the locks that protect each dispatch queue and the linked lists that comprise the queues. These miss sequences are highly repetitive because all processors scan the dispatch queues in the same order, starting with the real-time priority queue and then advancing through the linked list that connects the dispatch queues. Because the locks remain at fixed addresses, and the queues themselves change little between scans, the misses form temporal streams.

3. Analysis Methodology

We apply an information-theoretic approach to quantify the prevalence and characteristics of temporal streams. Like similar studies of repetition in L1 data accesses [7] and program paths [16], we use the SEQUITUR hierarchical data compression algorithm [10] to identify repetitive sub-sequences within the miss traces.

The SEQUITUR compression algorithm. SEQUITUR constructs a grammar whose production rules correspond to repetitions in its input. Each production rule maps a label to a sequence of symbols and other rule labels. SEQUITUR operates by incrementally extending the grammar's root production rule by one symbol at a time. As each symbol is appended,

Table 1: Application parameters.

Online Transaction Processing (TPC-C on DB2)	
OLTP	100 warehouses (10 GB), 64 clients, 450 MB buffer pool
Decision Support (TPC-H on DB2)	
Qry 1	Scan-dominated, 450 MB buffer pool
Qry 2	Join-dominated, 450 MB buffer pool
Qry 17	Balanced scan-join, 450 MB buffer pool
Web Server	
Apache	16K connections, FastCGI, worker threading model
Zeus	16K connections, FastCGI

the grammar is modified to create new production rules that capture any new repetition the appended symbol creates. SEQUITUR maintains two invariants as the grammar grows. First, no pair of symbols are adjacent more than once in the grammar. Second, every production rule in the grammar (except the root rule) is used more than once. As a result of these invariants, the grammar's rules correspond to distinct repetitive sequences (a.k.a. temporal streams).

System contexts. We analyze read miss traces from three system contexts: (1) off-chip misses in multi-chip multiprocessor (referred to as *"multi-chip"* in the results), (2) off-chip misses in single-chip multi-core system *("single-chip")*, and (3) intra-chip misses in the single-chip system; that is, hits in shared on-chip caches *("intra-chip")*. Our traces include all user and OS read misses.

Our multi-chip model is a 16-node distributed-shared-memory multiprocessor based on the system in [25]. Each node comprises a single processor with split 2-way 64KB L1 I and D caches and a unified 16-way 8MB L2 cache. We chose a 64KB L1 size because this is roughly the upper bound seen in recent commercial systems for one- to two-cycle access times. We chose an 8MB L2 because investigation of miss rates across cache sizes in our database workloads indicate that 8MB is sufficient to capture first-order temporal locality in the applications' data footprints [11]. We model an MSI coherence protocol.

Our single-chip model is a 4-core system with split 64KB L1 I and D caches and a shared 16-way 8MB L2 cache, chosen to approximate a contemporary multi-core. The L1s and shared L2 implement a MOSI coherence protocol based closely on Piranha [2]. The hierarchy is non-inclusive. Other system details do not materially affect traces.

Applications. Our analysis covers three commercial application classes. Table 1 enumerates their configuration details. We run all applications on top of Solaris 8.

We include OLTP and DSS workloads running on *IBM DB2 v8 ESE*. Our OLTP workload is an optimized TPC-C 3.0 toolkit provided by IBM. We select three queries from the TPC-H DSS workload based on the categorization in [20]: one scan-dominated query, one join-dominated query, and one query exhibiting mixed behavior. We evaluate web server behavior with the SPECweb99 benchmark on *Apache HTTP Server v2.0* and *Zeus Web Server v4.3*. We simulate a separate client system and collect memory traces only on the server.

Trace collection. We collect cache miss traces with the *FLEXUS* full-system simulation infrastructure [26]. FLEXUS builds on *Virtutech Simics* [17] and supports both rapid trace-based and detailed cycle-accurate simulation of a variety of uni- and multiprocessor system organizations.

We collect read miss traces in FLEXUS with in-order execution and no memory system stalls. For OLTP and web workloads, we warm main memory for at least 5000 transactions or requests prior to starting traces, and then trace at least 1000 transactions. For DSS queries, we analyze queries 2 and 17 in their entirety, and a trace of over three billion instructions taken from query 1 at steady-state. We have verified that varying trace start location has minimal impact on results.

Code module analysis. With the exception of Apache, we do not have access to the source code of any of the commercial applications in this study. Instead, we exploit the function names embedded in the commercial software and Solaris to tie misses to specific function invocations. By analyzing the call stack at each miss, we can identify an enclosing function with a recognizable purpose. We then group functions into larger categories based on module naming conventions adorning the function names. Fortunately, both Solaris and DB2 frequently prefix function names with an identifier of the module to which it belongs.

Our results are based on a best-effort categorization by iterative refinement of the assignment of functions to categories. In most cases, the purpose of a particular module is apparent from its function names. For Solaris, several public resources document the kernel implementation details (e.g., [18]). Furthermore, with the release of OpenSolaris in 2006, we can examine much of the kernel source code, with the caveat that the code may have changed since the Solaris release we study (Solaris 8). Our categorizations have not been reviewed or endorsed by the organizations that produced these products.

Our FLEXUS tracing infrastructure collects a descriptor for the currently-running thread and a call stack at each miss. We obtain an index of threads, symbol names, and corresponding virtual address ranges using the Solaris kernel debugger, `mdb`, and the `nm` utility.

4. Temporal Stream Characterization

We first report on the quantitative characteristics of temporal streams in our application suite. Like prior studies, these results take a "black box" analysis approach. We "open the box" to report on our code analysis in Section 5.

4.1 Miss Classification

We begin by classifying misses using a categorization based on the "four C's model" [12]. These high-level breakdowns demonstrate the substantial differences in miss behavior across *single-chip* and *multi-chip* contexts. Furthermore, the breakdown allows us to immediately identify miss subsets that are not repetitive, and gives insight into later results.

We classify off-chip misses as: *Coherence* if the cache block was written by another processor since last read at this processor; *I/O Coherence* if the block was written by a DMA transfer or OS-to-user bulk memory copy; *Compulsory* if the corresponding cache block has never previously been accessed; and

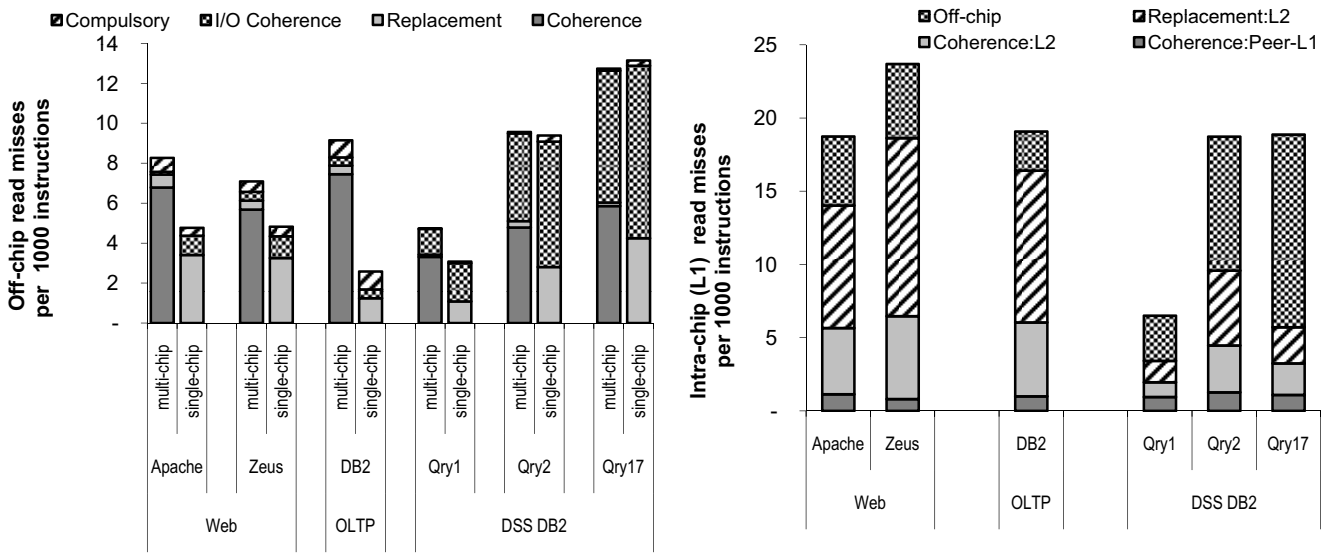

FIGURE 1: Miss classification for off-chip misses (left) and intra-chip misses (right).

Replacement for all remaining misses, which may be capacity or conflict misses (as our L2 caches are 16-way associative, most are capacity misses). We report the results in Figure 1 (left). The vertical axis shows the number of misses per thousand instructions.

In Figure 1 (right), we classify *intra-chip* misses based on their cause and the hierarchy level that supplied the response. Because of the allocation policies of the Piranha intra-chip coherence protocol, coherence misses may be satisfied by a peer L1 or the shared L2. All other L2 hits are the result of L1 replacement misses. We aggregate L2 misses into a single *Off-chip* category, which corresponds to the *single-chip* breakdown in Figure 1 (left).

Discussion. Our measurements corroborate prior conclusions that coherence misses tend to dominate in multi-chip systems with large L2 caches [3]. In the *intra-chip* context, we also observe a substantial fraction of misses from coherence activity between cores despite the small capacity of the L1 caches—roughly one third to one half of all L2 and peer-L1 accesses result from coherence. There is no (non-I/O) off-chip coherence activity in *single-chip*. In the DSS workloads, compulsory misses dominate across contexts, as many data are visited only once.

Coherence misses form temporal streams when locks and read-write shared data structures with repetitive traversal patterns are transferred among cores. Replacement misses comprise streams when a repetitive traversal accesses a data structure that exceeds the cache's capacity or traversals are separated by many intervening accesses. Unlike these two categories, compulsory misses, by definition, do not repeat a prior miss sequence and cannot form temporal streams. A key shortcoming of temporal-stream prefetching is that it cannot address compulsory misses (unlike stride-based prefetchers, which can eliminate compulsory misses).

I/O coherence misses arise from two sources, blocks invalidated by DMA transfers, and blocks written by bulk kernel-to-user buffer copies (the Solaris `default_copyout` family of functions; these functions use special store instructions that do not allocate in the cache hierarchy). About half of the misses arise from each source. In principle, misses in either category could form temporal streams if I/O buffers are reused. In practice, we find that DMA transfers rarely reuse buffers over the time-scales covered by our traces (seconds of execution), whereas kernel-to-user copies aggressively reuse buffers. Nonetheless, we expect that temporal streams provide no advantage over existing prefetching schemes as these accesses are typically strided.

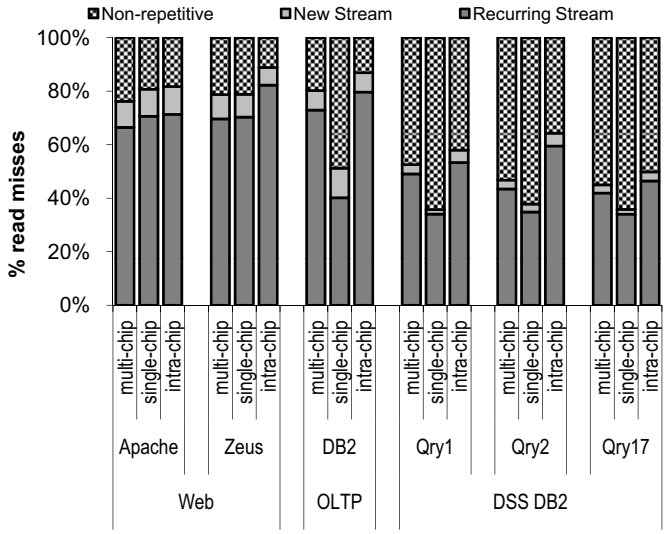

FIGURE 2: Fraction of misses in temporal streams.

4.2 Fraction of misses in temporal streams

Using the SEQUITUR algorithm, we identify temporal streams in our miss traces. In Figure 2, we show the fraction of misses that are part of the first occurrence of a temporal stream (*New stream*), the second or subsequent occurrence of a stream *(Recurring stream)*, or not part of any stream *(Non-repetitive*).

Discussion. Our results confirm prior studies [7, 8, 25]: a substantial miss fraction, 35%-90%, occurs in temporal streams.

In the web applications, temporal streams account for 75-80% of off-chip misses. In the *multi-chip* context, these applications are dominated by coherence activity, nearly all of which is repetitive. In the *single-chip* context, as much as 20% of off-chip accesses arise as a result of I/O coherence. Unlike database workloads, the I/O activity in web applications is repetitive, as it involves many kernel-to-user copies from reused buffers.

In contrast to the web workloads, OLTP exhibits a stark difference in miss repetition across contexts. Coherence activity dominates in the *multi-chip* and *intra-chip* contexts. Past studies attribute the high proportion of coherence activity in OLTP to locking and read-write meta-data structures (e.g., active transaction tables, cursors, log management) [3]. Our code module analysis (see Section 5) finds that OS scheduling and synchronization primitives also contribute substantially. OLTP coherence activity is remarkably repetitive. In the *single-chip* context, there is much less repetition: half are either compulsory or non-repetitive I/O coherence misses, leaving only limited opportunity in the remaining replacement misses for temporal streaming.

The DSS queries show an even larger fraction of non-repetitive compulsory misses. These queries all scan tables that exceed the database's buffer pool capacity. Tuples from the largest table are visited only once. Hence, we observe a substantial proportion of non-repetitive I/O coherence as data is scanned and discarded.

4.3 Strided patterns and temporal streams

Whether a sequence of accesses forms a temporal stream is orthogonal to whether it follows a constant stride. We present a joint breakdown of strided and repetitive miss sequences in Figure 3. The solid segments of the bar correspond to temporal streams *(Repetitive),* while the upper-most and lower-most segments correspond to stride-predictable accesses *(Strided)*.

Discussion. In the case of DSS queries, many accesses, both within and outside of temporal streams, follow strided patterns, particularly within the *single-chip* context. Temporal streams are unlikely to provide much benefit for *single-chip* DSS workloads. Coherence misses in the *multi-chip* and *intra-chip* context are not strided, but still tend to be repetitive, offering some opportunity for synergy between stride and temporal stream prefetching. In the remaining applications, only a small fraction of misses are strided. Moreover, strided patterns and temporal streams are largely disjoint.

4.4 Temporal stream length

Stream length is a critical factor affecting the usefulness of temporal streams. Long streams amortize prefetch costs (e.g., off-chip lookup latency [8, 9, 25]). However, long streams may also imply the need to throttle stream retrieval—a long stream cannot be buffered entirely on chip without displacing other potentially-useful data.

For each application and context, we construct a cumulative distribution of stream lengths weighted by their total contribution to temporal streams. Hence, the cumulative distribution shows the relative importance of long and short streams, and the 50th percentile corresponds to the median stream length.

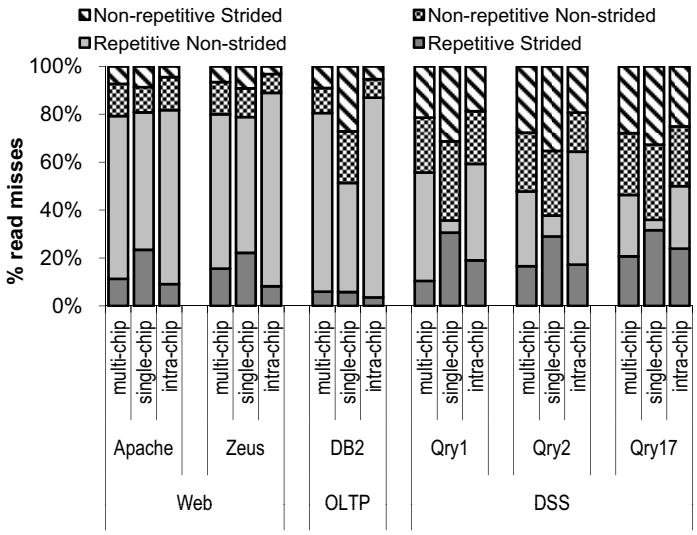

FIGURE 3: Strides and temporal streams.

The cumulative distributions appear in Figure 4 (left). Stream length is plotted logarithmically on the horizontal axis.

Discussion. The most critical observation we draw from these results is that streams are generally quite long—in all cases the median stream length exceeds the fixed prefetch depths of many prior proposals [8, 19, 21]. Overall, the median stream length is eight to ten cache blocks, and thousand-block streams are not unusual.

Furthermore, we observe that there is enormous variation in stream length, from as few as two to many thousands of blocks. The variation in stream length argues against fixed-depth fetch policies—there is no one size that fits all temporal streams, within or across applications.

Stream length variability also complicates stream storage. Assuming a fixed length for streams allows a simple set-associative table to map stream heads to their bodies [21]. Supporting streams with lengths that vary over three orders of magnitude precludes such a simple design.

The DSS applications tend to exhibit longer streams than the other applications, a trend that is particularly apparent in the *single-chip* context. The large step at the right edge arises from streams that access approximately 4KB, which corresponds to the OS page size. Nearly all of the DSS streams longer than 512 bytes arise from bulk memory copies and are easily captured by simple stride predictors.

4.5 Stream reuse distance

The distance between two occurrences of a temporal stream provides insight into the time-scale over which data structure traversals repeat and is a critical indicator of the storage requirements for streams.

To divorce the notion of stream reuse distance from time or performance, we measure reuse distance as the number of misses between two stream occurrences. As we study multiprocessors, the two occurrences may not be on the same processor. We count the number of intervening misses on the first processor, as this measure corresponds directly to the storage

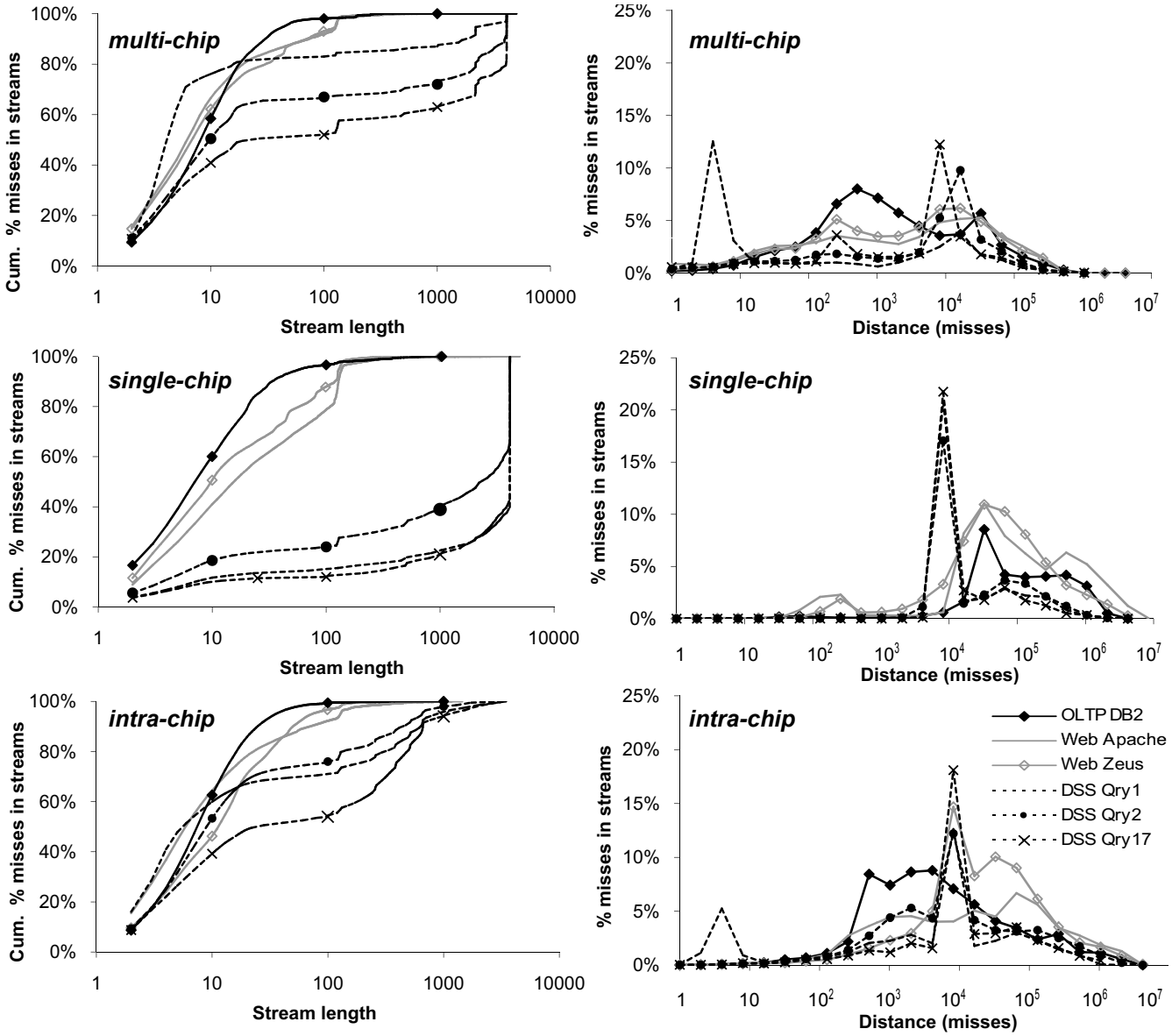

FIGURE 4: Temporal stream length (left) and reuse distance (right).

required to remember the stream in a log of all misses at the first processor [25].

We report the probability density function of reuse distance in Figure 4 (right). We truncate the distribution at ten million misses as such distances correspond to substantial real execution time (approaching one second of wall-clock time) and are unlikely to be exploited by prefetching.

Discussion. Coherence misses typically have far shorter reuse distances than replacement misses. A replacement miss implies that the corresponding block has been displaced by other cache allocations. As L2 capacities are large, typically thousands of L2 misses occur between a block's allocation and eviction. Blocks accessed more frequently than every ten thousand misses will not be evicted from L2, which places a soft lower bound on replacement miss reuse distance. In contrast, coherence-miss reuse distances are determined by the distance between production and consumption of shared data, and can be arbitrarily short (near zero in cases of contention).

The critical difference between the reuse behavior of these two miss classes results in the substantial difference in the center of mass of the reuse distributions across contexts. As the *multi-chip* context is dominated by coherence activity, reuse distances tend to be short, and nearly all stream reuse distances are below ~200,000 misses. *Intra-chip* temporal streams also frequently exhibit low reuse distances: in addition to coherence activity, intra-chip replacement streams may recur frequently because L1 capacity is limited (1000 lines).

In contrast, in the *single-chip* context, reuse distances tend to be much longer, with the longest distances approaching ten million misses. In *single-chip*, replacement misses dominate. Replacement misses have high reuse distances because of the substantial L2 capacity. The key implication of this result is that *single-chip* systems will require larger storage to track the same fraction of temporal streams as *multi-chip* systems.

Several peaks in the DSS reuse distance distribution arise from specific query behaviors, and are not representative of average

Table 2: Miss Categories.

Cross-application categories	
Bulk memory copies	Kernel and user memory copy functions, such as `memcpy`, `bcopy`, `__align_cpy_1`, and `default_copyout`. This last function is particularly notable as it copies the results of I/O arriving via DMA from kernel to user buffers, and figures prominently in the overall contribution of memory copies.
System call implementation	Kernel functionality invoked on behalf of user threads within system call interfaces. The most frequent system calls all involve I/O, with `poll`, `open`, `read`, `write`, and `stat` dominating.
Kernel task scheduler	Functions that perform kernel thread prioritization and dispatching, as briefly described in Section 2.1. Extensive discussion of the operation of the Solaris scheduler appears in [18].
Kernel MMU and trap handlers	Functions (other than system calls) that are entered via the kernel's trap vector table. The most frequent traps are the `instruction_access_MMU_miss` and `data_access_MMU_miss` traps, which fill virtual-to-physical translations into the on-chip MMU from software caches and page tables. Register window management traps, where a window of eight integer registers are read from or written to a software stack, also contribute substantially. SPARC trap handling is described in detail in [24].
Kernel synchronization primitives	Solaris-supplied mutex and condition variable primitives. This category also includes functions that manage the linked lists of threads waiting on a mutex or condition variable.
Kernel - other activity	All remaining functionality that can be definitively tied to the Solaris kernel but is not part of a category that stands out as a major contributor to memory access behavior in any application or context. Many of the functions in this category deal with various forms of kernel memory and resource management.
Web-specific categories	
Kernel STREAMS	Implementation of stream based I/O, such as `stdin` and `stdout`. Consists largely of functions that move pointers to strings or portions of strings among thread-safe queues.
Kernel IP packet assembly	Functions that divide data written to sockets into individual IP packets.
Web server worker threads	All activity within either Apache or Zeus—perhaps surprisingly, relatively little of the overall SpecWeb activity occurs in web server code: most operations are performed by the OS on behalf of the web server.
CGI - perl input processing	A single function, `Perl_sv_gets`, which parses the requests passed from the web server to perl.
CGI - perl execution engine	The `Perl_pp_*` functions, which implement the primitive operations that make up perl's control flow graph. Examples include `Perl_pp_const`, `Perl_pp_print`, and `Perl_pp_return`.
CGI - perl other	Other functionality of perl that is not readily identifiable.
DB2-specific categories	
Kernel block device driver	A small number of functions that manage I/O to block devices, such as disks.
DB2 index, page, and tuple accesses	Functions in the `sqli`, `sqld`, and `sqlpg` modules of DB2. The `sqli` module accounts for most of this category, and includes functions which manipulate and traverse indices. The example we describe in Section 2.1 arises in this category. The `sqld` module includes functions that access individual database rows, such as `sqldRowUpdate` or `sqldRowFetch`. The `sqlpg` module includes functions that manipulate entire buffer pool pages, such as flushing pages to disk or calculating page checksums.
DB2 SQL request control	The DB2 modules `sqlrr` and `sqlra` which manipulate context information for a particular database transaction/request (e.g., state of database cursors).
DB2 interprocess communication	Functions which pass data between the DB2 server and client processes.
DB2 SQL runtime interpreter	The `sqlri` module. The functions in this module implement primitive operations that appear in a parsed database execution plan, analogous to the `Perl_pp_*` functions of the perl interpreter.
DB2 - other activity	Any other DB2 functionality whose overall contribution to misses is small or where function names do not allow us to determine their functionality.

behavior. Query one exhibits a reuse distance peak below ten misses. These misses are the result of the lock contention. All three queries exhibit peaks just under 10,000 misses in all contexts. These peaks arise from the bulk kernel-to-user copies of data arriving from disk.

5. Code Module Analysis

Our previous results have demonstrated that the vast majority of misses in commercial applications are repetitive. However, the black-box approach of our trace analyses does not tell us why temporal streams arise. In this section, we provide quantitative evidence that ties temporal streams to specific application behaviors.

Miss categories. We briefly describe each miss category in Table 2. For easier comparison across applications, we divide categories into those that apply to all three application classes and those specific to each. In the following tables, we report each category's contribution to the overall miss breakdown, and the fraction of misses within a category that are part of temporal streams.

5.1 Web applications

One of the most surprising results of our web server analysis is that the http server software itself accounts for only a tiny fraction of memory activity—about 3% of off-chip misses and 5% of L1 misses. Instead, activity is dominated by the communication between perl scripts that generate dynamic page content, the web server software, and kernel interfaces that send http replies to the network.

Table 3: Temporal stream origins in Web applications.

	multi-chip		single-chip		intra-chip	
	% misses	in streams	% misses	in streams	% misses	in streams
Uncategorized / Unknown	8.4%	7.9%	7.1%	4.9%	7.6%	6.5%
Cross-application categories						
Bulk memory copies	12.6%	7.4%	20.3%	14.2%	6.3%	3.7%
System call implementation	11.0%	8.8%	18.0%	15.1%	22.8%	20.5%
Kernel task scheduler	8.8%	6.7%	3.5%	3.0%	2.6%	2.3%
Kernel MMU & trap handlers	3.7%	2.8%	4.4%	3.4%	13.6%	12.0%
Kernel synchronization primitives	7.5%	7.1%	2.1%	1.8%	2.5%	2.4%
Kernel - other activity	4.6%	3.4%	4.7%	3.8%	5.2%	4.7%
Web-specific categories						
Kernel STREAMS subsystem	16.3%	12.9%	7.8%	6.2%	12.5%	10.3%
Kernel IP packet assembly	8.4%	6.9%	4.1%	3.4%	7.3%	6.7%
Web server worker thread pool	2.5%	1.8%	3.2%	2.3%	5.4%	4.4%
CGI - perl input processing	1.9%	1.9%	3.9%	3.9%	0.8%	0.8%
CGI - perl execution engine	7.6%	5.2%	11.1%	9.2%	7.1%	5.3%
CGI - perl other activity	6.5%	4.9%	10.0%	8.8%	6.3%	4.9%
Overall% in streams		77.7%		80.0%		84.5%

SPECWeb specifies that some client web requests be satisfied by static pages served by the web server, and some by dynamic pages that are generated via the web server's Common Gateway Interface (CGI). The majority of web server activity arises from the creation and delivery of these dynamic pages. Much of the tuning of a SPECWeb implementation centers on careful orchestration of the CGI interaction. Our workloads employ the FastCGI interface, where a pool of perl processes await client requests, and the web server dispatches requests to an available process as requests arrive [5]. FastCGI drastically improves performance over traditional CGI by avoiding process creation overheads on each request. The web server and perl communicate using the standard I/O streams, which are implemented in Solaris's STREAMS sub-system.

The interprocess communication implemented in STREAMS results in many temporal streams. The STREAMS code breaks data passed via read and write system calls into individual messages, which pass through a series of modules that may perform various processing steps, such as packetization or header assembly [23]. The kernel STREAMS code manages synchronization and message passing among modules. Both the locks and the manipulation of message pointers within these queues result in highly-repetitive access sequences—about 80% occur in temporal streams.

The perl processes that generate the dynamic web content also exhibit many temporal streams, albeit somewhat less repetitive than kernel activities. The function that parses the input to the dynamic content generation scripts, `Perl_sv_gets`, is the single most repetitive function we have identified, with just under 99% of its misses repeating a prior temporal stream. The implementations of individual perl statements and expressions also result in about 75% repetitive miss behavior. We conjecture that misses in the perl execution engine repeat because each of the thousands of requests invoke the same script, and hence traverse the same control flow graph. Thus, accesses to control flow graph representation within the execution engine are likely to repeat.

Within OS activity, the largest miss contributor is system calls, in particular, `poll`. This system call is used by the web server to accept incoming connections and pass them to a worker thread for processing.

Because of the many http and perl threads created to keep up with incoming requests, the OS scheduler and synchronization primitives result in many temporal streams (see example two in Section 2.1). The repetitive synchronization activity arises mostly from Solaris condition variables, where manipulation of queues of sleeping threads are the likely source of temporal streams.

Finally, we note that bulk memory copies account for a substantial fraction of misses, particularly in the *single-chip* context. More than half of these copies are repetitive. We have found that the repetitive copies arise because of reused I/O buffers for incoming network data.

Although perl's behavior may be an artifact of SPECWeb, many of the other temporal streams we observe are inherent in web serving. For example, the STREAMS and IP packet assembly code must be exercised by any web server on Solaris. Scheduling and synchronization activity will also occur in any busy web server, as current web servers maintain hundreds to thousands of threads for servicing incoming connections. Hence, we believe our results broadly represent web serving beyond the specific workloads studied here.

5.2 Online transaction processing

The most significant miss sources in OLTP are the index, tuple, and page accesses issued to the database buffer pool, accounting for between one sixth and one fifth of all misses. Within this category, index accesses are the largest contributor (see example one in Section 2.1).

Table 4: Temporal stream origins in OLTP (DB2).

	multi-chip		single-chip		intra-chip	
	% misses	% in streams	% misses	% in streams	% misses	% in streams
Uncategorized / Unknown	11.4%	9.7%	10.9%	5.5%	13.6%	11.3%
Cross-application categories						
Bulk memory copies	4.4%	1.5%	10.8%	1.5%	2.9%	1.6%
System call implementation	7.4%	2.1%	19.1%	2.1%	5.9%	3.1%
Kernel task scheduler	9.1%	9.1%	0.5%	0.5%	3.2%	3.2%
Kernel MMU & trap handlers	11.2%	10.0%	9.3%	7.2%	16.9%	15.5%
Kernel synchronization primitives	5.7%	5.7%	0.2%	0.2%	2.4%	2.4%
Kernel - other activity	6.7%	5.0%	8.9%	6.0%	5.4%	4.5%
DB2-specific categories						
Kernel block device driver	2.3%	2.2%	0.7%	0.6%	3.3%	3.3%
DB2 index, page & tuple accesses	16.6%	13.1%	23.4%	16.8%	15.1%	13.1%
DB2 SQL request control	9.5%	9.1%	2.5%	2.3%	7.9%	7.6%
DB2 interprocess communication	3.5%	3.4%	1.4%	1.2%	7.6%	7.4%
DB2 SQL runtime interpreter	3.7%	3.5%	2.6%	2.4%	4.4%	4.2%
DB2 - other activity	8.4%	5.1%	9.8%	4.7%	11.2%	9.3%
Overall% in streams		79.5%		51.0%		86.5%

The higher layers of the DB2 execution engine, which build transaction and query processing capabilities on top of the storage manager interface, are more repetitive than the buffer pool management functionality. The transaction management, execution plan interpreter, and interprocess communication components all exhibit 90% repetition. These observations support prior conjecture that the coherence activity in databases comes from meta-data—data structures that do not reside on disk or within the buffer pool, such as locks, transaction tables, or the query plans manipulated by the optimizer and runtime interpreter [3]. We expect these repetitive activities to arise in any transaction-oriented database workload, even if accesses to the data managed within the database are not repetitive.

The Solaris scheduler and synchronization primitives contribute substantially to the miss profiles where coherence activity plays a significant role (*multi-chip*, *intra-chip*), but are absent in the *single-chip* system. This profile is characteristic of coherence activity to a relatively small number of addresses—cache lines migrate between processors, but are never evicted due to capacity constraints.

Within the Solaris kernel, we see that trap handlers, and in particular traps filling translations into the SPARC MMU, result in a large number of temporal streams. A single page table lookup can require several main memory accesses, and, since many mappings are loaded repetitively into the MMU, the misses incurred during the translation process repeat.

5.3 Decision support

The obvious characteristic of DSS miss breakdowns is the prominence of bulk memory copies. Half or more of all memory access activity arises from these copies. The copies tend to be of power-of-two sizes, with page-sized 4KB copies most frequent. Unlike in the web applications, where bulk copies are often repetitive, copies in DSS do not reuse buffers and are generally non-repetitive. It is the prevalence of these copies that renders temporal streams all but useless for DSS workloads—bulk copies are either bandwidth bound, or already addressed by far simpler stride prefetchers.

The second most important miss contributors in DSS are index and tuple accesses, as in OLTP workloads. However, unlike OLTP, off-chip misses in these functions are not repetitive—the DSS workloads typically scan over data only once. We do observe some *intra-chip* repetition. We attribute this to the nested-loop joins in queries 2 and 17, which loop over portions of a database table that exceed L1, but do not exceed L2 capacity. In general, we conclude that DSS workloads exhibit few temporal streams because most data are visited only once.

6. Conclusion

To our knowledge, our study is the first to identify specific application/OS behaviors that lead to the recurring temporal streams underlying a variety of prefetching techniques. Temporal streams are inherent in many programming idioms; one-half to three-quarters of misses occur in streams. Streams are frequently tens, hundreds, and even thousands of misses long. Furthermore, we observe drastic differences between the off-chip access patterns of single- and multi-chip multiprocessors, which have critical implications on the design and storage requirements of prefetching in general, and temporal streams in particular.

The memory access activity in commercial server applications is spread over a wide variety of functionality, with no particular activity standing out. We believe this flat distribution is a testament to the years of investment in optimizing these applications—any code that produced disproportionate misses has been stamped out. Furthermore, the lack of outlying "bad" behavior illustrates why it is so challenging to optimize these applications further, and why we must resort to mechanisms as

Table 5: Temporal stream origins in DSS (DB2).

	multi-chip		single-chip		intra-chip	
	% misses	% in streams	% misses	% in streams	% misses	% in streams
Uncategorized / Unknown	3.6%	3.0%	1.8%	0.6%	7.3%	2.0%
Cross-application categories						
Bulk memory copies	46.3%	16.2%	66.7%	28.7%	35.5%	16.3%
System call implementation	0.6%	0.6%	0.4%	0.4%	1.9%	1.8%
Kernel task scheduler	2.9%	2.4%	0.1%	0.1%	0.7%	0.7%
Kernel MMU & trap handlers	1.5%	1.0%	1.4%	0.8%	5.7%	4.8%
Kernel - other activity	13.1%	9.2%	6.0%	2.6%	12.8%	9.5%
DB2-specific categories						
Kernel block device driver	6.3%	6.0%	0.6%	0.5%	8.7%	8.3%
DB2 index, page & tuple accesses	18.0%	2.7%	19.2%	1.5%	14.6%	6.1%
DB2 SQL runtime interpreter	1.9%	1.8%	0.6%	0.5%	5.8%	5.6%
DB2 - other activity	5.2%	3.2%	3.3%	1.7%	6.9%	4.1%
Overall% in streams		46.1%		37.4%		59.2%

complex as temporal stream prefetching to achieve performance gains.

Acknowledgements

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

References

[1] 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*, Sept. 1999.

[2] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture base on single-chip multiprocessing. In *Proc. of the 27th Intn'l Symp. on Computer Architecture*, June 2000.

[3] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proc. of the 25th Intn'l Symp. on Computer Architecture*, June 1998.

[4] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3), Sep. 1972.

[5] M. Brown. FastCGI: A high-performance web server interface. http://its.mak.ac.ug/fastcgi/doc/fastcgi-whitepaper/fastcgi.htm.

[6] M. J. Charney and A. P. Reeves. Generalized correlation-based hardware prefetching. Technical Report EE-CEG-95-1, School of Electrical Engineering, Cornell University, Feb. 1995.

[7] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, June 2002.

[8] Y. Chou. Low-cost epoch-based correlation prefetching for commercial applications. In *Proc. of the 40th Intn'l Symp. on Microarchitecture*, Dec. 2007.

[9] M. Ferdman and B. Falsafi. Last-touch correlated data streaming. In *IEEE Intn'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2007.

[10] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7, 1997.

[11] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *3rd Biennial Conf. on Innovative Data Systems Research (CIDR)*, Jan. 2007.

[12] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. on Computers*, C-38(12), Dec. 1989.

[13] Z. Hu, M. Martonosi, and S. Kaxiras. TCP: Tag correlating prefetchers. In *Proc. of the 9th Symp. on High-Performance Computer Architecture*, 2003.

[14] D. Joseph and D. Grunwald. Prefetching using Markov Predictors. In *24th Intn'l Symp. on Computer Architecture*, Jun. 1997.

[15] A.-C. Lai and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proc. of the 28th Intn'l Symp. on Computer Architecture*, July 2001.

[16] J. R. Larus. Whole program paths. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, 1999.

[17] 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.

[18] J. Mauro and R. McDougall. *Solaris Internals*. Sun Microsystems Press, 2001.

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

[20] M. Shao, A. Ailamaki, and B. Falsafi. DBmbench: Fast and accurate database workload representation on modern microarchitecture. In *Proc. of the 15th IBM Center for Advanced Studies Conference*, Oct. 2005.

[21] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proc. of the 29th Annual Intn'l Symp. on Computer Architecture*, May 2002.

[22] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *Proc. of the 33rd Annual Intn'l Symp. on Computer Architecture*, June 2006.

[23] Sun Microsystems. *STREAMS Programming Guide*. Sun Microsystems Press, 2005.

[24] D. L. Weaver and T. Germond. *The SPARC Architecture Manual, Version 9*. SPARC International, 1994.

[25] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *Proc. of the 32nd Annual Intn'l Symp. on Computer Architecture*, June 2005.

[26] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4), Jul.-Aug. 2006.