Appears in Proceedings of the 41st Annual IEEE/ACM Int'l Symposium on Microarchitecture, 2008

Temporal Instruction Fetch Streaming

Michael Ferdman[1 3], Thomas F. Wenisch[2],
Anastasia Ailamaki[1 3], Babak Falsafi[3] and Andreas Moshovos[4]
[1]Computer Architecture Lab (CALCM), Carnegie Mellon University, Pittsburgh, PA, USA
[2]Advanced Computer Architecture Lab (ACAL), University of Michigan, Ann Arbor, MI, USA
[3]Parallel Systems Architecture Lab (PARSA), Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
[4]Department of ECE, University of Toronto, Toronto, Canada

Abstract—**L1 instruction-cache misses pose a critical performance bottleneck in commercial server workloads. Cache access latency constraints preclude L1 instruction caches large enough to capture the application, library, and OS instruction working sets of these workloads. To cope with capacity constraints, researchers have proposed instruction prefetchers that use branch predictors to explore future control flow. However, such prefetchers suffer from several fundamental flaws: their lookahead is limited by branch prediction bandwidth, their accuracy suffers from geometrically-compounding branch misprediction probability, and they are ignorant of the cache contents, frequently predicting blocks already present in L1. Hence, L1 instruction misses remain a bottleneck.**

We propose Temporal Instruction Fetch Streaming (TIFS)—a mechanism for prefetching temporally-correlated instruction streams from lower-level caches. Rather than explore a program's control flow graph, TIFS predicts future instruction-cache misses directly, through recording and replaying recurring L1 instruction miss sequences. In this paper, we first present an information-theoretic offline trace analysis of instruction-miss repetition to show that 94% of L1 instruction misses occur in long, recurring sequences. Then, we describe a practical mechanism to record these recurring sequences in the L2 cache and leverage them for instruction-cache prefetching. Our TIFS design requires less than 5% storage overhead over the baseline L2 cache and improves performance by 11% on average and 24% at best in a suite of commercial server workloads.

Keywords: instruction streaming, fetch-directed, caching, prefetching

1 INTRODUCTION

L1 instruction-cache misses pose a critical performance bottleneck in commercial server workloads [1, 11, 12, 15, 17, 31, 33]. Commercial server workloads span multiple binaries, shared libraries, and operating system code and exhibit large instruction working sets that overwhelm L1 instruction caches [12, 17, 31]. Although total on-chip cache capacity is growing, low access latency and high fetch bandwidth requirements preclude enlarging L1 instruction caches to fit commercial application working sets. As a result, commercial server workloads incur instruction-related delays in the memory system that account for as much as 25%-40% of execution time [12].

Instruction-cache misses are particularly expensive because they contribute to system performance loss in multiple ways. First, instruction fetch stalls directly prevent a core from making forward progress because instructions are not available for dispatch. Unlike data accesses, which can be overlapped through out-of-order execution, instruction fetch is on the critical path of program execution, and nearly the entire latency of an L1 instruction miss is exposed. Second, due to lower average ROB occupancy, instruction-fetch stalls reduce the number of load instructions simultaneously present in the ROB, thereby indirectly reducing performance through a decrease in memory-level parallelism.

To improve fetch unit performance despite limited instruction cache capacity, researchers have proposed a variety of hardware prefetching schemes. The widely-implemented next-line instruction prefetcher [29], despite its simplicity, substantially reduces L1 instruction-cache misses in commercial server workloads [17], but is only effective for straight-line code. More advanced instruction prefetchers can predict discontinuous control flow by using the branch predictor to explore a program's control flow graph ahead of the fetch unit [5, 24, 31, 32], prefetching the instruction-cache blocks encountered along the predicted path into the L1 cache.

Although state-of-the-art hardware prefetchers eliminate many instruction misses, they suffer from four fundamental flaws that limit their coverage and lookahead. First, the branch predictors at the core of these prefetchers predict only a few branches per cycle, limiting instruction prefetch lookahead. Second, branch predictors issue predictions at basic block rather than cache block granularity. Substantial long-range prediction efficiency is lost when exploring control flow within an instruction-cache block and around inner loops. Third, misprediction probability compounds geometrically, limiting prefetch accuracy. Finally, branch predictors are ignorant of cache contents, predicting all future control flow. As a result, the vast majority of predicted blocks are already present in L1, incurring prediction overhead or requiring filtering and lookup mechanisms.

Rather than exploring a program's control flow graph, we propose to predict the L1 instruction-cache miss sequences directly. Our key observation, inspired by recent studies of data prefetching [7, 8, 21, 30, 37], is that repetitive control flow graph traversals lead to recurring instruction-cache miss sequences, much like repetitive data-structure traversals lead to recurring data-cache miss sequences. We show that nearly all instruction-cache misses, including control transfers to non-contiguous instructions, occur as part of such recurring sequences, which we call *temporal instruction streams*. In contrast to the instruction streams in [22, 23, 27], which comprise a sequence of contiguous basic blocks, temporal

instruction streams comprise recurring sequences of instruction-cache blocks that span discontinuities (taken branches) within the instruction address sequence. We record temporal instruction streams as they are encountered, and replay them to predict misses, eliminating the inefficiencies that arise through reconstructing the sequences indirectly with branch predictions. Our hardware design, *Temporal Instruction Fetch Streaming (TIFS)*, embeds the storage required to track temporal streams in the L2 tag and data arrays, imposing less than 5% storage overhead. Through a combination of information-theoretic analysis of repetition in instruction-cache miss sequences, trace-driven hardware simulation, and cycle-accurate modeling of a CMP system running commercial server workloads, we demonstrate:

- **Recurring streams of instruction misses.** We demonstrate that 94% of instruction-cache misses repeat a prior miss stream, with a median stream length of over 20 cache blocks.
- **Lookahead limits of branch-predictor-directed prefetch.** We find that, for approximately one-third of instruction-cache misses, more than 16 non-inner-loop branches must be correctly predicted to achieve a four-cache-miss lookahead. TIFS lookahead is not limited by the branch predictor.
- **I-streaming effectiveness exceeding the state-of-the-art.** TIFS operates on instruction-cache misses rather than the basic-block sequence, improving accuracy, bandwidth efficiency, and lookahead of instruction-cache prefetching. TIFS improves commercial workload performance by 5% on average and 14% at best over a system with fetch-directed prefetching [24] and by 11% on average and 24% at best over one with next-line instruction prefetching.

The remainder of this paper is organized as follows. We demonstrate the need for improvement over next-line instruction prefetching in Section 2. We analyze examples of code patterns that can lead to improved performance with TIFS in Section 3. In Section 4, we present an information-theoretic opportunity study to characterize repetition in instruction-miss streams. Based on this study, we develop a hardware design for TIFS in Section 5. We evaluate the effectiveness of TIFS on real workloads in Section 6. In Section 7, we discuss related work, and in Section 8 we conclude.

2 The Need for Improved Instruction Prefetching

To motivate the need for further improvements to instruction prefetching, we begin with a preliminary performance study of the sensitivity of commercial server workloads to instruction prefetch accuracy beyond what is obtained with the next-line instruction prefetchers in current processors. To measure workload performance sensitivity, we simulate a simple probabilistic prefetching model and vary its coverage. For each L1 instruction miss (also missed by the next-line instruction prefetcher), if the requested block is available on chip, we determine randomly (based on the desired prefetch coverage) if the request should be treated as a prefetch hit. Such hits are instantly filled into the L1 cache. If the block is not available on chip (i.e., this is the first time the instruction is fetched), the miss proceeds normally. A probability of 100% approximates a perfect and timely instruction prefetcher. We include three commercial server workload classes in our study: online transaction processing (OLTP), decision support (DSS), and web serving (Web). We provide complete details of our workloads in Section 4.1 and our simulation infrastructure and performance measurement methodology in Section 6.1.

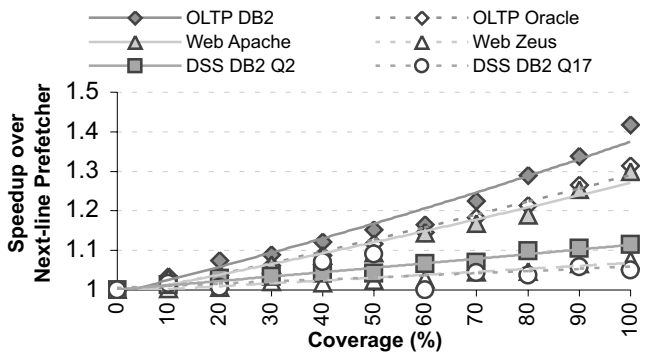

Figure 1. Opportunity. Application performance improvement as an increasing fraction of L1 instruction misses are eliminated.

We plot results of this study in Figure 1. As these results are derived from sampled simulations and are subject to sample variability (see Section 6.1), we plot linear regressions to illustrate their trend. Our results demonstrate that both online transaction processing (OLTP) workloads and Web-Apache are highly sensitive to instruction prefetching—their performance improves by over 30% with a perfect instruction prefetching mechanism. In decision support system (DSS) queries, and in Web-Zeus, the instruction working set is smaller, yielding less sensitivity. Overall, it is clear that augmenting next-line instruction prefetching with a more capable mechanism can have substantial performance impact.

3 Drawbacks of Existing Instruction Prefetchers

To provide better intuition for the drawbacks of existing instruction prefetch mechanisms, we perform an analysis of a few common-case scenarios where a processor equipped with next-line and branch-predictor–directed prefetchers may experience stalls due to instruction-cache misses. We perform this investigation by extracting call-stack traces at instruction-cache misses of interest, and then use application binary and kernel symbol tables to map the call stack to function names. The mapping enables us to locate and analyze the source code (for Apache and Solaris) at these misses.

The specific examples we describe here do not by themselves account for a majority of processor front-end stalls (though frequent, they still represent a small fraction of the overall miss profile). Rather, each example illustrates a common-case behavior that arises in many program contexts. Analysis of these cases offers a deeper look at the drawbacks of existing techniques.

3.1 Unpredictable sequential fetch

Surprisingly, both the next-line and the branch-predictor–directed prefetchers sometimes fail to hide instruction-cache miss latency even in functions with sequential instruction flow. For a next-line instruction prefetcher, each fetch discontinuity

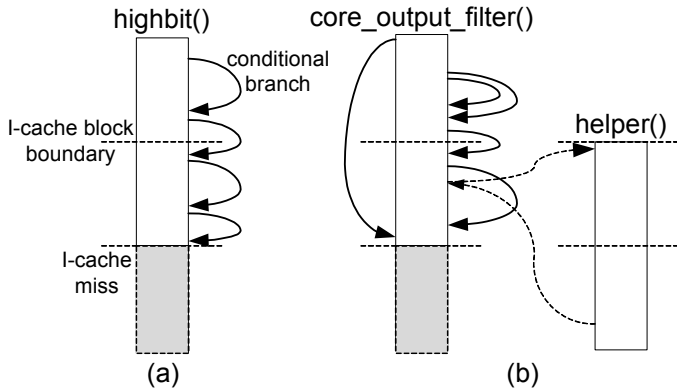

Figure 2. Instruction fetch example. Examples of cache misses incurred by next-line and branch-predictor–directed prefetching with unpredictable sequential fetch (a) and re-convergent hammocks (b).

can lead to such stalls. Upon a control transfer at the discontinuity, the fetch unit retrieves the target instruction, and the next-line instruction prefetcher initiates a prefetch of the next few cache blocks. Often, execution of instructions in the first cache block completes before prefetches of the subsequent blocks complete, and some instruction-fetch latency is exposed. It is this scenario that motivates the discontinuity prefetcher [31]. The discontinuity prefetcher and more sophisticated fetch-directed prefetchers [5, 24, 31, 32] have the potential to eliminate these stalls if they provide sufficient lookahead to predict the critical discontinuity. However, in branch-heavy code, these prefetchers cannot run far enough ahead of the fetch unit to hide all instruction-fetch latency.

The Solaris kernel scheduling code is critical to commercial server workload performance. An example of a frequently-invoked helper function from the scheduler code is `highbit()`, which calculates the index of the most-significant asserted bit in a double-word through a series of mask and arithmetic operations organized in a sequence of branch hammocks (shown in Figure 2 (a)). Although `highbit()` spans only a few consecutive instruction-cache blocks, and execution always proceeds through all blocks, the last block incurs an instruction-cache miss in a large fraction of invocations, even with a fetch-directed prefetcher. The complex control flow in the scheduling code preceding the invocation of `highbit()` and the dense branch hammocks within it exceed the prefetchers' lookahead capabilities. Moreover, because the scheduling code begins with synchronization instructions that drain the processor's reorder buffer, the instruction-cache miss in `highbit()` is fully exposed and incurs a large penalty.

The sequence of instruction-cache misses experienced while executing `highbit()` is always the same, and there are only a limited number of call sites from which this function is invoked. It is therefore possible for a hardware mechanism to record the sequences of misses that lead up to and include the `highbit()` function. Whenever a sequence of cache misses that includes the `highbit()` function is detected within the scheduler code, the previously-recorded sequence can be used to predict all of the needed cache blocks before any of the instructions of `highbit()` are executed. Therefore, unlike next-line and branch-predictor–directed prefetchers whose lookahead is limited by the number of branches that must be traversed, temporal instruction streaming can properly prefetch the function's instruction-cache misses long before their use, eliminating all of `highbit()`'s instruction-cache stalls.

3.2 Re-convergent hammocks

Predictor-directed prefetchers are not effective in code regions that exhibit series of if-then-else statements that re-converge in the program's control flow (hammocks). The data-dependent branches in such regions cause frequent branch mispredictions, which thwart branch-predictor-directed prefetch methods. The `core_output_filter()` function found in the Apache 2.x web server illustrates this behavior.

Because of its central nature, `core_output_filter()` is invoked from numerous call sites and contains a multitude of highly data-dependent code paths and loops. The complexity of `core_output_filter()` results in a large instruction working set, over 2.5KB for this function alone, in addition to a variety of helpers called from within its body. The "if" statements in `core_output_filter()` serve as an example of conditional branches that frequently cause processor stalls at the re-convergence points. Because branch outcomes are data-dependent and change with each invocation of the function, branch predictors cannot look beyond the loops, conditional paths, and sub-routine calls within branch hammocks in this code. Branch mispredictions impede correct prefetches by a fetch-directed mechanism; predicted prefetch paths are discarded and the fetch-directed prefetcher restarts its control-flow exploration each time a branch resolves incorrectly. Hence, despite a fetch-directed prefetcher, execution of `core_output_filter()` exhibits frequent instruction-cache misses at the hammock re-converge points (Figure 2 (b)).

The hammocks in `core_output_filter()` eventually re-converge on the same path. While the specific branch outcomes are unpredictable, having a global view of the previously encountered cache-miss sequence enables prediction of future misses at the hammock re-convergence points. In fact, because the re-convergence points are present in all previously recorded miss-address sequences, correct predictions can be made by using any one of the sequences. Furthermore, the complex, data-dependent computations that are present in `core_output_filter()` are slow, allowing sufficient time for a temporal instruction-stream prefetcher to fully hide all instruction-miss latency.

4 Opportunity for Temporal Instruction Fetch Streaming

Our analysis of the performance losses from instruction-cache misses demonstrates the need to address on-chip instruction access stalls such as those described in Section 3. The key idea of TIFS is to record streams of L1 instruction misses, predict when these streams recur, and then stream instructions from lower-level caches prior to explicit fetch requests. In this section, we describe an information-theoretic analysis of repetition in instruction-cache misses to determine the potential to eliminate misses with TIFS.

TABLE I. COMMERCIAL SERVER WORKLOAD PARAMETERS.

Online Transaction Processing (TPC-C)	
Oracle	Oracle 10g Enterprise Database Server, 100 warehouses (10 GB), 16 clients, 1.4 GB SGA
DB2	IBM DB2 v8 ESE, 100 warehouses (10 GB), 64 clients, 2 GB buffer pool
Decision Support (TPC-H on DB2 v8 ESE)	
Qry 2	Join-dominated, 480 MB buffer pool
Qry 17	Balanced scan-join, 480 MB buffer pool
Web Server (SPECweb99)	
Apache	Apache HTTP Server 2.0, 4K connections, FastCGI, worker threading model
Zeus	Zeus Web Server v4.3, 4K connections, FastCGI

4.1 Opportunity study methodology

TIFS relies on the repetition in L1 instruction miss sequences. To determine the opportunity to eliminate misses with TIFS, we study the amount of repetition in traces of the L1 instruction-cache misses of commercial server workloads. Table I enumerates the details of our workload suite. We collect traces with the *FLEXUS* full-system simulation infrastructure [38]. We trace the DSS queries in their entirety, and four billion instructions (one billion per core) from the remaining workloads. Our traces include all user and OS instruction fetches. We study a 4-core single-chip CMP with split 64 KB 2-way set-associative L1 caches and 64-byte blocks, and a next-line instruction prefetcher that continually prefetches two cache blocks ahead of the fetch unit (other model parameters do not affect trace collection). A "miss" is an instruction fetch that can't be satisfied by the L1 instruction cache or next-line instruction prefetcher.

We apply an information-theoretic approach to quantify miss-stream repetition. Like similar studies of repetitive streams in L1 data accesses [6], off-chip data misses [35, 36], and program paths [16], we use the SEQUITUR [10] hierarchical data compression algorithm to identify repetitive subsequences within the miss-address traces. SEQUITUR produces a grammar whose production rules correspond to repetitions in its input. Hence, grammar production rules correspond to recurring miss-address streams.

4.2 Miss repetition

We apply SEQUITUR to quantify the maximum opportunity for TIFS. Figure 3 shows a categorization of instruction misses based on whether or not they repeat a prior temporal miss stream. *Non-repetitive* misses do not occur as part of a repetitive stream (i.e., they never occur twice with the same preceding or succeeding miss address). On the first occurrence of a repetitive stream, we categorize the misses as *New*. On subsequent occurrences, the first miss in the stream is categorized as *Head* and the remaining misses as *Opportunity*. We distinguish *Head* and *New* from *Opportunity* because our hardware mechanisms are not be able to eliminate these misses due to training and prediction-trigger mechanisms. An example temporal miss stream and categorization appear in Figure 4.

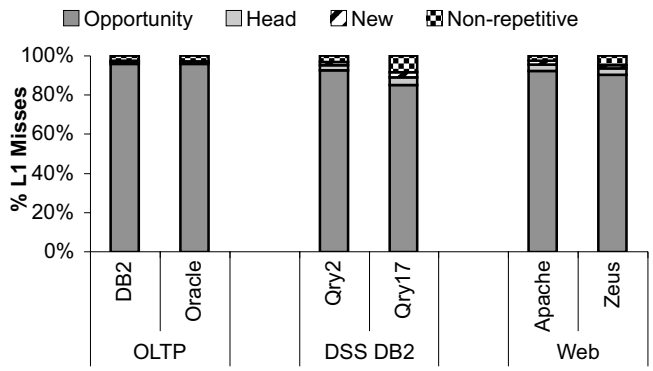

Figure 3. Opportunity. The opportunity to eliminate misses through TIFS as revealed by our information-theoretic analysis of miss-address repetition.

Our SEQUITUR analysis reveals that nearly all instruction misses, on average 94%, are part of a recurring stream of instruction-cache misses that may be eliminated by TIFS. Opportunity is highest in OLTP workloads, which have the largest instruction working sets among our workloads. This offline trace analysis demonstrates the highly repetitive nature of instruction execution—the same execution paths incur the same temporal miss streams over and over.

4.3 Stream length

The length of repetitive access sequences directly translates to prefetch coverage that a memory streaming mechanism can achieve by predicting that sequence. The ability to follow arbitrarily long streams by periodically requesting additional addresses distinguishes temporal streaming [35, 36, 37] from prefetching approaches that only retrieve a constant number of blocks in response to a miss (e.g., [7, 21, 30]). Without this ability, the system would incur one miss for each group of prefetched blocks. Furthermore, long streams improve prefetch timeliness—after a stream head is detected, a simple rate-matching mechanism enables the prefetcher to retrieve instruction-cache blocks ahead of the fetch unit for the rest of the stream [37].

Figure 5 shows the cumulative distribution of recurring stream lengths as identified by SEQUITUR. The stream lengths shown in Figure 5 reflect only non-sequential instruction block references; all sequential misses are removed from the traces to simulate the effect of a perfect next-line instruction prefetcher (stream lengths roughly double when sequential blocks are not removed). The SEQUITUR analysis confirms that instruction-miss streams are long—for example, in OLTP-Oracle, the median length is 80 perfectly-repeating non-sequential blocks. In comparison, prior work reports that off-chip temporal data-miss streams exhibit a median length of 8 to 10 blocks [36].

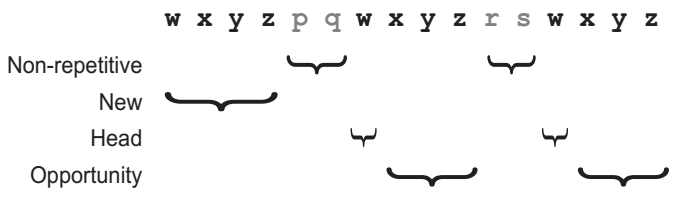

Figure 4. Example of opportunity accounting.

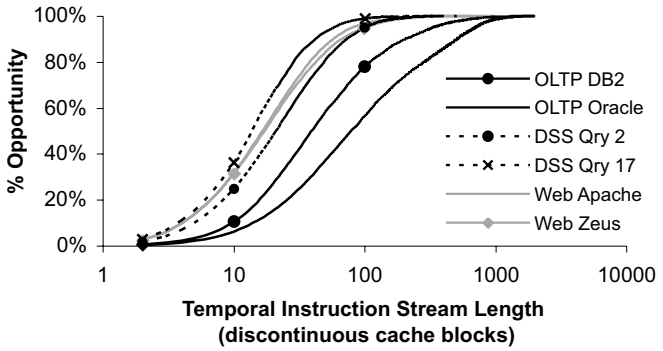

Figure 5. Stream Length. Cumulative distribution of stream lengths (as identified by SEQUITUR) for recurring temporal instruction streams.

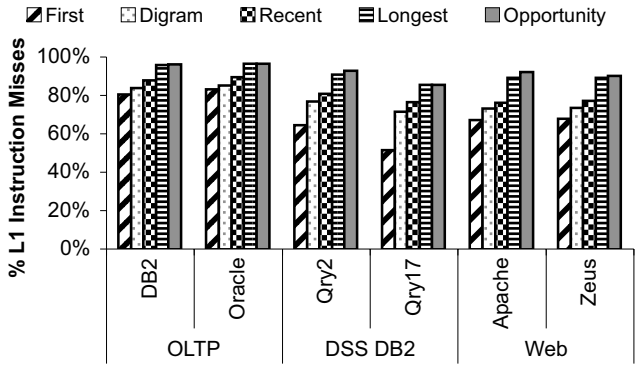

Figure 6. Stream Lookup Heuristics. Fraction of misses that can be eliminated by each stream lookup heuristic.

4.4 Stream lookup heuristics

The SEQUITUR opportunity study identifies all recurring streams present in the miss sequence. Under some circumstances, such as divergent control flow, multiple distinct repetitive streams begin with the same head address. SEQUITUR always identifies the best match among these alternatives—it measures the upper bound of repetition, and hence represents perfect stream lookup. In contrast, a practical hardware streaming mechanism must use a heuristic to determine which previously-seen stream among all distinct prior alternatives should be fetched upon a subsequent miss to the head address.

To guide the design of our streaming hardware, we consider several possible heuristics for choosing the correct stream. The results of our comparison are shown in Figure 6. The *First* policy associates a head address to the first stream (earliest in program order) that SEQUITUR identifies as starting with that address. *Digram* uses the second address, in addition to the head address, to identify which stream to follow. The *Recent* heuristic continually re-associates a miss address with the most-recently encountered stream headed by that address. Finally, *Longest* associates a miss address with the longest stream to ever occur following that address. *Opportunity* is the bound on repetition established in Figure 3.

Our study reveals that the *Longest* policy is most effective. Unfortunately, we are unaware of a practical implementation of this heuristic, because hardware mechanisms can only discover stream length after the fact, by counting successful prefetches. Hence, like past designs [21, 37], TIFS uses the next-best option, the *Recent* heuristic, as it is easy to implement in hardware while still yielding high predictor coverage.

5 DESIGN

We separate our presentation of the TIFS hardware design into two sections. For clarity of presentation, in Section 5.1 we present the logical structure and operation of TIFS while omitting the details of the hardware implementation. In Section 5.2 we present the physical organization of these structures in TIFS, placing emphasis on a low hardware-cost implementation. Where possible, we virtualize the TIFS storage structures [4]—that is, predictor storage is allocated within the L2 cache rather than in dedicated SRAM. Virtualization allows software to control and vary structure size allocation and partitioning on a per-application or per-core basis, up to and including disabling TIFS entirely to reclaim L2 capacity in the event it is not effective for a particular workload.

5.1 Logical hardware overview

Figure 7 illustrates the logical structure and operation of TIFS. The basic operation of TIFS mirrors prior address-correlated prefetching proposals that target data accesses [8, 21, 37]. TIFS adds three logical structures to the chip: a set of Streamed Value Buffers (SVBs), one per L1-I cache; a set of Instruction Miss Logs (IMLs), one per L1-I cache; and a single Index Table. As in prior stream buffer proposals [14], the SVB holds streamed blocks that have not yet been accessed and maintains the information necessary to continue fetching a stream. Each IML structure maintains a log of all instruction-cache miss addresses from the corresponding L1-I cache. The shared Index Table, used to locate streams, maintains a pointer from each address to the most recent global occurrence of that address in any IML. Because the Index Table is shared among all IMLs, an Index Table pointer is not limited to a particular IML, enabling SVBs to locate and follow streams logged by other cores.

5.1.1 Logging miss-address streams

As instructions retire, L1-I fetch-miss addresses are logged in an IML. Addresses are logged only at instruction retirement to reduce the impact of out-of-order and wrong-path execution. To facilitate logging, ROB entries are marked if the instruction misses in the L1-I cache during the fetch stage. IMLs record physical addresses to avoid the need for address translation during prefetch. As an address is appended to an IML, an entry in the Index Table is created/updated with a pointer to indicate the location inside the IML at which the address was logged.

5.1.2 Instruction streaming

The SVB tracks the current state of a stream flowing into its L1 cache and serves as the temporary storage for streamed blocks that have not yet been accessed by the processor core. On every L1-I cache miss, the core checks if the accessed block is present in the SVB. The SVB check is performed after the L1 access to avoid circuit complexity that might impact the fetch

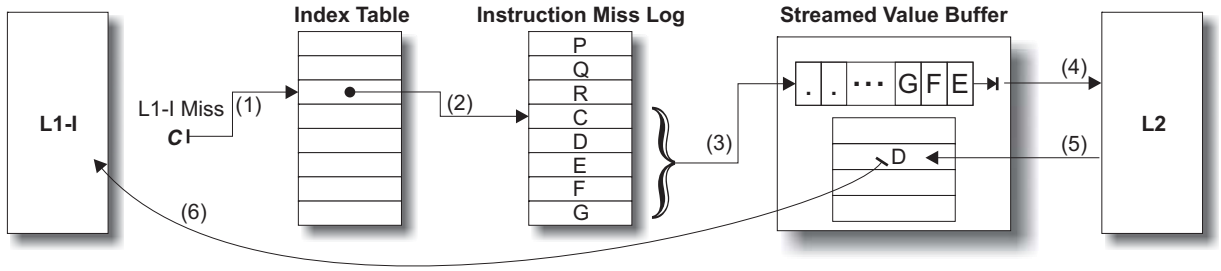

Figure 7. TIFS Operation. An L1-I miss to address *C* consults the index table (1), which points to an IML entry (2). The stream following *C* is read from the IML and sent to the SVB (3). The SVB requests the blocks in the stream from L2 (4), which returns the contents (5). Later, on a subsequent L1-I miss to *D*, the SVB returns the contents to the L1-I (6).

critical path. Upon an SVB hit, the block is immediately allocated in the L1 cache and the stream's next block is prefetched. SVB hits are also logged in the appropriate IML, ensuring that the block will be fetched during the next stream traversal.

If the requested block is not found in the SVB, a new stream is allocated to prefetch subsequent blocks. The SVB consults the Index Table to find a pointer to the IML location where the miss address was most recently observed. If found, a stream pointer in the SVB is set to the corresponding IML location, and the SVB begins reading the IML and prefetching blocks according to the logged stream. As further blocks in the stream are accessed, the SVB continues advancing the stream pointer and prefetching additional blocks.

5.1.3 End of stream detection

In prior stream buffers [14, 28], in the stride predictors of commercial systems [13], and in the simplest TIFS design, stream buffers make no effort to guess where a stream might end. However, as shown in Figure 5, streams vary drastically in length, from two to thousands of blocks. Stopping too early may result in lost coverage, while stopping too late may result in erroneous prefetches and wasted bandwidth.

The SVB terminates a stream by remembering where the stream ended the last time it was followed. As addresses are logged in an IML, an additional bit is stored to indicate if the address is recorded as the result of an SVB hit. When following a stream, the SVB immediately fetches blocks if this bit is set (indicating a correct prior prediction), but pauses after fetching the first block where the bit is cleared (indicating a potential stream end). If the block is accessed as a result of an L1-I miss, the SVB resumes stream fetch.

5.2 Hardware implementation

Figure 8 depicts our proposed implementation of TIFS in the context of a four-core chip multiprocessor. The physical implementation differs from the logical description of Section 5.1 in three respects. First, the SVB hardware simultaneously maintains several stream pointers and small queues of upcoming prefetch addresses to allow for multiple parallel in-progress streams. Multiple streams may arise because of traps, context switches, or other interruptions to the usual control flow. Second, although our logical design calls for a dedicated storage structure for each IML, we implement TIFS with minimal hardware overhead by virtualizing IMLs and storing their contents inside the L2 data array as proposed in [4].

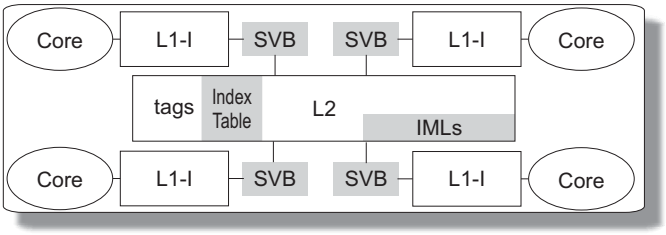

Figure 8. Temporal Instruction Fetch Streaming in a 4-core CMP.

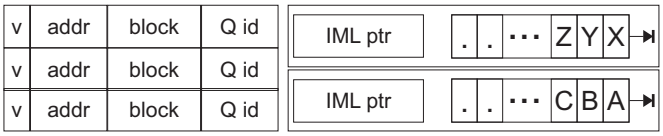

Figure 9. Streamed Value Buffer details.

Finally, rather than implementing the Index Table as a separate structure, we embed the IML pointers in the L2 cache as additional bits in the tag array.

5.2.1 Streamed value buffers

Figure 9 depicts the anatomy of the SVB. Our SVB design is adapted from [35, 37]. The SVB contains a small fully-associative buffer for temporary storage of streamed blocks. Each entry includes a valid bit, tag, and contents of the instruction block. Upon an SVB hit, the block is transferred to the L1 I-cache and the SVB entry is freed. If the SVB becomes full, entries are replaced using an LRU policy. To manage prefetching for in-flight streams, the SVB maintains FIFO queues of addresses awaiting prefetch and pointers into the IML indicating the continuation of each active stream. As a FIFO drains, further addresses are read from the IML and the IML pointer is advanced. In this fashion, the SVB can follow a stream of arbitrary length.

The SVB logic ensures that streams are followed (i.e., blocks are requested) sufficiently early to hide their retrieval latency. The SVB matches the rate that blocks are fetched to the rate at which the L1 cache consumes these blocks. The SVB attempts to maintain a constant number of streamed-but-not-yet-accessed blocks for each active stream. Across workloads, we find that four blocks per stream is sufficient. Hence, the SVB capacity can remain small (2 KB per core).

The key purpose of the SVB is to avoid polluting the cache hierarchy if a retrieved stream is never used. By making multiple blocks from a stream available simultaneously in a

fully-associative buffer, the SVB also serves as a window to mitigate small (e.g., a few cache blocks) deviations in the order of stream accesses that may arise due to minor data-dependent control flow irregularities.

5.2.2 Instruction miss logs

TIFS requires an allocation of on-chip storage to record cache-miss addresses. We virtualize the IML storage structures, placing cache-miss addresses directly in the L2 cache. From the cache's point of view, IML entries exist in a private region of the physical address space. IML reads and writes are issued to L2 as accesses to this address region, and are performed at cache-block granularity (i.e., 64-byte cache blocks containing twelve recorded miss addresses are read or written at a time).

The Index Table is realized as a part of the L2 tag array, as an additional IML pointer field per tag; the width of this pointer field places an upper bound on virtual IML size. Stream lookups are performed on every L1-I fetch miss. Co-locating the Index Table with the L2 tags provides a "free" lookup mechanism performed in parallel with the L2 access. When an access hits in the L2 tag array, the IML pointer is returned to the corresponding SVB to initiate stream fetch. The stream fetch proceeds in parallel with the L2 data-array access.

Each time an address is appended to an IML (at instruction retirement), the IML pointer stored as part of the L2 tag must be updated. IML pointer updates are queued separately from other cache requests and are scheduled to the tag pipelines with the lowest priority. If back-pressure results in full queues, updates are discarded. Because of the time delay between cache access and instruction retirement, IML pointer updates occasionally fail to find a matching address in the tag arrays; such updates are silently dropped.

6 Evaluation

We evaluate TIFS in the context of a four-core chip multiprocessor. We study a system with aggressive, out-of-order cores and decoupled front-end instruction fetch units [25] which partially hide instruction-fetch latency in commercial server workloads. We expect the impact of TIFS to be even higher in simpler cores without these mechanisms, where all instruction-fetch latency is exposed.

6.1 Methodology

We study TIFS using a combination of trace-based and cycle-accurate simulation. We model a four-core CMP with private L1 instruction and data caches and a shared, unified L2 cache. The L2 is divided into 16 banks with independently-scheduled tag and data pipelines. The L1 caches are connected to L2 banks with a full, non-blocking crossbar. Each bank's data pipeline may initiate a new access at most once every four cycles. The minimum total L2 hit latency is 20 cycles, but accesses may take longer due to bank conflicts or queueing delays. A total of at most 64 L2 accesses, L1 peer-to-peer transfers, and off-chip misses may be in flight at any time. We configure our core model to resemble the Intel Core 2 microarchitecture, and base main-memory latency and bandwidth parameters on the IBM Power 6. Further configuration details appear in Table II.

TABLE II. SYSTEM PARAMETERS.

Cores	UltraSPARC III ISA, Four 4.0 GHz OoO cores 4-wide dispatch / retirement 96-entry ROB, LSQ	**Main Memory**	3 GB total memory 28.4 GB/s peak bw 45ns access latency 64-byte transfer unit
L1-D Cache	64KB 2-way 2-cycle load-to-use 3 ports, 32 MSHRs 64-byte lines	**L2 Shared Cache**	8MB 16-way 20-cycle access latency 16 banks, 64 MSHRs 64-byte lines
I-Fetch Unit	64KB 2-way L1-I cache 16-entry pre-dispatch queue Hybrid branch predictor 16K gShare & 16K bimodal	**Stride Prefetch**	Next-line I-prefetcher 32-entry D-stream buffer Up to 16 distinct strides

Our base system includes a next-line instruction prefetcher in each instruction fetch unit, and a stride prefetcher at L2 for retrieving data from off chip. We account TIFS hits only in excess of those provided by the next-line instruction prefetcher (i.e., next-line hits are counted as L1 hits; we do not include these in TIFS coverage even if TIFS could also prefetch the blocks.)

6.2 Lookahead limitations of fetch-directed prefetching

A key advantage of TIFS is that it provides far greater lookahead than other instruction prefetchers. TIFS lookahead is bounded by temporal instruction stream length, and each IML lookup provides twelve prefetch addresses. In contrast, fetch-directed prefetching lookahead is limited by the number of branches that can be predicted accurately between instruction-cache misses.

To assess the lookahead limits of fetch-directed prefetching, we analyzed the number of branches that must be predicted correctly to prefetch four instruction-cache misses ahead of the fetch unit. We exclude backwards branches in inner-most loops, as a simple filter could detect such loops and prefetch along the fall-through path. The results of our analysis appear in Figure 10. For roughly a quarter of all instruction-cache misses, a fetch-directed prefetcher must traverse more than 16 non-inner-loop branches to achieve a lookahead of just four misses. With a branch predictor that can make only one or two predictions per cycle, fetch-directed prefetchers fall far short of the lookahead possible with TIFS.

6.3 Hardware requirements

Although they are large relative to L1 caches, application instruction working sets are small relative to L2, which allows TIFS to capture temporal instruction streams in IML storage that is a small fraction of L2. Figure 11 shows the TIFS predictor coverage as a function of IML storage capacity (for this analysis, we assume a perfect, dedicated Index Table). Our result confirms that a relatively small number of hot execution traces account for nearly all execution. For peak coverage, we find that each core's IML must record roughly 8K instruction block addresses, for an aggregate storage cost of 156 KB (8K entries / core, 38 physical address bits + 1 hit bit / entry, 4 cores;

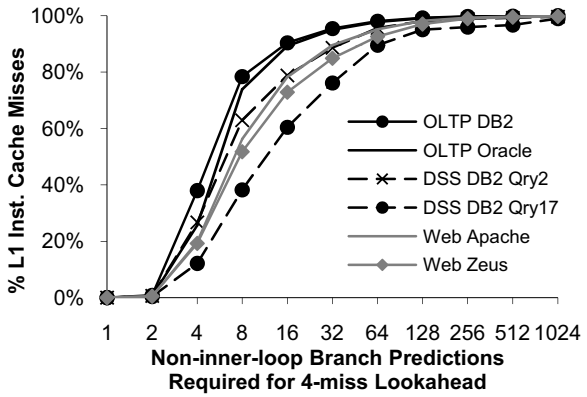

Figure 10. Limited lookahead of fetch-directed prefetching. Cumulative distribution of the number of correct non-inner-loop branch predictions a branch-predictor-directed prefetcher must make to predict the next four instruction-cache misses.

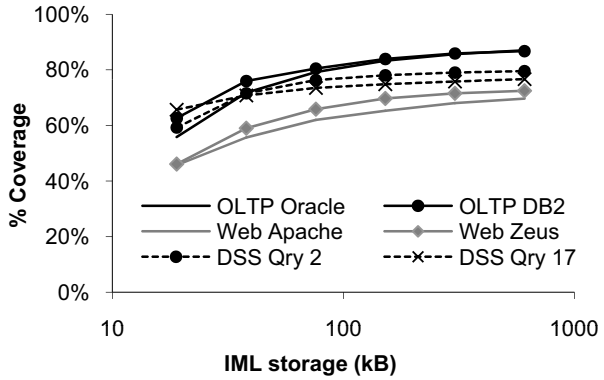

Figure 11. IML capacity requirements.

in total, less than 2% of the L2 cache capacity). Total IML capacity requirements scale with core count (~40KB per core).

The Index Table stores a mapping from addresses to IML pointers and is embedded in the L2 tag array. Each L2 tag is extended with a 15-bit IML pointer, for an aggregate storage cost of 240 KB (<3% increase in L2 storage). Storage costs for a dedicated Index Table are comparable, as a dedicated table requires fewer entries, but each entry must contain a tag as well as the IML pointer. Index Table storage requirements are independent of the number of cores, and instead scale with the number of L2 tags.

Finally, each core's SVB requires only minimal hardware: a 2KB buffer for instruction-cache blocks and several small FIFOs and registers for prefetch addresses and IML pointers.

6.4 Traffic overhead

TIFS increases L2 traffic in two ways: (1) it prefetches blocks which are never accessed by the CPU, which we call *discards*; and (2) it reads and writes IML blocks. Correctly prefetched blocks replace misses that occur in the base system, thus they cause no increase in traffic.

We first consider discards. A discard occurs whenever TIFS prefetches a block, and that block is replaced in the SVB by another prefetch (i.e., the block is not fetched). Discards occur most frequently when TIFS prefetches past the end of a repetitive stream; our end-of-stream detection mechanism mitigates this (Section 5.1). Figure 12 (left) shows TIFS coverage and discards for a predictor with previously-cited IML sizes. The coverage and discards are normalized to L1 fetch misses.

TIFS traffic overhead also includes reads and writes of the virtualized IMLs. In the best case, if all instruction-cache misses are streamed by TIFS, the IML read and write traffic are each 1/12th (<8.5%) of the fetch traffic. Discards and short streams increase these overheads. In Figure 12 (right), we show the total L2 traffic overhead as a fraction of the base L2 traffic (reads, fetches, and writebacks). TIFS increases L2 traffic on average by 13%.

6.5 Performance evaluation

We compare TIFS performance to state-of-the-art prefetching mechanisms and investigate the performance cost of virtualizing TIFS hardware structures in Figure 13. All bars are normalized to the performance of the next-line instruction prefetcher included in our base system design. *FDIP* is a state-of-the-art fetch-directed instruction prefetcher [24]. We make several minor adjustments to the original FDIP design to tune it for our workloads. First, the FDIP authors propose several optimizations to minimize FDIP's impact on L1 tag-port bandwidth. To simplify our experiments, we omit these optimizations, but provide FDIP unlimited tag bandwidth (i.e., no impact on fetch). Second, the authors recommend allowing FDIP to proceed roughly 30 instructions ahead of the fetch unit. To maximize FDIP effectiveness in our environment, we increase this depth to 96 instructions, but at most 6 branches. Finally, for fairness, we assume a fully-associative rather than FIFO prefetch buffer, as the SVB is fully-associative. In our environment, these changes strictly improve FDIP performance. The three *TIFS* bars represent our TIFS design with unbounded IMLs, 156 KB of dedicated IML storage, and 156 KB of virtualized IML storage in the L2 data array, respectively. Finally, *Perfect* indicates the upper bound achieved by a perfect instruction streaming mechanism.

The TIFS performance improvements match the coverages reported in Figure 12 and the sensitivity reported in Figure 1. TIFS outperforms FDIP on all studied workloads except DSS

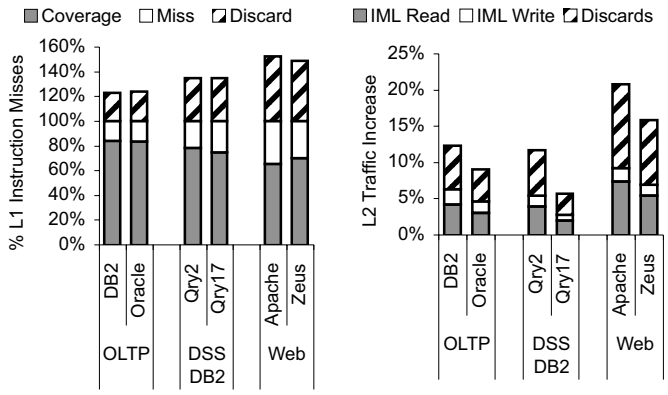

Figure 12. Coverage, Discards, and Traffic Overhead.

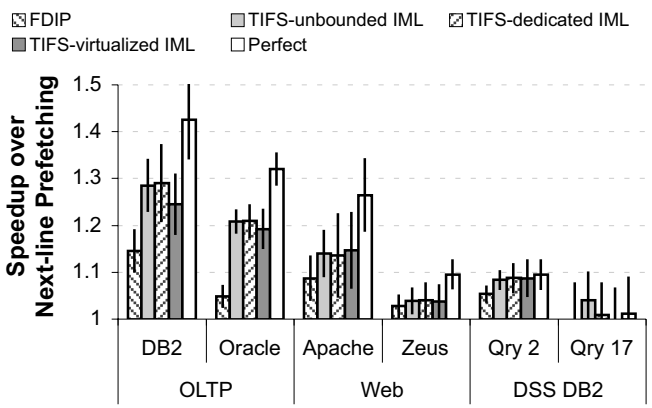

Figure 13. TIFS Performance Comparison.

DB2 Qry17, where instruction prefetching provides negligible benefit. TIFS provides the largest benefit in the OLTP workloads, which have the largest instruction working sets. Limiting IML capacity to 156 KB has no effect on performance. Virtualizing the IML marginally reduces performance in OLTP-DB2 because of a slight increase in L2 bank contention.

7 Related Work

The importance of instruction prefetching has long been recognized by computer architects. Sequential instruction (next-line) prefetching was implemented in the IBM System 360 Model 91 in the late 1960's [2], and prefetching into caches was analyzed by Smith in the late 1970's [29]. Although computer architectures and workloads have evolved drastically since then, simple next-line instruction prefetching remains critical to the performance of modern commercial server workloads [17]. More recent work on instruction-stream prefetching generalizes the notion of the next-line instruction prefetcher to arbitrary-length sequences of contiguous basic blocks [23, 27].

A greater challenge lies in prefetching at *fetch discontinuities*—interruptions in the sequential instruction-fetch sequence from procedure calls, taken branches, and traps. The discontinuity predictor [31] maintains a table of such fetch discontinuities. As a next-line instruction prefetcher explores ahead of the fetch unit, it consults the discontinuity table with each block address and, upon a match, prefetches both the sequential and discontinuous paths. Although it is simple and requires minimal hardware, the discontinuity predictor can bridge only a single fetch discontinuity; recursive lookups to explore additional paths result in an exponential growth in the number of prefetched blocks.

Rather than use a dedicated table, branch-predictor-directed prefetchers [5, 24, 32] reuse existing branch predictors to explore the future control flow of a program and identify cache blocks for prefetch. Pre-execution and speculative threading mechanisms [18, 34, 40] and Runahead Execution [20] similarly use branch prediction to enable speculative control-flow exploration and prefetching. TIFS differs from these approaches in that it records and replays instruction-cache miss sequences rather than branch outcomes, predicting directly the anticipated future sequence of fetch discontinuities, which improves prefetch timeliness. Furthermore, unlike TIFS, neither next-line nor branch-predictor-directed mechanisms can ensure timely prefetch of sequential blocks that follow a discontinuity. TIFS is not affected by the behavior of unpredictable data-dependent branches, relying only on actual instruction-cache miss sequences that occurred in the past for future predictions. We compared the performance of TIFS with Fetch Directed Instruction Prefetching (FDIP) [24] in Section 6.5. We quantified the relationship between branch prediction bandwidth and prefetch lookahead in Section 6.2.

Software approaches to reduce instruction-fetch bottlenecks relocate rarely-executed code and increase the number of sequentially executed instructions [22], improve cache line reuse through gang-scheduling or data batching [12, 39], add compiler-inserted instruction prefetches [19], or perform call graph prefetching [3]. These approaches target the same bottlenecks as TIFS, but all require detailed software analysis and modification, and some are application-specific.

The TIFS design is based on recent proposals for address-correlated prefetch of recurring temporal data streams [7, 8, 21, 30, 37]. These prefetchers target primarily off-chip data references and require large off-chip tables to capture prefetcher meta-data. As instruction working sets are orders-of-magnitude smaller than data working sets, TIFS meta-data fits on chip (see Section 6.3). To further reduce the hardware overhead of TIFS, we employ a variant of *predictor virtualization* [4], a technique for storing prefetcher meta-data in the L2 cache (see Section 5).

The term *temporal stream*, introduced in [37], refers to extended sequences of data references that recur over the course of program execution. Similar repetition in the sequence of basic blocks visited by a program has been reported by Larus [16] and underlies trace scheduling [9] and trace caches [26]. Temporal instruction streams differ from previously-defined instruction streams [22, 23, 27] in two key respects: (1) temporal instruction streams are defined at cache-block rather than basic-block granularity, and (2) temporal instruction streams span fetch discontinuities. TIFS efficiently records and exploits long recurring temporal instruction streams.

8 Conclusions

L1 instruction misses are a critical performance bottleneck in commercial server workloads. In this work, we made the observation that instruction-cache misses repeat in long recurring streams. We employed this observation to construct Temporal Instruction Fetch Streaming (TIFS)—a mechanism for prefetching temporally-correlated instruction streams from lower-level caches. Unlike prior fetch-directed prefetchers, which explore a program's control flow graph, TIFS predicts future instruction-cache misses directly, through recording and replaying recurring L1 instruction-miss sequences. Through full-system simulation of commercial server workloads, we demonstrated that TIFS improves performance by 11% on average and 24% at best.

Acknowledgements

The authors would like to thank Brian T. Gold, Nikolaos Hardavellas, Stephen Somogyi, and 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? *The VLDB Journal*, Sept. 1999.

[2] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM system/360 model 91: Machine philosophy and instruction handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967.

[3] M. Annavaram, J. M. Patel, and E. S. Davidson. Call graph prefetching for database applications. *ACM Transactions on Computer Systems*, 21(4):412–444, 2003.

[4] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi. Predictor virtualization. *Proc. 13th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[5] I-C. K. Chen, C.-C. Lee, and T. N. Mudge. Instruction prefetching using branch prediction information. *Int'l Conference on Computer Design*, 1997.

[6] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. *Proc. SIGPLAN '01 Conference on Programming Language Design and Implementation*, 2001.

[7] Y. Chou. Low-cost epoch-based correlation prefetching for commercial applications. *40th Annual Int'l Symposium on Microarchitecture*, Dec. 2007.

[8] M. Ferdman and B. Falsafi. Last-touch correlated data streaming. *Int'l Symposium on Performance Analysis of Systems and Software*, 2007.

[9] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, Jul. 1981.

[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. *3rd Biennial Conference on Innovative Data Systems Research*, 2007.

[12] S. Harizopoulos and A. Ailamaki. Steps towards cache-resident transaction processing. *Proc. 30th Int'l Conference on Very Large Data Bases (VLDB'04)*, Aug. 2004.

[13] R. Hedge. Optimizing application performance on Intel Core microarchitecture using hardware-implemented prefetchers. http://www.intel.com/cd/ids/developer/asmo-na/eng/298229.htm.

[14] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *Proc. 17th Annual Int'l Symposium on Computer Architecture*, May 1990.

[15] 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. *Proc. 25th Annual Int'l Symposium on Computer Architecture*, Jun. 1998.

[16] J. R. Larus. Whole program paths. *Proc. SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[17] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. *Proc. 25th Annual Int'l Symposium on Computer Architecture*, Jun. 1998.

[18] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. *Proc. 28th Annual Int'l Symposium on Computer Architecture*, Jun. 2001.

[19] C.-K. Luk and T. C. Mowry. Cooperative prefetching: compiler and hardware support for effective instruction prefetching in modern processors. *Proc. 31st Annual Int'l Symposium on Microarchitecture*, Dec. 1998.

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

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

[22] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, P.G. Lowney, and M. Valero. Code layout optimizations for transaction processing workloads. *Proc. 28th Annual Int'l Symposium on Computer Architecture*, Jun. 2001.

[23] A. Ramirez, O. J Santana, J. L Larriba-Pey, and M. Valero. Fetching instruction streams. *35th Annual Int'l Symposium on Microarchitecture*, Dec. 2002.

[24] G. Reinman, B. Calder, and T. Austin. Fetch-directed instruction prefetching, *Proc. 32nd Annual Int'l Symposium on Microarchitecture*, Dec. 1999.

[25] G. Reinman, B. Calder, and T. M. Austin. Optimizations enabled by a decoupled front-end architecture. *IEEE Transactions Computers*, 50(4):338–355, 2001.

[26] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. *Proc. 29th Annual Int'l Symposium on Microarchitecture*, Dec. 1996.

[27] O. J. Santana, A. Ramirez, and M. Valero. Enlarging instruction streams. *IEEE Transactions Computers*, 56(10):1342–1357, 2007.

[28] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. *33rd Annual Int'l Symposium on Microarchitecture*, Dec. 2000.

[29] A. J. Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, 1978.

[30] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. *Proc. 29th Annual Int'l Symposium on Computer Architecture*, May 2002.

[31] L. Spracklen, Y. Chou, and S. G. Abraham. Effective instruction prefetching in chip multiprocessors for modern commercial applications. *Proc. 11th Int'l Symposium on High-Performance Computer Architecture*, 2005.

[32] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and Thomas R. Puzak. Branch history guided instruction prefetching. *7th Int'l Symposium on High-Performance Computer Architecture*, 2001.

[33] R. Stets, K. Gharachorloo, and L. A. Barroso. A detailed comparison of two transaction processing workloads. *IEEE Int'l Workshop on Workload Characterization.* Nov. 2002.

[34] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: improving both performance and fault tolerance. *Proc. 9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, Nov. 2000.

[35] T. F. Wenisch. *Temporal Memory Streaming*. PhD Thesis, Carnegie Mellon University, Aug. 2007.

[36] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal streams in commercial server applications. *Proc. IEEE Int'l Symposium on Workload Characterization*, Sept. 2008.

[37] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. *Proc. 32nd Annual Int'l Symposium on Computer Architecture*, Jun. 2005.

[38] 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):18–31, Jul.-Aug. 2006.

[39] J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. *Proc. 2004 ACM SIGMOD Int'l Conference on Management of Data*, 2004.

[40] C. B. Zilles and G. S. Sohi. Execution-based prediction using speculative slices. *Proc. 28th Annual Int'l Symposium on Computer Architecture*, Jun. 2001.