SHIFT: Shared History Instruction Fetch for Lean-Core Server Processors

Cansu Kaynak
EcoCloud, EPFL

Boris Grot*
University of Edinburgh

Babak Falsafi
EcoCloud, EPFL

ABSTRACT

In server workloads, large instruction working sets result in high L1 instruction cache miss rates. Fast access requirements preclude large instruction caches that can accommodate the deep software stacks prevalent in server applications. Prefetching has been a promising approach to mitigate instruction-fetch stalls by relying on recurring instruction streams of server workloads to predict future instruction misses. By recording and replaying instruction streams from dedicated storage next to each core, stream-based prefetchers have been shown to overcome instruction fetch stalls. Problematically, existing stream-based prefetchers incur high history storage costs resulting from large instruction working sets and complex control flow inherent in server workloads. The high storage requirements of these prefetchers prohibit their use in emerging lean-core server processors.

We introduce Shared History Instruction Fetch, SHIFT, an instruction prefetcher suitable for lean-core server processors. By sharing the history across cores, SHIFT minimizes the cost per core without sacrificing miss coverage. Moreover, by embedding the shared instruction history in the LLC, SHIFT obviates the need for dedicated instruction history storage, while transparently enabling multiple instruction histories in the presence of workload consolidation. In a 16-core server CMP, SHIFT eliminates 81% (up to 93%) of instruction cache misses, achieving 19% (up to 42%) speedup on average. SHIFT captures 90% of the performance benefit of the state-of-the-art instruction prefetcher at 14x less storage cost.

Categories and Subject Descriptors
B.3.2 [**Memory Structures**]: Design Styles – cache memories.

General Terms
Design, Performance

Keywords
Instruction streaming, prefetching, caching, branch prediction

*This work was done while the author was at EPFL.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
MICRO-46, December 07 - 11 2013, Davis, CA, USA
Copyright 2013 ACM 978-1-4503-2638-4/13/12…$15.00.
http://dx.doi.org/10.1145/2540708.2540732

1. INTRODUCTION

Servers power today's information-centric world. Server workloads range from traditional databases that perform business analytics or online transaction processing, to emerging scale-out workloads such as web search and media streaming. A characteristic feature of server workloads is their multi-megabyte instruction working sets that defy the capacities of private first- and second-level caches. As a result, server workloads suffer from frequent instruction fetches from the last-level cache that cause the frontend of the processor to stall. Prior work has shown that frontend stalls due to large instruction working sets of server workloads account for up to 40% of execution time in server processors [17, 29, 35, 39].

Prefetching is a well-known mechanism for overcoming the instruction stall bottleneck. Because server workloads exhibit highly recurring behavior in processing a large number of similar requests, they are amenable to prefetching as their control flow tends to be predictable at the request level. However, the control flow of each individual request may be quite complex and span multiple layers of the software stack, including the application itself, a database engine, a web server and the OS. As a result, naïve techniques, such as next-line prefetching, offer only marginal performance benefit.

State-of-the-art instruction prefetchers for server workloads rely on *temporal streaming* to record, and subsequently replay, entire sequences of instructions [14, 15]. As a class, these stream-based prefetchers have been shown to be highly effective at eliminating the vast majority of frontend stalls stemming from instruction cache misses. However, due to the large instruction footprints and small, yet numerous differences in the control flow among the various types of requests of a given workload, stream-based prefetchers require significant storage capacities for high miss coverage. Thus, the state-of-the-art stream-based prefetcher for servers, Proactive Instruction Fetch (PIF), can eliminate an average of 90% of instruction cache misses, but necessitates over 200KB per core for its history storage [14].

Stream-based instruction prefetchers were proposed for conventional fat-core processors, where the prefetcher's high storage cost is acceptable in view of large private cache capacities and big core area footprints. Meanwhile, recent research results [22] and industry trends point in the direction of server processors with many lean cores, rather than a handful of fat cores, which is a characteristic of conventional server processor designs. For instance, Tilera's Tile-series processors integrate as many as 64 simple cores and have been shown to be highly effective on Facebook's workload [7]. In general, manycore processors are well-matched to rich request-level parallelism present in server workloads, and achieve

high energy efficiency on memory-intensive server applications through their simple core microarchitectures.

We observe that stream-based prefetchers are poorly matched to lean-core processor designs in which the area footprint of the prefetcher history storage may approach that of the core and its L1 caches. For instance, in 40nm technology, an ARM Cortex-A8 (a dual-issue in-order core) together with its L1 caches occupies an area of 1.3mm^2, whereas PIF's per-core storage cost is 0.9mm^2.

This work attacks the problem of effective instruction prefetch in lean-core server processors. Individual server workloads are *homogeneous*, meaning that each core executes the same types of requests as all other cores. Consequently, over time, the various cores of a processor executing a common homogeneous server workload tend to generate similar instruction access sequences. Building on this observation, we make a critical insight that commonality and recurrence in the instruction-level behavior across cores can be exploited to generate a common instruction history, which can then be shared by all of the cores running a given workload. By sharing the instruction history and its associated storage among multiple cores, this work provides an effective approach for mitigating the severe area overhead of existing instruction prefetcher designs, while preserving their performance benefit. As a result, this work proposes a practical instruction prefetcher to mitigate the frontend stalls resulting from instruction cache misses in lean-core server processors.

The contributions of this work are as follows:

- We demonstrate significant commonality (over 90%) in the instruction history across multiple cores running a common server workload.
- We show that a single core chosen at random can generate the instruction history, which can then be shared across other cores running the same workload. In a 16-core CMP, the shared history eliminates 81%, on average, of all instruction cache misses across a variety of traditional and emerging server workloads.
- We introduce Shared History Instruction Fetch, SHIFT, a new instruction prefetcher design, which combines shared instruction history with light-weight per-core control logic. By sharing the history, SHIFT virtually eliminates the high history storage cost associated with earlier approaches, improving performance per unit of core area by 16% and 59% for two lean-core designs over the state-of-the-art instruction prefetcher [14]. The absolute performance improvement on a suite of diverse server workloads is 19%, on average.
- By embedding the history in the memory hierarchy, SHIFT eliminates the need for dedicated storage and provides the flexibility needed to support consolidated workloads via a per-workload history.

The rest of the paper is organized as follows. We motivate the need for effective and low-cost instruction prefetching for lean-core server processors in Section 2. We quantify the commonality of instruction fetch streams across cores running a server workload in Section 3 and describe the SHIFT design in Section 4. In Section 5, we evaluate SHIFT against prior proposals and study SHIFT's sensitivity to the design parameters. We discuss additional issues and prior work in Section 6 and Section 7. Finally, we conclude in Section 8.

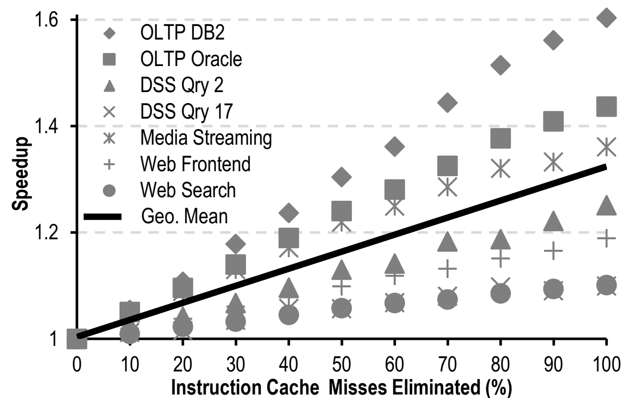

Figure 1. Speedup as a function of cache misses eliminated.

2. MOTIVATION

2.1 Instruction Stalls in Server Workloads

Server workloads have vast instruction working sets as a result of the deep software stacks they employ and their heavy reliance on the operating system functionality. The large instruction footprints are beyond the reach of today's practical first-level instruction caches [17, 20, 35]. Instruction cache misses served from lower levels of the cache hierarchy incur delays that cause server cores to stall up to 40% of the execution time [17, 29, 35, 39], exposing instruction-fetch stalls as a dominant performance bottleneck in servers [2, 13, 16, 18, 20, 22, 35].

Figure 1 shows the performance improvements that server workloads can achieve as a function of the fraction of instruction cache misses eliminated. The system configuration and the workloads used for the experiment are detailed in Section 5.1. In this experiment, each instruction cache miss is converted into a hit (i.e., the miss latency is not exposed to the core) with some probability, determined based on the desired fraction of instruction misses eliminated. 100% instruction cache miss elimination is equivalent to a perfect instruction cache.

As Figure 1 shows, both traditional database and web workloads (OLTP, DSS and web frontend), as well as emerging server workloads (media streaming and web search), achieve significant performance improvements as a result of eliminating instruction misses. As indicated by the trend line, performance increases linearly as the fraction of instruction misses decreases, reaching 31% speedup on average, for our workload suite.

Despite the linear relationship between performance and miss coverage, the correspondence is not one-to-one. This indicates that while high miss coverage is important, perfection is not necessary to realize much of the performance benefit. For instance, improving the miss coverage from 80% to 90% yields an average performance gain of 2.3%. Consequently, the amount of resources dedicated to eliminating some fraction of misses needs to be balanced with the expected performance gain.

2.2 Instruction Prefetching

Instruction prefetching is an established approach to alleviate instruction-fetch stalls prevalent in server workloads. Next-line prefetcher, a common design choice in today's processors, improves performance by 9% by eliminating 35% of instruction cache misses, on average, for our workload suite. Given the performance potential from eliminating more instruction cache misses,

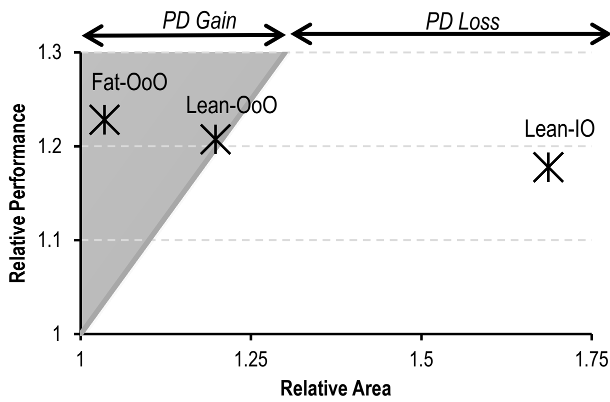

Figure 2. Comparison of PIF [14] area overhead and performance improvement for various core types.

server processors call for more sophisticated instruction prefetchers.

To overcome the next-line prefetcher's inability to predict instruction cache misses that are not contiguous, stream-based prefetchers [14, 15] exploit the recurring control-flow graph traversals in server workloads. As program control flow recurs, the core frontend generates repeating sequences of fetch addresses. The instruction fetch addresses that appear together and in the same order are temporally correlated and together form a so-called *temporal stream*. For instance, in the instruction cache access sequence *A, B, C, D*, *X, Y*, *A, B, C, D*, *Z*, the address sequence *A, B, C, D* constitutes a temporal instruction stream. Once a temporal stream is recorded, it can be identified by its first address, the stream head (address *A* in our example) and replayed to issue prefetch requests in advance of the core frontend to hide the instruction cache miss latency from the core. If the actual instruction stream matches the replayed stream (addresses *B, C, D*), the instruction misses are eliminated by the prefetcher.

The state-of-the-art stream-based instruction prefetcher is Proactive Instruction Fetch (PIF) [14], which extends earlier work on temporal streaming [15]. PIF's key innovation over prior work is its reliance on access streams instead of miss streams. By recording all accesses to the instruction cache, as opposed to just those that miss, PIF eliminates the dependence on the content of the cache. While cache content can change over time, reference patterns that PIF records remain stable. The drawback of recording access streams, instead of just miss streams, is the high history storage required. For instance, to eliminate an average of 90% of instruction cache misses, thus approaching the performance of a perfect I-cache, PIF requires over 210KB of history storage per core (Section 5.1 details PIF's storage cost).

2.3 Toward Lean Cores

Today's mainstream server processors, such as Intel Xeon and AMD Opteron, feature a handful of powerful cores targeting high single-threaded performance. Recent research has demonstrated a mismatch between these processor organizations and server workloads whose chief characteristic is low instruction- and memory-level parallelism but high request-level parallelism [13, 22]. Server workloads are best served by processors with many lean cores to maximize throughput and efficiency. A number of processors in today's server space exemplify the lean-core philosophy. These include the Tilera Tile series [45] and the Calxeda ARM-based Server-on-Chip [9]. While the performance benefits of stream-based instruction prefetching are consistently high across the core microarchitecture spectrum, the relative cost varies greatly. For instance, the 213KB of storage required by the PIF prefetcher described above consumes 0.9mm^2 of the die real-estate in 40nm process technology. Meanwhile, a Xeon Nehalem core along with its private L1 caches has an area footprint of 25mm^2. The 4% of area overhead that PIF introduces when coupled to a Xeon core is a relative bargain next to the 23% performance gain it delivers. This makes PIF a good design choice for conventional server processors.

On the opposite end of the microarchitectural spectrum is a very lean core like the ARM Cortex-A8 [6]. The A8 is a dual-issue in-order design with an area of 1.3mm^2, including the private L1 caches. In comparison to the A8, PIF's 0.9mm^2 storage overhead is prohibitive given the 17% performance boost that it delivers. Given that server workloads have abundant request-level parallelism, making it easy to scale performance with core count, the area occupied by PIF is better spent on another A8 core. The extra core can double the performance over the baseline core, whereas PIF only offers a 17% performance benefit.

To succinctly capture the performance-area trade-off, we use the metric of *performance-density (PD)*, defined as performance per unit area [22]. By simply adding cores, server processors can effectively scale performance, while maintaining a constant PD (i.e., twice the performance in twice the area). As a result, the desirable microarchitectural features are those that grow performance-density, as they offer a better-than-linear return on the area investment.

Figure 2 shows the relative performance-density merits of three PIF-enabled core microarchitectures over their baseline (without PIF) counterparts. Two of the cores are the Xeon and ARM Cortex-A8 discussed above; the third is an ARM Cortex-A15 [40], a lean out-of-order core with an area of 4.5mm^2. For simplicity, we refer to the three designs as Fat-OoO (Xeon), Lean-OoO (A15), and Lean-IO (A8).

In the figure, the trend line indicates constant PD (PD = 1), which corresponds to scaling performance by adding cores. The shaded area that falls into the left-hand side of the trend line is the region of PD gain, where the relative performance improvement is greater than the relative area overhead. The right-hand side of the trend line corresponds to PD loss, where the relative performance gain is less than the relative area. In summary, designs that fall in the shaded region improve performance-density over the baseline; those outside strictly diminish it.

The results in the figure match the intuition. For the Fat-OoO core, PIF improves performance-density. In contrast, for the Lean-OoO core, PIF fails to improve PD, and for the Lean-IO core, PIF actually diminishes PD, providing less-than-linear performance density benefit that cannot compensate for its area overhead. Thus, we conclude that lean-core processors benefit from instruction prefetching nearly as much as fat-core designs, but mandate area-efficient mechanisms to minimize the overhead.

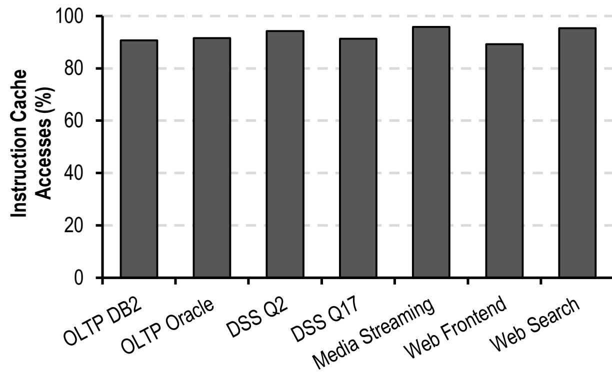

Figure 3. Instruction cache accesses within common temporal streams.

3. INSTRUCTION STREAM COMMONALITY ACROSS CORES

Our performance density analysis of the state-of-the-art instruction prefetcher shows it to be unsuitable for lean-core server processors due to its high area overhead. This work overcomes the high per-core storage cost of existing designs by sharing the cost among all cores executing a common server workload. Our insight is that instruction fetch streams across cores are common and are amenable to sharing.

To show the potential for sharing the instruction history, we first quantify the similarity between the instruction streams of cores running a homogeneous server workload. For the purposes of this study, only one core picked at random records its instruction cache access stream. All the other cores, upon referencing the first address in a stream, replay the most recent occurrence of that stream in the recorded history. The commonality between two streams is quantified as the number of matching instruction block addresses between the replayed stream and those issued by the core replaying the stream. For this study, we use instruction fetch traces from 16 cores and average the results across all of the cores.

Figure 3 illustrates the commonality of instruction streams between cores executing a given homogeneous server workload. More than 90% (up to 96%) of instruction cache accesses (comprised of both application and operating system instructions) from all sixteen cores belong to temporal streams that are recorded by a single core picked at random. This result indicates that program control flow commonality between cores yields temporal instruction stream commonality, suggesting that multiple cores running a homogeneous server workload can benefit from a single shared instruction history for instruction prefetching.

4. SHIFT DESIGN

SHIFT exploits the commonality of instruction fetch streams across cores running a common homogeneous server workload by enabling a single instruction fetch stream history to be shared by all the cores. We base the SHIFT history storage on the Global History Buffer [26] prefetcher to record instruction fetch streams in a similar vein to prior stream-based instruction prefetchers [14, 15]. We augment each core with simple logic to read instruction streams from the shared history buffer and issue prefetch requests.

In the rest of this section, we first present the baseline SHIFT design with dedicated storage, and then explain how to virtualize the storage (i.e., embed the storage in the LLC). Finally, we demonstrate SHIFT with multiple history buffers to enable support for workload consolidation.

4.1 Baseline SHIFT Design

SHIFT employs two microarchitectural components shared by all the cores running a common workload to record and replay the common instruction streams: the history buffer and the index table. The *history buffer* records the history of instruction streams; the *index table* provides fast lookups for the records stored in the history buffer.

The per-core private *stream address buffer* reads instruction streams from the shared history buffer and coordinates prefetch requests in accordance with instruction cache misses.

Recording. SHIFT's distinguishing feature is to maintain a single shared history buffer and employ only one core, *history generator core*, running the target workload to generate the instruction fetch stream history.

The history generator core records retire-order instruction cache access streams to eliminate the microarchitectural noise in streams introduced by the instruction cache replacement policy and branch mispredictions [14]. To mitigate increased history storage requirements resulting from recording instruction cache accesses rather than instruction cache misses, the history generator core collapses retired instruction addresses by forming spatial regions of instruction cache blocks.

Step 1 in Figure 4(a) depicts how a spatial region is generated by recording retire-order instruction cache accesses obtained from the history generator core's backend. In this example, a spatial region consists of five consecutive instruction cache blocks; the trigger block and four adjacent blocks. The first access to the spatial region, an instruction within block A, is the trigger access and defines the new spatial region composed of the instruction blocks between block A and $A+4$. Subsequent accesses to the same spatial region are recorded by setting the corresponding bits in the bit vector until an access to a block outside the current region occurs.

Upon an access to a new spatial region, the old spatial region record is sent to the shared history buffer to be recorded. The history buffer, logically organized as a circular buffer, maintains the stream of retired instructions as a queue of spatial region records. A new spatial region is recorded in the shared history buffer into the entry pointed by the write pointer, as illustrated in step 2 in Figure 4(a). The write pointer is incremented by one after every history write operation and wraps around when it reaches the end of the history buffer.

To enable fast lookup for the most recent occurrence of a trigger address, SHIFT employs an index table for the shared history buffer, where each entry is tagged with a trigger instruction block address and stores a pointer to that block's most recent occurrence in the history buffer. Whenever a new spatial region record is inserted into the history buffer, SHIFT modifies the index table entry for the trigger address of the new record to point to the insertion position (step 3 in Figure 4(a)).

Replaying. The per-core stream address buffer maintains a queue of spatial region records and is responsible for reading a small portion of the instruction stream history from the shared history buffer in anticipation of future instruction cache misses. When an instruction block is not found in the instruction cache, the stream address buffer issues an index lookup for the instruction's block address to

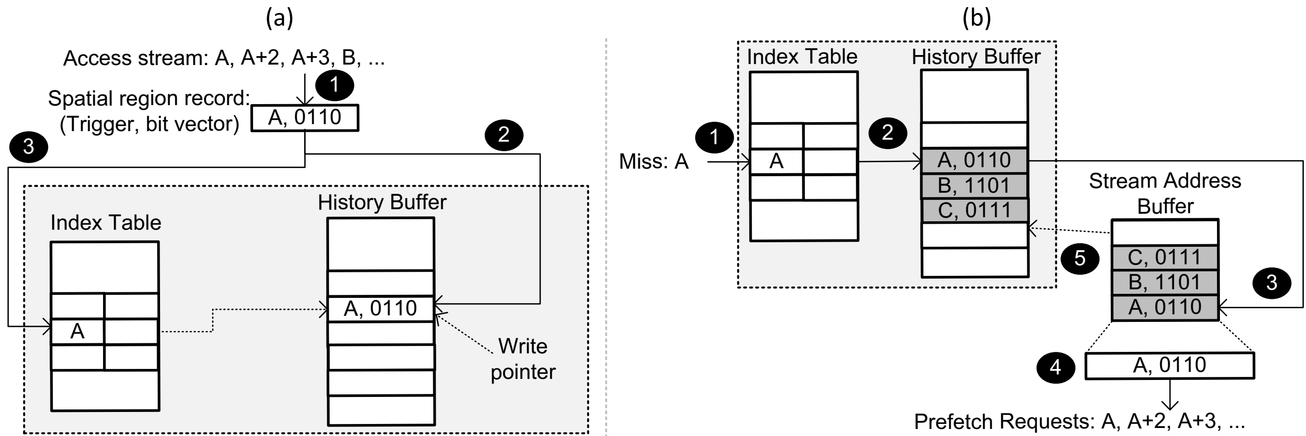

Figure 4. SHIFT's logical components and data flow to (a) record and (b) replay temporal instruction streams.

the index table (step 1 in Figure 4(b)). If a matching entry is found, it supplies the pointer to the most recent occurrence of the address in the history buffer. Once the spatial region corresponding to the instruction block that triggered the lookup is located in the history buffer (step 2 in Figure 4(b)), the stream address buffer reads out the record and a number of consecutive records following it as a lookahead optimization. The records are then placed into the stream address buffer (step 3 in Figure 4(b)).

Next, the stream address buffer reconstructs the instruction block addresses encoded by the spatial region entries based on the trigger address and the bit vector. Then, the stream address buffer issues prefetch requests for the reconstructed instruction block addresses if they do not exist in the instruction cache (step 4 in Figure 4(b)).

The stream address buffer also monitors the retired instructions. A retired instruction that falls into a spatial region maintained by the stream address buffer advances the stream by triggering additional spatial region record reads from the history buffer (step 5 in Figure 4(b)) and issues prefetch requests for the instruction blocks in the new spatial regions.

As an optimization, SHIFT employs multiple stream buffers (four in our design) to replay multiple streams, which may arise due to frequent traps and context switches in server workloads. The least-recently-used stream is evicted upon allocating a new stream. When the active stream address buffer reaches its capacity, the oldest spatial region record is evicted to make space for the incoming record.

For the actual design parameters we performed the corresponding sensitivity analysis and found that a spatial region size of eight, a lookahead of five and a stream address buffer capacity of twelve achieve the maximum performance (results not shown in the paper due to space limitation).

4.2 Virtualized SHIFT Design

The baseline SHIFT design described in Section 4.1 relies on dedicated history storage. The principal advantage of dedicated storage is that it ensures non-interference with the cache hierarchy. However, this design choice carries several drawbacks, including (1) new storage structures for the history buffer and the index table, (2) lack of flexibility with respect to capacity allocation, and (3) considerable storage expense to support multiple histories as required for workload consolidation. To overcome these limitations, we embed the SHIFT history buffer in the LLC leveraging the virtualization framework [8].

History Virtualization. To virtualize the instruction history buffer, SHIFT first allocates a portion of the physical address space for the history buffer. History buffer entries are stored in the LLC along with regular instruction and cache blocks. For the index table entries, SHIFT extends the LLC tag array to augment the existing instruction block tags with pointers to the shared history buffer records.

SHIFT reserves a small portion of the physical address space that is hidden from the operating system. The reserved address space starts from a physical address called the History Buffer Base (HBBase) and spans a contiguous portion of the physical address space. The size of the reserved physical address space for the history buffer can change based on the instruction working set size of a workload.

The history buffer is logically a circular buffer; however, the actual storage is organized as cache blocks. To access a spatial region record in the history buffer, the value of the pointer to the spatial region record is added to HBBase to form a physical address and an LLC lookup is performed for that physical address. Each cache block that belongs to the history buffer contains multiple spatial region records, therefore, each history buffer read and write operation spans multiple spatial region records. The LLC blocks that belong to the history buffer are non-evictable, which ensures that the entire history buffer is always present in the LLC. Non-eviction support is provided at the cache controller through trivial logic that compares a block's address to the address range reserved for the history. As an alternative, a cache partitioning scheme (e.g., Vantage [31]) can easily guarantee the required cache partition for the history buffer.

The index table, which contains pointers to the spatial region records in the history buffer, is embedded in the LLC by extending the tag array with pointer bits. This eliminates the need for a dedicated index table and provides an index lookup mechanism for free by coupling index lookups with instruction block requests to LLC (details below). Although each tag is augmented with a pointer, the pointers are used only for instruction blocks. Each instruction block tag in the LLC can point to the most recent occurrence of the corresponding instruction block address in the history buffer. The width of the pointer is a function of the history buffer size and is independent of the LLC capacity. In our design, each pointer is 15 bits allowing for an up to 32K-entry history buffer.

In Figure 5, the shaded areas indicate the changes in the LLC due to history virtualization. The LLC tag array is augmented with

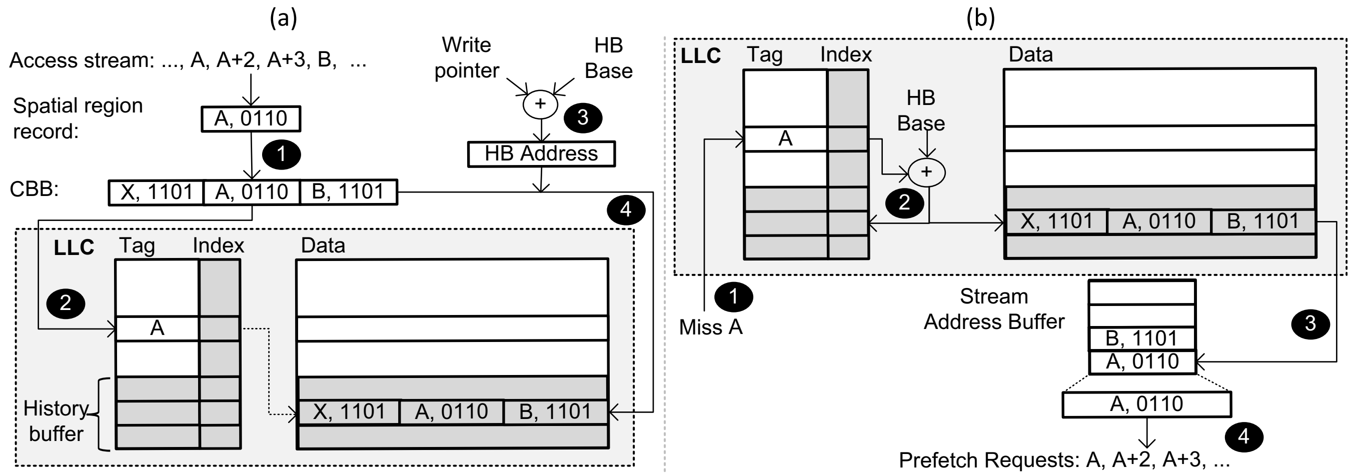

Figure 5. SHIFT's virtualized history and data flow to (a) record and (b) replay temporal instruction streams. Virtualized history components (index pointers and shared history buffer) are shaded in the LLC.

pointers and a portion of the LLC blocks are reserved for the history buffer. The LLC blocks reserved for the history buffer are distributed across different sets and banks; however, we show the history buffer as a contiguous space in the LLC to simplify the figure.

Recording. Figure 5(a) illustrates how the SHIFT logic next to the history generator core records the retire-order instruction stream in the shared and virtualized history buffer. First, the history generator core forms spatial region records as described in Section 4.1. Because the history buffer is accessed at cache-block granularity in virtualized SHIFT, the history generator core accumulates the spatial region records in a cache-block buffer (CBB), instead of sending each spatial region record to the LLC one by one (step 1). However, upon each spatial region record insertion into the CBB, the history generator core issues an index update request to the LLC for the spatial region's trigger address by sending the current value of the write pointer (step 2). The LLC performs a tag lookup for the trigger instruction block address and if the block is found in the LLC, its pointer is set to the write pointer value sent by the history generator core. After sending the index update request, the history generator core increments the write pointer.

Once the CBB becomes full, its content needs to be inserted into the virtualized history buffer. To accomplish this, the SHIFT logic next to the history generator core computes the write address by adding the value of the write pointer to HBBase (step 3), and then flushes the CBB into the LLC to the computed write address (step 4).

Replaying. While the history generator core is continuously writing its instruction access history into the LLC-resident history buffer, the rest of the cores executing the workload read the history buffer to anticipate their instruction demands. Figure 5(b) illustrates how each core replays the shared instruction stream.

SHIFT starts replaying a new stream when there is a miss in the instruction cache. For every demand request for an instruction block, the LLC sends the instruction block and the index pointer stored next to the tag of the instruction block to the requesting core (step 1). The SHIFT logic next to the core constructs the physical address for the history buffer by adding the index pointer value to HBBase and sends a request for the corresponding history buffer block (step 2). Finally, the LLC sends the history buffer block to the stream address buffer of the core (step 3).

Upon arrival of a history buffer block, the stream address buffer allocates a new stream and places the spatial region records in the history buffer block into the new stream. The stream address buffer constructs instruction block addresses and issues prefetch requests for them (step 4) as described in Section 4.1. If a retired instruction matches with an address in the stream address buffer, the stream is advanced by issuing a history read request to the LLC following the index pointer maintained as part of the stream in the stream address buffer (i.e., by incrementing the index pointer by the number of history buffer entries in a block and constructing the physical address as in step 2) .

Hardware cost. Each spatial region record, which spans eight consecutive instruction blocks, maintains the trigger instruction block address (34 bits) and 7 bits in the bit vector (assuming a 40-bit physical address space and 64-byte cache blocks). A 64-byte cache block can accommodate 12 such spatial region records. A SHIFT design with 32K history buffer entries (i.e., spatial region records) necessitates 2,731 cache lines for an aggregate LLC footprint of 171KB.

With history entries stored inside the existing cache lines and given the trivial per-core prefetch control logic, the only source of meaningful area overhead in SHIFT is due to the index table appended to the LLC tag array. The index table augments each LLC tag with a 15-bit pointer into the 32K-entry virtualized history buffer. In an 8MB LLC, these extra bits in the tag array constitute the 240KB storage overhead (accounting for the unused pointers for associated with regular data blocks in the LLC).

4.3 SHIFT Design for Workload Consolidation

Multiple server workloads running concurrently on a manycore CMP also benefit from SHIFT, as SHIFT relies on virtualization allowing for a flexible history buffer storage mechanism. SHIFT requires two minor adjustments in the context of workload consolidation. First, a history buffer per workload. should be instantiated in the LLC. SHIFT's 171KB (2% of an 8MB LLC) history buffer size is dwarfed by the LLC capacities of contemporary server processors and the performance degradation due to the LLC capacity reserved for SHIFT is negligible. So, we instantiate one history buffer per workload. Second, the operating system or the hypervi-

Table I. System and application parameters.

Processing Nodes	UltraSPARC III ISA, sixteen 2GHz cores *Fat-OoO (25mm²):* 4-wide dispatch/retirement, 128-entry ROB, 32-entry LSQ *Lean-OoO (4.5mm²):* 3-wide dispatch/retirement, 60-entry ROB, 16-entry LSQ *Lean-IO (1.3mm²):* 2-wide dispatch/retirement
I-Fetch Unit	32KB, 2-way, 64B-blocks, 2-cycle load-to-use L1-I cache Hybrid branch predictor (16K gShare & 16K bimodal)
L1D Caches	32KB, 2-way, 64B blocks, 2-cycle load-to-use, 32 MSHRs
L2 NUCA Cache	Unified, 512KB per core, 16-way, 64B blocks, 16 banks, 5-cycle hit latency, 64 MSHRs
Main Memory	45 ns access latency
Interconnect	4x4 2D mesh, 3 cycles/hop

OLTP – Online Transaction Processing (TPC-C)	
DB2	*IBM DB2 v8 ESE, 100 warehouses (10GB), 64 clients, 2 GB buffer pool*
Oracle	*Oracle 10g Enterprise Database Server,* 100 warehouses (10 GB), 16 clients, 1.4 GB SGA
DSS – Decision Support Systems (TPC-H)	
Qry 2, Qry 17	*IBM DB2 v8 ESE*, 480MB buffer pool, 1GB database
Media Streaming	
Darwin	*Darwin Streaming Server 6.0.3*, 7500 clients, 60GB dataset, high bitrates
Web Frontend (SPECweb99)	
Apache	*Apache HTTP Server v2.0*, 16K connections, fastCGI, worker threading model
Web Search	
Nutch	*Nutch 1.2/Lucene 3.0.1*, 230 clients, 1.4 GB index, 15 GB data segment

sor needs to assign one history generator core per workload and set the history buffer base address (HBBase) to the HBBase of the corresponding history buffer for all cores in the system. After these two adjustments, the record and replay of instruction streams work as described in Section 4.2.

Even with extreme heterogeneity (i.e., a unique workload per core), SHIFT provides a storage advantage over PIF as the history buffers are embedded in the LLC data array and the size of the index table embedded in the LLC tag array does not change. Because the size of the index pointers only depends on the size of the corresponding history buffer, it does not change with the number of active per-workload history buffers.

5. EVALUATION

5.1 Methodology

We evaluate SHIFT and compare it to the state-of-the-art instruction prefetcher with per-core private instruction history, PIF [14], using trace-based and cycle-accurate simulations of a 16-core CMP, running server workloads. For our evaluation, we use Flexus [44], a Virtutech Simics-based, full-system multiprocessor simulator, which models the SPARC v9 instruction set architecture. We simulate CMPs running the Solaris operating system and executing the server workload suite listed in Table I.

We use trace-based experiments for our opportunity study and initial predictor results by using traces with 32 billion instructions (two billion per core) in steady state. For the DSS workloads, we collect traces for the entire query execution. Our traces include both the application and the operating system instructions.

For performance evaluation, we use the SimFlex multiprocessor sampling methodology [44], which extends the SMARTS sampling framework [46]. Our samples are collected over 10-30 seconds of workload execution (for the DSS workloads, they are collected over the entire execution). For each measurement point, we start the cycle-accurate simulation from checkpoints with warmed architectural state and run 100K cycles of cycle-accurate simulation to warm up the queues and the interconnect state, then collect measurements from the subsequent 50K cycles. We use the ratio of the number of application instructions to the total number of cycles (including the cycles spent executing operating system code) to measure performance; this metric has been shown to accurately reflect overall system throughput [44]. Performance measurements are computed with an average error of less than 5% at the 95% confidence level.

We model a tiled SHIFT architecture with a lean-OoO core modeled after an ARM-Cortex A15 [40]. For the performance density study, we also consider a fat-OoO core (representative of contemporary Xeon-class cores) and a lean in-order core (similar to an ARM Cortex-A8 [6]), which all operate at 2GHz to simplify the comparison. The design and architectural parameter assumptions are listed in Table I. For performance density studies, we scale the published area numbers for the target cores to the 40nm technology. Cache parameters, including the SRAMs for PIF's history buffer and index table, are estimated using CACTI [24].

State-of-the-art prefetcher configuration. We compare SHIFT's effectiveness and history storage requirements with the state-of-the-art instruction prefetcher, PIF [14]. Like other stream-based instruction and data prefetchers [15, 43], PIF employs a per-core history buffer and index table. PIF records and replays spatial region records with eight instruction blocks. Each spatial region record maintains the trigger instruction block address (34 bits) and 7 bits in the bit vector. Hence, each record in the history buffer contains 41 bits. PIF requires 32K spatial region records in the history buffer targeting 90% instruction miss coverage [14], also validated with our experiments. As a result, the history buffer is 164KB for each core in total.

Each entry in PIF's index table contains an instruction block address (34 bits) and a pointer to the history buffer (15 bits) adding up to 49 bits per entry. According to our sensitivity analysis, the index table requires 8K entries for the target 90% instruction miss coverage. The actual storage required for the 8K entry index table is 49KB per core. PIF's per-core history buffer and index table together occupy 0.9mm^2 area.

We also evaluate a PIF design with a total storage cost equal to that of SHIFT. Since SHIFT stores the history buffer entries inside existing LLC cache blocks, its only source of storage overhead is the 240KB index table embedded in the LLC tag array. An equal-cost PIF design affords 2K spatial region records in the history buffer and 512 entries in the index table per core. We refer to this

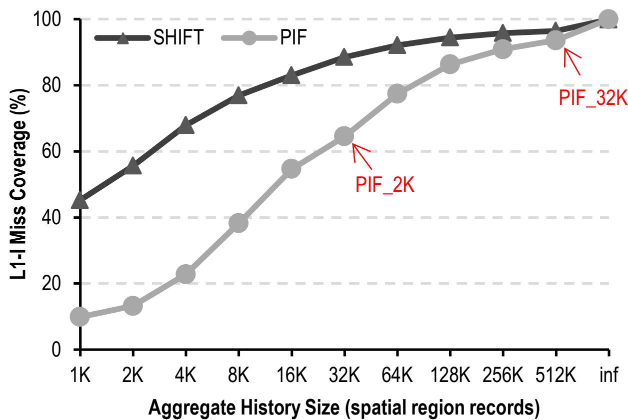

Figure 6. Percentage of instruction misses predicted.

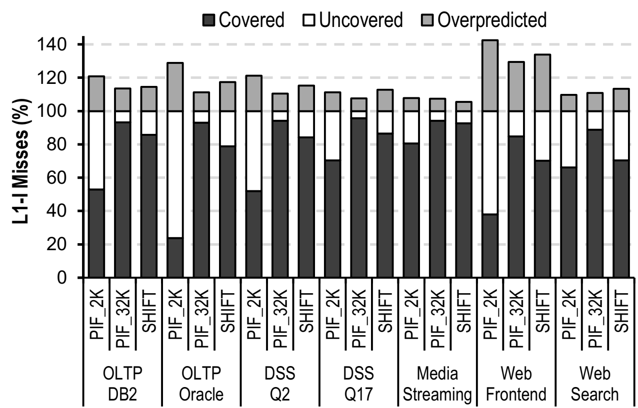

Figure 7. Percentage of instruction misses covered and overpredicted.

PIF design as PIF_2K and the original PIF design as PIF_32K to differentiate between the two design points in the rest of the paper.

5.2 Instruction Miss Coverage

To show SHIFT's effectiveness, we first compare the fraction of instruction cache misses predicted by SHIFT to PIF [14]. For the purposes of this study, we only track the predictions that would be made through replaying recurring instruction streams in stream address buffers and do not prefetch or perturb the instruction cache state.

Figure 6 shows the fraction of instruction cache misses correctly predicted for all of the cores in the system averaged across all workloads, as the number of spatial region records in the history buffer increases. The history size shown is the aggregate for PIF in the 16-core system evaluated, whereas for SHIFT, it is the overall size of the single shared history buffer.

Because the history buffer can maintain more recurring temporal instruction streams as its size increases, the prediction capabilities of both designs increase monotonically with the allocated history buffer size. Because the aggregate history buffer capacity is distributed across cores, PIF's coverage always lags behind SHIFT. For relatively small aggregate history buffer sizes, PIF's small per-core history buffer can only maintain a small fraction of the instruction working set size. As the history buffer size increases, PIF's per-core history buffer captures a larger fraction of the instruction working set. For all aggregate history buffer sizes, SHIFT can maintain a higher fraction of the instruction working set compared to PIF by employing a single history buffer, rather than distributing it across cores. As all the cores can replay SHIFT's shared instruction stream history, SHIFT's miss coverage is always greater than PIF for equal aggregate storage capacities. Because the history sizes beyond 32K return diminishing performance benefits, for the actual SHIFT design, we pick a history buffer size of 32K records.

We compare SHIFT's actual miss coverage with PIF for each workload, this time accounting for the mispredictions as well, as mispredictions might evict useful but not-yet-referenced blocks in the cache. For this comparison, we use the two PIF design points described in Section 5.1.

Figure 7 shows the instruction cache misses eliminated (covered) and the mispredicted instruction blocks (overpredicted) normalized to the instruction cache misses in the baseline design without any prefetching. On average, SHIFT eliminates 81% of the instruction cache misses with 16% overprediction, while PIF_32K eliminates 92% of the instruction cache misses with 13% overprediction, corroborating prior results [14]. However, PIF_2K, which has the same aggregate storage overhead as SHIFT, can eliminate only 53% of instruction cache misses on average, with a 20% overprediction ratio. The discrepancy in miss coverage between PIF_2K and SHIFT is caused by the limited instruction stream history stored by each core's smaller history buffer in PIF, which falls short of capturing the instruction working set.

In conclusion, by sharing the instruction history generated by a single core, all cores running a common server workload attain similar benefits to per-core instruction prefetching, but with a much lower storage overhead.

5.3 Performance Comparison

We compare SHIFT's performance against PIF_32K, PIF_2K and the next-line prefetcher normalized to the performance of the baseline system with lean-OoO cores, where no instruction prefetching mechanism is employed in Figure 8.

The difference in PIF_32K and SHIFT's speedups stems from two main differences. First, SHIFT has slightly lower miss coverage compared to PIF_32K, as shown in Figure 7. Second, SHIFT's history buffer is embedded in the LLC resulting in accesses to the LLC for reading the history buffer, which delays the replay of streams until the history buffer block arrives at the core. Moreover, history buffer reads and writes to LLC incur extra LLC traffic as compared to PIF_32K. To compare the performance benefits resulting solely from SHIFT's prediction capabilities, we also plot the performance results achieved by SHIFT assuming a dedicated history buffer with zero access latency (ZeroLat-SHIFT).

The relative performance improvements of zero-latency SHIFT and PIF_32K match the miss coverages shown in Figure 7. On average, zero-latency SHIFT provides 20% performance improvement, while PIF_32K improves performance by 21%. A realistic SHIFT design results in 1.5% speedup loss compared to zero-latency SHIFT, due to the latency of accessing the history buffer in the LLC and the traffic created by transferring history buffer data over the network. Overall, despite its low history storage cost, SHIFT retains over 90% of the performance benefit (98% of the overall absolute performance) that PIF_32K provides.

In comparison to PIF_2K, SHIFT achieves higher speedups for all the workloads as a result of its higher miss coverage as Figure 7 shows. Due to its greater effective history buffer capacity, SHIFT

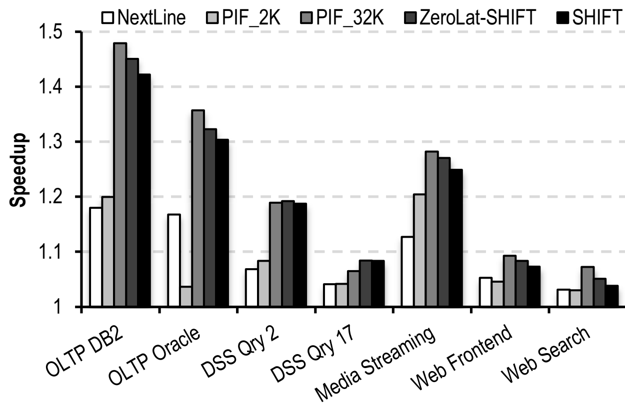

Figure 8. Performance comparison.

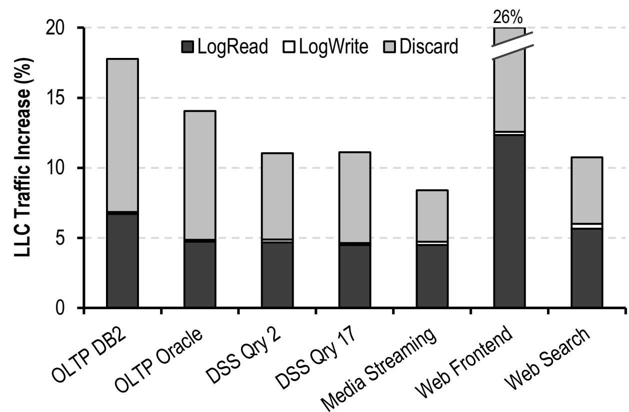

Figure 9. LLC traffic overhead.

outperforms PIF_2K for all the workloads (by 9% on average). For the workloads with bigger instruction working sets (e.g., OLTP on Oracle), SHIFT outperforms PIF_2K by up to 26%.

Finally, we compare SHIFT's performance to the next-line prefetcher. Although the next-line prefetcher does not incur any storage overheads, it only provides 9% performance improvement due to its low miss coverage (35%) stemming from its incapability of predicting misses to discontinuous instruction blocks.

5.4 LLC Overheads

SHIFT introduces two types of LLC overhead. First, the history buffer occupies a portion of the LLC, effectively reducing its capacity. Our results indicate that the performance impact of reduced capacity is negligible. With SHIFT occupying just 171KB of the LLC capacity, the measured performance loss with an 8MB LLC is under 1%.

The second source of overhead is due to the extra LLC traffic, generated by (1) read and write requests to the history buffer; (2) useless LLC reads as a result of mispredicted instruction blocks, which are discarded before used by the core; and (3) index updates issued by the history generator core. Figure 9 illustrates the extra LLC traffic generated by SHIFT normalized to the LLC traffic (due to both instruction and data requests) in the baseline system without any prefetching. History buffer reads and writes increase the LLC traffic by 6%, while discards account for the 7% of the baseline LLC traffic on average. The index updates (not shown in the graph) are only 2.5% of the baseline LLC traffic; however, they only increase the traffic in the LLC tag array.

In general, we note that LLC bandwidth is ample in our system, as server workloads have low ILP and MLP, plus the tiled design provides for a one-to-one ratio of cores to banks, affording very high aggregate LLC bandwidth. With average LLC bandwidth utilization well under 10%, the additional LLC traffic even for the worst-case workload (web frontend) is easily absorbed and has no bearing on performance in our studies.

5.5 Workload Consolidation

Figure 10 shows the performance improvement attained by SHIFT in comparison to the next-line prefetcher, PIF_2K and PIF_32K, in the presence of multiple workloads running on a server CMP. In this experiment, we use the 16-core Lean-OoO server processor described in Section 5.1. We consolidate two traditional (OLTP on Oracle and web frontend) and two emerging (media streaming and web search) server workloads. Each workload runs on four cores and has its own software stack (i.e., separate OS images). For SHIFT, each workload has a single shared history buffer with 32K records embedded in the LLC.

We see that the speedup trends for the various design options follow the same trend as the standalone workloads, as shown in Section 5.3. SHIFT delivers 95% of PIF_32K's absolute performance and outperforms PIF_2K by 12% and the next-line prefetcher by 11% on average.

Zero-latency SHIFT delivers 25% performance improvement over the baseline, while SHIFT achieves 22% speedup on average. The 3% difference mainly results from the extra LLC traffic generated by virtualized SHIFT. The LLC traffic due to log reads remains the same in the case of workload consolidation as all the cores read from their corresponding shared history embedded in the LLC. However, the index updates and log writes increase with the number of workloads, as there is one history generator core per workload. While log writes increase the fetch traffic by 1.1%, index updates, which only increase the traffic in the LLC tag array, correspond to 15% of the baseline fetch accesses.

Overall, we conclude that in the presence of multiple workloads, SHIFT's benefits are unperturbed and remain comparable to the single-workload case.

5.6 Performance Density Implications

To quantify the performance benefits of the different prefetchers as a function of their area cost, we compare SHIFT's performance-density (PD) with PIF_32K and PIF_2K. We consider the three core designs described in Section 2.3 namely, Fat-OoO, Lean-OoO, and Lean-IO.

SHIFT improves performance-density over PIF_32K as a result of eliminating the per-core instruction history, while retaining similar performance benefits. Compared to PIF_32K, SHIFT improves the overall performance by 16 to 20%, depending on the core type, at a negligible area cost per core (0.96mm^2 in total, as opposed to PIF_32K's 14.4mm^2 cost in aggregate in a 16-core CMP). While PIF_32K offers higher absolute performance, the relative benefit is diminished due to the high area cost. As a result, SHIFT improves PD over PIF_32K for all three core microarchitectures. As expected, the biggest PD improvement is registered for lean cores (16% and 59% for Lean-OoO and Lean-IO respectively); however,

Figure 10. Speedup for workload consolidation.

even the fat-core design enjoys a 2% improvement in PD due to SHIFT's low area cost.

In comparison to PIF_2K, SHIFT achieves higher miss coverage due to the better capacity management of the aggregate history buffer storage. Although PIF_2K occupies the same aggregate storage area as SHIFT, SHIFT almost doubles the performance improvement for the three core types, as a result of its higher miss coverage. Consequently, SHIFT improves performance density over PIF_2K by around 9% on average for all the core types.

SHIFT improves the absolute performance-density for both lean-core designs and the fat-core design over a no-prefetch system, while providing 98% of performance of the state-of-the-art instruction prefetcher, demonstrating the feasibility of area-efficient high-performance instruction prefetching for servers.

5.7 Power Implications

SHIFT introduces power overhead to the baseline system due to two factors: (1) history buffer reads and writes to/from the LLC and (2) index reads and writes to/from the LLC. To quantify the overall power overhead induced by SHIFT, we use CACTI [24] to estimate the LLC power (both for index pointers in the tag array and history buffers in the data array) and custom NoC power models to estimate the link, router switch fabric and buffer power in the NoC [21]. We find the additional power overhead due to history buffer and index activities in the LLC to be less than 150mW in total for a 16-core CMP. This corresponds to less than 2% power increase per Lean-IO core, which is the lowest-power core evaluated in our studies. We thus conclude that the power consumption due to SHIFT is negligible.

6. DISCUSSIONS

6.1 Choice of History Generator Core

We show SHIFT's miss coverage and performance improvement by employing one history generator core picked at random throughout the paper. In our experience, in a sixteen-core system, there is no sensitivity to the choice of the history generator core.

Although the cores executing a homogeneous server workload exhibit common temporal instruction streams, there are also spontaneous events that might take place both in the core generating the shared instruction history and the cores reading from the shared instruction history, such as the OS scheduler, TLB miss handlers, garbage collector, and hardware interrupts. In our experience, such events are rare and only briefly hurt the instruction cache miss coverage due to the pollution and fragmentation of temporal streams.

In case of a long-lasting deviation in the program control flow of the history generator core, a sampling mechanism that monitors the instruction miss coverage and changes the history generator core accordingly can overcome the disturbance in the shared instruction history.

6.2 Virtualized PIF

Although virtualization could be readily used with prefetchers using per-core history, such designs would induce high capacity and bandwidth pressure in the LLC. For example, virtualizing PIF's per-core history buffers would require 2.7MB of LLC capacity and this requirement grows linearly with the number of cores. Furthermore, as each core records its own history, the bandwidth and power consumption in the LLC also increase linearly with the number of cores. By sharing the instruction prefetcher history, SHIFT not only saves area but also minimizes the pressure on the LLC compared to virtualized per-core instruction prefetchers.

7. RELATED WORK

Instruction fetch stalls have long been recognized as a dominant performance bottleneck in servers [2, 10, 13, 18, 20, 29, 41]. Simple next-line instruction prefetchers varying in prefetch degree have been ubiquitously employed in commercial processors to eliminate misses to subsequent blocks [28, 32, 33] and fail to eliminate instruction cache misses due to discontinuities in the program control flow caused by function calls, taken branches and interrupts.

A class of instruction prefetchers rely on the branch predictor running ahead of the fetch unit to predict instruction cache misses caused by discontinuities [12, 30, 36, 42]. To explore future code paths, an idle thread [20], a helper thread [1], speculative threading mechanisms [37, 48] or run-ahead execution [25] can be employed. Although these techniques do not require additional history, they are limited by the lookahead and accuracy of the branch predictor.

To overcome the lookahead limitation, the discontinuity prefetcher [35] maintains the history of discontinuous transitions between two instruction blocks. However, the lookahead of the discontinuity prefetcher is limited to one target instruction block for each source block. TIFS [15] records streams of discontinuities in its history, enhancing the lookahead of discontinuity prefetching. PIF [14], records the complete retire-order instruction cache access history, capturing both discontinuities and next-line misses. SHIFT maintains the retire-order instruction cache access history like PIF. Unlike prior stream-based instruction prefetchers, SHIFT maintains a single shared history, allowing all cores running a common workload to use the shared history to predict future instruction misses.

The SHIFT design adopts its key history record and replay mechanisms from previously proposed per-core data and instruction prefetchers [14, 15, 34, 43]. To facilitate sharing the instruction history, SHIFT embeds the history buffer in the LLC as proposed in predictor virtualization [8].

A number of orthogonal studies mitigate instruction cache misses by exploiting code commonality across multiple threads [4, 11]. These approaches distribute the code footprint across private instruction caches to leverage the aggregate on-chip instruction cache capacity. In a similar manner, SHIFT relies on the code path commonality, but it does not depend on the aggregate instruction cache capacity, which might be insufficient to accommodate large

instruction footprints. Moreover, SHIFT supports multiple workloads running concurrently, while these techniques might lose their effectiveness due to the contention for instruction cache capacity in the presence of multiple workloads.

Another way to exploit the code commonality across multiple threads is to group similar requests and time-multiplex their execution on a single core, so that the threads in a group can reuse the instructions, which are already brought into the instruction cache by the lead thread [5, 17]. Unfortunately, these approaches are likely to hurt response latency of individual threads, as each thread is queued for execution and has to wait for the other threads in the group to execute.

Prior software-base approaches proposed optimizing the code layout to avoid conflict misses [27, 38, 47], inserting instruction prefetch instructions at compile time [23], and exploiting the recurring call-graph history [3]. These techniques can be used with SHIFT to further improve instruction miss coverage and reduce the storage cost.

Concurrent with our work, RDIP [19] correlates instruction cache miss sequences with the call stack content. RDIP associates the history of miss sequences with a signature, which summarizes the return address stack content. In doing so, RDIP reduces the history storage requirements by not recording the entire instruction streams as in stream-based prefetchers. However, RDIP's miss coverage is still limited by the amount of per-core storage. SHIFT, on the other hand, amortizes the cost of the entire instruction stream history across multiple cores, obviating the need for per-core history storage reduction.

8. CONCLUSION

Instruction fetch stalls are a well-known cause of performance loss in server processors due to the large instruction working sets of server workloads. Sophisticated instruction prefetch mechanisms developed by researchers specifically for this workload class have been shown to be highly effective in mitigating the instruction stall bottleneck by recording, and subsequently replaying, entire instruction sequences. However, for high miss coverage, existing prefetchers require prohibitive storage for the instruction history due to the large instruction working sets and complex control flow. While high storage overhead is acceptable in fat-core processors whose area requirements dwarf those of the prefetcher, we find the opposite to be true in lean-core server chips.

This work confronted the problem of high storage overhead in stream-based prefetchers. We observed that the instruction history among all of the cores executing a server workload exhibits significant commonality and showed that it is amenable to sharing. Building on this insight, we introduced SHIFT – a shared history instruction prefetcher. SHIFT records the instruction access history of a single core and shares it among all of the cores running the same workload. In a 16-core CMP, SHIFT delivers over 90% of the performance benefit of PIF, a state-of-the-art instruction prefetcher, while largely eliminating PIF's prohibitive per-core storage overhead. With a lean in-order core microarchitecture, SHIFT improves performance per mm^2 by up to 59% compared to PIF, indicating SHIFT's advantage in lean-core processor designs.

9. ACKNOWLEDGEMENTS

The authors would like to thank Christos Kozyrakis, Onur Kocberber, Stavros Volos, Djordje Jevdjic, Javier Picorel, Almutaz Adileh, Pejman Lotfi-Kamran, Sotiria Fytraki, and the anonymous reviewers for their insightful feedback on earlier drafts of this paper. This work was partially supported by Swiss National Science Foundation, Project No. 200021_127021.

10. REFERENCES

[1] T. M. Aamodt, P. Chow, P. Hammarlund, H. Wang, and J. P. Shen. Hardware support for prescient instruction prefetch. In *Proceedings of the International Symposium on High Performance Computer Architecture*, Feb. 2004.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the International Conference on Very Large Data Bases*, Sept. 1999.

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

[4] I. Atta, P. Tözün, A. Ailamaki, and A. Moshovos. SLICC: Self-assembly of instruction cache collectives for OLTP workloads. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2012.

[5] I. Atta, P. Tözün, X. Tong, A. Ailamaki, and A. Moshovos. STREX: Boosting instruction cache reuse in OLTP workloads through stratified execution. In *Proceedings of the International Symposium on Computer Architecture*, June 2013.

[6] M. Baron. The F1: T1's 65nm Cortex-A8. *Microprocessor Report*, 20(7):1–9, July 2006.

[7] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *Proceedings of the International Green Computing Conference and Workshops*, 2011.

[8] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi. Predictor virtualization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2008.

[9] Calxeda. http://www.calxeda.com/.

[10] Q. Cao, P. Trancoso, J.-L. Larriba-Pey, J. Torrellas, R. Knighten, and Y. Won. Detailed characterization of a Quad Pentium Pro server running TPC-D. In *International Conference on Computer Design*, Oct. 1999.

[11] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[12] I.-C. K. Chen, C.-C. Lee, and T. N. Mudge. Instruction prefetching using branch prediction information. In *Proceedings of the International Conference on Computer Design*, Oct. 1997.

[13] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2012.

[14] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive instruction fetch. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2011.

[15] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal instruction fetch streaming. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2008.

[16] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *Proceedings of the Conference on Innovative Data Systems Research*, Jan. 2007.

[17] S. Harizopoulos and A. Ailamaki. STEPS towards cache-resident transaction processing. In *Proceedings of the International Conference on Very Large Data Bases*, Aug. 2004.

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

[19] A. Kolli, A. Saidi, and T. Wenisch. RDIP: Return-address-stack directed instruction prefetching. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2013.

[20] 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. In *Proceedings of the International Symposium on Computer Architecture*, June 1998.

[21] P. Lotfi-Kamran, B. Grot, and B. Falsafi. NOC-Out: Microarchitecting a scale-out processor. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2012.

[22] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi. Scale-out processors. In *Proceedings of the International Symposium on Computer Architecture*, June 2012.

[23] C.-K. Luk and T. C. Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 1998.

[24] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2007.

[25] 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.

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

[27] 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. In *Proceedings of the International Symposium on Computer Architecture*, June 2001.

[28] A. Ramirez, O. J. Santana, J. L. Larriba-Pey, and M. Valero. Fetching instruction streams. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2002.

[29] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 1998.

[30] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 1999.

[31] D. Sanchez and C. Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the International Symposium on Computer Architecture*, June 2011.

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

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

[34] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-temporal memory streaming. In *Proceedings of the International Symposium on Computer Architecture*, June 2009.

[35] L. Spracklen, Y. Chou, and S. G. Abraham. Effective instruction prefetching in chip multiprocessors for modern commercial applications. In *Proceedings of the International Symposium on High Performance Computer Architecture*, Feb. 2005.

[36] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak. Branch history guided instruction prefetching. In *Proceedings of the International Symposium on High Performance Computer Architecture*, Jan. 2001.

[37] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[38] J. Torrellas, C. Xia, and R. Daigle. Optimizing instruction cache performance for operating system intensive workloads. In *Proceedings of the International Symposium on High Performance Computer Architecture*, Jan. 1995.

[39] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki. From A to E: Analyzing TPC's OLTP benchmarks: the obsolete, the ubiquitous, the unexplored. In *Proceedings of the International Conference on Extending Database Technology*, Mar. 2013.

[40] J. Turley. Cortex-A15 "Eagle" flies the coop. *Microprocessor Report*, 24(11):1–11, Nov. 2010.

[41] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Instruction fetching: Coping with code bloat. In *Proceedings of the International Symposium on Computer Architecture*, June 1995.

[42] A. V. Veidenbaum. Instruction cache prefetching using multilevel branch prediction. In *Proceedings of the International Symposium on High-Performance Computing*, Nov. 1997.

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

[44] 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, July-Aug. 2006.

[45] B. Wheeler. Tilera sees opening in clouds. *Microprocessor Report*, 25(7):13–16, July 2011.

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

[47] C. Xia and J. Torrellas. Instruction prefetching of systems codes with layout optimized for reduced cache misses. In *Proceedings of the International Symposium on Computer Architecture*, June 1996.

[48] C. B. Zilles and G. S. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the International Symposium on Computer Architecture*, June 2001.