Proactive Instruction Fetch

Michael Ferdman[1 2], Cansu Kaynak[2], and Babak Falsafi[2]
{michael.ferdman, cansu.kaynak, babak.falsafi}@epfl.ch
[1]Computer Architecture Lab (CALCM), Carnegie Mellon University, Pittsburgh, PA, USA
[2]Parallel Systems Architecture Lab (PARSA), Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

ABSTRACT

Fast access requirements preclude building L1 instruction caches large enough to capture the working set of server workloads. Efforts exist to mitigate limited L1 instruction cache capacity by relying on the stability and repetitiveness of the instruction stream to predict and prefetch future instruction blocks prior to their use. However, dynamic variation in cache miss sequences prevents correct and timely prediction, leaving many instruction-fetch stalls exposed, resulting in a key performance bottleneck for servers.

We observe that, while the vast majority of application instruction references are amenable to prediction, even minor control-flow variations are amplified by microarchitectural components, resulting in a major source of instability and randomness that significantly limit prefetcher utility. Control-flow variation disturbs the L1 instruction cache replacement order and branch predictor state, causing the L1 instruction cache to randomly filter the instruction stream while the branch predictor and spontaneous hardware interrupts inject the stream with unpredictable noise. Based on this observation, we show that an instruction prefetcher, previously plagued by microarchitectural instability, becomes nearly perfect when modified to operate on the correct-path, retire-order instruction stream. We propose Proactive Instruction Fetch, an instruction prefetch mechanism that achieves higher than 99.5% instruction-cache hit rate, improving server throughput by 27% and nearly matching the performance of a perfect L1 instruction cache that never misses.

Categories and Subject Descriptors

B.3.2 [Memory Structures]: Design Styles – cache memories.

General Terms

Design, Performance

Keywords

instruction streaming, prefetching, caching, branch prediction

1 INTRODUCTION

Although aggregate on-chip cache capacity grows with each technology generation, fast access requirements prohibit building L1 instruction caches large enough to accommodate the working sets of server workloads [10, 14, 27]. Data-dependent branches, shared library and operating system calls, and hardware interrupts cause control transfers across multi-megabyte binaries, leading to high L1 instruction-cache miss rates that account for over 40% of the execution time [10, 27], emerging as one of the dominant performance bottlenecks in server systems [1, 9, 10, 13, 14, 27, 29].

To bridge the gap between inadequate L1 instruction cache capacity and the need for low-latency access to instructions, researchers have proposed mechanisms that predict the future execution path and prefetch instruction blocks along the predicted path. The next-line instruction prefetchers [12, 25] effectively mitigate the latency of L1 instruction-cache misses of spatially contiguous blocks [14]. Correlating prefetchers predict discontinuous instruction-cache misses by allowing the branch predictor to run ahead of instruction fetch [5, 22, 28, 31] or by maintaining a separate record of the previously observed discontinuities and predicting their repetition [6, 27].

Although effective at reducing the L1 instruction-cache miss rate, existing prefetchers fall short of an ideal L1 cache. Prefetchers rely on the stability and repetitiveness of the instruction reference stream to learn the access sequence and predict when the same sequence will repeat. However, when faced with instability and uncertainty, the ability to learn and correctly predict diminishes; prediction is severely hampered when previously observed accesses disappear or when previously unseen accesses emerge in the middle of the learned sequences.

We observe that, while the vast majority of the application instruction sequences are extremely repetitive, microarchitectural components (L1 instruction cache, branch predictor, and interrupt handling) are responsible for the instability and lack of repetition in the instruction miss stream. Even small changes in control flow affect the cache replacement order, resulting in different miss sequences for precisely the same sequences of instruction fetches. Furthermore, data dependencies cause branch mispredictions, injecting an arbitrary number of wrong-path instructions, determined by the data-dependent delays of resolving mispredictions. In turn, wrong-path cache accesses and hardware interrupts further amplify the instability that the cache introduces. We find that much of the ineffectiveness of prior techniques is due to vainly attempting to predict an instruction sequence that is artificially fragmented, filtered, and inflated. Moreover, this observation explains why prior work could not achieve the expected speedups observed in trace studies [6, 27].

In this work, we propose *Proactive Instruction Fetch (PIF)*, an instruction prefetcher that avoids the instability and randomness of the instruction sequence introduced by the microarchitecture. We use the correct-path, retire-order instruction sequence and processor trap levels to record the exact instruction-fetch sequence, unaffected by the filtering and wrong-path injection effects of the cache, branch predictor, or hardware interrupts.

MICRO'11, December 3-7, 2011, Porto Alegre, Brazil

When a previously recorded address recurs, we simply prefetch the upcoming requests by replaying the recorded sequence starting at the most recent location of the recurring address in the recorded sequence. Eliminating the microarchitectural randomness and noise exposed to the prefetcher enables high prediction coverage and accuracy.

We evaluate the stability and predictability of the instruction-fetch sequences collected at various locations of the processor. Drawing on the observed properties of the sequences, we propose a prefetcher implementation that eliminates most instruction-fetch stalls. Using full-system simulation of a large-scale, next-generation CMP running unmodified commercial server software and industry-standard benchmarks, we demonstrate:

- **Microarchitectural effects on instruction history.** We observe that the instruction cache, the branch predictor, and spontaneous interrupts hurt the effectiveness of prior techniques by recording non-repetitive instruction history.
- **Near-perfect repetitive instruction streams.** We find that the history of retired instructions provides a nearly ideal repetitive instruction stream, enabling over 99.5% prediction coverage of the instruction-fetch accesses.
- **Compaction of accesses to reduce history storage.** We find that considerable history storage compaction is possible through recording spatially-correlated and temporally-correlated groups of accesses rather than individual references.
- **Benefits of Proactive Instruction Fetch.** We propose hardware to leverage the repetitiveness of compacted retire-order instruction streams to predict future instruction accesses. Our technique eliminates nearly all instruction-fetch stalls, converging to the performance of a perfect L1 instruction cache.

The rest of this paper is organized as follows. We explore the alternatives for collecting the instruction streams suitable for accurate prediction of the instruction-fetch sequence in Section 2. We present a brief evaluation of the key properties motivating the Proactive Instruction Fetch design in Section 3 and describe the hardware design in Section 4. In Section 5, we present a sensitivity analysis of the key design parameters and compare our design to prior work. We discuss prominent prior work in Section 6 and conclude in Section 7.

2 MICROARCHITECTURAL EFFECTS ON INSTRUCTION STREAMS

Instruction-fetch stalls are a known critical bottleneck for server workloads [10, 27]. Prefetching the instruction blocks into the instruction cache prior to a core's request can avoid stalling the processor, mitigating the bottleneck. Existing instruction prefetchers leverage the regularity and repetitiveness of the instruction cache reference stream to predict future instruction cache requests. However, we find that microarchitectural components such as the instruction cache, the branch predictor, and hardware interrupt handling limit an instruction prefetcher's ability to correctly predict future accesses.

Instruction caches have no mechanism to ensure that blocks that are accessed together either all remain in the cache or are all evicted from the cache. The replacement policy operates at block granularity, treating each block independently and selecting victim blocks without regard to their content or relationship to the other cached blocks. The selection of victim instruction blocks changes dynamically throughout program execution, arbitrarily filtering and fragmenting the cache miss stream as compared to the more regular and repetitive stream of instruction fetches.

The branch predictor speculatively forges ahead of the known control flow, mispredicting the instruction path when unstable data-dependent branches are encountered. While a reorder buffer can tolerate branch mispredictions that are quickly resolved, wrong-path instruction references are arbitrarily injected into the instruction reference stream, appearing both as instruction accesses and as instruction-cache misses, damaging the regularity and repetitiveness of the observed instruction stream and precluding prediction by mechanisms that rely on stable history.

We explore the microarchitectural side effects on the instruction streams in detail and find that retire-order instruction streams exhibit near-perfect repetitiveness compared to both instruction-cache access and cache-miss streams. Retire-order instruction streams are not affected by the instruction-cache filtering nor by the noise injected by the branch predictor. Retire-order streams provide a clean instruction stream for effective and timely prediction of future accesses, enabling an instruction prefetcher that can eliminate nearly all of the instruction-fetch stalls.

2.1 Eliminating the Instruction Cache as a Filter

Despite being an effective mechanism for reducing the instruction-fetch stalls, the instruction cache cannot remove all stalls because the low latency requirements preclude caches large enough to capture the entire—multi-megabyte [8]—instruction working sets of server workloads. Moreover, by filtering and fragmenting the instruction stream, the cache limits the effectiveness of instruction prefetchers that rely on the stability of the reference stream.

Because program control flow is repetitive, the processor front-end typically fetches the same sequence of instruction blocks in the same order, resulting in temporal correlation among consecutively accessed instruction blocks. Thus, recording and replaying temporal instruction streams, sequences of instruction blocks that appear together and in the same order during program execution, was shown to be effective at correctly and accurately predicting instruction-cache misses [6].

In the case of the LRU replacement policy, a block's chance of being present in the cache is determined by how frequently it is referenced and how frequently the other blocks in the same set are referenced. The instruction cache tracks all blocks independently, ignoring the temporal correlation between consecutive accesses to the instruction blocks.

In Figure 1 (left), we show an example of how the instruction cache fragments access sequences and creates non-repetitive miss sequences. For demonstration purposes, we use a four-block, direct-mapped instruction cache. We assume an initially empty cache and two unique access sequences: *ABCD* and *RS*. The first time the *ABCD* access sequence arrives at time T_1, the miss sequence is exactly the same as the access sequence, because the cache initially does not contain any of the accessed blocks. When the *RS* access sequence arrives, *R* replaces *A* and *S* replaces *C*. By time T_2, *A* and *C* are evicted from the instruction cache, while their temporally-correlated blocks *B* and *D* remain in the cache. The second time the *ABCD* access sequence appears at time T_3, the corresponding miss sequence, *AC*, is different from the access

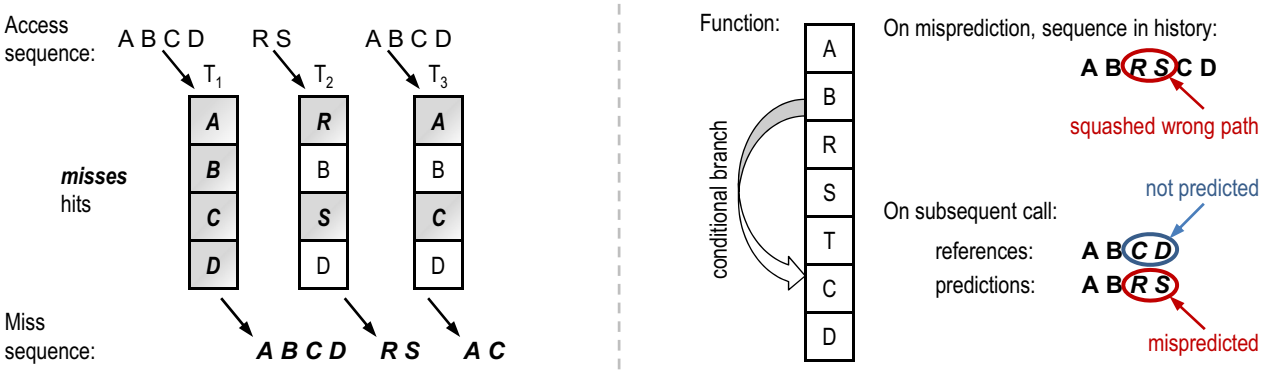

Figure 1. Fragmentation of the instruction sequences by the instruction cache (left) and Branch predictor noise (right)

sequence, because the instruction cache still contains some of the blocks in the access stream. As a result, the next time *ABCD* occurs, an instruction prefetcher that follows the most-recently recorded miss stream starting with *A* (stream *AC*), will prefetch *C* but will not prefetch *B* and *D*. The instruction cache evicts the blocks independently of the rest of their temporal access stream, separating the temporally-correlated block addresses from each other in the miss stream. Thus, the instruction cache fragments the access streams, losing the correlation information between blocks.

Figure 2 shows the fraction of the correct-path instruction-cache misses that can be predicted by recording the temporal instruction streams at various places in the processor and replaying the most recent stream when the first address of that stream recurs. For this study, we use a cycle-accurate model of a 16-core out-of-order CMP with 2-way 64KB L1 instruction caches. We trace the instruction references for 50M cycles of cycle-accurate execution of server workloads at a steady state (system and workload details are listed in Section 5), presenting the results averaged across the 16 simulated cores. The processor behavior is undisturbed by the experiment, as we only track the predictions that would be made, but do not prefetch or perturb the cache state in any way.

In Figure 2, the left-most *Miss* bar shows the predictor coverage[1] achieved by predicting the instruction-cache miss streams, while the *Access* bar shows the instruction-cache miss coverage achieved by predicting the cache accesses. The disparity in the predictor coverage for the two streams highlighted by this experiment arises directly from the randomized filtering and fragmenting effects of the instruction cache, as all other aspects (including the actual instruction stream) are exactly identical.

We find that losing the temporal correlation information of the instruction access streams always results in less regular and less repetitive streams, limiting the prediction capability of a prefetcher. For example, the instruction-cache miss streams that result from the same instruction reference stream lose the opportunity to predict more than 20% of the correct-path instruction-cache misses in the *Web* workloads.

1. *Coverage* is the fraction of the *correct-path* instruction cache *misses* predicted through temporal correlation. In some cases, wrong-path accesses may prefetch the correct-path instruction blocks. We take this effect into account in our experiment; the correct-path accesses following wrong-path misses are counted as cache hits and not included in our coverage computation.

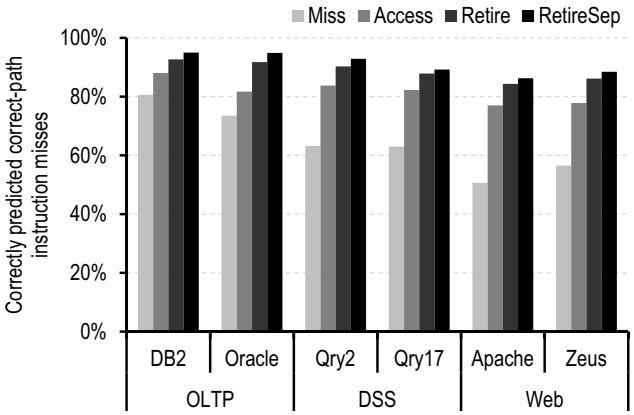

Figure 2. Percentage of correctly predicted L1-I misses

2.2 Eliminating the Branch Predictor Noise

The instruction-cache access streams are made up of instruction-block addresses generated by the processor front-end, including both the correct-path and wrong-path references. However, wrong-path accesses are unstable and change frequently between visits to the same code. The dynamic behavior of the branch predictor sometimes injects wrong-path accesses into the instruction stream and sometimes does not. Moreover, because of the variance in the pipeline behavior (due to data-cache misses, data-dependent latencies, load/store dependencies, resource stalls, etc.), the latency to detect a branch misprediction and squash the wrong-path instructions from the pipeline is unpredictable, resulting in an arbitrary time spent exploring the wrong path and, therefore, an arbitrary number of wrong-path accesses injected into the reference stream.

Figure 1 (right) illustrates how branch predictor noise is injected into the instruction access sequence. The function in the example spans multiple instruction blocks and has a conditional branch instruction in block *B* which will cause the blocks *R, S,* and *T* to be skipped if the branch is taken. In our example, the branch predictor mispredicts and decides not to take the branch. The fetch unit sends requests for the instruction blocks *A, B, R,* and *S* before the branch misprediction is resolved, and then continues with *C* and *D*, after the instructions from the blocks *R* and *S* are squashed in the pipeline. The recorded instruction access sequence now contains the correct-path access sequence *A, B, C, D,* interspersed with noise (blocks *R* and *S*). The next time the same function executes, the branch predictor is likely to correctly predict the branch as taken. An instruction prefetcher that follows the previously recorded stream will send requests for *A, B, R, S*. However,

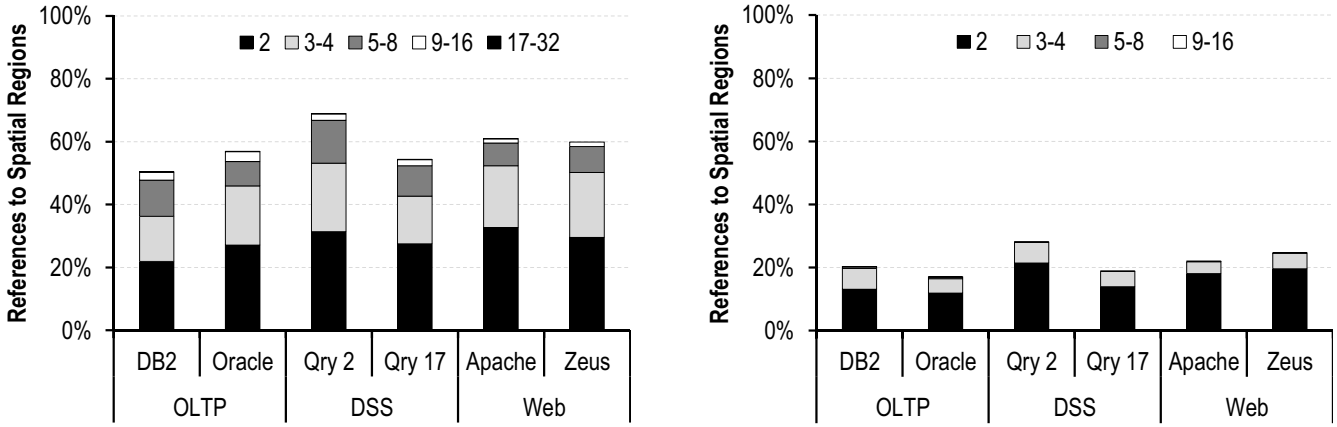

Figure 3. Density of spatial regions (left) and discontinuous (non—next-line) accesses in spatial regions (right)

the actual access sequence is *A, B, C, D* and the prefetcher will not prefetch *C* and *D*. At the same time, blocks *R* and *S* will be prefetched and will pollute the cache and delay the arrival of blocks *C* and *D*. As we show in the example, the branch predictor introduces noise to the instruction access streams, preventing the prefetcher from issuing accurate prefetch requests.

Figure 2 presents the effect of branch mispredictions on the instruction stream by comparing the correct-path miss coverage achieved by predicting the front-end instruction accesses, the *Access* bar, with the correct-path miss coverage achieved by predicting the sequence of retired instructions, the *Retire* bar. The retired-instruction streams contain only the correct-path instructions, regardless of the outcome of intermediate branch predictions or other events that may dynamically reset the reorder buffer or pipeline. We observe up to 10% miss coverage loss in *OLTP on Oracle* from predicting the instruction-cache access sequence compared to predicting the correct-path retire-order sequence.

2.3 Interrupt Handlers and Application References

Unlike scientific and engineering workloads, server workloads actively use disk and network I/O and frequently interact with the operating system. Spontaneous hardware-interrupt handlers (e.g., network card interrupts and TLB misses) occur in the pipeline frequently [11], temporarily redirecting the instruction stream to execute the handlers. The instruction access stream is fragmented at arbitrary places by the injection of references to the handler code, reducing the regularity and repetitiveness of the application references and imposing further challenges to correctly predicting the instruction stream.

To eliminate the effects of the noise introduced by interrupt handlers, we separate the instruction reference streams belonging to different processor trap levels and record them in separate temporal streams. In Figure 2, the *RetireSep* bar illustrates the effectiveness of this approach, indicating that up to an additional 2% of predictor coverage can be achieved, yielding nearly perfect opportunity to eliminate all correct-path instruction-cache misses. Notably, although the coverage loss due to interrupts is small, the misses that occur shortly after a handler returns are often the most costly to performance, as the reorder buffer is empty and the core is entirely stalled, waiting for an instruction-cache fill.

3 COMPACTING STREAMS

To take advantage of temporal streaming, an instruction prefetcher must record address streams and predict future accesses by replaying recorded streams. Unfortunately, addresses in temporal streams exhibit no simple patterns such as strides [7] or delta correlation [18]. Functions are typically distributed throughout the instruction memory at the whim of the programmer, the compiler, and the dynamic library loader, yielding data-dependent jumps in arbitrary directions and for arbitrary distances as instructions are executed. Options for encoding the temporal streams are therefore limited, requiring to record addresses one by one, resulting in large storage overhead for predictors relying on temporal streams [18, 32]. In turn, storage capacity emerges as a major constraint, limiting the full prefetcher potential because predictor coverage must be traded off for capacity.

Fortunately, many instruction blocks are not only temporally correlated, but also exhibit spatial and temporal locality. Functions nearly always comprise multiple consecutive instruction blocks, with many tight loops and local branches constrained within a relatively small number of instruction blocks. While prior proposals [6, 32], both for instruction and data prefetching, record temporally correlated stream addresses into a circular buffer one by one, we can take advantage of the locality between instruction blocks by representing them in a compact form, recording an address per spatial region (typically, a function) rather than per address and avoiding redundant storage of multiple iterations of tight loops.

3.1 Leveraging Locality in Instruction Streams

A function's instructions are likely to be stored in consecutive instruction cache blocks, executed sequentially until a long-distance discontinuity is encountered (e.g., a function call, a distant branch in a large function, or a system call). We explore the space savings of recording only a single record per *spatial region*, a group of spatially-adjacent instruction cache blocks. Each spatial region record has a *trigger* address (the first instruction's address accessed within the spatial region) and a bit vector, where each bit represents an adjacent cache block. We divide the bit vector in two parts, with the left part of the bit vector representing the blocks that precede the trigger and the right part representing the blocks that succeed the trigger.

We quantify the spatial region densities in Figure 3 (left). For each spatial region, we count only unique accesses to that region,

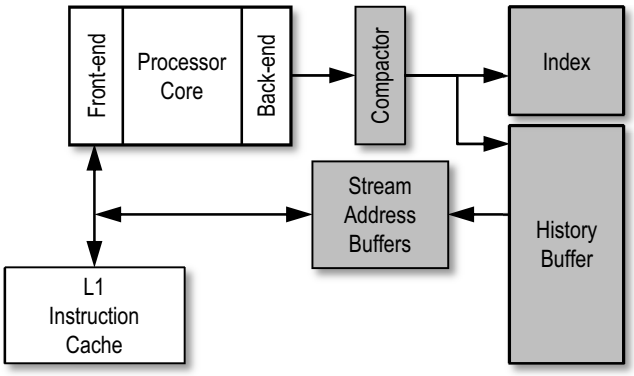

Figure 4. PIF hardware and data flow (single core)

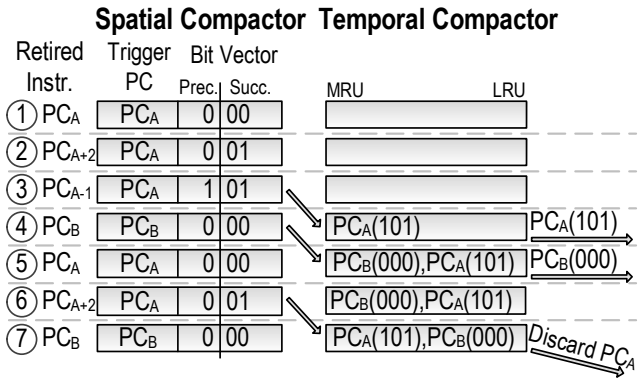

Figure 5. Compacting the instruction address sequence

avoiding over-counting due to small loops that fall within the boundaries of the spatial region. For all workloads studied, we find that more than 50% of the spatial regions have more than one block accessed, with many regions having up to 8 accessed instruction blocks. We conclude that considerable space saving opportunity exists for storing temporal streams of spatial region records rather than individual instruction addresses.

We note that spatial locality does not imply that only sequential blocks are accessed within a region. Gaps may exist due to conditional branching behavior such as never-executed error handling code. We quantify the number of discontinuous groups of sequential blocks inside spatial regions in Figure 3 (right). We find that approximately one fifth of all spatial regions observe discontinuous accesses. Although, in some of these cases (short forward jumps), an aggressive next-line prefetcher can correctly predict future accesses, this approach would considerably increase the number of unnecessarily fetched blocks and cache pollution.

Recording accesses as spatial regions introduces another advantage for stream space reduction by enabling compact storage of accesses exhibiting temporal locality. When a loop body fits into a single instruction block, the accesses to the block are recorded only once, regardless of the number of iterations of the loop. However, small loop bodies frequently span multiple consecutive instruction blocks within one region or span a small number of different regions (e.g., when a tight loop includes a call to a helper function). The instruction blocks comprising the loop are accessed on every iteration of the loop. By encoding accesses to regions and skipping regions that repeat in tight loops, unnecessary redundancy is avoided in the temporal stream storage, providing further space savings.

3.2 Effects of Space Reduction on Coverage

Compacting the instruction stream and avoiding storing repetition not only reduce the temporal stream storage, but also improve the repetitiveness and predictability of the temporal streams. In a traditional temporal stream, the iterative accesses are recorded unnecessarily, as they cannot improve cache-miss coverage because instruction blocks are brought into the cache on the first iteration of the loop and remain there until after the loop terminates. Moreover, if the number of loop iterations recorded in the temporal history does not match the subsequent encounter of the same loop, the temporal stream fails to predict blocks following the loop termination. Encoding loop blocks as a single record enables correct prediction of these blocks, as well as the blocks that follow the loop, regardless of the data-dependent number of loop iterations. Therefore, counter-intuitively, we find that compacting the storage actually leads to more repetitive streams, thus, higher coverage.

4 DESIGN

The Proactive Instruction Fetch mechanism is based on recording and replaying the retire-order instruction access streams to predict future correct-path instruction fetches. We base our design on the Global History Buffer [18] G/AC prefetcher. Although designing a new mechanism specifically suited to our observations and targeting instruction prefetch would enable a number of engineering optimizations, particularly with regard to reducing the predictor storage overheads, we instead adapt an existing design to isolate the benefits of avoiding microarchitectural noise, promoting a clearer understanding of our contributions in the context of prior prefetching literature [6, 18, 32]. For example, although storage benefits can be attained by sharing predictor structures among multiple cores or virtualizing the predictor storage in the L2 cache [4], we avoid these designs in favor of simplicity, simulating completely independent dedicated predictor hardware for each core.

Proactive Instruction Fetch introduces four hardware structures, shown shaded in Figure 4. The *Compactor* tracks retired instruction addresses, leveraging the spatial and temporal locality among instructions for compaction. The compacted records are stored in the *History Buffer*. The history buffer stores a continuous sequence of spatial regions, while the *Index* provides a fast search mechanism for instruction streams in the history buffer. The *Stream Address Buffers* read records from the history buffer and coordinate prefetch requests by monitoring the instruction cache accesses of the core's front-end.

4.1 Compacting the Retire-Order Sequence

The compactor monitors the back-end of the processor core and records the addresses of retiring instructions. However, storing individual PCs that flow out of the core would result in wasted space, because the history buffer is used to predict instruction block addresses rather than addresses of individual instructions. We therefore collapse all consecutively retired PCs belonging to the same instruction block into a single address. The block address of the retiring instruction is checked against the block address of the previous retired instruction; if the new retiring PC is in the same block, the new PC is discarded.

The compactor comprises the spatial and temporal compaction mechanisms whose operation is depicted in Figure 5. Taking

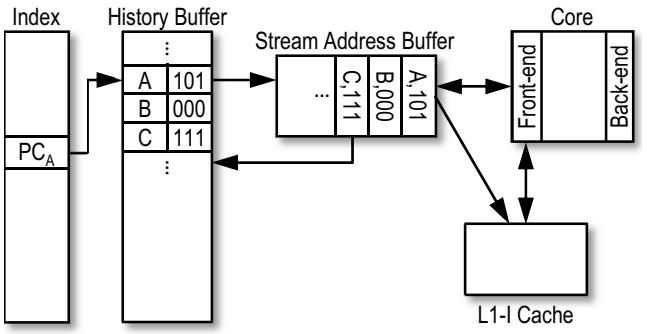

Figure 6. Predicting future instruction accesses

advantage of the spatial locality across instruction blocks, the spatial compactor combines instruction-block addresses that fall within a *spatial region*, a group of adjacent instruction blocks. The spatial compactor defines the boundaries of a spatial region according the first access within the region, which we call a trigger. The new spatial region comprises *N* blocks preceding the trigger and *M* blocks succeeding the trigger, corresponding to *N+M+1* instruction blocks in total. In the example in Figure 5, PC_A, an instruction in the block *A,* is the trigger of the new spatial region that spans one instruction block preceding the trigger instruction's block, *A-1*, and two instruction blocks succeeding the trigger instruction's block, *A+1* and *A+2*. When a new spatial region is encountered, the compactor defines the boundaries of the new region, records the trigger PC, and clears the bit vector where each bit represents an instruction block within the spatial region (step 1 in Figure 5). As instructions within the current spatial region retire, corresponding bits are set in the bit vector (step 2 and step 3). When an instruction outside the current spatial region retires, the existing spatial region record (trigger PC and bit vector) is sent to the temporal compactor and the spatial compactor starts monitoring the blocks within the new spatial region (step 4).

Tight code loops constitute a large fraction of the executed instructions, with the instruction footprint of the loops typically spanning several spatial regions. When an instruction prefetcher is employed, ideally, all instruction blocks of the loop are prefetched into the L1 instruction cache prior to the first loop iteration. There is no benefit from predicting the subsequent iterations because the instruction blocks have already been brought into the cache. To avoid recording the spatial region records that belong to such subsequent iterations, the temporal compactor tracks a small number of the most-recently-observed spatial region records. If an incoming spatial region record does not match any of the records in the temporal compactor (if the trigger PC is not found in the temporal compactor or if the incoming bit vector is not a subset of the bit vector of the spatial region record in the temporal compactor), the temporal compactor stores the evicted spatial region record, evicting the least-recently-used record, and sends the new record to the history buffer to be recorded and used for later predictions (step 4 and step 5). If a matching spatial region record is found in the temporal compactor, the spatial region record that comes from the spatial compactor is discarded and the corresponding record in the temporal compactor is promoted to the MRU position (step 7).

4.2 Learning the Instruction Sequences

The history buffer is a circular buffer that stores the sequence of retired instructions in FIFO order. A tail pointer determines the next location to be written in the history buffer. Each history buffer location stores the block address of a trigger instruction and the bit vector of the surrounding spatial region.

The index table, a small cache-like structure, facilitates fast search of the history buffer. The index table maintains a mapping between a trigger PC and the location of its most-recent record in the history buffer. Instructions that were not explicitly prefetched are tagged at the fetch stage and carry the tag through the pipeline and compactors. At the time of insertion of a spatial region record into the history buffer, if the trigger PC of the spatial region is tagged, the trigger PC (the trigger of the new stream) is also inserted into the index table pointing to the current tail pointer (the location at which new the spatial region record is being inserted). We note that only the index table insertion is conditional; an insertion into the history buffer is always performed independent of whether the region contains a tagged instruction.

4.3 Making Predictions for Future Accesses

If a sequence of instructions is executed once, the same sequence of instructions is likely to be executed again. Proactive Instruction Fetch predicts future instruction-cache accesses by replaying previously recorded instruction sequences as depicted in Figure 6. We identify a sequence of instructions by the trigger PC of the first spatial region. When the core issues an instruction fetch that was not prefetched, the prediction mechanism is triggered. The prediction mechanism searches for the PC of the accessed instruction in the index table. If a valid index table entry exists for the searched PC, a new active prediction stream is allocated.

Every active prediction stream is tracked by a *Stream Address Buffer* (SAB). An SAB tracks a window of consecutive spatial regions.[2] The SAB maintains a pointer to the sequence in the history buffer, initially set to the pointer taken from the index table lookup. The prediction mechanism reads the start of the sequence from the history buffer into the SAB. For each record, the SAB calculates the addresses of the instruction blocks that are encoded by the bit vector and issues prefetch requests for these addresses. Predictions are made by traversing the bit vector from left to right, as this typically predicts the accesses in the order they will be issued by the core.

Before queuing for prefetch, predictions first probe the instruction cache to confirm that the block is not present in the cache. As in prior work [27], we find that a line buffer between the core and the L1 instruction cache ensures ample bandwidth to the instruction cache tags for both the instruction-fetch and prefetch mechanisms without the need to duplicate the instruction-cache tags.

After issuing predictions from the initial window of the stream, the SAB monitors fetch requests from the core to the L1 instruction cache. L1 requests that fall within an active stream advance the

2. We maintain four SABs for concurrent active prediction streams, replacing the least-recently-used SAB when a new active stream is allocated. We empirically find that, for maximum performance with our core microarchitecture and cache latencies, the window in the SAB should track seven consecutive regions.

Table I.System and application parameters

Processing Nodes	UltraSPARC III ISA Sixteen 2 GHz OoO cores 3-wide dispatch / retirement 96-entry ROB, 64-entry LSQ TSO memory model
I-Fetch Unit	64KB, 2-way, 64B-block, 2-cycle load-to-use L1-I cache 24-entry pre-dispatch queue Hybrid branch predictor 16K gShare & 16K bimodal
L1D Caches	64KB, 2-way, 64B blocks, 2-cycle load-to-use, 2 ports, 32 MSHRs
L2 NUCA Cache	Unified, 512KB per core,16-way, 64B blocks, 16 banks, 15-cycle hit latency, 64 MSHRs
Main Memory	3 GB total memory, 45 ns access latency, 64B coherence unit
Interconnect	4x4 2D mesh

OLTP – Online Transaction Processing (TPC-C v3.0)	
DB2	*IBM DB2 v8 ESE,* 100 warehouses (10 GB), 64 clients, 2 GB buffer pool
Oracle	*Oracle 10g Enterprise Database Server,* 100 warehouses (10 GB), 16 clients, 1.4 GB SGA
Web Server (SPECweb99)	
Apache	*Apache HTTP Server v2.0,* 16K connections, fastCGI, worker threading model
Zeus	*Zeus Web Server v4.3,* 16K connections, fastCGI
DSS – Decision Support Systems (TPC-H)	
Qry 2, Qry 17	*IBM DB2 v8 ESE,* 480 MB buffer pool, 1GB database

SAB's history buffer pointer, reading subsequent records from the history buffer into the SAB and issuing prefetch requests for the corresponding cache blocks.

5 EVALUATION

We evaluate Proactive Instruction Fetch and compare it to the most-recently proposed temporal instruction prefetcher using trace-based and cycle-accurate full-system simulation of a 16-core CMP modeled in *Flexus* [33]. The details of the simulated architecture are listed in Table I (left). Flexus models the SPARC v9 instruction set architecture and is able to run unmodified server operating systems and applications. We simulate systems running the Solaris 8 operating system and executing the server workload suite described in Table I (right).

For the trace-based analyses, we use correct-path, in-order instruction reference traces. We collect traces of sixteen billion instructions (one billion per core) for the transaction processing and web serving workloads running in steady state. For the DSS workloads, we collect traces for the entire time of query execution. Our traces contain all application and operating system instructions, including hardware interrupt handlers.

For performance evaluation, we use the SimFlex multiprocessor sampling methodology [33]. Our samples are drawn over an interval of 10 to 30 seconds of simulated time for the OLTP and Web workloads and over the entire query execution for the DSS workloads. For each measurement, we launch simulation from checkpoints with warmed caches, instruction prefetcher tables, and branch predictors, and run 100,000 cycles to of detailed cycle-accurate simulation to warm queues and interconnect states before collecting measurements for the subsequent 50,000 cycles. As the performance metric, we use user instructions committed per cycle (UIPC), which is proportional to the overall system throughput [33]. We present speedups based on the average UIPC computed at a 95% confidence level with less than ±5% error.

5.1 The Need for Deep History Storage

To demonstrate the need for accurate storage of long temporal streams, Figure 7 presents the number of instruction block accesses between two occurrences of the same stream (the *jump distance)*, weighted by the number of correct predictions made by the corresponding stream. Short jump distances belong to frequently repeating streams, while long jump distances belong to rarely repeating streams. Although recent streams are used frequently to predict, we observe that medium-aged and old streams contribute to as many correct predictions as recent streams, highlighting the necessity for long temporal history capable of retaining and using old streams for prediction.

5.2 Optimal Spatial Region Size

Proactive Instruction Fetch uses bit vectors to encode accesses to adjacent blocks, after the trigger access within a spatial region occurs. The left portion of the bit vector represents the blocks before the trigger access and the right portion represents the blocks after the trigger access.

We measure the access frequencies to instruction blocks with different offsets from the trigger access in the spatial region.

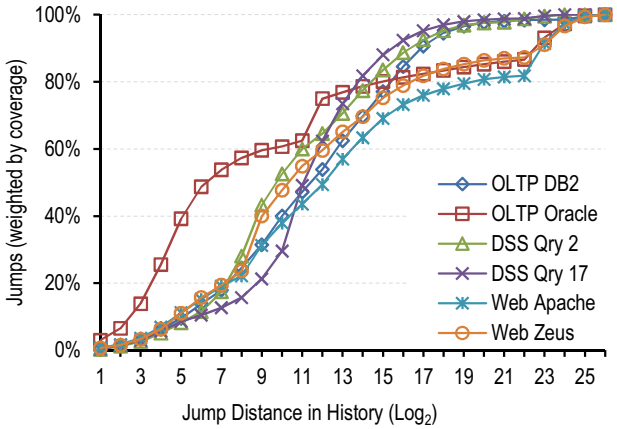

Figure 7. Weighted jump distance in history

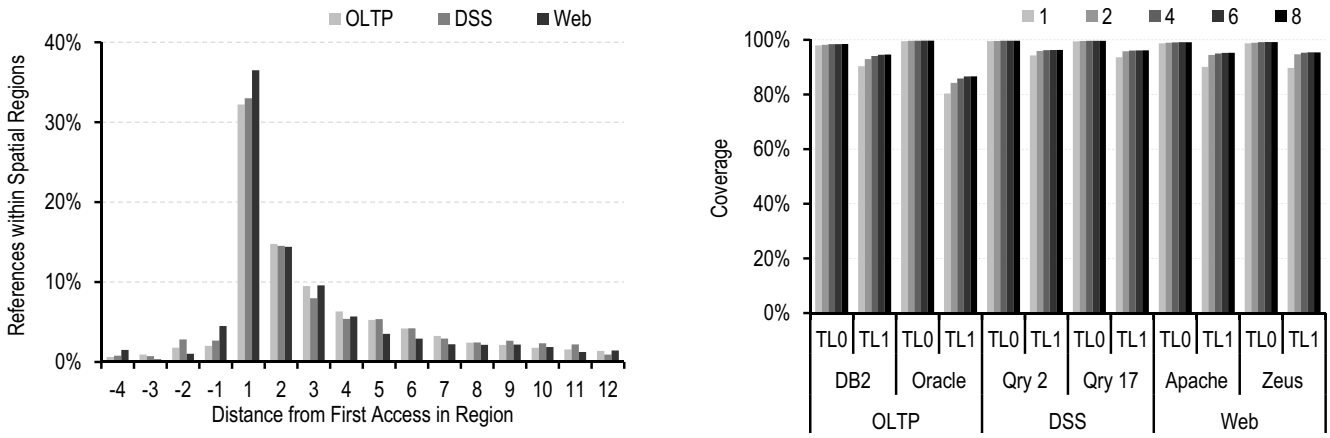

Figure 8. Distribution of accesses around the trigger block (left) and Spatial region size sensitivity (Trap Levels 0 and 1) (right)

Figure 8 (left) depicts the frequency of accesses to the blocks around the trigger accesses. As expected, the most frequently accessed blocks immediately follow the trigger access, with farther blocks not accessed as frequently. Most importantly, we see that there is a need to keep track of a small number of blocks before the trigger access, as backward jumps occur with significant frequency. Keeping two blocks preceding the trigger access in the same spatial region prevents storing additional entries in the temporal stream for backward jumps. Additionally, keeping the preceding block within the same spatial region allows correct prediction of subsequent blocks regardless of whether the backward jump is taken or not, as is the case when a function call is performed near the end of a loop body.

As expected, Figure 8 (left) shows that access frequency decreases significantly as the distance from the trigger block increases, indicating little benefit in storing spatial regions larger than 8 blocks because most of the bits would be zero. Additionally, we conclude that regions should be skewed toward blocks following the trigger access; we therefore consider only two blocks preceding the trigger as being part of the region.

Compacting storage with spatial regions not only reduces the temporal history space, but also increases coverage. Figure 8 (right) shows the predictor coverage as the region size is varied from one to eight. Trap-Level 0 (TL0) coverage is the coverage of application instruction misses, while Trap-Level 1 (TL1) coverage is the hardware interrupt coverage. We find that TL0 coverage increases slightly as the region size increases, whereas TL1 coverage improves significantly due to shorter sequences in the presence of very compact code with carefully crafted data-dependent jumps, often optimized to skip entire blocks of instructions to avoid instruction cache pollution.

5.3 Temporal Stream Lengths

The temporal prefetcher is likely to miss on the first access to each temporal stream, losing opportunity on the first access and potentially suffering from untimely prefetches for the subsequent one or two blocks within that stream. Temporal correlation therefore relies on long and repetitive streams that offer higher coverage and better timeliness, as prefetches along the temporal stream can be made farther in advance of demand, with the delay of bringing the first few blocks into the cache amortized over the stream length.

We analyze the temporal stream lengths and their contribution to correct predictions in Figure 9 (left). We see that temporal streams have highly variable lengths. More importantly, we note that medium and long streams contribute more to correct predictions than short streams. Because we analyze correct-path streams, avoiding the filtering effects of the instruction cache, separating interrupt handlers from regular program flow, and avoiding local control-flow ambiguities by grouping spatial regions, we see many streams comprising thousands of instruction blocks (despite the fact that spatial regions filter most of the local loop repetitions).

5.4 Sensitivity to Temporal Stream Storage

The prediction effectiveness depends directly on the ability of the history buffer to maintain as many repeating temporal streams as possible. Figure 9 (right) presents the predictor coverage as the history buffer size is varied. Coverage increases monotonically with the allotted storage. As an engineering trade-off, there is little justification for growing temporal stream storage beyond 32K regions. We note that prior work indicates (and we corroborate these findings) that the working sets of server applications are on the order of megabytes [8]. Although the history buffer that stores 32K regions consumes considerable chip real-estate, an additional level of instruction cache of the same capacity would offer practically no performance benefit, especially for OLTP workloads, and may actually harm performance due to the additional latency to probe an intermediate level of instruction cache.

It is important to note that, to eliminate the ambiguity introduced by the instruction cache, the results in Figure 9 (right) depict the predictor coverage, rather than the cache-miss coverage. In our design, temporal stream heads may be present in the instruction cache, thus not experiencing a cache miss on the first access to the temporal stream while still enabling correct prediction and prefetch of subsequent blocks. Therefore, unlike prior temporal streaming proposals, only a small fraction of the coverage loss experienced by the predictor is observed as cache misses.

5.5 Competitive Comparison

We compare the cache-miss coverage of Proactive Instruction Fetch with an aggressive next-line prefetcher and Temporal Instruction Fetch Streaming (TIFS), a state-of-the-art temporal instruction-fetch streaming proposal. To stress the fundamental difference in predictor effectiveness, we present the coverage of both techniques without history storage limitations, showing the maximum possible opportunity. Figure 10 (left) shows that PIF

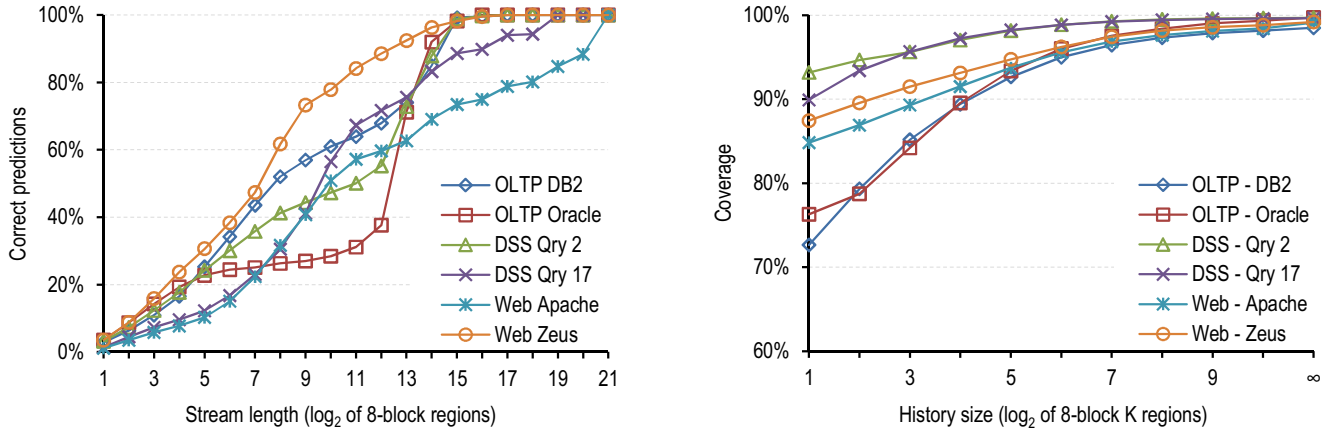

Figure 9. Temporal stream size contribution to prediction (left) and History size sensitivity (right)

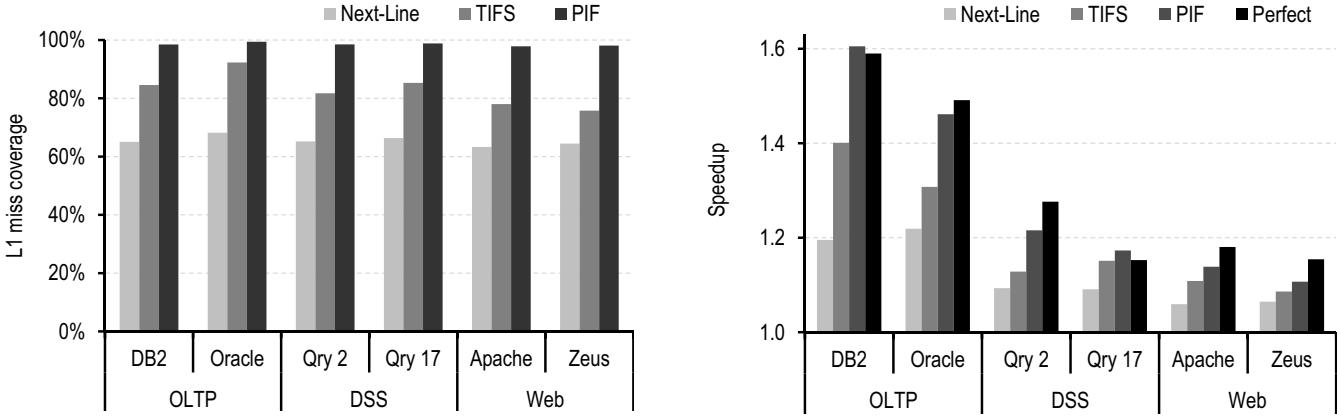

Figure 10. Competitive coverage comparison (left), competitive performance comparison (right)

has nearly perfect coverage across all workloads studied, while TIFS coverage is between 65-90%, corroborating prior work [6].

Importantly, the coverage shown for TIFS in Figure 10 (left) cannot be achieved in systems with branch prediction, as further coverage loss is experienced by TIFS due to the injection of wrong-path accesses into the temporal stream. Conversely, the coverage shown for PIF closely tracks the correct-path prediction coverage observed in cycle-accurate simulation.

5.6 Performance Improvement

We compare the Proactive Instruction Fetch performance against the next-line prefetcher and TIFS [6], and present the maximum speedup that can be achieved with a perfect-latency instruction cache.[3] Figure 10 (right) shows the performance improvements achieved by the corresponding mechanisms, normalized to the baseline case, where no instruction prefetching mechanism is employed.

The relative performance improvements of the instruction streaming mechanisms match the coverages in Figure 10 (left). As we expect, Proactive Instruction Fetch outperforms the next-line prefetcher and TIFS, converging to the performance of a perfect instruction cache. Compared to the next-line prefetcher, Proactive Instruction Fetch is able to prefetch the discontinuity points as well as the adjacent instruction blocks that are captured by the next-line prefetcher. Compared to TIFS, Proactive Instruction Fetch is more accurate for discontinuities, because it has a concrete instruction access history unfiltered by the instruction cache and without wrong-path noise injected by the branch predictor. Although the Proactive Instruction Fetch implementation we present is no more complex than the TIFS design, the Proactive Instruction Fetch design hides nearly all of the instruction access latencies from the core. As a result, Proactive Instruction Fetch achieves 27% performance improvement on average, nearly matching the 29% performance improvement of a perfect instruction cache. For two benchmarks, the PIF performance is marginally higher than the perfect-latency cache design; this is primarily due to the PIF design inducing less pressure on the on-chip interconnect compared to our perfect-latency cache configuration.

6 RELATED WORK

Instruction-fetch-related stalls are a key performance bottleneck for server workloads. Computer architects initially addressed this problem with simple next-line instruction prefetchers to take advantage of inherently sequential instruction accesses [2]. Subsequent next-line prefetcher designs expanded on this concept, issuing prefetch requests upon various conditions (e.g., upon an instruction access or miss) and observing varying degrees of prefetch depth and lookahead [21, 24, 25]. Proactive Instruction

3. The perfect-latency cache we simulate always returns the requested instruction block with the latency of a cache hit, with all other externally-observed behaviors of the cache matching the Next-Line configuration.

Fetch effectively predicts both the sequential accesses to instruction blocks and the control transfers that are beyond the reach of next-line prefetchers, while limiting over-predictions that arise with next-line prefetchers as they continue to issue sequential prefetches beyond the end of the accessed region.

A branch predictor running ahead of the fetch unit can predict control transfers by issuing requests for possible future instruction accesses [5, 19, 22, 23, 28]. An idle thread [15] or speculative threading mechanisms [30, 35] can be employed to generate future instruction accesses. Run-ahead execution prefetches the instruction blocks that will later be requested by the fetch unit [17]. Unlike the branch-predictor directed approaches, PIF relies on previously seen instruction streams, instead of relying on branch speculation. Proactive Instruction Fetch operates at instruction region granularity, rather than the granularity of each branch, offering much greater prediction accuracy and lookahead [6].

The discontinuity prefetcher [27] and TIFS [6] also address the lookahead limitations of the branch-predictor directed prefetchers by operating at instruction-block granularity and keeping the history of non-sequential transitions. However, the discontinuity prefetcher has a lookahead limitation of handling only one transition at a time to prevent gross over-prediction. On the contrary, Proactive Instruction Fetch maintains a complete history of accesses and has no lookahead limitation while following a temporal stream. Unlike TIFS, PIF maintains a clean and highly repetitive instruction-fetch history, unencumbered by microarchitectural effects such as instruction-cache filtering, injection of noise by the branch predictor, or the spontaneous occurrence of interrupt handlers. Furthermore, Proactive Instruction Fetch significantly improves on temporal storage efficiency and on predictor coverage by temporal streams using a compact stream representation rather than explicitly recording all addresses [6].

Many orthogonal software approaches have attempted to eliminate the instruction-fetch-related stalls by optimizing the application code for higher locality [10, 20, 34], employing the compiler to insert prefetch instructions [3], and performing call graph prefetching [16]. These approaches make the instruction sequence more regular and predictable, in some cases providing hints of future accesses to the hardware. The Proactive Instruction Fetch implementation can potentially benefit from these techniques to further improve coverage and reduce temporal stream storage.

7 CONCLUSIONS

We showed that microarchitectural components such as the instruction cache, the branch predictor, and hardware trap handlers disrupt the repetitive behavior of instruction streams. While the L1 cache filters the reference stream, the branch predictor and hardware interrupts inject wrong-path references, limiting the predictability of future instruction references. We demonstrated that the retire-order instruction stream, unfiltered by the instruction cache and unaffected by the branch predictor, can be used to correctly predict over 99.5% of the instruction fetches. In this work, we proposed Proactive Instruction Fetch, a hardware mechanism that efficiently records correct-path instruction fetch streams and replays them to predict future instruction accesses, hiding instruction-fetch stalls from the core. Through cycle-accurate simulation of a 16-core CMP running server workloads, we demonstrated that Proactive Instruction Fetch outperforms a next-line prefetcher and TIFS, a state-of-the-art temporal instruction prefetcher, improving the performance on our server benchmark suite by 27% on average, and converging to the performance of a perfect cache.

Acknowledgements

The authors would like to thank the members of PARSA at EPFL and the anonymous reviewers for their feedback on drafts of this paper. This work was partially supported by grants from Intel Corporation and the Swiss National Science Foundation.

References

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

[2] D. W. Anderson, F. J. Sparacio, and Robert 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] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Call graph prefetching for database applications. *ACM Transactions on Computer Systems*, 21(4):412–444, 2003.

[4] Ioana Burcea, Stephen Somogyi, Andreas Moshovos, and Babak Falsafi. Predictor virtualization. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2008.

[5] I-Cheng K. Chen, Chih-Chieh Lee, and Trevor N. Mudge. Instruction prefetching using branch prediction information. In *Proceedings of the International Conference on Computer Design*, October 1997.

[6] Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal instruction fetch streaming. In *Proceedings of the 41st International Symposium on Microarchitecture*, December 2008.

[7] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, November 1992.

[8] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, June 2009.

[9] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *Proceedings of the 3rd Conference on Innovative Data Systems Research*, January 2007.

[10] Stavros Harizopoulos and Anastassia Ailamaki. STEPS towards cache-resident transaction processing. In *Proceedings of the 30th International Conference on Very Large Databases*, August 2004.

[11] Aamer Jaleel and Bruce L. Jacob. In-line interrupt handling for software-managed TLBs. In *Proceedings of the International Conference on Computer Design*, September 2001.

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

[13] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance characterization of a Quad Pentium Pro SMP using OLTP workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[14] Jack L. Lo, Luiz Andre Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[15] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th International Symposium on Computer Architecture*, June 2001.

[16] Chi-Keung Luk and Todd C. Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, December 1998.

[17] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An effective alternative to large instruction windows. *IEEE Micro*, 23(6):20–25, Nov.-Dec. 2003.

[18] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture*, February 2004.

[19] Jim Pierce and Trevor Mudge. Wrong-path instruction prefetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996.

[20] Alex Ramirez, Luiz Andre Barroso, Kourosh Gharachorloo, Robert Cohn, Josep Larriba-Pey, P. Geoffrey Lowney, and Mateo Valero. Code layout optimizations for transaction processing workloads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, June 2001.

[21] Alex Ramirez, Oliverio J. Santana, Josep L. Larriba-Pey, and Mateo Valero. Fetching instruction streams. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, December 2002.

[22] Glenn Reinman, Brad Calder, and Todd Austin. Fetch directed instruction prefetching. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, December 1999.

[23] Glenn Reinman, Brad Calder, and Todd M. Austin. Optimizations enabled by a decoupled front-end architecture. *IEEE Transactions on Computers*, 50(4):338–355, 2001.

[24] Oliverio J. Santana, Alex Ramirez, and Mateo Valero. Enlarging instruction streams. *IEEE Transactions on Computers*, 56(10):1342–1357, 2007.

[25] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, 1978.

[26] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, June 2009.

[27] Lawrence Spracklen, Yuan Chou, and Santosh G. Abraham. Effective instruction prefetching in chip multiprocessors for modern commercial applications. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, February 2005.

[28] Viji Srinivasan, Edward S. Davidson, Gary S. Tyson, Mark J. Charney, and Thomas R. Puzak. Branch history guided instruction prefetching. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, January 2001.

[29] Robert Stets, Kourosh Gharachorloo, and Luiz Andre Barroso. A detailed comparison of two transaction processing workloads. In *Proceedings of the International Workshop on Workload Characterization*, November 2002.

[30] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

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

[32] Thomas F. Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Temporal streaming of shared memory. In *Proceedings of the 32nd International Symposium on Computer Architecture*, June 2005.

[33] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, July-Aug. 2006.

[34] Jingren Zhou and Kenneth A. Ross. Buffering database operations for enhanced instruction cache performance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 2004.

[35] Craig B. Zilles and Gurindar S. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th International Symposium on Computer Architecture*, June 2001.