RDIP: Return-address-stack Directed Instruction Prefetching

Aasheesh Kolli
University of Michigan
akolli@umich.edu

Ali Saidi
ARM
ali.saidi@arm.com

Thomas F. Wenisch
University of Michigan
twenisch@umich.edu

ABSTRACT

L1 instruction fetch misses remain a critical performance bottleneck, accounting for up to 40% slowdowns in server applications. Whereas instruction footprints typically fit within last-level caches, they overwhelm L1 caches, whose capacity is limited by latency constraints. Past work has shown that server application instruction miss sequences are highly repetitive. By recording, indexing, and prefetching according to these sequences, nearly all L1 instruction misses can be eliminated. However, existing schemes require impractical storage and considerable complexity to correct for minor control-flow variations that disrupt sequences.

In this work, we simplify and reduce the energy requirements of accurate instruction prefetching via two observations: (1) program context as captured in the call stack correlates strongly with L1 instruction misses, and (2) the return address stack (RAS), already present in all high performance processors, succinctly summarizes program context. We propose RAS-Directed Instruction Prefetching (RDIP), which associates prefetch operations with signatures formed from the contents of the RAS. RDIP achieves 70% of the potential speedup of an ideal L1 cache, outperforms a prefetcherless baseline by 11.5% and reduces energy and complexity relative to sequence-based prefetching. RDIP's performance is within 2% of the state-of-the-art Proactive Instruction Fetch, with nearly 3X reduction in storage and 1.9X reduction in energy overheads.

Categories and Subject Descriptors

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

General Terms

Design, Performance

Keywords

RAS, caching, instruction fetch, prefetching, accuracy

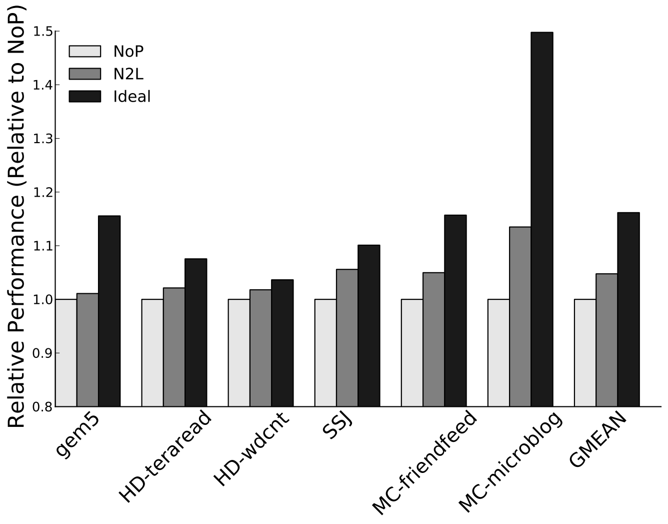


Figure 1: Speedup under next-2-line (N2L) and an idealized instruction cache (Ideal) normalized to a cache without prefetching (NoP).

1. INTRODUCTION

Recent research shows that L1 instruction fetch misses remain a critical performance bottleneck in traditional server workloads [9, 8, 11, 12, 32], cloud computing workloads [7, 19], and even smartphone applications [10], accounting for up to 40% slowdowns. Whereas instruction footprints typically fit comfortably within last-level caches, they overwhelm L1 caches, whose capacity is tightly limited by latency constraints. Intermediate instruction caches may reduce miss penalties, but are either too small to capture the entire instruction footprint or have high access latencies, which are then exposed on the execution critical path [7, 11]. Unlike L1 data misses, out of order mechanisms are ineffective in hiding instruction miss penalties. Figure 1 shows the performance gap between a 32kB L1 cache without prefetching ("NoP"), the same cache with a conventional next-2-line prefetcher ("N2L"), and an idealized (unbounded size, but single-cycle latency; "Ideal") cache for a range of cloud computing and server applications. Instruction prefetching has the potential to improve performance by an average of 16.2%, but next-line prefetching falls well short of this potential. (For a brief description of the workloads and corresponding performance metrics, see Section 4).

The importance of instruction prefetching has long been recognized. Nearly all shipping processors, from embedded- to server-class systems, include some form of next-line or stream-based instruction prefetcher [13, 14, 26, 30, 31, 37]. Such prefetchers are simple and require negligible storage, but fail to prefetch across fetch discontinuities (e.g., function calls and returns), which incur numerous stalls. Early research attempts at more sophisticated instruction prefetch leverage the branch predictor to run ahead of fetch [5, 24, 27, 28, 33, 36]. However, poor branch predictability and insufficient lookahead limit the effectiveness of these designs [9]. To our knowledge, such instruction prefetchers have never been commercially deployed.

More recent hardware prefetching proposals rely on the observation that instruction cache miss or instruction execution sequences are highly repetitive [8, 9]. These designs log the miss/execution streams in a large circular buffer and maintain an index on this buffer to locate and replay past sequences on subsequent triggers. The most recent proposal, Proactive Instruction Fetch (PIF) [8], can eliminate nearly all L1 instruction misses, but requires impractical storage and substantial complexity to correct for minor control-flow variations that disrupt otherwise-repetitive sequences. Storage (and hence, energy) requirements grow rapidly with code footprint because these mechanisms log all instruction-block fetches and must maintain an index by block address.

In this work, we seek to simplify and reduce the storage and energy requirements of accurate hardware instruction prefetching. We exploit the observation that program context, as captured in the call stack, correlates strongly with L1 instruction misses. Moreover, we note that modern high-performance processors already contain a structure that succinctly summarizes program context: the return address stack (RAS). Upon each call or return operation, *RAS-Directed Instruction Prefetching (RDIP)* encodes the RAS state into a compact signature comprising the active call stack, and direction/destination of the current call or return. We find a strong correlation between these signatures and L1 instruction misses, and associate each signature with a sequence of misses recently seen for that signature. By generating signatures on both calls and returns, we construct signatures with fine-grained context information that can prefetch not only on traversals down the call graph, but can also prefetch accurately within large functions and when deep call hierarchies unwind—a key advantage over past hardware and software prefetching schemes that exploit caller-callee relationships [2, 20]. Furthermore, we find that RAS signatures are highly predictable—current signatures accurately predict upcoming signatures, enabling higher prefetch lookahead.

RDIP reduces area and overheads relative to past sequence-based prefetchers because it records less state and requires simpler indexing; we find that coverage saturates with only 64kB of storage (relative to over 200kB for [8]). Using the gem5 [4] full-system simulation infrastructure running a suite of cloud computing and server applications, we show that RDIP achieves a performance improvement of up to 36% (avg 11.5%) over a prefetcher-less design, as compared to 13.5% (avg 4.8%) for a carefully tuned next-line prefetcher and 40.1% (avg 12.9%) for PIF. PIF incurs over 3X higher storage overhead and 1.9X dynamic energy overhead than RDIP.

2. RELATED WORK

Due to the performance criticality of instruction cache misses, there is a rich body of prior work on instruction prefetching. Even early computer systems included next-line instruction prefetchers to exploit the common case of sequential instruction fetch [1]. This early concept evolved into next-N-line and instruction stream prefetchers [14, 26, 30, 37], which make use of a variety of trigger events and control mechanisms to modulate the aggressiveness and lookahead of the prefetcher. However, next-line prefetchers provide poor accuracy and lack prefetch lookahead for codes with frequent branching and function calls. Nevertheless, because of their simplicity, next-line and stream prefetchers are widely deployed in industrial designs (e.g., [13]). To our knowledge, more sophisticated hardware prefetchers proposed in the literature have never been deployed.

To improve prefetch accuracy and lookahead in branch- and call-heavy code, researchers have proposed several branch predictor based prefetchers [5, 27, 28, 33]. Run-ahead execution [22], wrong path instruction prefetching [24], and speculative threading mechanisms [34, 39] also prefetch instructions by using control flow speculation to explore ahead of the instruction fetch unit. As shown by Ferdman *et al.* [8], these prefetchers suffer from interference caused by wrong path execution and insufficient lookahead when the branch predictor traverses loop branches. RDIP instead bases its predictions on non-speculative RAS state, which is more stable. It is important to note that execution-path based correlation has been shown to be effective for branch prediction [23], dead-block prediction [17] and last touch prediction [16], but none of these leverage the RAS to make their respective predictions.

The discontinuity prefetcher [32] handles fetch discontinuities but its lookahead is limited to one fetch discontinuity at a time to prevent run-away growth in the number of prefetch candidates. The branch history guided prefetcher (BHGP) [33] keeps track of the branch instructions executed and near future instruction cache misses, and prefetches them upon next occurrence of the branch. BHGP cannot differentiate among invocations of the same branch, which results in either unnecessary prefetches or reduced coverage. In RDIP, by using the RAS (instead of a single entry) to make prefetching decisions, different invocations of a function call are uniquely identified leading to more accurate prefetching.

TIFS [9] and PIF [8] address the limitations of branch-predictor-directed prefetching by directly recording the instruction fetch miss and instruction commit sequences, respectively. Hence, their accuracy and lookahead are independent of the predictability of individual branches. Both designs maintain an ordered log of instruction block addresses and an index that maps particular fetch events (called *trigger accesses*) to locations in the log. Whenever a trigger access results in a hit in the index, the prefetcher begins issuing prefetch requests according to the recorded history in the log. PIF improves over TIFS by recording the committed instruction sequence, and hence is immune to disruptions from wrong-path execution that harm the accuracy and coverage of TIFS. To similarly avoid disruptions from wrong-path calls and returns, RDIP updates its signatures as call and return instructions commit (rather than at fetch as for a conventional RAS), which substantially improves its ability to use current signatures to predict future signatures.

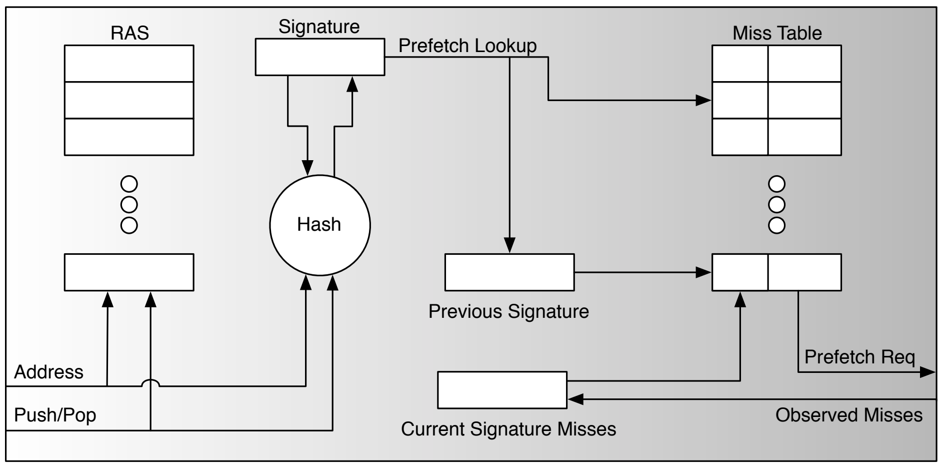


Figure 2: Prefetcher Design.

Both TIFS and PIF require considerable on-chip storage, which in turn leads to high energy requirements. TIFS adds a 15-bit overhead to every L2 tag array entry to maintain its index (totaling to 240kB in an 8MB L2) and requires an additional 156kB (also within the L2) to store its instruction miss log. Hence, each access to prefetcher meta-data incurs up to two L2 accesses, at a considerable cost in energy. PIF employs dedicated storage structures, but even those structures total over 200kB, and are accessed frequently. In contrast, as we show later, RDIP requires only 64kB of dedicated storage. We analyze energy overheads of each approach in greater detail in Section 5.4.3.

SHIFT [15] is a concurrent work that employs a similar record and replay mechanism as PIF, while reducing storage overhead per core. The key insight behind SHIFT is that different cores in a multi-core server system usually execute similar instruction sequences. By adopting shared prefetcher-related storage for the different cores, SHIFT reduces overall storage overhead per core. In contrast, RDIP does not rely on the assumption that neighboring cores execute the same programs, and is more storage efficient when a server system allocates less than 4 cores per workload.

Some prior hardware-software hybrid prefetchers track function caller-callee relationships and initiate next-N-line instruction prefetching for a function's callees upon entry to the caller [2, 20]. Like RDIP, these techniques exploit relationships in a program's control flow graph. However, RDIP uses additional RAS history, rather than just its leaf, which allows it to distinguish multiple invocations of the same functions from different call sites, improving accuracy. Moreover, RDIP prefetches on traversals both up and down the call stack, enabling timely and accurate prefetch when control returns into the body of a large function.

Other software techniques involve relocating infrequently executed code to increase sequential code execution [12, 25, 38] or compiler-inserted instruction prefetches (e.g., [20, 21]). These techniques are orthogonal to RDIP and can provide synergistic benefits.

3. DESIGN

We next explain the design and rationale for RDIP, which is based on three key observations:

- Instruction cache misses correlate strongly to program context; that is, the same misses recur each time a context is revisited.
- Program call graphs follow repetitive patterns, hence a current program context can be used to accurately predict upcoming program contexts.
- The state of the return address stack succinctly summarizes program context.

The current call stack accurately reflects the program execution path taken to reach a particular point in the code. The contents of the instruction cache are determined by the instruction fetches encountered along this execution path. So, both the initial cache state upon entering a particular context and the misses incurred during that context will tend to be similar to prior invocations of the same context. The central idea of RDIP is to record the instruction cache misses seen during a particular program context and prefetch these upon the next occurrence of the same program context.

However, issuing prefetch requests upon entering a program context (i.e., after executing the call instruction that branches into a function) is too late—the processor will likely stall on a miss to the call's target. Instead, prefetch requests arising during a particular program context must be issued during a preceding context. Fortunately, the call graphs of server applications tend to be predictable and hence near-future contexts can be predicted accurately, providing adequate lookahead.

RDIP operates in three steps:

1. Record the instruction cache misses seen during a particular program context.
2. Predict a future program context based on the current program context.
3. Prefetch cache misses correlated to the future program context.

Throughout our description, we refer to Figure 2, which depicts the RDIP design.

PC	Type	RAS	CurSig	PrevSig	I$-Miss?	Description
0x400	∗	0x048,	0x048|0	0x0|0		Initial state including call from 0x048
0x404	∗	0x048,				
0x408	∗	0x048,			YES	Log Miss: Sig=0x0|0, Addr=0x408
0x40C	∗	0x048,				
0x410	CALL	0x048,	0x414⊕0x048|0	0x048|0		Function Call
0x100	∗	0x048,0x414			YES	Log Miss: Sig=0x048|0, Addr = 0x100
0x104	RET	0x048,0x414	0x414⊕0x048|1	0x414⊕0x048|0		Function Return
0x414	∗	0x048,				
0x418	∗	0x048,			YES	Log Miss: Sig=0x414⊕0x048|0, Addr = 0x418
0x420	∗	0x048,				

Figure 3: Example of RDIP Operation.

3.1 Program Context Representation

The active program context is reflected by the current content of the RAS. As the number of bits in the RAS is large and contains a great deal of redundancy, we can compact it into a smaller representation through hashing. In our implementation we simply XOR the bit-string representing the RAS state to a desired bit width (we find that 32 bits suffice). We call the resulting hash value a *signature*. Because we use a commutative operation, signatures can maintained in a single register and updated incrementally in response to RAS pushes, pops, and overflows.

The call stack reflected in the RAS, and its corresponding signature, is an indication of the execution path taken to reach a particular point in the program, which in turn will determine the content of the L1 instruction cache. However, when a single caller function invokes many callees, upon return from each callee, the RAS state (and signature) is identical, but subsequent instructions will differ as they correspond to different segments of the caller. While each segment could suffer different instruction cache misses, they all associate with the same signature.

To distinguish these return sites, we form signatures before processing return instructions (i.e., before popping the RAS) but after call instructions (i.e., after pushing the RAS) and further extend the signature with a bit to indicate the direction of the control transfer (call or return). Hence, signatures always make use of all available information, and RDIP triggers prefetches on traversals both up and down the call stack.

3.2 Logging Misses

The *Miss Table* records the association between a signature and a set of instruction cache misses that have been recently observed for that signature. It is a set-associative structure accessed by signature, with least-recently-used replacement. During execution, instruction cache miss addresses are recorded in a circular buffer, the *Current Signature Misses* buffer. Whenever a call or return instruction is retired, and therefore the active RAS signature changes, we first record the misses logged in the Current Signature Misses buffer into an appropriate entry in the Miss Table (we discuss the signature to which these misses should be associated in the next subsection). We then clear the Current Signature Misses buffer. The Current Signature Misses buffer is organized as a circular buffer, hence, misses will be discarded if the buffer overflows.

To reduce the size of miss table entries, we first compress the sequence of misses using a scheme similar to that proposed for PIF [8]. The key observation is that the misses associated with a particular signature usually comprise a small number of contiguous blocks separated by discontinuities due to local control flow. For each nearby region of blocks, we record a single base address and then a bit vector indicating the other nearby blocks (at higher and lower addresses) that should also be prefetched. We provision storage for two (or more) such regions per signature, to account for a larger discontinuity within a program context. We examine sensitivity to the bit vector encoding and number of regions in Section 5.

When updating an existing miss table entry, we merge newly recorded misses with the existing encoding of regions when possible (i.e., setting additional bits for an existing base address), replacing regions in round-robin fashion when merging is impossible (e.g., if the newly recorded misses are not near the previously recorded regions).

3.3 Program Context Prediction

Nominally, we might associate misses with the active signature when those misses were observed. However, under such a design, when we later try to use the Miss Table to predict future misses, our prefetches will not be timely; the interval between the signature change (call or return) and the misses is too short to hide their latency. Instead, when we log misses, we record them with a *Previous Signature*, as shown in Figure 2. By maintaining a FIFO queue of previous signatures, rather than a single previous signature, we can increase RDIP's prefetch lookahead. We examine sensitivity to the number of signatures of lookahead in our evaluation. We will show that a lookahead of one signature is sufficient to assure timely prefetch.

Alternatively, one might design RDIP with separate signature and miss prediction tables (as in two-level branch predictors), which would allow signature lookahead to be varied dynamically at run time. However, such a design requires far more storage for the additional signature table, hence, we do not consider it further.

Upon every signature change, RDIP consults the miss table. If it finds a match, it issues the prefetch requests indicated in the table entry. It is important to note that accessing the Miss Table and issuing prefetch requests lie off the critical path of instruction fetch and hence the hardware structures can be pipelined to meet cycle time constraints. Pipelining the Miss Table delays the issue of prefetches by up to a few cycles. However, RDIP provides sufficient lookahead to hide small increases in prefetch issue latency.

3.4 Example

Figure 3 shows an example of how RDIP constructs signatures and logs misses. Each row of the table represents the execution of a single instruction. *PC* indicates the instruction PC, while *Type* indicates the type of instruction (Call, Return, or other). *RAS* shows the state of a 2-entry RAS, while *Signature* shows a representation of the new signature value each time the signature changes. The signatures in the table are shown as a list of return addresses, separated by $\oplus$ operators, followed by a 0 indicating a call or 1 indicating a return operation caused the signature change. *ICache Miss* indicates if a miss is incurred when fetching a particular instruction, while the *Description* describes the actions taken by RDIP.

Initially, the RAS has a single valid return address (0x048), execution begins at address 0x400, and the initial active signature is (0x048|0), while the previous signature (not shown) is (0x0|0). When the instruction at address 0x408 is fetched, an instruction cache miss occurs, and address 0x408 is logged. The call instruction (0x410) causes the active signature to change to (0x414$\oplus$0x48|0), and the logged misses are recorded in the signature table in the entry for the previous signature (0x0|0). We again log a miss for instruction 0x100. Upon execution of the return instruction (0x104), the active signature again changes, to (0x414$\oplus$0x48|1). As before, the logged misses are entered into the miss table and associated with the previous signature (0x048|0).

4. METHODOLOGY

We next describe the simulation infrastructure and workloads we use to investigate RDIP.

4.1 Simulation Infrastructure

We use the gem5 [4] simulation infrastructure for all of our analysis. All results were obtained by simulating an out-of-order core executing the ARM instruction set, the configuration details of which can be found in Table 1. Our prefetcher was evaluated using a combination of trace-based studies for the sensitivity analyses and detailed full-system simulation for performance studies. In all cases, we capture a checkpoint once steady state behavior has been reached. From this checkpoint, we either generate an instruction fetch trace for offline analysis or run performance simulations with a detailed CPU model and memory system.

For our trace-based studies, we collect execution traces of one billion instructions for each benchmark. We analyze these traces to guide the final design of RDIP. In particular, we use this trace-based methodology to tune the size of the miss table, choose a hashing algorithm for the signature compression, and experiment with techniques for compressing a sequence of misses. We validate these design decisions using execution-driven full-system simulation.

All performance results derive from full-system simulation. We launch the simulation from a checkpoint and simulate two billion instructions. We report the metric used to measure performance for each workload in the descriptions of the workloads.

We obtain energy-per-access, static power, and access time estimates for RDIP's miss table and structures used in other prefetchers using CACTI 5.3 [35].

Core	2GHz OoO 6-wide Dispatch, 8-wide Commit 96-entry ROB, 16-entry RAS 32-entry Issue Queue 12-entry Skid Buffer
I-Cache	32kB, 2-way, 64B 1ns cycle hit latency, 4 MSHRs
D-Cache	64kB, 2-way, 64B 2ns hit latency, 6 MSHRs
L2-Cache	2MB, 8-way, 64B 12ns hit latency, 16 MSHRs
Main Memory	2GB, 54ns access latency

Table 1: Simulator Configuration.

4.2 Workloads

We study a number of server and cloud workloads that have large instruction footprints. Several of our workloads use operating system services intensively, hence, we use full-system simulation and run our workloads on Ubuntu 12.04 (Linux kernel v3.3). Unlike oft-used user-level CPU-intensive benchmark suites (e.g., SPEC), our workloads comprise many thousands of code lines overwhelming L1 instruction caches. Descriptions of each follow:

gem5. To investigate a design automation workload, we simulate the gem5 simulator running within itself. The simulated gem5 instance is simulating the twolf benchmark from the SPEC2000 suite, using a simple CPU model and a timing-accurate memory system. The gem5 simulator is a large C++ program with over 200k lines of code that makes extensive use of virtual function calls. The code size coupled with pervasive virtual function indirection renders simple prefetchers ineffective. The performance metric is IPC when simulating gem5 (which is, in turn, simulating twolf).

Hadoop. To analyze massive amounts of unstructured data, companies have turned to MapReduce implementations such as Apache's Hadoop. Hadoop is a shared-nothing parallel processing framework designed to run on scale-out systems and its use over the last few years has exploded. For our simulations we use an in-house proxy for hadoop (written in Java), and carrie out the following Hadoop mapping tasks:

- *Teraread (hdtr):* This hadoop workload reads through terasort input data and writes it back to a different part of the filesystem. This workload represents small mapping tasks and exercises the I/O sub-system and associated APIs.
- *Word Count (hdwc):* Word Count builds a map of words to counts found in the input corpus, useful for building indexes.

The performance metric for Hadoop is the rate at which data is processed and written back to the file system.

Memcached. In memory key-value stores such as memcached are widely used by internet services operators to enable scalable web services. Memcached clusters reduce pressure on back-end databases and drastically reduce the average latency to retrieve cached data. Memcached clusters generally serve as a best-effort cache for a database backing store, as memcached does not offer the same resiliency guarantees provided by a database. Facebook reports using memcached clusters of over 800 servers that cache over 28 TB of data [29]. The behavior of a memcached server varies significantly with the kind of traffic it services, in particular the distribution of requests sent by clients and the size distribution of the cached objects [3]. Note that, although memcached itself contains relatively little code (under 10,000 lines), it makes extensive use of operating system services and the networking stack, resulting in a kernel instruction footprint that overwhelms the L1 cache [18]. We simulate two different workloads running on top of Memcached:

- *MicroBlog (mcmb):* This workload represents queries for short snippets of text (e.g., user status updates). We base the object size and popularity distribution on a sample of "tweets" (brief messages shared between Twitter users) collected from Twitter. The text of a tweet is restricted to 140 characters, however, associated meta-data brings the average object size to 1kB with little variance. Even the largest tweet objects are under 2.5kB in size.
- *FriendFeed (mcff):* Our second workload demonstrates heavy use of MULTI-GET requests. Facebook has disclosed that MULTI-GETs play a central role in its use of memcached [6]. This workload seeks to emulate the requests required to construct a user's Facebook Wall, a list of a user's friends' most recent posts and activity. We use the distribution of requests from the MicroBlog workload, as Facebook status updates and tweets have similar size and popularity characteristics [3]. However, instead of issuing single GET requests individually, a group of roughly 100 GETs is batched together to approximate a user having as many friends. MULTI-GETs improve efficiency because they incur fewer trips through the TCP/IP stack, which reduces pressure on the ICache.

We quantify performance by reporting the number of requests per second the server achieves when offered a load that saturates the CPU (no idle time), but does not saturate the network interface.

SSJ. Server Side Java (SSJ) tests Java performance in the SPECpower benchmark, and is similar to the SPECjbb benchmark. The benchmark is written in Java and runs on top of a commercial JVM. The JVM we study performs just-in-time compilation and aggressively inlines method invocations when generating native code sequences in its code cache. Hence, the JIT process flattens much of the control flow hierarchy present in the original Java code, with control transfers through branches and jumps rather than calls and returns. As such, this workload represents a particularly challenging case for RDIP. The workload contains a mix of six different transaction types and we quantify performance by measuring the transaction completion rate.

5. RESULTS

We next evaluate the effectiveness and impact of RDIP relative to existing prefetching approaches and the potential gains from an ideal instruction cache. First, we validate the key premises underlying RDIP. Then, we perform sensitivity studies to size the hardware structures in RDIP. Next, we show the coverage and performance achieved by RDIP and contrast them with the state-of-the-art PIF prefetcher and a basic Next-2-Line prefetcher. We finally compare our storage and energy overheads against PIF.

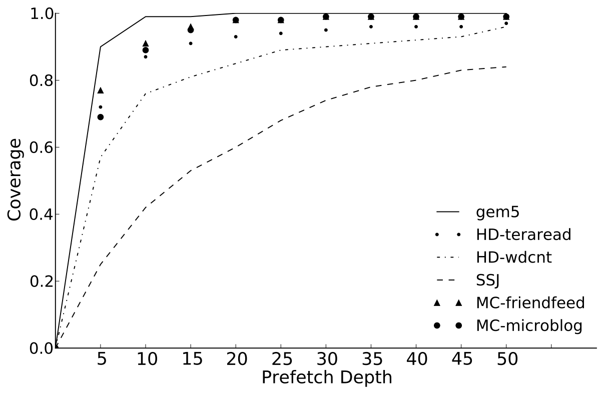


Figure 4: Coverage vs. Prefetch Depth. As the number of past misses (and correspondingly, number of prefetches) associated with each signature increases, coverage improves, saturating between 10 and 20 for nearly all workloads.

5.1 Prefetch Accuracy & Coverage

We begin by validating the key premises on which RDIP rests, namely that instruction misses correlate strongly to signatures and that signatures themselves are predictable.

5.1.1 Potential of Signature Based Prefetching

The key premise of RDIP is that the set of instruction cache misses incurred each time a signature recurs is repetitive. This phenomenon allows us to record previously seen misses for a signature and prefetch them upon the next occurrence of the signature, thus eliminating potential misses. The intuition underlying this premise is that the signature is an indication both of the past control-flow, which determines the L1 instruction cache content, and upcoming control flow, which determines the set of cache blocks that will be needed in the near future.

We examine this key issue in Figure 4. The graph shows, for all of our benchmarks, potential RDIP coverage as a function of the history depth. Here, we define coverage as the fraction of misses incurred while a signature is active that appear within the previously recorded history for that signature. For this experiment, we do not restrict the storage in the Miss Table, rather, we assume an unbounded number of entries and maintain up to 50 distinct miss addresses per signature, updated in an LRU fashion. Figure 4 shows that recording as few as 10 cache block addresses achieves nearly 80% coverage for most benchmarks. These results demonstrate that, with the notable exception of SSJ, our main premise holds—misses are strongly correlated to signatures, and RDIP has the potential to eliminate most of the instruction cache misses. We explore the Miss Table entry organization in greater detail in subsequent sections.

In SSJ, the impact of the JVM's just-in-time compilation can be seen: far more misses must be tracked for each signature to achieve high coverage. Because of inlining and loop unrolling, the JIT tends to generate long functions without

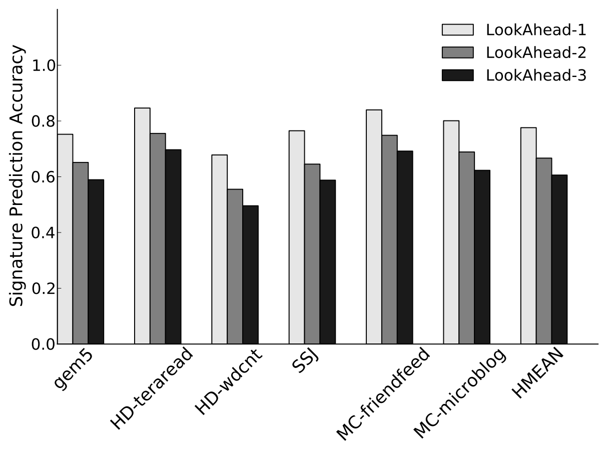


Figure 5: Future Signature Prediction Accuracy: The accuracy with which future signatures can be predicted given the current signature. LookAhead-X indicates a prediction X signatures into the future.

internal calls and returns. Control is returned to the JVM upon a method invocation from within generated code. The RAS state therefore reflects the state of the JVM, rather than the executing Java program, when control is transferred into the code cache. Hence, the control flow structure of the original Java code is obscured. Nevertheless, RDIP still has the potential to eliminate about half of instruction cache misses within practical storage constraints.

5.1.2 *Signature Prediction*

The second premise of RDIP is that the current RAS signature effectively predicts upcoming signatures. That is, that a program's call graph is repetitive. It is this property that enables RDIP to achieve sufficient lookahead to hide prefetch latency. Inaccurate signature prediction leads to spurious prefetches—the wrong blocks will be fetched from L2, wasting bandwidth and polluting the cache. We begin our evaluation by demonstrating that signatures can be predicted.

Figure 5 shows the RAS signature prediction accuracy as a function of lookahead, that is, how many signatures into the future we attempt to predict. Predicting the next RAS signature is synonymous with predicting the next program context—the next call or return operation. The three bars for each benchmark correspond to lookaheads of one, two, and three signatures, respectively. The further into the future we predict, the less accurate the prediction, as a considerable number of control flow decisions may be made between three consecutive call/return operations. The vertical axis indicates the prediction accuracy, where 1.0 indicates that future signatures are always correctly predicted.

We find that signature prediction accuracy is high—80% for a one-signature lookahead on average. Greater lookahead reduces accuracy. Fortunately, however, we find in our timing simulations that a lookahead of a single signature is sufficient to hide the L2 instruction fetch latency.

For SSJ, though our previous results indicated instruction cache misses correlate poorly to RAS state for generated code, we nevertheless find here that the call graph within the JVM remains predictable, as demonstrated by the high signature prediction accuracy.

As discussed in Section 3, we do not explicitly predict signatures using a signature table. Rather, we associate misses in the Miss Table with the preceding signature, this association implicitly relies on signature predictability.

5.2 Practical Design

In this subsection, we optimize the storage requirements of RDIP to arrive at a practical design. The storage requirements for RDIP are determined by two factors: (1) the number of signatures to be tracked, and (2) the number of associated misses per signature. Below, we analyze both these factors.

5.2.1 *Few Signatures Account for Most Misses*

The number of distinct signatures encountered in each benchmark is quite high (16000 for SSJ). Tracking all of these signatures would require prohibitive storage. Fortunately, we observe that only a few signatures account for the vast majority of instruction cache misses in each benchmark, as shown in Figure 6. Each graph shows the cumulative fraction of misses attributable to each distinct signature, with distinct signatures sorted along the x-axis in descending frequency of occurrence. The figure demonstrates that a few thousand most frequently occurring signatures account for the vast majority of misses in all cases. Thus, tracking just these signatures is sufficient to achieve near-peak coverage. In the following subsection, we look at the actual number of signatures that must be tracked in the Miss Table.

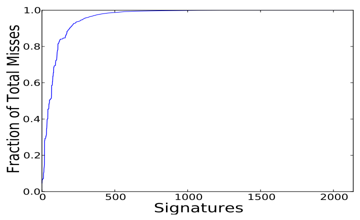
5.2.2 *Miss Table Size*

Clearly, an unbounded Miss Table is infeasible. The Miss Table is indexed in a fashion similar to a cache in that the last N bits of the signature are used to index the table (where N is log_2 size of the table) and the remaining bits are used as a tag. If a signature misses in the table, no prefetches are issued.

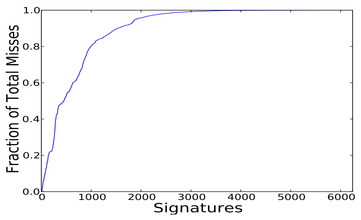
We sweep a range of practical sizes for the Miss Table, from 256 to 64K entries. We assume a 4-way set-associative structure. The 64K-entry table achieves 99% hit rate and results in no performance loss when compared to an infinite-storage Miss Table. We tried all power-of-two sizes between 256 and 64K and report the most relevant subset in Figure 7. We select a 4K-entry miss table for our final design, as it achieves a 96% hit rate on average while reducing storage by 94% when compared to a 64K-entry miss table. Next we look at the number of misses that are associated with each signature in the Miss Table.

5.2.3 *Miss Table Entry Encoding*

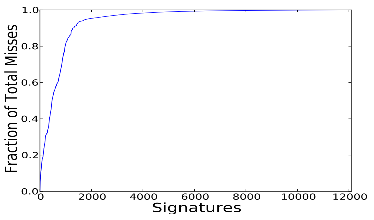
As shown in Figure 4, storing just a few instruction cache misses (about 15) can result in coverages >80% for RDIP. However, storing complete addresses for 15 cache blocks in each Miss Table entry remains prohibitive. Fortunately, as observed by Ferdman *et al.* [8], misses tend to be closely clustered with only a few discontinuities. Hence, we adopt a compression scheme similar to theirs to reduce the number of bits required in each Miss Table Entry. Each Miss Table Entry stores one or more trigger addresses (each 26 bits long) and an associated bit vector (8 bits long) that indicates which surrounding blocks to prefetch.



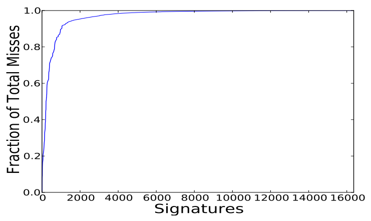
(a) gem5



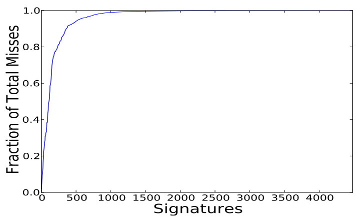
(b) HD-teraread



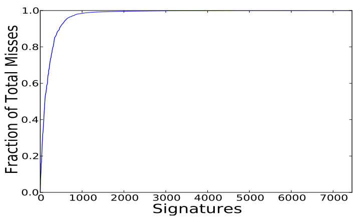
(c) HD-wdcnt



(d) SSJ



(e) MC-friendfeed



(f) MC-microblog

Figure 6: Not all Signatures are Equal: For each benchmark, we show the cumulative fraction of misses attributable to each distinct signature, with distinct signatures sorted along the x-axis in descending frequency of occurrence. For each benchmark, few frequently occurring signatures account for the vast majority of misses.

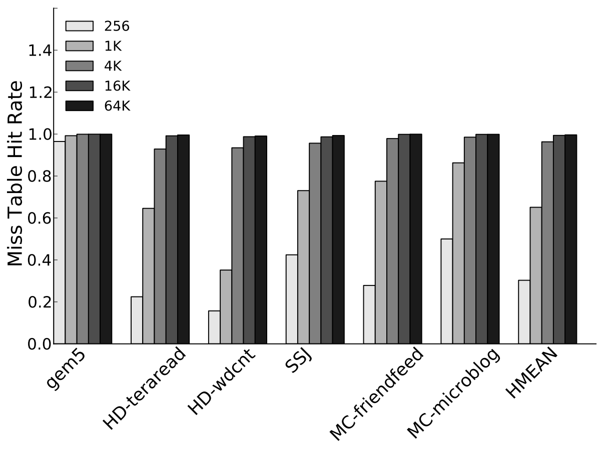


Figure 7: Miss Table: Size vs Hit Rate. We select a 4K-entry (4-way associative) design, which achieves 96% of possible Miss Table Hit Rate.

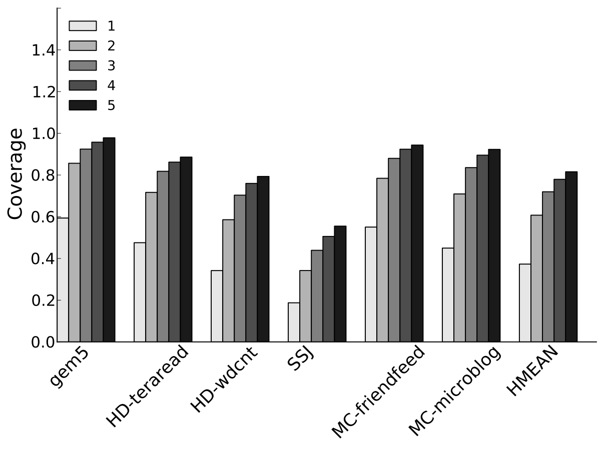


Figure 8: Miss Table Entry: No. of Trigger Addresses vs Coverage. Average coverage achieved is around 75% when 3 or more triggers are used. We use 3 trigger addresses and associated bit-vectors for each Miss Table Entry.

We perform a sensitivity study to discover the impact of varying the number of trigger addresses. We also study the width of the bit vector indicating the surrounding blocks, but found no improvement when increasing the bit vector length above eight bits. The coverage achieved for a varying number of trigger addresses is presented in Figure 8. Coverage, here, is defined as the fraction of misses that can be eliminated by RDIP when restricted to an encoding scheme with the specified number of trigger addresses, varying from one to five. Increasing the number of trigger addresses allows the prefetcher to correctly capture miss sequences for functions with more complex internal control flow (i.e., more jumps and fetch discontinuities) and also account for virtual function calls. As can be seen in the graph, adding additional trigger addresses substantially improves performance over a single trigger address. However having additional triggers implies additional storage. We select 3 triggers as a sensible trade-off between storage cost and coverage. As expected, SSJ has the lowest coverage among all benchmarks. We have found that even 10 triggers is insufficient to achieve perfect coverage for SSJ.

5.2.4 *Sensitivity to RAS size*

Next, we analyze the sensitivity of RDIP to the number RAS entries used to form a signature. RDIP only considers non-speculative call/return instructions when constructing the signature to minimize spurious signature changes and prefetch requests. Processors maintain some notion of non-speculative RAS to recover from mispredicted branches.

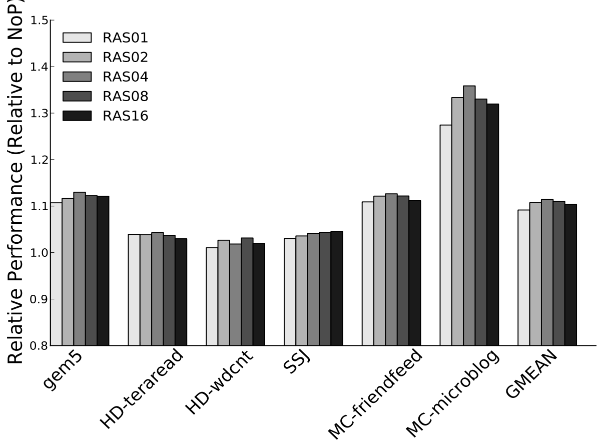


Figure 9: RDIP sensitivity to RAS size: A 4-entry RAS performs best.

This non-speculative RAS state (which is referred to simply as "RAS" from here on) is used by RDIP for signature generation. For the results presented in Figure 9, we use the top N entries of the RAS. RDIP with a larger RAS trades off miss table hit rate (due to more distinct signatures competing for the same miss table capacity) against future signature predictability (a larger RAS implies fewer overflows and more distinct signatures to identify program contexts). As can be seen from Figure 9, RDIP's performance is maximized with a 4-entry RAS; a smaller RAS sacrifices too much signature predictability while a larger RAS increases miss table conflicts. We use a 4-entry RAS for all other RDIP results. (Note that the branch predictor continues to use a 16-entry RAS in all cases).

5.2.5 Practical Configuration

In summary, we configure RDIP with a 4K-entry miss table, each having three trigger addresses and an 8-bit vector of blocks to prefetch. The total storage requirement is about 63kB. In the following section we analyze the performance of RDIP relative to alternative prefetching schemes.

5.3 Performance

In this section, we contrast the performance of RDIP with other relevant prefetcher designs. Throughout this section, we consider five prefetcher designs:

- *NoP* represents a system lacking an I-Cache prefetcher.
- *N2L* is a standard next-2-line prefetcher, typical of current processors. We have tried several other prefetch depths, and found next-2-line to perform best.
- *PIF* is our implementation of Proactive Instruction Fetch proposed by Ferdman et al. [8], which is the most effective instruction prefetching design reported to date.
- *RDIP* is our prefetcher design.

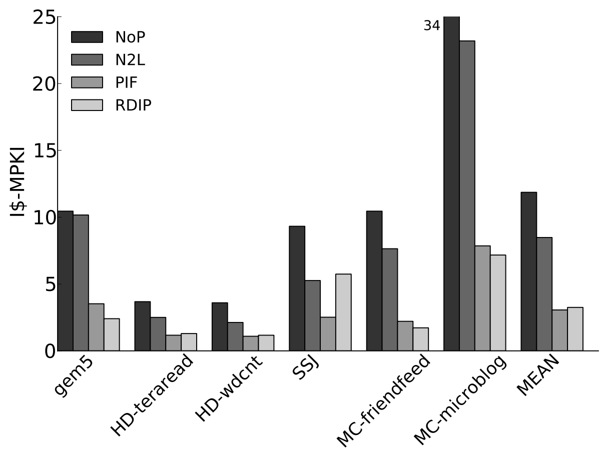


Figure 10: I-Cache Miss Rate across prefetching schemes

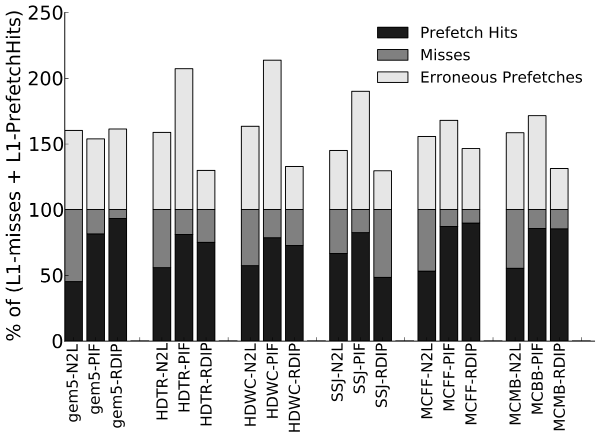


Figure 11: Coverage and Erroneous Prefetches. Each bar is divided into three segments, *Prefetch Hits* eliminated by the prefetching method, remaining *Misses*, and, above 100%, the number of *Erroneous Prefetches* normalized to the number of hits+misses.

- *Ideal* is an unrealizable 1MB instruction cache with the same latency as the 32kB cache. It easily captures working sets for all workloads and acts as an upper bound for performance of history-based schemes like RDIP and PIF.

We first discuss Figure 10, which shows the decrease in instruction cache miss rate for various prefetcher configurations across benchmarks. While RDIP reduces overall miss rate by 72.4%, N2L reduces it by only 28.5% and PIF by 74.2% over the baseline NoP case. These reductions result in corresponding improvements in performance as shown later.

Next, we discuss the overall effectiveness of the prefetchers (N2L, PIF, RDIP) in terms of the fraction of misses eliminated and the number of erroneously prefetched blocks transferred from L2 into the L1 cache. In Figure 11, we show the number of prefetch hits, misses, and erroneous prefetches.

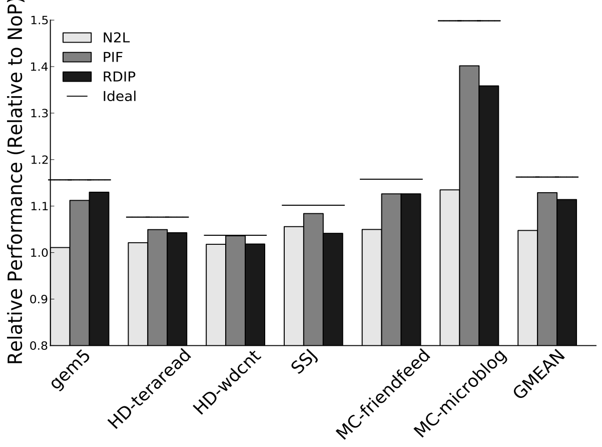


Figure 12: Performance gains across prefetching schemes.

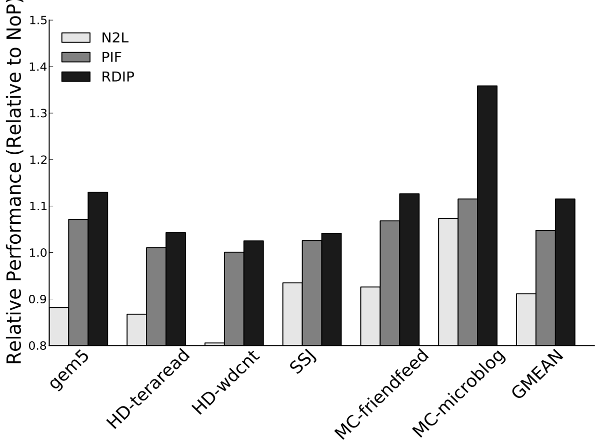


Figure 13: Iso-storage study (64kB).

	RDIP	PIF	
Structure	MissTable	HistoryBuffer	IndexTable
Size	64kB	136kB	68kB
Access Energy	13pJ	32pJ	11pJ
Static Power	12mW	39mW	7mW
Access Time	0.33ns	0.45ns	0.30ns

Table 2: Energy and Power Estimates for Hardware structures in RDIP and PIF.

Erroneous prefetches hurt performance in two ways: (1) they waste L1-L2 bandwidth, and (2) they displace potentially useful blocks in the L1 cache. In all benchmarks except gem5, RDIP issues fewer erroneous prefetches than even N2L. PIF almost always issues many more prefetch requests than RDIP. This result is to be expected as PIF has no way of predicting when the current temporal stream being prefetched terminates. To provide timely prefetches, the PIF prefetch engine traverses well-ahead of the actual commit stream. Thus, erroneous prefetches are issued past the end of every temporal stream. Additionally, PIF uses the PC of the head of a temporal stream to identify/distinguish streams. However, the same PC sometimes results in a miss in different program contexts; PIF's stream detection mechanisms is unable to distinguish such differing contexts, leading to a decrease in accuracy (but, generally, little decrease in coverage since the correct stream is often identified on the next miss).

RDIP achieves much of the coverage possible with PIF, while issuing far fewer requests. RDIP's greater accuracy leads to significant energy savings, as we show in Section 5.4.

As can be seen from Figure 12, averaged across the workloads, the ideal cache (Ideal) improves performance by 16.2%, which is substantially better than the 4.8% available from a next-2-line prefetcher. The frequent function calls and fetch discontinuities in these benchmarks give rise to the relatively poor performance of a next-2-line prefetcher. PIF improves performance by as much as 40% and by 12.9% on average. RDIP outperforms the next-2-line prefetcher and performs comparably to PIF in all cases except SSJ. N2L is relatively effective for SSJ because the JIT frequently generates long sequential functions, due to inlining and loop-unrolling. PIF is similarly able to learn these long miss sequences, since it relies on miss addresses rather than calls and returns to distinguish streams. A hybrid of RDIP and N2L might reclaim some of the lost opportunity on SSJ. Overall, RDIP improves performance by as much as 33% (11.5% on average), and realizes over 70% of the possible performance achieved with an Ideal instruction cache.

5.4 Energy and Storage Overheads

In this section, we present the storage and energy requirements of RDIP and contrast them with the requirements of PIF. The original report on PIF does not explicitly enumerate the storage requirements of the design, so we must infer some structure sizes from other results reported in the work.

5.4.1 Hardware Overhead

RDIP's storage overhead is almost entirely in the Miss Table. Based on studies presented earlier in this section, we size the Miss Table to have 4K entries with each entry as described previously. Each tag is 22 bits long, each trigger address is a cache block address and hence is 26 bits long. We use an 8-bit vector to identify nearby misses. Thus, the total storage per entry is 16 bytes *(22+3*(26+8) bits)*, which results in a total structure size of about 63kB (we use 64kB in CACTI for energy calculations). The storage required by PIF includes a 32K entry history buffer, with each entry storing a cache block region *(26+8 bits)*, which implies a 136kB (128kB in CACTI) history buffer as well as an index table, which we estimate at 68kB (64kB in CACTI; the index table must be large enough to refer to each unique trigger address in the history buffer). The estimate for index table size is based on earlier work [9], which employs similar structures. RDIP reduces storage by at least 3X compared to PIF.

5.4.2 ISO-storage Comparison

In this section, we present an iso-storage comparison between N2L, PIF and RDIP. We configure each prefetcher with a storage overhead budget of 64kB. In case of N2L, we employ a 96kB I-Cache with a hit latency of 3ns as opposed to 32kB and 1ns for PIF and RDIP. PIF was configured to have 8K-entry history buffer and index table. As can be seen from Figure 13, RDIP performs best of the three.

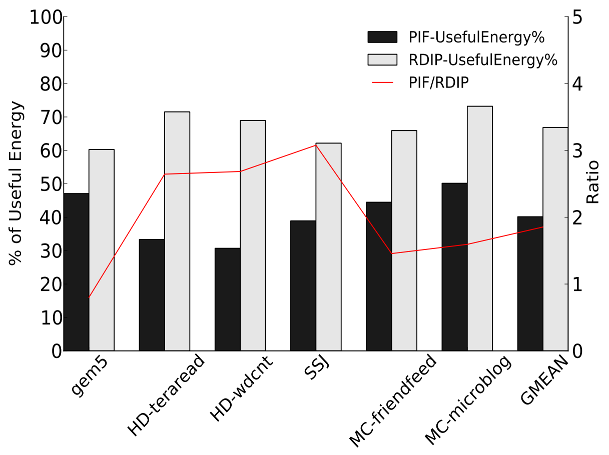


Figure 14: Energy Efficiency PIF vs RDIP.

Benchmark	PIF	RDIP	Ratio
gem5	23.28	29.11	0.80
Hadoop-teraread	10.76	4.05	2.65
Hadoop-wdcnt	9.44	3.16	2.98
SSJ	22.07	7.17	3.07
MC-FriendFeed	23.74	16.3	1.46
MC-Microblog	66.68	41.84	1.59

Table 3: Dynamic Energy Overhead per Instruction in pJ.

The performance of N2L with a large I-Cache highlights the criticality of maintaining low I-Cache hit latencies. RDIP comes out better than PIF for the given storage budget as RDIP tracks only I-cache misses where as PIF tracks all the accesses made.

5.4.3 *Energy Overhead*

Here we compare the energy and power requirements of RDIP and PIF. In Table 2, we present static power requirements, which we obtain from CACTI [35]. Based on Table 2 and access counts to various structures (including I-Cache and L2-Cache), we estimate the average dynamic energy overhead per instruction for PIF and RDIP. We report them in Table 3.

The line (right y-axis) in Figure 14 *(PIF/RDIP)* shows that PIF consumes nearly 1.9X more dynamic energy per instruction, on average, when compared to RDIP. This overhead is due to its large structures and more accesses to these structures and the L2-Cache. Interestingly, RDIP's energy overhead is only 7% more than N2L. Despite the additional storage structure, RDIP's greater accuracy as compared to N2L results in reduced L2 accesses, offsetting the storage energy overhead. Figure 14 also shows the percent of total dynamic energy which was "usefully" spent, for both PIF and RDIP. "Useful" energy is defined as the energy spent in prefetching cache blocks from L2 to L1 that subsequently resulted in prefetch hits. The "non-useful" component of energy arises due to the static energy of prefetcher-specific hardware structures and erroneous prefetches. For RDIP, nearly 67% of the energy spent goes towards transporting useful cache blocks, while for PIF, only 40% of energy is usefully spent.

These results show that RDIP, though marginally outperformed by PIF, is more energy and power efficient, reducing prefetcher related energy consumption by 1.9X.

6. CONCLUSIONS

Recent research shows that L1 instruction fetch misses remain a critical performance bottleneck, accounting for up to 40% slowdowns in server applications. While instruction footprints comfortably fit in the last-level caches, they overwhelm typical L1 instruction caches, which are limited by strict latency constraints.

In this work we showed that the RAS captures the program context and correlates strongly with L1 instruction cache misses. This observation allowed us to develop a prefetcher that associates prefetch operations with signatures formed from the RAS state. Our RAS-Directed Instruction Prefetcher is able to realize nearly 70% of the total possible performance improvement of an impractically large L1 instruction cache resulting in a 11.5% improvement in overall performance over a prefetcher-less baseline. RDIP also comes within 2% of the performance achieved by the state of the art prefetcher (PIF). Unlike PIF, which requires impractical storage and considerable complexity, RDIP is able to realize this performance improvement at one third the hardware overhead and half the energy overhead.

Acknowledgments

This work was funded by grants from ARM and NSF CCF-0815457. The authors would like to thank Faissal Sleiman and Steven Pelley for their help in putting this work together and the anonymous reviewers for their feedback.

7. REFERENCES

[1] D. Anderson, F. Sparacio, and R. Tomasulo. The IBM System/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1), 1967.

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

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of ACM SIGMETRICS/Performance*, 2012.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

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

[6] Facebook. Memcached Tech Talk with M. Zuckerberg, 2010.

[7] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proc. of the International Conf. on*

Architectural Support for Programming Languages and Operating Systems, 2012.

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

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

[10] A. Gutierrez, R. Dreslinski, T. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-system analysis and characterization of interactive smartphone applications. In *IEEE International Symp. on Workload Characterization*, 2011.

[11] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *Proc. of the International Symp. on Computer Architecture*, 2009.

[12] S. Harizopoulos and A. Ailamaki. STEPS towards cache-resident transaction processing. In *Proc. of the International Conf. on Very Large Databases*, 2004.

[13] R. Hegde. Technical report, Intel, 2008.

[14] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of the International Symp. on Computer Architecture*, 1990.

[15] C. Kaynak, B. Grot, and B. Falsafi. Shift: Shared history instruction fetch for lean-core server processors. In *Proceedings of 46th International Symposium on Microarchitecture*, 2013.

[16] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of 27th International Symposium on Computer Architecture*, 2000.

[17] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of 28th International Symposium on Computer Architecture*, 2001.

[18] K. Lim, D. Meisner, A. Saidi, P. Ranganathan, and T. Wenisch. Thin servers with smart pipes: Designing soc accelerators for memcached. In *Proceedings of 40th International Symposium on Computer Architecture*, 2013.

[19] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, et al. Scale-out processors. In *Proc. of the International Symp. on Computer Architecture*, 2012.

[20] C. Luk and T. Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Proc. of the International Symp. on Microarchitecture*, 1998.

[21] C. Luk and T. Mowry. Architectural and compiler support for effective instruction prefetching: a cooperative approach. *ACM Trans. on Computer Systems*, 19(1), 2001.

[22] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An effective alternative to large instruction windows for out-of-order processors. In *Proc. of the International Symp. on High-Performance Computer Architecture*, 2003.

[23] R. Nair. Dynamic path-based branch correlation. In *Proceedings of 28th International Symposium on Microarchitecture*, 1995.

[24] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *Proc. of the International Symp. on Microarchitecture*, 1996.

[25] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohen, J. L. Larriba-pey, P. G. Lowney, and M. Valero. Code layout optimizations for transaction processing workloads. In *Proc. of the International Symp. on Computer Architecture*, 2001.

[26] A. Ramirez, O. J. Santana, J. L. Larriba-pey, and M. Valero. Fetching instruction streams. In *Proc. of the International Symposium on Microarchitecture*, 2002.

[27] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *Proc. International Symp. on Microarchitecture*, 1999.

[28] G. Reinman, B. Calder, and T. Austin. Optimizations enabled by a decoupled front-end architecture. *Computers, IEEE Transactions on*, 50(4), 2001.

[29] P. Saab. Scaling memcached at Facebook, 2008.

[30] O. Santana, A. Ramirez, and M. Valero. Enlarging instruction streams. *IEEE Transactions on Computers*, 56(10), 2007.

[31] A. Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12), 1978.

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

[33] V. Srinivasan, E. Davidson, G. Tyson, M. Charney, and T. Puzak. Branch history guided instruction prefetching. In *Proc. of the International Symp. on High-Performance Computer Architecture*, 2001.

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

[35] S. Thoziyoor and N. Muralimanohar. Cacti 5.0, 2007.

[36] A. V. Veidenbaum. Instruction cache prefetching using multilevel branch prediction. In *Proc. of the International Symposium on High Performance Systems*, 1997.

[37] C. Zhang and S. McKee. Hardware-only stream prefetching and dynamic access ordering. In *Proc. of the 14th international conference on Supercomputing*, 2000.

[38] J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. In *Proc. of the ACM SIGMOD International Conf. on Management of Data*, 2004.

[39] C. B. Zilles and G. S. Sohi. Execution-based prediction using speculative slices. In *Proc. of the International Symp. on Computer Architecture*, 2001.