I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing

Tanvir Ahmed Khan[*] Akshitha Sriraman[*] Joseph Devietti[†] Gilles Pokam[‡] Heiner Litz[§] Baris Kasikci[*]
[*]University of Michigan [†]University of Pennsylvania [‡]Intel Corporation [§]University of California, Santa Cruz
[*]{takh, akshitha, barisk}@umich.edu [†]devietti@cis.upenn.edu [‡]gilles.a.pokam@intel.com [§]hlitz@ucsc.edu

***Abstract*—Modern data center applications have rapidly expanding instruction footprints that lead to frequent instruction cache misses, increasing cost and degrading data center performance and energy efficiency. Mitigating instruction cache misses is challenging since existing techniques (1) require significant hardware modifications, (2) expect impractical on-chip storage, or (3) prefetch instructions based on inaccurate understanding of program miss behavior.**

To overcome these limitations, we first investigate the challenges of effective instruction prefetching. We then use insights derived from our investigation to develop *I-SPY*, a novel profile-driven prefetching technique. *I-SPY* uses dynamic miss profiles to drive an offline analysis of I-cache miss behavior, which it uses to inform prefetching decisions. Two key techniques underlie *I-SPY*'s design: (1) *conditional prefetching*, which only prefetches instructions if the program context is known to lead to misses, and (2) *prefetch coalescing*, which merges multiple prefetches of non-contiguous cache lines into a single prefetch instruction. *I-SPY* exposes these techniques via a family of light-weight hardware code prefetch instructions.

We study *I-SPY* in the context of nine data center applications and show that it provides an average of 15.5% (up to 45.9%) speedup and 95.9% (and up to 98.4%) reduction in instruction cache misses, outperforming the state-of-the-art prefetching technique by 22.5%. We show that *I-SPY* achieves performance improvements that are on average 90.5% of the performance of an ideal cache with no misses.

***Index Terms*—Prefetching, frontend stalls, memory systems.**

I. Introduction

The expanding user base and feature portfolio of modern data center applications is driving a precipitous growth in their complexity [1]. Data center applications are increasingly composed of deep and complex software stacks with several layers of kernel networking and storage modules, data retrieval, processing elements, and logging components [2–4]. As a result, code footprints are often a hundred times larger than a typical L1 instruction cache (I-cache) [5], and further increase rapidly every year [1].

I-cache misses are becoming a critical performance bottleneck due to increasing instruction footprints [1, 2, 6]. Even modern out-of-order mechanisms do not hide instruction misses that show up as glaring stalls in the critical path of execution. Hence, reducing I-cache misses can significantly improve data center application performance, leading to millions of dollars in cost and energy savings [2, 7].

The importance of mechanisms that reduce I-cache misses (e.g., instruction prefetching) has long been recognized. Prior works have proposed next-line or history-based hardware instruction prefetchers [3, 4, 8–14] and several software mechanisms have been proposed to perform code layout optimizations for improving instruction locality [15–19]. While these techniques are promising, they (1) demand impractical on-chip storage [10, 12, 13], (2) require significant hardware modifications [3, 4], or (3) face inaccuracies due to approximations used in computing a cache-optimal code layout [18, 20].

A recent profile-guided prefetching proposal, AsmDB [2], was able to reduce I-cache misses in Google workloads. However, we find that even AsmDB can fall short of an ideal prefetcher by 25.5% on average. To completely eliminate I-cache misses, it is important to first understand: why do existing state-of-the-art prefetching mechanisms achieve sub-optimal performance? What are the challenges in building a prefetcher that achieves near-ideal application speedup?

To this end, we perform a comprehensive characterization of the challenges in developing an ideal instruction prefetcher. We find that an ideal instruction prefetcher must make careful decisions about (1) *what* information is needed to efficiently predict an I-cache miss, (2) *when* to prefetch an instruction, (3) *where* to introduce a prefetch operation in the application code, and (4) *how* to sparingly prefetch instructions. Each of these design decisions introduces non-trivial trade-offs affecting performance and increasing the burden of developing an ideal prefetcher. For example, the state-of-the-art prefetcher, AsmDB, injects prefetches at link time based on application's miss profiles. However, control flow may not be predicted at link time or may diverge from the profile at run time (e.g., due to input dependencies), resulting in many prefetched cache lines that never get used and pollute the cache. Moreover, AsmDB suffers from static and dynamic code bloat due to additional prefetch instructions injected into the code.

In this work, we aim to reduce I-cache misses with *I-SPY*—a prefetching technique that carefully identifies I-cache misses, sparingly injects "code prefetch" instructions in suitable program locations at link time, and selectively executes injected prefetch instructions at run time. *I-SPY* proposes two novel mechanisms that enable on average 90.4% of ideal speedup: *conditional prefetching* and *prefetch coalescing*.

Conditional prefetching. Prior techniques [2, 21] either prefetch excessively to hide more I-cache misses, or prefetch conservatively to prevent unnecessary prefetch operations that pollute the I-cache. To hide more I-cache misses as well as to reduce unnecessary prefetches, we propose *conditional prefetching*, wherein we use profiled execution context to inject code prefetch instructions that cover each miss, at link time. At run-time, we reduce unnecessary prefetches by executing an injected prefetch instruction only when the miss-inducing context is observed again.

To implement conditional prefetching with *I-SPY*, we propose two new hardware modifications. First, we propose simple CPU modifications that use Intel's Last Branch Record (LBR) [22] to enable a server to selectively execute an injected prefetch instruction based on the likelihood of the prefetch being successful. We also propose a "code prefetch" instruction called `Cprefetch` that holds miss-inducing context information in its operands, to enable an *I-SPY*-aware CPU to conditionally execute the prefetch instruction.

Prefetch coalescing. Whereas conditional prefetching facilitates eliminating more I-cache misses without prefetching unnecessarily at run time, it can still inject too many prefetch instructions that might further increase the static code footprint. Since data center applications face significant I-cache misses [1, 7], injecting even a single prefetch instruction for each I-cache miss can significantly increase an already-large static code footprint. To avoid a significant code footprint increase, we propose *prefetch coalescing*, wherein we prefetch multiple cache lines with a single instruction. We find that several applications face I-cache misses from non-contiguous cache lines, i.e., in a window of N lines after a miss, only a subset of the N lines will incur a miss. We propose a new instruction called `Lprefetch` to prefetch these non-contiguous cache lines using a single instruction.

We study *I-SPY* in the context of nine popular data center applications that face frequent I-cache misses. Across all applications, we demonstrate an average performance improvement of 15.5% (up to 45.9%) due to a mean 95.9% (up to 98.4%) L1 I-cache miss reduction. We also show that *I-SPY* improves application performance by 22.4% compared to the state-of-the-art instruction prefetcher [2]. *I-SPY* increases the dynamically-executed instruction count by 5.1% on average and incurs an 8.2% mean static code footprint increase.

In summary, we make the following contributions:

- A detailed analysis of the challenges involved in building a prefetcher that provides close-to-ideal speedups.
- *Conditional prefetching:* A novel profile-guided prefetching technique that accurately identifies miss-inducing program contexts to prefetch I-cache lines only when needed.
- *Prefetch coalescing:* A technique that coalesces multiple non-contiguous cache line prefetches based on run-time information obtained from execution profiles.
- *I-SPY*: An end-to-end system that combines conditional prefetching with prefetch coalescing using a new family of instructions to achieve near-ideal speedup.

II. Understanding the Challenges of Instruction Prefetching

In this section, we present a detailed characterization of the challenges in developing an ideal instruction prefetching technique. We define an ideal prefetcher as one that achieves the performance of an I-cache with no misses, i.e., where every access hits in the L1 I-cache (a theoretical upper bound). We characterize prefetching challenges by exploring four important questions: (1) ***What*** information is needed to efficiently predict an I-cache miss?, (2) ***When*** must an instruction be prefetched to avoid an I-cache miss? (3) ***Where*** should a prefetcher inject a code prefetch instruction in the program?, and (4) ***How*** can a prefetcher sparingly prefetch instructions while still eliminating most I-cache misses?

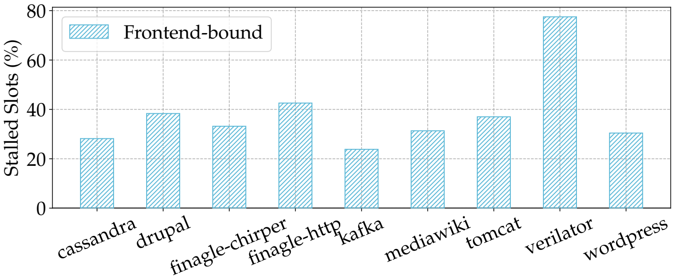


Fig. 1: Several widely-used data center applications spend a significant fraction of their pipeline slots on "Frontend-bound" stalls, waiting for I-cache misses to return (measured using the Top-down methodology [23]).

We characterize challenges using nine popular real-world applications that exhibit significant I-cache misses. In Fig. 1, we show the "frontend" pipeline stalls that the nine applications exhibit when waiting for I-cache misses to return. We observe that these data center applications can spend 23% - 80% of their pipeline slots in waiting for I-cache misses to return. Hence, we include these applications in our study.

From Facebook's HHVM OSS-performance [24] benchmark suite, we analyze (1) *Drupal*: a PHP content management system, (2) *Mediawiki*: an open-source Wiki engine, and (3) *Wordpress*: a PHP-based content management system used by services such as Bloomberg Professional and Microsoft News. From the Java DaCapo [25] benchmark suite, we analyze (a) *Cassandra* [26]: a NoSQL database management system used by companies such as Instagram and Netflix, (b) *Kafka*: Apache's stream-processing software platform used by companies such as Uber and Linkedin, and (c) *Tomcat* [27]: Apache's implementation of the Java Servlet and WebSocket. From the Java Renaissance [28] benchmark suite, we analyze *Finagle-Chirper* and *Finagle-HTTP* [29]: Twitter Finagle's micro-blogging service and HTTP server, respectively. We also study *Verilator* [30], a tool used by cloud companies to simulate custom hardware designs. We describe our complete experimental setup and simulation parameters in Sec. V.

A. What Information is Needed to Efficiently Predict an I-Cache Miss?

An ideal prefetcher must predict all I-cache misses before they occur, to prefetch them into the I-cache in time. To this end, prior work [2, 8–10] (e.g., next-in-line prefetching) has shown that an I-cache miss can be predicted using the program instructions executed before the miss. Since any arbitrary instruction (e.g., direct/indirect branches or function returns)

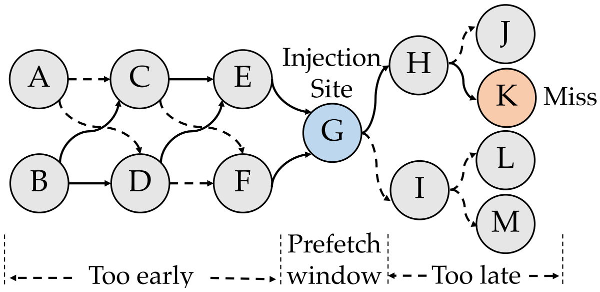


Fig. 2: A partial example of a miss-annotated dynamic control flow graph. Dashed edges represent execution paths that do not lead to a miss.

could execute before a miss, the application's dynamic control flow must be tracked to predict a miss using the program paths that lead to it. An application's execution can be represented by a dynamic Control Flow Graph (CFG). In a dynamic CFG, the nodes represent basic blocks (sequence of instructions without a branch) and the edges represent branches. Fig. 2 shows a dynamic CFG, where the cache miss at basic block K can be reached via various paths. The CFG's edges are typically weighted by a branch's execution count. For brevity, we assume all the weights are equal to one in this example.

Software-driven prefetchers [2, 17, 19] construct an application's dynamic CFG and identify miss locations that can be eliminated using a suitable prefetch instruction. For example, AsmDB [2] uses DynamoRIO's [31] memory trace client to capture an application's dynamic CFG for locating I-cache misses in the captured trace. Unfortunately, DynamoRIO [31] incurs undue overhead [32], making it costly to deploy in production. To efficiently generate miss-annotated dynamic CFGs, we propose augmenting dynamic CFG traces from Intel's LBR [22] with L1 I-cache miss profiles collected with Intel's Precise Event Based Sampling (PEBS) [33] performance counters. Generating dynamic CFGs using such lightweight monitoring enables profiling applications in production.

Observation: Representing a program's execution using a dynamic CFG and augmenting it with L1 I-cache miss profiles enables determining prefetch candidates.
Insight: *Generating a lightweight miss-annotated dynamic CFG using Intel's LBR and PEBS incurs low run-time performance overhead and enables predicting miss locations in production systems.*

B. When To Prefetch an Instruction?

A prefetch is successful only if it is timely. In the dynamic CFG in Fig. 2, a prefetch instruction injected at predecessor basic blocks H or I is too late: the prefetcher will not be able to bring the line into the I-cache in time and a miss will occur at K. In contrast, if a prefetch instruction is injected at predecessors E or F, the prefetched line may not be needed soon enough, and it may (1) either evict other lines that will be accessed sooner, or (2) itself get prematurely evicted before it is accessed. Instead, the prefetch must be injected in an appropriate *prefetch window*. In our example, we assume block G is a timely injection candidate in the prefetch window.

Prior work [2] empirically determines an ideal prefetch window using average application-specific IPC to inject a prefetch instruction that hides a cache miss. *I-SPY* relies on this approach and injects prefetch instructions 27 - 200 cycles before a miss, a window we determine in our evaluation.

Observation: An instruction must be prefetched in a timely manner to avoid a miss.
Insight: *Empirically determining the prefetch window such that a prefetch is not too early or too late, can effectively eliminate a miss.*

C. Where to Inject a Prefetch?

An ideal prefetcher would eliminate all I-cache misses, achieving full *miss coverage*. To achieve full miss coverage, a prefetcher such as the one proposed by Luk and Mowry [21], might inject a "code prefetch" instruction into every basic block preceding an I-cache miss. However, the problem of this approach is that due to dynamic control flow changes, naively injecting a prefetch into a predecessor basic block causes a high number of inaccurate prefetches whenever the predecessor does not lead to the miss. Prefetching irrelevant lines hurts *prefetch accuracy* (the fraction of useful prefetches) and leads to I-cache pollution, degrading application performance.

Prefetch accuracy can be improved by assessing the usefulness of a prefetch and by restricting the injection of prefetches to those that are likely to improve performance. To determine the likelihood of a prefetch being useful, we can analyze the *fan-out* of the prefetch's injection site. We define fan-out as the percentage of paths that do not lead to a target miss from a given injection site. For example, in Fig. 2, the candidate injection site G has a fan-out of 75% as only one out of four paths leads to the miss K.

By limiting prefetch injection to nodes whose fan-out is below a certain threshold, accuracy can be improved, however, coverage is also reduced. The fan-out threshold that decides whether to inject a prefetch represents a control knob to trade-off coverage vs. accuracy. To determine this threshold, Fig. 3 analyzes the impact of fan-out on accuracy and coverage for the *wordpress* application. As it can be seen, for real applications with large CFGs, a high fan-out of 99% is required to achieve the best performance, although accuracy starts to drop sharply at this point. Hence, prior works (including AsmDB) that rely on static analysis for injecting prefetches fall short of achieving close to ideal performance (65% in the case of *wordpress*).

With *I-SPY*, we aim to break this trade-off by optimizing for prefetch accuracy and miss coverage simultaneously. To this end, we propose context sensitive conditional prefetching, a technique that statically injects prefetches to cover each miss (i.e., high miss coverage), but dynamically executes injected prefetches only when the prefetch is likely to be successful, minimizing unused prefetches (i.e., high prefetch accuracy). In Section III-A, we describe our conditional prefetching

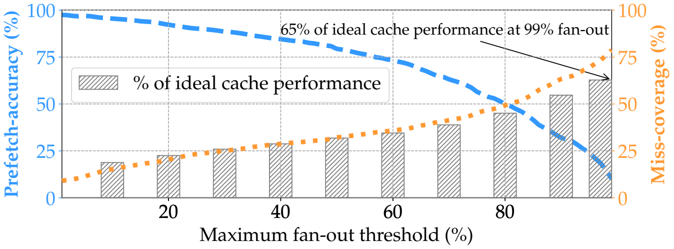


Fig. 3: Prefetch accuracy vs. miss coverage tradeoff in AsmDB and its relation to ideal cache performance: Miss-coverage increases with an increase in fan-out threshold, but prefetch accuracy starts to reduce. Only 65% of ideal cache performance can be reached at 99% fan-out due to low prefetch accuracy.

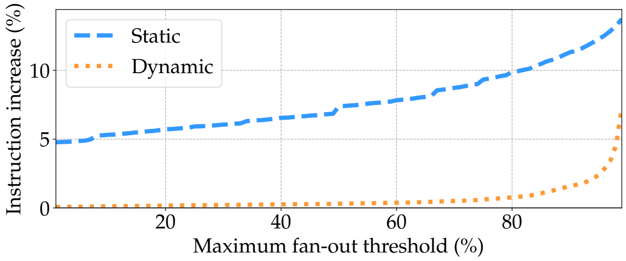


Fig. 4: AsmDB's static and dynamic code footprint increase: Injecting prefetches in high fan-out predecessors significantly increases static and dynamic code footprints.

technique and our approach that leverages dynamic context information to decide whether to execute a prefetch or not.

Observation: It is challenging to achieve both high miss coverage and prefetch accuracy if we determine prefetch injection candidate blocks based on a static CFG analysis alone.
Insight: *Leveraging dynamic run-time information to conditionally execute statically-injected prefetch instructions can help improve both miss coverage and prefetch accuracy.*

D. How to Sparingly Prefetch Instructions?

Several profile-guided prefetchers [2, 21] require at least one code prefetch instruction to mitigate an I-cache miss. For example, the state-of-the-art prefetcher, AsmDB [2], covers each miss by injecting a prefetch instruction into a high fan-out (≤99%) predecessor. However, statically injecting numerous prefetch instructions and executing them at run time, increases the static and dynamic application code footprint by 13.7% and 7.3% respectively, as portrayed in Fig. 4. An increase in static and dynamic code footprints can pollute the I-cache and cause unnecessary cache line evictions, further degrading application performance. Hence, it is critical to sparingly prefetch instructions to minimize code footprints.

Prefetch coalescing. Our conditional prefetching proposal allows statically injecting more prefetch instructions to eliminate more I-cache misses, without having to dynamically perform inaccurate prefetches. However, a large number of statically-injected code prefetch instructions can still increase an application's static code footprint.

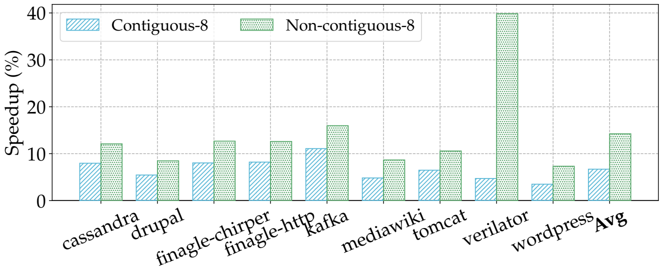


Fig. 5: Speedup of Contiguous-8 (prefetches all 8 contiguous lines after a miss) vs. Non-contiguous-8 (prefetches only the misses in an 8-line window after a miss): Prefetching non-contiguous cache lines offers a greater speedup opportunity.

A naïve approach to statically inject fewer instructions is to leverage the spatial locality of I-cache misses to prefetch multiple contiguous cache lines with a single prefetch instruction rather than a single line at a time [8, 9]. In contrast, another approach [3] finds value in prefetching multiple non-contiguous cache lines together. Similarly, we posit that it is unlikely that all the contiguous cache lines in a window of n lines after a given miss will incur misses. It is more likely that a subset of the next-n lines will incur misses, whereas others will not. To validate this hypothesis, we consider a window of eight cache lines immediately following a miss to implement two prefetchers: (1) *Contiguous-8*, that prefetches all eight contiguous cache lines after a miss and (2) *Non-contiguous-8*, that prefetches only the missed cache lines in the eight cache line window.

We profile all our benchmarks to detect I-cache misses and measure the speedup achieved by both prefetchers in Fig. 5. We find that Non-contiguous-8 provides an average 7.6% speedup over Contiguous-8. We conclude that prefetch coalescing of non-contiguous, but spatially nearby I-cache misses, via a single prefetch instruction can improve performance while minimizing the number of static and dynamic prefetch instructions. We note that our conclusion holds for larger windows of cache lines (e.g., 16 and 32). We find that a window of eight lines offers a good trade-off between speedup and circuit complexity required to support a larger window size. We provide a sensitivity analysis for window sizes in §VI-B.

Observation: Injecting too many prefetch instructions can increase static and dynamic code footprints, inducing additional cache line evictions.
Insight: *Conditional prefetching can minimize dynamic code footprints; coalescing spatially-near non-contiguous I-cache miss lines into a single prefetch instruction can minimize both static and dynamic code footprints.*

III. I-SPY

I-SPY proposes two novel techniques to improve profile-guided instruction prefetching. *I-SPY* introduces *conditional prefetching* to address the dichotomy between high coverage and accuracy discussed in §II-C. Furthermore, *I-SPY* proposes *prefetch coalescing* to reduce the static code footprint increase due to injected prefetch instructions explored in §II-D. *I-SPY* relies on profile-guided analysis at link-time to determine frequently missing blocks and prefetch injection sites using Intel LBR [22] and PEBS [33]. We provide a detailed description of *I-SPY*'s usage model in §IV. *I-SPY* also introduces minor hardware modifications to improve prefetch efficiency at run time. As a result, our proposed techniques close the gap between static and dynamic prefetching by combining the performance of dynamic hardware-based mechanisms with the low complexity of static software prefetching schemes.

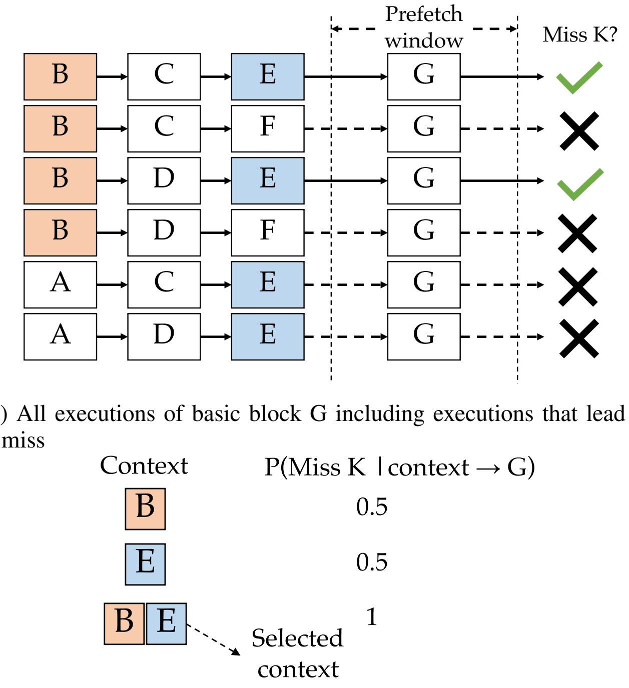
A. Conditional Prefetching

Conditionally executing prefetches has a two-fold benefit: *I-SPY* can liberally inject conditional prefetch instructions to cover each miss (i.e., achieve high miss coverage) while simultaneously minimizing unused prefetches (i.e., achieve high accuracy). *I-SPY* uses the execution context to decide whether to conditionally execute a prefetch or not. We first discuss how *I-SPY* computes contexts leading to misses. We then explain how *I-SPY* 's conditional prefetching instruction is implemented, and finally discuss micro-architectural details.

Miss context discovery. Similar to many other branch prediction schemes [3, 4, 34, 35], *I-SPY* uses the basic block execution history to compute the execution context. Initially, we attempted to use the exact basic block sequence to predict a miss. However, we found this approach intractable since the number of block sequences (i.e., the number of execution paths) leading to a miss grows exponentially with the increase in sequence length. As a result, *I-SPY* only considers the presence of certain important basic blocks in the recent context history to inform its prefetching decisions. This approach is in line with prior work [36] that observes that prediction accuracy is largely insensitive to the basic block order sequence.

We use the dynamic CFG in Fig. 2 to describe the miss context discovery process. Recall that in this example, the miss occurs in basic block K and block G is the injection site in the prefetch window. As shown in Fig. 6a, there are six execution paths including the candidate injection site G and two of these paths lead to the basic block K, where the miss occurs.

I-SPY starts miss context discovery by identifying *predictor basic blocks*—blocks with the highest frequency of occurrence in the execution paths leading to each miss. In our example, B and E are predictor blocks. Since *I-SPY* only relies on the presence of blocks to identify the context (as opposed to relying on the order of blocks), it computes combinations of predictor blocks as potential *contexts* for a given miss. Then, *I-SPY* calculates the conditional probability of each *context* leading to a miss in a block B, i.e., $P(\text{Miss } \textit{in Block} \text{ ``}B\text{''}|\text{context})$ as per the Bayes theorem. As shown in Fig. 6b, *I-SPY* computes $P(\text{Miss } K|\text{B})$, $P(\text{Miss } K|\text{E})$, and $P(\text{Miss } K|\text{B} \cap \text{E})$, i.e., the probability of leading to the miss in block K, given an execution context of either (B), or (E), or both (B and E).



(a) All executions of basic block G including executions that lead to a miss

(b) Probability calculation for a context leading to a miss

Fig. 6: An example of *I-SPY*'s context discovery process

I-SPY then selects the combination with the highest probability as the context for a given miss. In our example, this context, namely (B and E) will be encoded into the conditional prefetch instruction injected at G. At run time, the conditional prefetch will be executed if the run-time branch history contains the recorded context. We now detail *I-SPY*'s conditional prefetch instruction.

Conditional prefetch instruction. We propose a new prefetch instruction, `Cprefetch` that requires an extra operand to specify the execution context. Each basic block in the context is identified by its address, i.e., the address of the first instruction in the basic block. *I-SPY* computes the basic block address using the LBR data.

To reduce the code size of `Cprefetch`, *I-SPY* hashes the individual basic block addresses in the context into an n-byte `immediate` operand (`context-hash`) using hash functions, FNV-1 [37] and MurmurHash3 [38]. When a `Cprefetch` is executed at run time, the processor recomputes a hash value (`runtime-hash`) using the last 32 predecessor basic blocks (Intel LBR [22] provides the addresses of 32 most recently executed basic blocks), and compares it against the `context-hash`. The prefetch operation is performed only if the set-bits in `context-hash` are a subset of the set-bits in the `runtime-hash`.

Both `runtime-hash` and `context-hash` are compressed representations of multiple basic block addresses. While compressing multiple 64 bit basic block addresses into fewer bits reduces the code bloat, it might also introduce

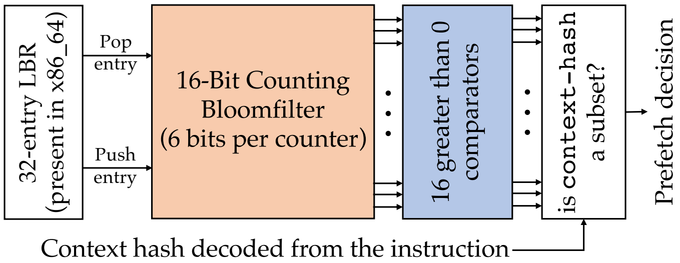

32-entry LBR (present in x86_64)
Pop entry
Push entry
16-Bit Counting Bloomfilter (6 bits per counter)
16 greater than 0 comparators
is context-hash a subset?
Prefetch decision
Context hash decoded from the instruction

Fig. 7: Micro-architectural changes needed to execute the context-sensitive conditional prefetch instruction, `Cprefetch`

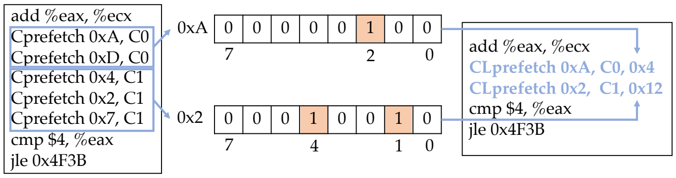

```
add %eax, %ecx
Cprefetch 0xA, C0
Cprefetch 0xD, C0
Cprefetch 0x4, C1
Cprefetch 0x2, C1
Cprefetch 0x7, C1
cmp $4, %eax
jle 0x4F3B
```
```
add %eax, %ecx
CLprefetch 0xA, C0, 0x4
CLprefetch 0x2,  C1, 0x12
cmp $4, %eax
jle 0x4F3B
```

Fig. 8: An example of *I-SPY*'s prefetch coalescing process

false positives. A false positive might occur when the set-bits in `context-hash` are a subset of the set-bits in `runtime-hash`, however, not all the basic blocks represented by `context-hash` are present among the 32 most recently-executed basic blocks represented by `runtime-hash`. We analyze a range of values for the `context-hash` size in Fig. 21 and determine that a 16 bit immediate offers a good tradeoff between code bloat and false positive rates.

Micro-architectural modifications `Cprefetch` requires minor micro-architectural modifications. Intel's Xeon data center processors support an LBR [22] control flow tracing facility, which tracks the program counter and target address of the 32 most recently executed branches.

I-SPY extends the LBR to maintain a rolling `runtime-hash` of its contents. Fig. 7 shows the micro-architectural requirements of *I-SPY*'s context-sensitive prefetch instruction for 32 predecessor basic blocks and a 16 bit `context-hash`. Since the LBR is a FIFO, we maintain the `runtime-hash` incrementally. Using a counting Bloom filter [39, 40], we assign a 6-bit counter to each of the 16 bits of the `runtime-hash` (96 bits in total). Whenever a new entry is added into the LBR, we hash the corresponding block address and increment the corresponding counters in the `runtime-hash`; the counters for the hash of the evicted LBR entry are decremented. The counters never overflow and the `runtime-hash` precisely tracks the LBR contents since there are only ever 32 branches recorded in the `runtime-hash`. We also add a small amount of logic to reduce each counter to a single "is-zero" bit; in those 16 bits, we check if the `context-hash` bits are a subset of the `runtime-hash`. If they are, the prefetch fires, otherwise it is disabled.

To clarify how Bloom filters help *I-SPY* match `runtime-hash` to `context-hash`, let's consider the same example in Fig. 6. Let's assume the 16-bit hashes of *B* and *E* are `0x2` and `0x10`, respectively. Therefore, the `context-hash` would be `0x12`, where the Least Significant Bits (LSB) 1 and 4 are set. To enable prefetching, `runtime-hash` must also have these bits set. At run time, if *B* is present in the last 32 predecessors, the bloom filter counter corresponding to LSB-1 must be greater than 0. Similarly for *E*, the counter corresponding to LSB-4 must be greater than 0. Hence, the result of subset comparison between `context-hash` and `runtime-hash` will be true and a prefetch will be triggered.

B. Prefetching Coalescing

Conditional prefetching enables high-accuracy prefetching. Nevertheless, it leads to static code bloat as every prefetch instruction increases the size of the application's text segment. *Prefetch coalescing* reduces the static code bloat as well as the number of dynamically-executed prefetch instructions by combining multiple prefetches into a single instruction. We first describe how *I-SPY* decides which lines should be coalesced, followed by details of *I-SPY*'s coalesced prefetching instruction. We then detail the micro-architectural modifications required to support prefetch coalescing.

To perform coalescing, *I-SPY* analyzes all prefetch instructions injected into a basic block and groups them by context. As shown in Fig. 8, prefetches for addresses `0xA` and `0xD` are grouped together since they are conditional on the same context, `C0`. Similarly, `0x4`, `0x2`, and `0x7` are grouped together since they share the same context `C1`.

Next, *I-SPY* attempts to merge a group of prefetch instructions into a single prefetch instruction. *I-SPY* uses an *n*-bit bitmap to select a subset of cache lines within a window of *n* consecutive cache lines. In the example shown in Fig. 8, the coalesced prefetch for context `C0` has two bits set in the bitmask to encode lines `0x4` and `0x7` where the base address of the prefetch is `0x2`. While a larger bitmask allows coalescing more prefetches, it also increases hardware complexity. We study the effect of bitmask size in Fig. 17.

Coalesced prefetch instruction. Our proposed coalesced prefetch instruction, `Lprefetch`, requires an additional operand for specifying the coalescing bit-vector. Prefetch instructions in current hardware (e.g., `prefetcht*` on x86 and `pli` on ARM) follow the format, (`prefetch`, `address`), which takes `address` as an operand and prefetches the cache line corresponding to `address`. `Lprefetch` takes an extra operand, `bit-vector`. The `prefetcht*` instruction on x86 has a size of 7 bytes, hence, with the addition of an $n = 8$ bits bitmask, `Lprefetch` has a size of 8 bytes.

I-SPY combines prefetch coalescing and conditional prefetching via another instruction, `CLprefetch`, with the format (`prefetch`, `address`, `context-hash`, `bit-vector`) as shown in Fig. 8. `CLprefetch` prefetches all the prefetch targets specified by `bit-vector` only if the current context matches with the context encoded in the `context-hash`.

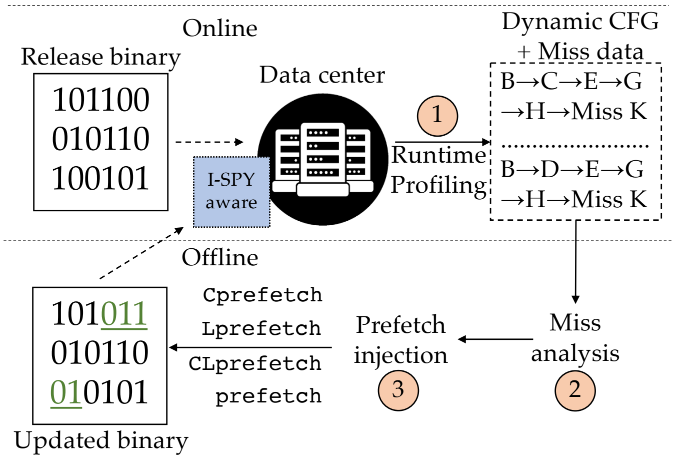


Fig. 9: Usage model of *I-SPY*

This new instruction has a size of 10 bytes (2 extra bytes to specify `context-hash`).

Micro-architectural modifications. Coalesced prefetch instructions require minor micro-architectural modifications that mainly consists of a series of simple incrementers. These incrementers decode the 8-bit coalescing vector and enable prefetching up to 9 cache lines (the initial prefetch target, plus up to 8 bit-vector-dependent targets). The resultant cache line addresses are then forwarded to the prefetch engine.

Replacement policy for prefetched lines. *I-SPY*'s prefetch instructions also update the replacement policy priority of the prefetched cache line. Instead of assigning the highest priority to the prefetched cache line (as done for demand-loads), *I-SPY*'s prefetch instructions assign the prefetched cache line a priority equal to the half of the highest priority. *I-SPY*'s goal with this policy is to reduce the adverse effects of a potentially inaccurate prefetch operation.

IV. Usage Model

We provide an overview of the high-level usage model of *I-SPY* in Fig. 9. *I-SPY* profiles an application's execution at run time, and uses these profiles to perform an offline analysis of I-cache misses to suitably inject code prefetch instructions.

Online profiling. *I-SPY* first profiles an application's execution at run time (step ①). It uses Intel's LBR [22] to construct a dynamic CFG (such as the one shown in Fig. 2), and augments the dynamic CFG with L1 I-cache miss profiles collected with Intel's PEBS [33] hardware performance counters. At every I-cache miss, *I-SPY* records the program counters of the previous 32 branches that the program executed (on x86_64, LBR has a 32-entry limit). Run-time profiling using Intel LBR's and Intel PEBS's lightweight monitoring [22, 41] enables profiling applications online, in production.

Offline analysis. Next, *I-SPY* performs an offline analysis (②) of the miss-annotated dynamic CFG that it generates at run time. For each miss, *I-SPY* considers all predecessor basic blocks within the prefetch window. Unlike prior work [2], *I-SPY* does not require the per-application IPC metric to find predecessors within the prefetch window as the LBR profile already includes dynamic cycle information for each basic block. Apart from this, the algorithm to find the best prefetch injection site is similar to prior work [2] and has a worst-case complexity of $O(n \log n)$.

After finding the best prefetch injection site to cover each miss, *I-SPY* runs two extra analyses, context discovery and prefetch coalescing. First, if the prefetch injection site has a non-zero fan-out, *I-SPY* analyzes the predecessors of the injection site to reduce its fan-out (Fig. 6). Next, if the same injection site is selected for prefetching multiple cache lines, *I-SPY* applies prefetch coalescing to reduce the number of prefetch instructions (Fig. 8).

Once *I-SPY* finishes identifying opportunities for conditional prefetching and prefetch coalescing, it injects appropriate prefetch instructions to cover all misses. Specifically, *I-SPY* injects four kinds of prefetch instructions (③).

If the context of a given prefetch instruction differs from the contexts of all other prefetch instructions, then this prefetch instruction cannot be coalesced with others. In that case, *I-SPY* injects a `Cprefetch` instruction.

Conditionally prefetching a line based on the execution context may not improve the prefetch accuracy. In this case, *I-SPY* will try to inject an `Lprefetch` instruction. If multiple cache lines are within a range of n lines (where n is the size of `bit-vector` used to perform coalescing as in §III-B) from the nearest prefetch target, *I-SPY* will inject an `Lprefetch`. Otherwise, *I-SPY* will inject multiple AsmDB-style `prefetch` instructions that simply prefetch a single target cache line.

If conditional prefetching improves prefetching accuracy and multiple cache lines can be coalesced, *I-SPY* injects `CLprefetch` instructions.

The new binary updated with code prefetch instructions is deployed on I-SPY-aware data center servers that can conditionally execute and (or) coalesce the injected prefetches.

V. Evaluation Methodology

We envision an end-to-end *I-SPY* system that uses application profile information and our proposed family of hardware code prefetch instructions. We evaluate *I-SPY* using simulation since existing server-class processors do not support our proposed hardware modifications for conditional prefetching and prefetch coalescing. Additionally, simulation enables replaying memory traces to conduct limit studies and compare *I-SPY*'s performance against an ideal prefetch mechanism. We prototype the state-of-the-art prefetcher, AsmDB [2], and compare *I-SPY* against it. We now describe (1) the experimental setup that we use to collect an application's execution profile, (2) our simulation infrastructure, (3) *I-SPY*'s system parameters, and (4) the data center applications we study.

Data collection. During *I-SPY*'s offline phase, we use Intel's LBR [22] and PEBS counters [42] (more specifically (`frontend_retired.l1i_miss`)) to collect an application's execution profile and L1 I-cache miss information. We record up to 100 million instructions executed in steady-state.

TABLE I: Simulated System

Parameter	Value
CPU	Intel Xeon Haswell
Number of cores per socket	20
L1 instruction cache	32 KiB, 8-way
L1 data cache	32 KiB, 8-way
L2 unified cache	1 MB, 16-way
L3 unified cache	Shared 10 MiB/core, 20-way
All-core turbo frequency	2.5 GHz
L1 I-cache latency	3 cycles
L1 D-cache latency	4 cycles
L2 cache latency	12 cycles
L3 cache latency	36 cycles
Memory latency	260 cycles
Memory bandwidth	6.25 GB/s

We combine our captured miss profiles and instruction traces to construct an application's miss-annotated dynamic CFG.

Simulation. We use the ZSim simulator [43] to evaluate *I-SPY*. We modify ZSim [43] to support conditional prefetching and prefetch coalescing. We use ZSim in a trace-driven execution mode, modeling an out-of-order processor. The detailed system parameters are summarized in Table I. Additionally, we extend ZSim to support our family of hardware code prefetch instructions. Our implemented code prefetch instructions insert prefetched cache lines with a lower replacement policy priority than any demand load requests.

System parameters. Based on the sensitivity analysis (see Fig. 18), we use 27 cycles as minimum prefetch distance, and 200 cycles as maximum prefetch distance. Additionally, we empirically determine that coalescing non-contiguous prefetches that occur within a cache line window of 8 cache lines yields the best performance.

Data center applications. We evaluate nine popular data center applications described in Sec. II. We allow an application's binary to be built with classic compiler code layout optimizations such as in-lining [44], hot/cold splitting [45], or profile-guided code alignment [19]. We study these applications with different input parameters offered to the client's load generator (e.g., number of requests per second or the number of threads).

Evaluation metrics. We use six evaluation metrics to evaluate *I-SPY*'s effectiveness. First, we compare *I-SPY*'s performance improvement against an ideal cache and AsmDB. Second, we study how well *I-SPY* reduces L1 I-cache MPKI compared to the state-of-the-art prefetcher, AsmDB [2]. Third, we analyze how much performance improvement stems from conditional prefetching and prefetch coalescing, individually. Fourth, we compare *I-SPY*'s prefetch accuracy with AsmDB. Fifth, we analyze the static and dynamic code footprint increase induced by *I-SPY*. Sixth, we determine whether *I-SPY* achieves high performance across various application inputs. Since, data center applications often run continuously, application inputs can drastically vary (e.g., diurnal load trends or load transients [46, 47]). Hence, a profile-guided optimization for data center applications must be able to improve performance across diverse inputs.

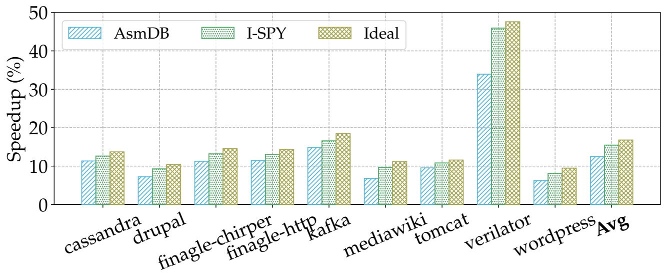


Fig. 10: *I-SPY*'s speedup compared to an ideal cache and AsmDB: *I-SPY* achieves an average speedup that is 90.4% of ideal.

We also perform a sensitivity analysis of *I-SPY*'s system parameters by evaluating the effect of varying the (1) number of predecessors in `context-hash`, (2) minimum and maximum prefetch distances, (3) coalescing size, and (4) context size used to conditionally prefetch.

VI. Evaluation

In this section, we evaluate how *I-SPY* improves application performance compared to an ideal cache implementation and the state-of-the-art prefetcher [2], AsmDB, using the evaluation metrics defined in §V. We then perform sensitivity studies to determine the effect of varying *I-SPY*'s configurations.

A. *I-SPY: Performance Analysis*

Speedup. We first evaluate the speedup achieved by *I-SPY* across all applications. In Fig. 10, we show *I-SPY*'s speedup (green bars) compared against an ideal cache that faces no misses (brown bars) and AsmDB [2] (blue bars).

We find that *I-SPY* attains a near-ideal speedup, achieving an average speedup that is 90.4% (up to 96.4%) of an ideal cache that always hits in the L1 I-cache. *I-SPY* falls slightly short of an ideal cache since (1) it executes more instructions due to the injected prefetch instructions and (2) a previously unobserved execution context might not trigger a prefetch, precipitating a miss. Additionally, *I-SPY* outperforms AsmDB by 22.4% on average (up to 41.2%), since it eliminates more I-cache misses than AsmDB as we show next.

L1 I-cache MPKI reduction. We next evaluate how well *I-SPY* reduces L1 I-cache misses compared to AsmDB [2] in Fig. 11. We evaluate across all nine applications.

We observe that *I-SPY* achieves a high miss coverage, reducing L1 I-cache MPKI by an average of 95.8% across all applications. Furthermore, *I-SPY* reduces MPKI compared to AsmDB by an average of 15.7% across all applications (the greatest improvement is 28.4% for *verilator*). The MPKI reduction is due to conditionally executing prefetches and coalescing them, thereby eliminating more I-cache misses. In contrast, AsmDB executes a large number of unused prefetches that evict useful data from the cache.

Performance of conditional prefetching and prefetch coalescing. In Fig. 12, we quantify how much *I-SPY*'s conditional

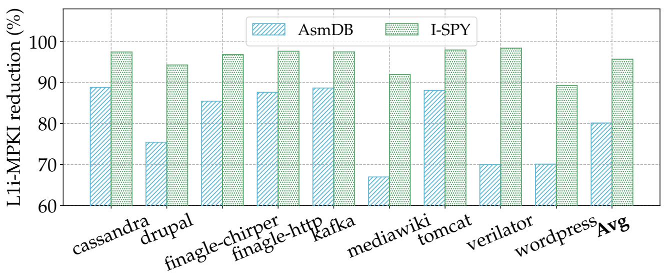


Fig. 11: *I-SPY*'s L1 I-cache MPKI reduction compared with AsmDB: *I-SPY* removes 15.7% more misses than AsmDB.

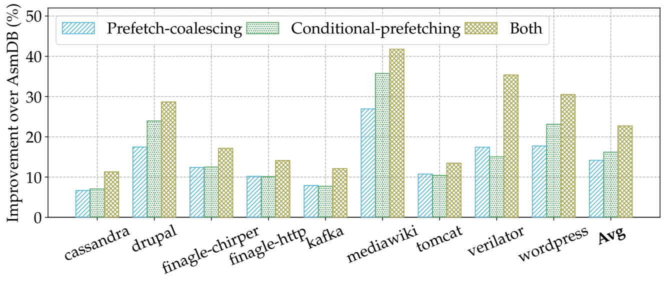


Fig. 12: Speedup achieved by conditional prefetching and prefetch coalescing over AsmDB: Conditional prefetching often offers better speedup than coalescing, but their combined speedup is significantly better.

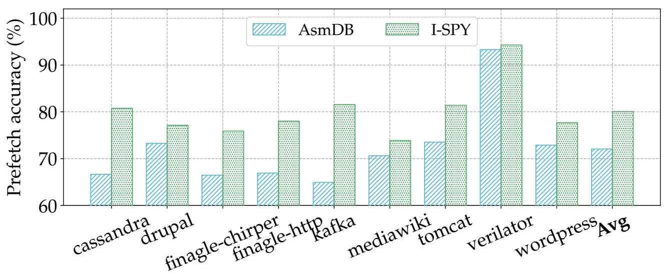


Fig. 13: *I-SPY*'s prefetch accuracy compared with AsmDB: *I-SPY* achieves an average of 8.2% better accuracy than AsmDB.

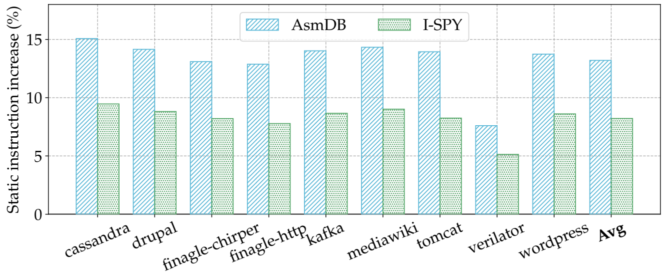


Fig. 14: *I-SPY*'s static code footprint increase compared to AsmDB: *I-SPY* statically injects 37% (average) fewer instructions than AsmDB.

prefetching and prefetch coalescing mechanisms contribute to net application speedup. We show the performance improvement achieved by these novel mechanisms over AsmDB, across all nine applications. We make two observations.

First, we note that both conditional prefetching and prefetch coalescing provide gains over AsmDB across all applications. Conditional prefetching improves performance more than coalescing for eight of our applications, since it covers more I-cache misses with better accuracy. In *verilator*, we observe that coalescing offers a better performance since 75% of *verilator*'s misses have a high spatial locality even within a cache line window of 8 lines.

Second, we find that the performance gains achieved by conditional prefetching and prefetch coalescing are not strictly additive. As *I-SPY* only coalesces prefetches that have the same condition, many prefetch instructions that depend on different conditions are not coalesced. Yet, combining both techniques offers better speedup than their individual counterparts.

Prefetch accuracy. We portray the prefetch accuracy achieved by *I-SPY* across all nine applications in Fig. 13. We also compare *I-SPY*'s prefetch accuracy against AsmDB.

We find that *I-SPY* achieves an average of 80.3% prefetch accuracy. Furthermore, *I-SPY*'s accuracy is 8.2% (average) better than AsmDB's accuracy, since *I-SPY*'s conditional prefetching avoids trading off prefetch accuracy for miss coverage, unlike AsmDB.

Static and dynamic code footprint increase. We next evaluate by how much *I-SPY* increases applications' static and dynamic code footprints. First, we illustrate the static code footprint increase induced by *I-SPY* in Fig. 14. We also compare against AsmDB's static code footprint.

We observe that *I-SPY* increases the static code footprint by 5.1% - 9.5% across all applications. By coalescing multiple prefetches into a single prefetch instruction, *I-SPY* introduces fewer prefetch instructions into the application's binary. In contrast, we find that AsmDB increases the static code footprint much more starkly—7.6% - 15.1%.

Next, we study by how much *I-SPY* increases the dynamic application footprint in Fig. 15 across all nine applications. We note that *I-SPY* executes 3.7% - 7.2% additional dynamic instructions since it covers I-cache misses by executing injected code prefetch instructions. We observe that AsmDB has a higher dynamic instruction footprint across eight applications (ranging from 5.5% - 11.6%), since it does not coalesce prefetches like *I-SPY*. For *verilator*, *I-SPY*'s dynamic footprint is higher than AsmDB since *I-SPY* covers 28.4% more misses than AsmDB by executing more prefetch instructions, while also providing 35.9% performance improvement over AsmDB.

Generalization across application inputs. To determine whether *I-SPY* achieves a performance improvement with an application input that is different from the profiled input, we characterize *I-SPY*'s performance for five different inputs fed to three of our applications—*drupal*, *mediawiki*, *wordpress* (Fig. 16). We choose these three applications, because they have the greatest variety of readily-available test inputs that we can run. We compare *I-SPY* against AsmDB in terms of ideal cache performance.

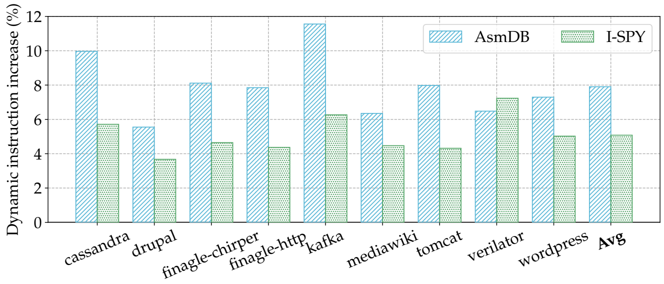


Fig. 15: *I-SPY*'s dynamic code footprint increase compared to AsmDB: On average, *I-SPY* executes 36% fewer prefetch instructions than AsmDB.

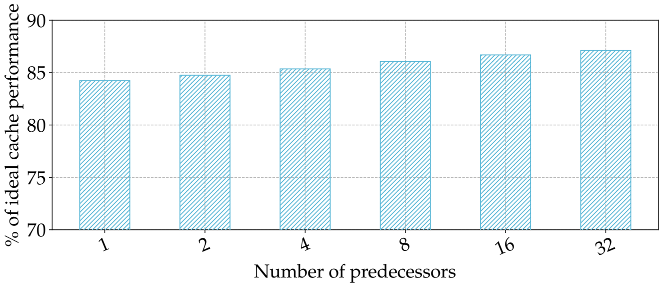


Fig. 17: *I-SPY*'s conditional prefetching achieves better performance with an increase in the number of predecessors comprising the context.

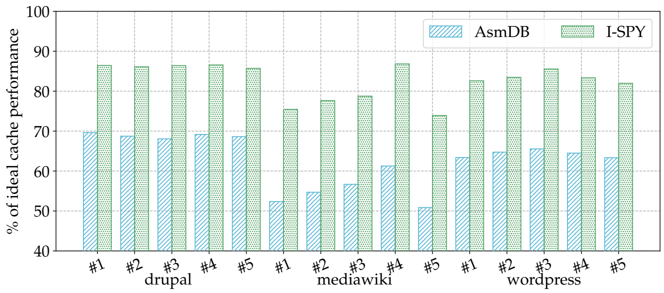


Fig. 16: *I-SPY*'s performance compared against AsmDB for different application test inputs: *I-SPY* outperforms AsmDB when the application input differs from the profiled input.

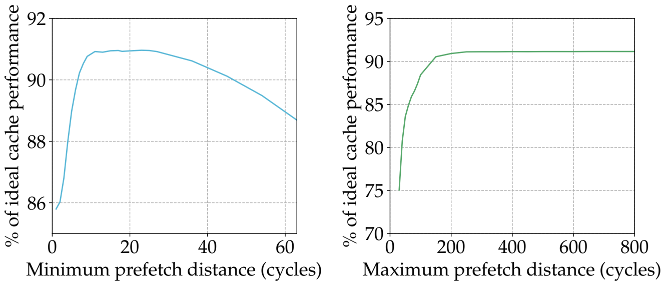


Fig. 18: *I-SPY*'s average performance variation in response to changes in the minimum (left) and the maximum (right) prefetch distance.

We observe that *I-SPY* achieves a speedup that is closer to the ideal speedup than the speedup provided by AsmDB across all test inputs. *I-SPY* is more resilient to the input changes than AsmDB because of conditional prefetching. *I-SPY* achieves at least 70% (up to 86.84%) of ideal cache performance on inputs that are different from the profiled input.

B. *I-SPY: Sensitivity Analysis*

We next evaluate how *I-SPY*'s performance varies in response to variations of the different system parameters.

Number of predecessors comprising the context. In Fig. 17, we observe how the *I-SPY* conditional prefetching's performance varies in response to a variation in the number of predecessors comprising the context condition (see Sec III-A). We vary predecessor counts from 1 to 32 (with a geometric progression of 2) and show the *I-SPY* conditional prefetching's average performance improvement across all nine applications.

We find that the *I-SPY* conditional prefetching's performance improves with an increase in the number of predecessors composing the context condition. Using more predecessors enables a more complete context description, and slightly improves performance by predicting I-cache misses more accurately. However, a large number of predecessors impose a significant context-discovery computation overhead. Specifically, the search space of possible predecessor candidates grows exponentially with the number of predecessors comprising the context condition. Consequently, the context discovery process takes tens of minutes to complete with more than 4 predecessors, which can be a bottleneck in the build process. Since *I-SPY*'s conditional prefetching achieves more than 85% of ideal cache performance even with four predecessors, *I-SPY*'s design uses four predecessors to define context and keeps the computational overhead of context discovery low.

Minimum and maximum prefetch distance. We next analyze how *I-SPY*'s performance varies with an increase in the minimum and maximum prefetch distances, in Fig. 18. We observe that *I-SPY* achieves maximum performance for a minimum prefetch distance of 20-30 cycles (which is greater than typical L2 access latency but less than L3 access latency). On the other hand, an increase in the maximum prefetch distance always improves *I-SPY*'s performance. However, the increase is very slow after 200 cycles. Based on these results, we use 27 cycles as the minimum prefetch distance, and 200 cycles as the maximum prefetch distance for *I-SPY*.

Coalescing size. We next study the sensitivity of *I-SPY*'s prefetch coalescing to the coalesce bitmask size (see §III-B) in Fig. 19. We vary the coalesce bitmask size from 1 bit to 64 bits, prefetching up to 2 and 65 cache lines using a single instruction, respectively. We then measure the percentage of ideal speedup achieved by *I-SPY*'s prefetch coalescing as an average across all applications.

We note that *I-SPY*'s performance improves slightly with a larger bitmask, since larger bitmasks enable coalescing more cache lines, reducing spurious evictions. However, a large

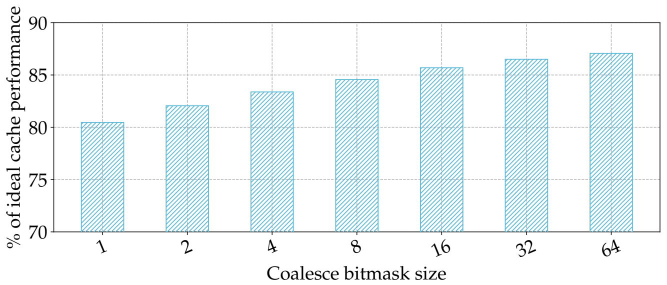


Fig. 19: *I-SPY*'s average performance variation in response to increasing the coalescing size: Larger coalescing sizes achieve higher gains.

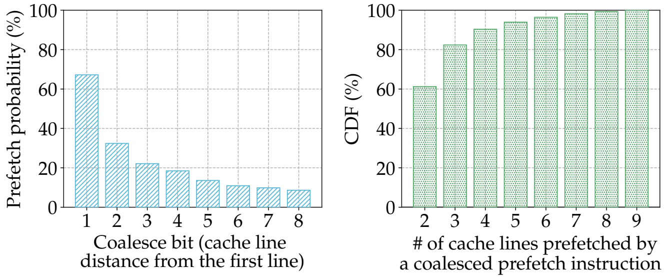


Fig. 20: (left) The probability of coalesced prefetching reduces with an increase in cache line distance. (right) Coalesced prefetch instructions usually bring in less than 4 cache lines.

bitmask will introduce hardware design complexities since the microarchitecture must now support additional in-flight prefetch operations. Similar to prior work [3], to minimize hardware complexity, we design *I-SPY* with an 8-bit coalescing bitmask, since it can be implemented with minor hardware modifications (as described in §III-B).

Additionally, we examine which and how many nearby cache lines a coalesced prefetch instruction typically prefetches for all nine applications. As shown in Fig. 20, the probability of coalesced prefetching reduces with an increase in cache line distance. Moreover, most coalesced prefetch instructions (82.4% averaged across nine applications) prefetch less than four cache lines.

Context hash size. We next analyze how *I-SPY*'s false positive rate varies with an increase in the context hash size, in Fig. 21. We study the *wordpress* benchmark since its speedup is heavily impacted by prefetch accuracy (see Fig. 3).

We observe that increasing the number of bits in the context hash reduces the false positive rate. However, an increase in the context hash size increases the static code footprint, as shown in Fig. 21. To minimize the static code footprint while still achieving a low false positive rate, *I-SPY*'s design uses a 16-bit context hash—13% false positive rate and 4.6% static code increase.

VII. Discussion

In this section we discuss some limitations of *I-SPY* and offer potential solutions.

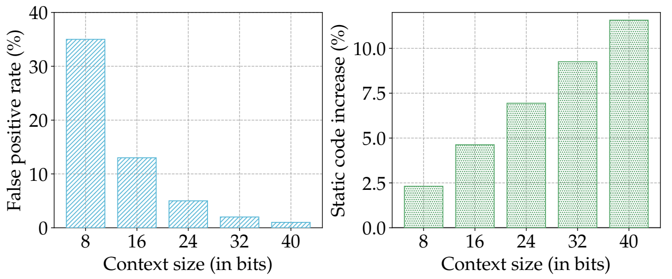


Fig. 21: (left) *I-SPY*'s false positive rate variation in response to an increase in context size: False positives are reduced with a larger context; (right) *I-SPY*'s static code footprint size variation in response to context size: Static code footprint increases with an increase in context size.

Prefetching already resident cache lines. Although our process of discovering high-probability contexts that lead to cache misses is effective, we also found that many times, the target cache line of a `Cprefetch` is already resident in the cache. However, the overhead of such resident prefetch operations is low since they do not poison the cache by bringing in new unnecessary cache lines. To make this overhead even lower, we design our proposed prefetch instructions such that they are always inserted with a lower priority as demand loads in regards to the replacement policy.

Prefetching within JITted code. Most instruction cache misses in code generated at run time are out of *I-SPY*'s scope. While *I-SPY* is able to prefetch for some of these misses via `Cprefetch` instructions inserted into non-JITted code, there are still up to 10% of code misses in JITted code (mostly for the three HHVM applications, *wordpress*, *drupal*, and *mediawiki*) that are not covered. To handle these additional misses, *I-SPY* could be integrated with a JIT compiler since all of *I-SPY*'s offline machinery (which leverages hardware performance monitoring mechanisms) can, in principle, be used online by the runtime instead.

VIII. Related Work

The performance criticality of instruction cache misses has resulted in a rich body of prior literature. We discuss three categories of related work.

Software prefetching. Several software techniques [17, 21, 48–53] improve instruction locality by relocating infrequently executed code via Profile-Guided Optimizations (PGO) at compile time [17], link time [16, 18], or post link time [15, 19]. However, finding the optimal cache-conscious layout is intractable in practice [2], since it requires meandering through a vast number of control-flow combinations. Hence, existing techniques must oftentimes make inaccurate control-flow approximations. Whereas PGO-based techniques have been shown to improve data center application performance [17, 19], they still eliminate only a small subset of all instruction cache misses [2].

Hardware prefetching. Hardware instruction prefetching techniques began with next-line instruction prefetchers that exploit the common case of fetching sequential instructions [54].

These next-line prefetchers soon evolved into next-N-line and instruction stream prefetchers [8–14] that use trigger events and control mechanisms to prefetch by adaptively looking a few instructions ahead. Next-line and stream prefetchers have been widely deployed in industrial designs because of their implementation simplicity. However, such next-line prefetchers are often inaccurate for complex data center applications that implement frequent branching and function calls.

Branch predictor based prefetchers [3, 4, 10, 34, 35, 55, 56] improve prefetch accuracy in branch- and call-heavy code. Runahead execution [57], wrong path instruction prefetching [58], and speculative prefetching mechanisms [59, 60] can also explore ahead of the instruction fetch unit. However, such prefetchers are susceptible to interference precipitated by wrong path execution and insufficient look ahead when the branch predictor traverses loop branches [12].

TIFS [12] and PIF [10] record the instruction fetch miss and instruction commit sequences to overcome the limitations of branch predictor based prefetching. Whereas these mechanisms have improved accuracy and miss coverage, they require considerable on-chip storage to maintain an ordered log of instruction block addresses. Increasing on-chip storage is impractical at data center scale due to strict energy requirements.

More sophisticated hardware instruction prefetchers proposed by prior works (e.g., trace caches and special hardware replacement policies) [14, 61–63] are too complex to be deployed. We conclude that hardware prefetching mechanisms either provide low accuracy and coverage or they require significant on-chip storage and are too complex to implement in real hardware.

In comparison, *I-SPY* covers most instruction cache misses with minor micro-architectural modifications. *I-SPY* requires only 96-bits of extra storage while state-of-the-art hardware prefetchers (e.g., SHIFT [13], Confluence [64], and Shotgun [3]) require kilobytes to megabytes of extra storage.

Hybrid hardware-software prefetching. Hybrid hardware-software techniques [2, 21] attempt to overcome the limitations of hardware-only and software-only prefetching mechanisms. These mechanisms propose hardware code prefetch instructions [65] that are similar to existing data prefetch instructions [66]. They use software-based control flow analyses to inject hardware code prefetch instructions.

Although existing hybrid instruction prefetching mechanisms have been the most effective in reducing I-cache misses in datacenter applications [2], they suffer from key limitations that hurt prefetch accuracy. First, such hybrid techniques rely on a single predecessor basic block as the execution context to predict a future cache miss. However, as we show in Section II, we find that miss patterns are more complex and multiple predecessor basic blocks are needed to construct the execution context to accurately predict a future cache miss. Second, existing hybrid prefetching techniques often execute far too many dynamic prefetch instructions, further increasing application code footprints. In contrast, *I-SPY* achieves near-ideal prefetch accuracy via conditional prefetching, while allowing only a small increase in application footprint.

IX. Conclusion

Large instruction working sets in modern data center applications have resulted in frequent I-cache misses that significantly degrade data center performance. We investigated instruction prefetching to address this problem and analyze the challenges of designing an ideal instruction prefetcher. We then used insights derived from our investigation to develop *I-SPY*, a novel profile-driven prefetching technique. *I-SPY* exposes two new instruction prefetching techniques: *conditional prefetching* and *prefetch coalescing* via a family of light-weight hardware code prefetch instructions. We evaluated *I-SPY* on nine widely-used data center applications to demonstrate an average of 15.5% (up to 45.9%) speedup and 95.9% (and up to 98.4%) reduction in instruction cache misses, outperforming the state-of-the-art prefetching technique by 22.5%.

Acknowledgments

We thank anonymous reviewers for their insightful feedback and suggestions. This work was supported by the Intel Corporation, the NSF FoMR grants #1823559 and #2011168, and the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. We thank Gagan Gupta and Rathijit Sen from Microsoft Corporation for insightful suggestions on the characterization of data center applications. We thank Grant Ayers from Google for excellent discussions and helpful feedback.

References

[1] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 158–169.

[2] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 462–473.

[3] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the front-end bottleneck with shotgun," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 30–42, 2018.

[4] R. Kumar, C.-C. Huang, B. Grot, and V. Nagarajan, "Boomerang: A metadata-free architecture for control flow delivery," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 493–504.

[5] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory hierarchy for web search," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 643–656.

[6] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," *Acm sigplan notices*, vol. 47, no. 4, pp. 37–48, 2012.

[7] A. Sriraman, A. Dhanotia, and T. F. Wenisch, "Softsku: Optimizing server architectures for microservice diversity@ scale," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 513–526.

[8] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, no. 12, pp. 7–21, 1978.

[9] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1999, pp. 16–27.

[10] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *International Symposium on Microarchitecture*, 2008.

[11] R. Panda, P. V. Gratz, and D. A. Jiménez, "B-fetch: Branch prediction directed prefetching for in-order processors," *IEEE Computer Architecture Letters*, vol. 11, no. 2, pp. 41–44, 2011.

[12] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *International Symposium on Microarchitecture*, 2011.

[13] C. Kaynak, B. Grot, and B. Falsafi, "Shift: Shared history instruction fetch for lean-core server processors," in *International Symposium on Microarchitecture*, 2013.

[14] A. Kolli, A. Saidi, and T. F. Wenisch, "Rdip: return-address-stack directed instruction prefetching," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2013, pp. 260–271.

[15] C.-K. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney, "Ispike: a post-link optimizer for the intel/spl reg/itanium/spl reg/architecture," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 15–26.

[16] D. X. Li, R. Ashok, and R. Hundt, "Lightweight feedback-directed cross-module optimization," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010, pp. 53–61.

[17] D. Chen, T. Moseley, and D. X. Li, "Autofdo: Automatic feedback-directed optimization for warehouse-scale applications," in *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2016, pp. 12–23.

[18] G. Ottoni and B. Maher, "Optimizing function placement for large-scale data-center applications," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017, pp. 233–244.

[19] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "Bolt: a practical binary optimizer for data centers and beyond," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 2–14.

[20] E. Petrank and D. Rawitz, "The hardness of cache conscious data placement," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 101–112. [Online]. Available: https://doi.org/10.1145/503272.503283

[21] C.-K. Luk and T. C. Mowry, "Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors," in *International Symposium on Microarchitecture*, 1998.

[22] "An introduction to last branch records," https://lwn.net/Articles/680985/.

[23] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 35–44.

[24] "facebookarchive/oss-performance: Scripts for benchmarking various php implementations when running open source software," https://github.com/facebookarchive/oss-performance, 2019, (Online; last accessed 15-November-2019).

[25] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer *et al.*, "The dacapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2006, pp. 169–190.

[26] "Apache cassandra," http://cassandra.apache.org/.

[27] "Apache tomcat," https://tomcat.apache.org/.

[28] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tůma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon, T. Würthinger, and W. Binder, "Renaissance: Benchmarking suite for parallel applications on the jvm," in *Programming Language Design and Implementation*, 2019.

[29] "Twitter finagle," https://twitter.github.io/finagle/.

[30] "Verilator," https://www.veripool.org/wiki/verilator.

[31] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *International Symposium on Code Generation and Optimization*, 2003.

[32] K. Hazelwood and J. E. Smith, "Exploring code cache eviction granularities in dynamic optimization systems," in *International Symposium on Code Generation and Optimization*, 2004.

[33] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation intel core: New microarchitecture code-named skylake," *IEEE Micro*, 2017.

[34] J. Bonanno, A. Collura, D. Lipetz, U. Mayer, B. Prasky, and A. Saporito, "Two level bulk preload branch prediction," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2013, pp. 71–82.

[35] I. Burcea and A. Moshovos, "Phantom-btb: a virtualized branch target buffer design," *Acm Sigplan Notices*, vol. 44, no. 3, pp. 313–324, 2009.

[36] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 413–425.

[37] Wikipedia contributors, "Fowler–noll–vo hash function — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Fowler%E2%80%93Noll%E2%80%93Vo_hash_function&oldid=931348563, 2019, [Online; accessed 17-April-2020].

[38] ——, "Murmurhash — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=MurmurHash&oldid=950347972, 2020, [Online; accessed 17-April-2020].

[39] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM transactions on networking*, vol. 8, no. 3, pp. 281–293, 2000.

[40] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *European Symposium on Algorithms*. Springer, 2006, pp. 684–695.

[41] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," *ACM SIGPLAN Notices*, 2017.

[42] I. Corparation, "Intel (r) 64 and ia-32 architectures software developer's manual," *Combined Volumes, Dec*, 2016.

[43] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *International Symposium on Computer Architecture*, 2013.

[44] A. Ayers, R. Schooler, and R. Gottlieb, "Aggressive inlining," *ACM SIGPLAN Notices*, 1997.

[45] T. M. Chilimbi, B. Davidson, and J. R. Larus, "Cache-conscious structure definition," in *ACM SIGPLAN conference on Programming language design and implementation*, 1999.

[46] A. Sriraman and T. F. Wenisch, "utune: Auto-tuned threading for OLDI microservices," in *Symposium on Operating Systems Design and Implementation*, 2018.

[47] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," *arXiv preprint arXiv:2003.03423*, 2020.

[48] S. Harizopoulos and A. Ailamaki, "Steps towards cache-resident transaction processing," in *International conference on Very large data bases*, 2004.

[49] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, P. G. Lowney, and M. Valero, "Code layout optimizations for transaction processing workloads," *ACM SIGARCH Computer Architecture News*, 2001.

[50] J. Zhou and K. A. Ross, "Buffering databse operations for enhanced instruction cache performance," in *International conference on Management of data*, 2004.

[51] L. L. Peterson, "Architectural and compiler support for effective instruction prefetching: a cooperative approach," *ACM Transactions on Computer Systems*, 2001.

[52] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying memory access patterns for prefetching," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 513–526.

[53] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," *arXiv preprint arXiv:1803.02329*, 2018.

[54] D. Anderson, F. Sparacio, and R. M. Tomasulo, "The ibm system/360 model 91: Machine philosophy and instruction-handling," *IBM Journal of Research and Development*, 1967.

[55] L. Spracklen, Y. Chou, and S. G. Abraham, "Effective instruction prefetching in chip multiprocessors for modern commercial applications," in *International Symposium on High-Performance Computer Architecture*, 2005.

[56] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak, "Branch history guided instruction prefetching," in *International Symposium on High-Performance Computer Architecture*, 2001.

[57] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An effective alternative to large instruction windows," *IEEE Micro*, 2003.

[58] J. Pierce and T. Mudge, "Wrong-path instruction prefetching," in *International Symposium on Microarchitecture*, 1996.

[59] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, “Slipstream processors: Improving both performance and fault tolerance,” *ACM SIGPLAN Notices*, 2000.

[60] C. Zilles and G. Sohi, “Execution-based prediction using speculative slices,” in *International Symposium on Computer Architecture*, 2001.

[61] E. Rotenberg, S. Bennett, and J. E. Smith, “Trace cache: a low latency approach to high bandwidth instruction fetching,” in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE, 1996, pp. 24–34.

[62] Q. Jacobson, E. Rotenberg, and J. E. Smith, “Path-based next trace prediction,” in *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 1997, pp. 14–23.

[63] S. M. Ajorpaz, E. Garza, S. Jindal, and D. A. Jiménez, “Exploring predictive replacement policies for instruction cache and branch target buffer,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 519–532.

[64] C. Kaynak, B. Grot, and B. Falsafi, “Confluence: unified instruction supply for scale-out servers,” in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 166–177.

[65] “Prfm (literal) — a64,” http://shell-storm.org/armv8-a/ISA_v85A_A64_xml_00bet8_OPT/xhtml/prfm_reg.html, [Online; accessed 28-March-2020].

[66] “Prefetchh: Prefetch data into caches (x86 instruction set reference),” https://c9x.me/x86/html/file_module_x86_id_252.html, [Online; accessed 28-March-2020].