AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers

Grant Ayers[*]
Stanford University

Nayana Prasad Nagendra[*]
Princeton University

David I. August
Princeton University

Hyoun Kyu Cho
Google

Svilen Kanev
Google

Christos Kozyrakis
Stanford University

Trivikram Krishnamurthy[*]
Nvidia

Heiner Litz[*]
UC Santa Cruz

Tipp Moseley
Google

Parthasarathy Ranganathan
Google

ABSTRACT

The large instruction working sets of private and public cloud workloads lead to frequent instruction cache misses and costs in the millions of dollars. While prior work has identified the growing importance of this problem, to date, there has been little analysis of where the misses come from, and what the opportunities are to improve them. To address this challenge, this paper makes three contributions. First, we present the design and deployment of a new, always-on, fleet-wide monitoring system, AsmDB, that tracks front-end bottlenecks. AsmDB uses hardware support to collect bursty execution traces, fleet-wide temporal and spatial sampling, and sophisticated offline post-processing to construct full-program dynamic control-flow graphs. Second, based on a longitudinal analysis of AsmDB data from real-world online services, we present two detailed insights on the sources of front-end stalls: (1) cold code that is brought in along with hot code leads to significant cache fragmentation and a corresponding large number of instruction cache misses; (2) distant branches and calls that are not amenable to traditional cache locality or next-line prefetching strategies account for a large fraction of cache misses. Third, we prototype two optimizations that target these insights. For misses caused by fragmentation, we focus on `memcmp`, one of the hottest functions contributing to cache misses, and show how fine-grained layout optimizations lead to significant benefits. For misses at the targets of distant jumps, we propose new hardware support for software code prefetching and prototype a new feedback-directed compiler optimization that combines static program flow analysis with dynamic miss profiles to demonstrate significant benefits for several large warehouse-scale workloads. Improving upon prior work, our proposal avoids invasive hardware modifications by prefetching via software in an efficient and scalable way. Simulation results show that such an approach can eliminate up to 96% of instruction cache misses with negligible overheads.

ACM Reference Format:
Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307650.3322234

1 INTRODUCTION

Nearly every device across the world, from IOT and mobile devices to self-driving cars, is now being served by services hosted in huge datacenters, which are dubbed Warehouse-Scale Computers (WSC) [3, 13, 26]. The continued growth in cloud-based, digital services has led WSCs to process increasingly large datasets with increasingly complex applications. WSC workloads are characterized by deep software stacks in which individual requests can traverse many layers of data retrieval, data processing, communication, logging, and monitoring. As a result, data and instruction footprints have been growing for decades.

The instruction footprint of WSC workloads in particular is often over 100 times larger than the size of a L1 instruction cache (i-cache) and can easily overwhelm it [2]. Studies show it expanding at rates of over 20% per year [16]. This results in instruction cache miss rates which are orders of magnitude higher than the worst cases in desktop-class applications, commonly represented by SPEC CPU benchmarks [9].

Because the performance of a general-purpose processor is critically dependent on its ability to feed itself with useful instructions, poor i-cache performance manifests itself in large unrealized performance gains due to front-end stalls. We corroborate this challenge for our WSCs on a web search binary. Figure 1 presents a Top-Down [33] breakdown of a web search loadtest running on an Intel Haswell CPU. 13.8% of total performance potential is wasted due to "Front-end latency," which is dominated by instruction cache misses. We also measured L1 i-cache miss rates of 11 misses per kilo-instruction (MPKI). Using the simulation methodology described in Section 5.5, we measured a hot steady-state instruction working

[*] Work performed while these authors were at Google.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6669-4/19/06.
https://doi.org/10.1145/3307650.3322234

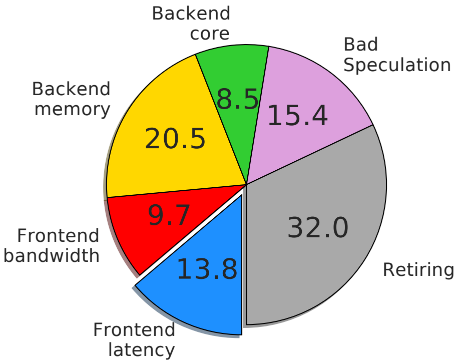

Figure 1: CPU performance potential breakdown (Top-Down) on a web search binary.

set of approximately 4 MiB. This is significantly larger than L1 instruction caches and L2 caches on today's server CPUs, but small and hot enough to easily fit and remain in the shared L3 cache (typically 10s of MiB). Thus, it is reasonable to assume that most i-cache misses are filled by the L3 cache in the worst case.

In this paper, we focus on understanding and improving the instruction cache behavior of WSC applications. Specifically, we focus on tools and techniques for *"broad" acceleration*[1] of thousands of WSC workloads. At the scale of a typical WSC server fleet, performance improvements of few percentage points (and even sub-1% improvements) lead to millions of dollars in cost and energy savings, *as long as they are widely applicable across workloads.* In order to enable the necessary horizontal analysis and optimization across the server fleet, we built a continuously-updated Assembly Database (AsmDB) with instruction- and basic-block-level information for most observed CPU cycles across the thousands of production binaries executing them (Section 2). We further correlate AsmDB with hardware performance counter profiles collected by a datacenter-wide profiling system – Google-Wide Profiling (GWP) [30] – in order to reason about specific patterns that affect front-end performance. Collecting and processing profiling data from hundreds of thousands of machines is a daunting task by itself. In this paper, we present the architecture of a system that can capture and process profiling data in a cost-efficient way, while generating terabytes of data each week.

A fleet-wide assembly database such as AsmDB provides a scalable way to search for performance anti-patterns and opens up new opportunities for performance and total-cost-of-ownership (TCO) optimizations. WSC servers typically execute thousands of different applications, so the kernels that matter most across the fleet (the "datacenter tax" [16]) may not be significant for a single workload, and are easy to overlook in application-by-application investigations. We leverage AsmDB's fleet-wide data in several case studies to understand and improve i-cache utilization and IPC of WSC applications.

We start with extensive analysis of AsmDB data, identifying the instructions that miss in the i-cache. We find that, while not particularly concentrated in specific code regions, most i-cache misses still share common characteristics (Section 3). Specifically, missing instructions are often the targets of control-flow-changing instructions with large jump distances. This points us towards exploring scalable automated solutions – with compiler and/or hardware support and no developer intervention – which can exploit these behaviors.

On the compiler side (Section 4), we outline several opportunities to target code bloat and fragmentation, which put unnecessary pressure on i-cache capacity. We find that intra-function fragmentation – where cold cache lines are brought in unnecessarily along with the hot portions of a function – is especially prevalent. Even after compiling with the necessary feedback-directed optimization (FDO) to eliminate guesswork, 50% of the majority of functions' code is cold, but frequently mixed with the hot parts. On a finer granularity, individual cachelines are also often fragmented and waste cache capacity, especially so in small functions. This suggests that classic compiler code layout optimizations such as inlining and hot/cold splitting can be more effectively and aggressively tuned, perhaps at link- or post-link time, when precise information about global control flow is available.

We prove this concept by manually applying similar optimizations to the most extreme case of code bloat uncovered by AsmDB – the library function memcmp. We demonstrate that, given the high front-end pressure of WSC applications, optimizing memcmp for better i-cache behavior can be a net performance win, even despite lower throughput numbers in isolated microbenchmarks.

Finally, we propose and evaluate a profile-driven optimization technique that intelligently injects software prefetch instructions *for code* (program instructions) into the binary during compilation (Section 5). We outline the design of the necessary "code prefetch" instruction, which is similar in nature to existing data prefetch instructions, except that it fetches into the L1-I cache and utilizes the I-TLB instead of the D-TLB. The implementation of such an instruction has negligible hardware cost and complexity compared to purely hardware methods and is commercially viable today. While it can be implemented on top of a wide variety of hardware front-ends, we demonstrate its viability on a system which employs only a next-line instruction prefetcher. Our prefetch insertion algorithm uses profile feedback information from AsmDB and performance counter profiles to ensure timely prefetches with minimal overhead. We prototype its effects on memory traces from several WSC workloads and show that it is possible to eliminate up to 96% of all L1-I cache misses while only adding 1.5% additional dynamic instructions for code prefetches.

2 ASMDB

We built the Assembly Database (AsmDB) with the goal to provide assembly-level information for nearly every instruction executed in our WSCs in an easy-to-query format. AsmDB aggregates instruction- and control-flow-data collected from hundreds of thousands of machines each day, and grows by multiple TiB each week. We have been continuously populating it over several years. In this section, we highlight the system design decisions which enable such scale and compare it with previous systems for datacenter-wide performance monitoring.

[1] *"Deep" acceleration* would involve focusing on a handful of workloads and trying to recover most of the ≈ 15% performance opportunity.

```
SELECT
  SUM(count) /
    (SELECT SUM(count) FROM ASMDB.last3days)
  AS execution_frac,
  asm.feature_name AS feature
FROM ASMDB.last3days
GROUP BY feature ORDER BY execution_frac DESC;
```

Figure 2: Example AsmDB SQL query which ranks x86 extensions (e.g. SSE, AVX) by execution frequency across our WSC fleet.

While the initial motivation for building AsmDB was manually answering simple horizontal questions about instruction mixes ("is x87 usage negligible?"), we have been increasingly using it for more sophisticated analyses, and especially so for finding instruction-cache-related optimization opportunities. This paper demonstrates several cases where AsmDB proves invaluable for that purpose: for spotting opportunities for manual optimizations, finding areas for improvement in existing compiler passes, as well as serving as a data source for new compiler-driven techniques to improve i-cache hit rates.

AsmDB is a specialization of generic datacenter-wide profiling systems like Google-wide-profiling (GWP) [30], which collect many different types of performance profiles. AsmDB is implemented in the GWP framework in order to share large portions of the underlying infrastructure. However, unlike more traditional performance profiles, AsmDB data requires extensive offline post-processing to reach a form that is easy to query by end users. It also has loftier coverage goals ("nearly every instruction executed in our WSCs"), which both enables new types of analyses, as well presents new scalability challenges, especially in dealing with the necessary storage.

Schema. AsmDB is a horizontal columnar database. Each row represents a unique instruction along with observed dynamic information from production – the containing basic block's execution counts, as well as any observed prior and next instructions. Each row also contains disassembled metadata for the instruction (assembly opcode, number/type of operands, etc.). This makes population studies trivial, as illustrated by the query in Figure 2 which ranks the relative usage of x87/SSE/AVX/etc. instruction set extensions. In addition, each row has metadata for the function and binary containing every instruction, which allows for queries that are more narrowly targeted than the full fleet (Section 4.4). Finally, each instruction is tagged with loop- and basic-block-level information from dynamically-reconstructed control-flow graphs (CFGs). This enables much more complex queries that use full control-flow information (Section 5).

Collection. AsmDB uses profiling data collected from a random sample of production machines using hardware last-branch-records (LBRs) – which capture bursts of execution traces of up to 32 basic blocks each. Most importantly, unlike traditional performance counters, LBR samples contain the destinations of all control-flow-changing instructions. This enables faithful and accurate reconstruction of dynamic program control flow. Collection is built on top of the data source for AutoFDO [5], which similarly uses LBRs to reconstruct basic block execution counts for compiler feedback-directed optimization. Overheads are similarly low: <1% when tracing, which only occurs for about 10 seconds per machine, per day.

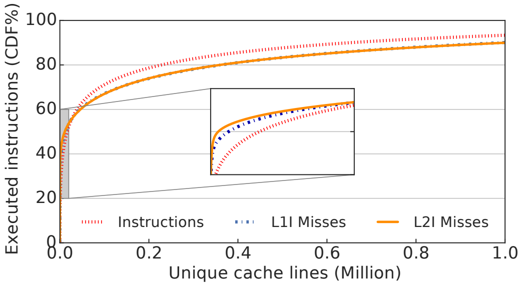

Figure 3: Fleet-wide distribution of executed instructions, L1-, and L2-instruction misses over unique cache lines. Like instructions, misses also follow a long tail.

Post-processing. Collected LBR samples only contain the addresses and counts of basic blocks. AsmDB requires an additional post-processing pass to produce instruction-level information suitable for querying. This offline pass fetches the binary bytes for each basic block, disassembles the contents, and fills out the metadata described earlier. We extend our existing symbolization service (which discovers, parses and indexes debugging symbols for every binary built in our WSCs) to also handle the actual binary bytes as well and to serve them on demand. Since debugging symbols are usually 10× larger than the binary they represent, this only increases complexity and does not hinder scalability.

In addition, post-processing discovers basic block predecessors and successors and identifies loops using Havlak's algorithm [12, 14] for full control-flow-graph (CFG) reconstruction.

With tens of thousands of binaries in a WSC, this step can be extremely costly in both computation and storage. We keep resource consumption reasonable by ignoring the very long tail and only populating AsmDB from the top 1000 binaries by execution cycles. This still captures 90% of observed cycles. Post-processing a single days' worth of collected data takes ≈ 8 hours with a 400-machine MapReduce [7], and produces ≈ 600 GiB compressed output. Thus, a years' worth of AsmDB data only takes up ≈ 200 TiB.

Design considerations. The main design goal for AsmDB is wide coverage – it contains assembly for 90+% of fleet-wide execution cycles. In addition, coverage is high within a binary itself – using LBR traces along with offline postprocessing allows us to capture relatively cold portions of the binary, which traditional sampling-based profiling (e.g. GWP) is likely to miss. This difference is particularly important for studies on cold code (Section 4).

In addition to post-processing feasibility, the main design constraint is restricting data cardinality to keep storage costs from exploding, while keeping enough metadata to enable useful queries. The recorded metadata in AsmDB is carefully limited to fields that compress well in columnar stores. For example, it explicitly excludes any runtime attributes (job names, datacenter names, etc.), for which the data typically has high variance and would compress poorly.

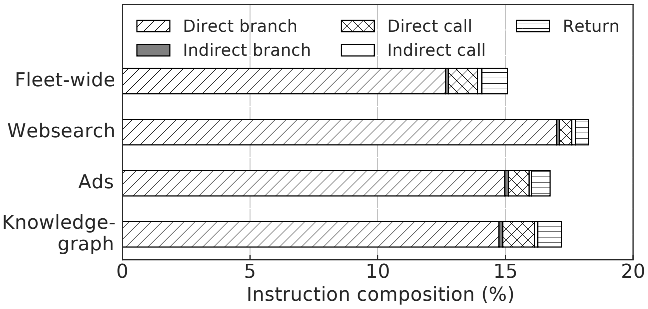

Figure 4: Control-flow instruction mixes for several WSC workloads. The remaining 80+% are sequential instructions.

3 WHERE ARE THE MISSES COMING FROM?

We begin our investigation into front-end stalls by characterizing and root-causing instruction-cache misses. We first use fleetwide miss profiles to confirm that, as many other WSC phenomena, i-cache misses also follow a long tail, and sooner or later that must be addressed by some form of automation. We then start looking for patterns that automated optimization can exploit by combining datasets from Google-Wide Profiling (GWP) and AsmDB. We focus on *miss-causing* instructions and find that indirect control-flow, as well as distant calls and branches are much more likely to be the root causes for misses.

3.1 Miss working sets

Working set sizes can tell us in broad strokes how to prioritize optimization efforts. For example, image processing workloads typically have tiny instruction working sets and manually hand-optimizing their code (similar to Section 4.4), is usually beneficial. On the contrary, WSC applications are well-known for their long tails and flat execution profiles [16], which are best addressed with scalable automatic optimization over many code locations.

Figure 3 shows that i-cache misses in WSCs have similarly long tails. It plots the cumulative distributions of dynamic instructions, L1-I, and L2-I misses over unique i-cache lines over a week of execution, fleetwide. Misses initially rise significantly steeper than instructions (inset), which suggests there are some pointwise manual optimizations with outsized performance gains. However, the distribution of misses tapers off, and addressing even two-thirds of dynamic misses requires transformations in $\approx 1M$ code locations, which is only conceivable with automation. In the rest of the paper, we show how a global database of assembly instructions can be useful in both the manual (Section 4.4) and automated cases (Section 5).

3.2 Miss-causing instructions

When optimizing instruction cache misses, it is not only important to identify the instructions that miss themselves, but also the execution paths that lead to them. These are the *miss-causing* instructions.

In the vast majority of cases, the predecessor of a particular instruction in execution is simply the previous sequential instruction. Figure 4 illustrates this for several large WSC binaries, along with a fleetwide average from AsmDB. More than 80% of all executed instructions are sequential (continuous or non-branching). The majority of the remainder (>10%) are direct branches, which are most typically associated with intra-function control-flow. Direct calls and returns, which jump into and out of functions, each represent only 1-2% of total execution. Indirect jumps and calls are even rarer.

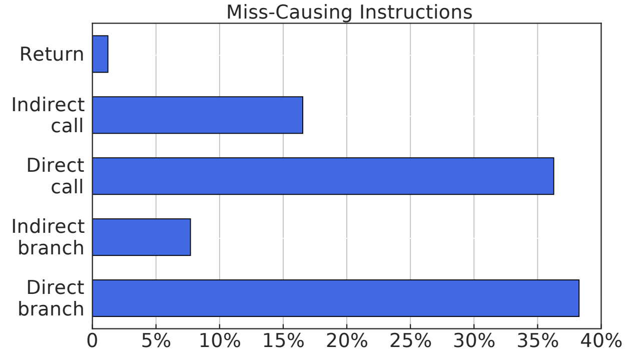

Figure 5: Instructions that lead to i-cache misses on a web search binary.

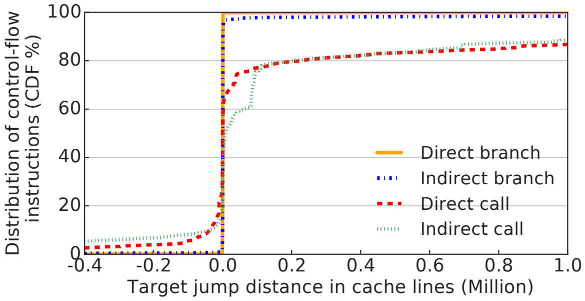

Figure 6: Fleetwide distribution of jump target distances.

Sequential instructions have high spatial locality and are thus inherently predictable. They can be trivially prefetched by a simple next-line prefetcher (NLP) in hardware. While there are no public details about instruction prefetching on current server processors, NLP is widely believed to be employed. And because NLP can virtually eliminate all cache misses for sequential instructions, it is the relatively small fraction of control-flow-changing instructions – branches, calls, and returns – which ultimately cause instruction cache misses and performance degradation.

Intuitively, branches typically jump within the same function so their targets are more likely to be resident in the cache due to reuse (temporal locality) for backward-pointing branches such as loops, and either reuse or NLP (with sufficiently small target offsets) for forward-pointing branches. On the other hand, we expect calls to miss more often because they jump across functions which can span a wide range of distances in the address space. This defeats NLP and is more difficult to capture by reuse due to limited cache sizes and flat callgraphs of WSC binaries.

In order to test this intuition, we send a full instruction trace from a web search binary through a simple L1 cache simulator with a next-two-line prefetcher, and mark each access that is a miss. For each miss, we look at the previous instruction in the program sequence which, assuming "perfect" (triggering on every access)

Discontinuity Type	Miss Percent	Miss Intensity
Direct Branch	38.26%	2.08
Indirect Branch	7.71%	59.30
Direct Call	36.27%	54.95
Indirect Call	16.54%	71.91
Return	1.22%	1.37

Table 1: Instruction miss intensities for web search

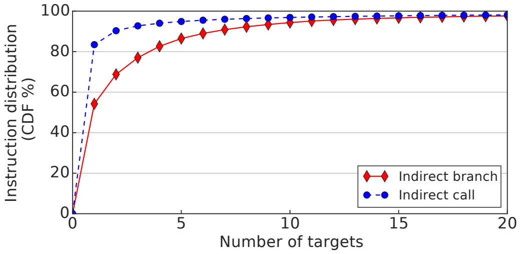

Figure 7: Cumulative distribution of number of targets for indirect jumps and calls.

NLP, is necessarily a control-flow-changing instruction. By counting these by type we built a profile of “miss-causing” instructions shown in Figure 5.

Interestingly, despite having higher temporal locality and being more amenable to NLP, direct branches are still responsible for 38% of all misses. These misses are comprised mostly of the small but long tail of direct branch targets that are greater than two cache lines away in the forward direction (18% of the profile). Perhaps more surprising is that direct calls account for 36% of all cache misses despite being only 1-2% of all executed instructions. This confirms that the probability of causing a miss is much higher for each call instruction, compared to that of a branch. Indirect calls, which are often used to implement function pointers and vtables, are even less frequently executed but also contribute significantly to misses. In contrast, returns rarely cause misses because they jump back to code that was often recently used. We summarize these probabilities in a dimensionless “miss intensity” metric, defined as the ratio of the number of caused misses to the execution counts for a particular instruction type. In other words, miss intensity helps us rank instruction classes by how likely they are to cause misses in the cache. We see from Table 1 that the miss intensities of direct branches and returns are much lower than the other types, which indicates their targets are more inherently cacheable.

We can confirm this is not limited to web search or due to the cache model used to tag misses by looking into fleetwide AsmDB jump distances. Figure 6 presents this data as a cumulative distribution function (CDF) of distances in bytes to the next instruction. Around 99% of all direct branches fleetwide jump to targets that are fewer than 500 cache lines away, and hence they are sharply centered around zero in the figure which is depicted at the scale of million cache lines. On the other hand, over 70% of direct calls have targets more than 100,000 cache lines (6.4 MiB) away. While such distances do not guarantee a cache miss, they do increase the likelihood that simple prefetchers without knowledge of branch behavior will be insufficient to keep the data cache-resident.

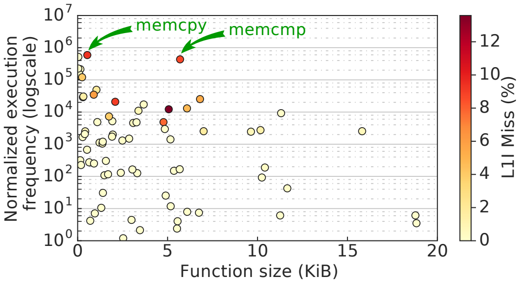

Figure 8: Normalized execution frequency vs. function size for the top 100 hottest fleetwide functions. memcmp is a clear outlier.

Indirect calls and branches roughly track the behavior of their direct counterparts. However, they are so infrequent that their targets are relatively cold (and unlikely to be resident in the cache), leading to the high miss intensity in Table 1. Note that, in practice, indirect calls and branches tend to have a very small number of targets per instruction (Figure 7 – 80+% of indirect calls and 58% of indirect branches always jump to a single address), which implies that they are very easily predictable with profile guided optimization.

In summary, the key takeaways from our instruction miss analysis are that 1) calls are the most significant cause of cache misses, and 2) branches with large offsets contribute significantly to misses because they are not fully covered by reuse and simple next-line prefetching.

4 CODE BLOAT AND FRAGMENTATION

Before we focus our attention to optimizing the targets of very distant branches in Section 5, we outline some opportunities for improving i-cache behavior on a much finer granularity. Namely, we identify reducing cache fragmentation on the intra-function and intra-cacheline level with feedback-driven code layout. A global database of assembly instructions and their frequencies such as AsmDB critically enables both prototyping and eventually productionizing such optimizations.

Briefly, fragmentation results in wasted limited cache resources when cold code is brought into the cache in addition to the necessary hot instructions. Such negative effects get increasingly prominent when functions frequently mix hot and cold regions of code. In this section we show that even among the hottest and most well-optimized functions in our server fleet, more than 50% of code is completely cold. We attribute this to the deep inlining that the compiler needs to perform when optimizing typical WSC flat execution profiles. This suggests that combining inlining with more aggressive hot/cold code splitting can achieve better i-cache utilization and free up scarce capacity.

4.1 Code bloat

One common symptom for excessive i-cache pressure is code bloat, or unnecessary complexity, especially in frequently-executed code. Figure 8 is an attempt to diagnose bloat from AsmDB data – it plots normalized function hotness (how often a particular function is called over a fixed period) versus the function's size in bytes for the 100 hottest functions in our WSCs. Perhaps unsurprisingly, it

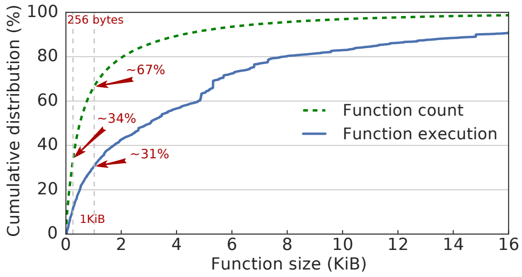

Figure 9: Distribution of execution over function size.

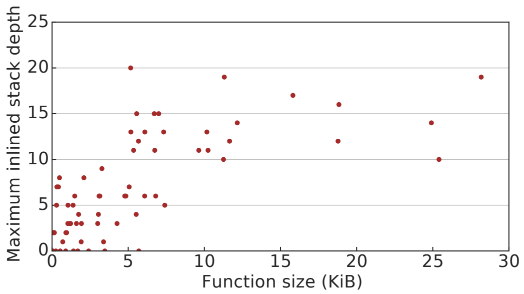

Figure 10: Maximum inlining depth versus function size for the 100 hottest fleetwide functions.

shows a loose negative correlation: Smaller functions are called more frequently. It also corroborates prior findings that low-level library functions ("datacenter tax" [16]), and specifically `memcpy` and `memcmp` (which copy and compare two byte arrays, respectively) are among the hottest in the workloads we examined.

However, despite smaller functions being significantly more frequent, they are not the major source of i-cache misses. Overlaying miss profiles from GWP onto Figure 8 (shading), we notice that most observed cache misses lie in functions larger than 1 KiB in code size, with over half in functions larger than 5 KiB.

This contradicts traditional optimization rules of thumb, like "Most [relevant] functions are small". But it also holds for execution cycles – as illustrated in Figure 9 – only 31% of execution fleetwide is in functions smaller than 1 KiB. Small functions are still prevalent: 67% of all observed functions by count are smaller than 1 KiB. However, a large portion of them are very cold. This suggests that, as expected for performance, small hot functions get frequently inlined with the help of profile feedback[2].

The ubiquity and overall aggressiveness of inlining is best illustrated in Figure 10, which plots the depth of observed inline stacks over the 100 hottest functions. Most functions 5 KiB or larger have inlined children more than 10 layers deep. While deep inlining is crucial for performance in workloads with flat callgraphs, it brings in exponentially more code at each inline level, not all of which is necessarily hot. This can cause fragmentation and suboptimal utilization of the available i-cache.

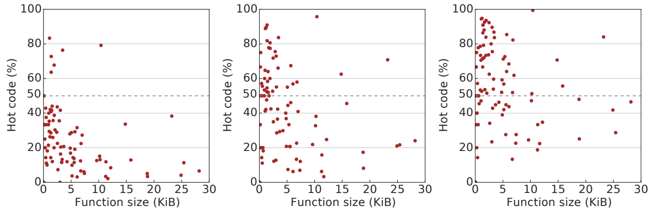

Figure 11: Fraction of hot code within a function among the 100 hottest fleetwide functions. From left to right, "hot code" defined as covering 90%, 99% and 99.9% of execution.

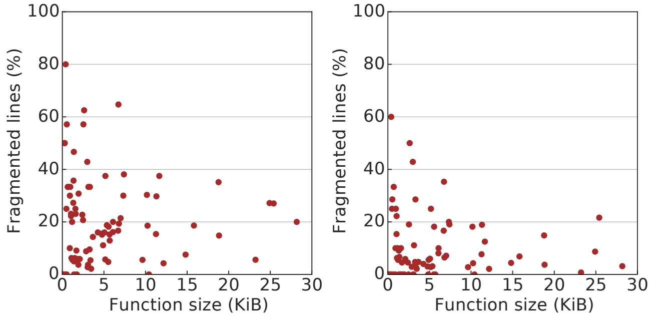

Figure 12: Intra-cacheline fragmentation vs function size for hotness thresholds of 90%, and 99%.

4.2 Intra-function fragmentation

In order to understand the magnitude of the potential problem, we quantify code fragmentation on the function level. We more formally define fragmentation to be the fraction of code that is definitely cold, that is the amount of code (in bytes) necessary to cover the last 10%, 1%, or 0.1% of execution of a function. Because functions are sequentially laid out in memory, these cold bytes are very likely to be brought into the cache by next-line prefetching. Intuitively, this definition measures the fraction of i-cache capacity potentially wasted by bringing them in.

Using `AsmDB` data, we calculate this measure for the top 100 functions by execution counts in our sever fleet. Figure 11 plots it against the containing function size. If we consider code covering the last 1% of execution "cold", 66 functions out of the 100 are comprised of more than 50% cold code. Even with a stricter definition of cold (< 0.1%), 46 functions have more than 50% cold code. Perhaps not surprisingly, there is a loose correlation with function size – larger (more complex) functions tend to have a larger fraction of cold code. Generally, in roughly half of even the hottest functions, more than half of the code bytes are practically never executed, but likely to be in the cache.

4.3 Intra-cacheline fragmentation

Fragmentation in the i-cache also manifests itself at an even finer granularity – for the bytes within each individual cacheline. Unlike cold cache lines within a function, cold bytes in a cache line are always brought in along the hot ones, and present a more severe performance problem. We defined a similar metric to quantify intra-cacheline fragmentation: counting the number of bytes (out of 64) necessary to cover 90% or 99% of the line's accesses. Similarly to

[2] At the time of collection, over 50% of fleet samples were built with some flavor of profile-guided optimization.

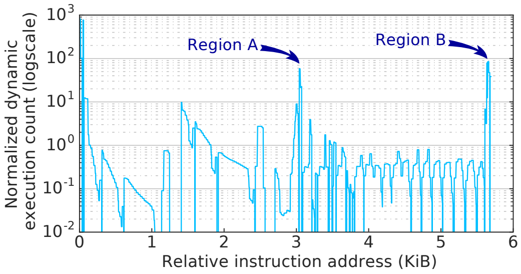

Figure 13: Instruction execution profile for memcmp. 90% of dynamic instructions are contained in 2 cache lines; covering 99% of instructions requires 41 i-cache lines.

the last section, we declare a line fragmented if it only uses 50% or fewer of its bytes to cover execution. Figure 12 shows the fraction of fragmented lines for each of the top 100 functions in our server fleet. At least 10% of functions have more than 20% of cache lines that are fragmented, and fragmentation is more common for small functions. In other words, while these functions are executing, at least 10% of i-cache capacity is stranded by fragmented lines. This suggests opportunities in basic-block layout, perhaps at link, or post-link time, when compiler profile information is precise enough to reason about specific cache lines.

4.4 Memcmp and the perils of micro-optimization

To illustrate the potential gains from more aggressive layout optimization, we focus on the most extreme case of bloat we observed in `AsmDB` – the library function `memcmp`.

`memcmp` clearly stands out of the correlation between call frequency and function size in Figure 8. It is both extremely frequent, and, at almost 6 KiB of code, 10× larger than `memcpy` which is conceptually of similar complexity. Examining its layout and execution patterns (Figure 13) suggests that it does suffer from the high amount of fragmentation we observed fleetwide in the previous section. While covering 90% of executed instructions in `memcmp` only requires two cache lines, getting up to 99% coverage requires 41 lines, or 2.6 KiB of cache capacity. Not only is more than 50% of code cold, but it is also interspersed between the relatively hot regions, and likely unnecessarily brought in by prefetchers. Such bloat is costly – performance counter data collected by GWP indicates that 8.2% of all i-cache misses among the 100 hottest functions are from `memcmp` alone.

A closer look at the actual code from glibc can explain the execution patterns in Figure 13. It is hand-written in assembly and precompiled, with extensive manual loop unrolling, many conditional cases for the various alignments of the two source arrays, and large jump tables.

In our experience, code usually evolves into similar state from over-reliance on micro-optimization and micro-benchmarking. While writing in assembly can in rare cases be a necessary evil, it prevents the compiler from doing even the most basic feedback-directed code layout optimizations. For example, it cannot duplicate or move the "compare remainders" and "exit" basic blocks marked `RegionA` and `RegionB` in Figure 13 closer to the cases that happen to call them the most (in this case the beginning of the function). This results in expensive and hard-to-prefetch jumps, and cache pollution. Similarly, when mostly evaluated on microbenchmarks, all relevant code usually fits in the i-cache, which is certainly not the case for large applications. This encourages developers to add more elaborate corner cases (e.g. for alignment) that improve the microbenchmark without regard to bloat.

We tested this hypothesis by macro-benchmarking a version of `memcmp` that is specifically optimized for code size (only 2 i-cache lines) and locality. In short, it only special-cases very small string sizes (to aid the compiler in inlining very fast cases) and falls back to `rep cmps` for larger compares. Even though it achieves slightly lower throughput numbers than the glibc version in microbenchmarks, this simple proof-of-concept is a net performance gain. On large-footprint workloads like websearch, it reduced cycles in `memcmp` more than twofold and showed an overall 0.5%-1% end-to-end performance improvement.

`AsmDB` allows us to spot extreme cases such as `memcmp` over thousands of applications. In this case, `memcmp` was the single immediately apparent outlier both in terms of code bloat and i-cache footprint. Manually optimizing it for code size was practical and immediately beneficial.

However, manual layout optimization does not scale past extreme outliers. Generalizing similar gains is in the domain of compilers. Compiler optimizations like hot/cold splitting [6] and partial inlining [32] aim to address fragmentation by only inlining the hot basic blocks of a function. However, they have recently been shown to be particularly susceptible to the accuracy of feedback profiles [25], especially with sampling approaches like AutoFDO [5].

The high degree of fragmentation we observed suggests there is significant opportunity to improve i-cache utilization by more aggressive and more precise versions of these optimizations than found in today's compilers. Alternatively, post-link optimization could be a viable option which does not suffer from profile accuracy loss. The latter approach has been shown to speed up some large datacenter applications by 2-8% [25].

5 SOFTWARE PREFETCHING FOR CODE

While addressing fragmentation and code bloat can recover a fraction of the i-cache capacity being wasted, the multi-megabyte instruction working sets in WSC applications suggest that even removing all fragmentation will not suffice to eliminate frontend bottlenecks. In Section 3 we showed that distant branches and calls cause the largest fraction of cache misses. Prior work has addressed these misses with significant hardware architectural modifications [10, 11, 18–20, 20, 22] or static control flow analysis [24, 28]. However, in WSC environments it has been commercially infeasible to implement these large hardware changes, and it is likewise intractable to optimize the nearly boundless number of possible control flow combinations in binaries hundreds of megabytes in size.

We improve upon prior work by leveraging the control flow information from `AsmDB`, and propose a novel compiler optimization that automatically inserts code prefetch instructions into performance-critical execution paths within the application binary. Specifically,

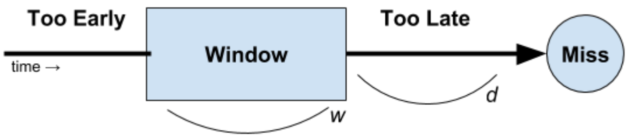

Figure 14: A prefetch is never late, nor is it early, it arrives precisely when it means to.

our approach reconstructs the control flow graph of the applications from AsmDB and enriches it with instruction miss profiles to select prefetch targets. This combined information allows us to inject low-overhead, high-accuracy software prefetching for the front-end using a new code prefetch instruction. As a result, we are able to achieve as much as 96% miss coverage with negligible application binary growth.

5.1 Requirements

Prefetching represents a prediction problem with a limited window of opportunity. Effective prefetches are both accurate and timely – they only bring in useful miss targets and do so neither too early nor too late in order to minimize early evictions and cache pollution. A prefetcher is effective if it generates effective prefetches and has high overall miss coverage. We begin this section with a discussion of the required information and hardware support which is necessary to ensure accuracy, timeliness, and coverage for WSC applications.

Finding prefetch targets. In the simplest sense, the set of targets to prefetch is merely the set of instruction addresses that miss in the cache. This set is often estimated heuristically. For instance, next-line prefetchers always predict the next cache block will be used, and static code analysis can assume some control flow and determine constructs such as loops which might be amenable to prefetching [24]. In contrast, our approach leverages AsmDB to augment heuristic information with *empirical* observations about top miss candidates and dominant control flow. This allows us to achieve high coverage while minimizing the overheads of prefetching in terms of bandwidth, energy, and performance in case the prefetch is not useful or accurate. Alternatively, performance counters or binary instrumentation tools can also provide information about top misses [4, 23].

Determining prefetch injection sites. The placement of prefetches in the execution stream determines their timeliness. An effective prefetch will initiate within a window of time prior to the miss that is neither too early (and thus evicted before use), nor too late (and doesn't arrive before the miss), as represented by w in Figure 14. Existing compiler based approaches [24] insert prefetches at a fixed number of instructions before the miss to roughly match the memory access latency. However, on modern OOO architectures, IPC can vary by orders of magnitude for different applications which leads to untimely prefetch injections. (For instance, in SPEC CPU2006, mcf has an IPC of 0.2 whereas dealII has an IPC of almost 2.0 [27].) Our approach addresses this issue by leveraging per-application IPC obtained from profiling data to calculate optimal prefetch distances.

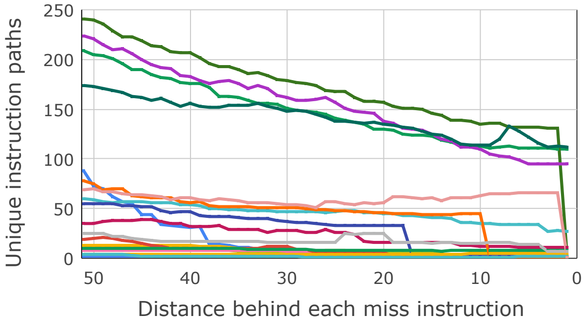

Figure 15: Fan-in for some misses can grow very fast with distance, especially for library functions.

In addition to being on time, a prefetch instruction needs to be on an execution path that is likely to actually lead to the target miss. This requires knowledge of the program's control flow, along with probabilities for specific paths. Some form of profile feedback is required to obtain this information – whether by hardware-enabled tracing (e.g., last-branch records or Intel Processor Trace), or software binary instrumentation (statically-inserted by a compiler or dynamically-inserted through systems like Pin or DynamoRIO). Static program analysis alone is insufficient since it must guess branch outcomes, realized indirect branch targets (this is especially problematic given that they're a frequent cause of misses, Section 3), and ultimately cannot determine the important control flow paths of the application. Our approach leverages AsmDB and additional profiling data to construct the control flow graph. Importantly, we do not rely on obtaining complete graphs; in fact we further prune the graph data to contain only paths of high execution frequency.

5.2 Software prefetch challenges

Knowledge of cache misses, execution history, and system details are necessary but not sufficient for effective software instruction prefetching. A number of second-order challenges arise that can make practical implementations difficult.

Fan-in. When moving backward from a miss target in the execution sequence, the number of potential instructions leading to that miss generally increases. In fact, it grows exponentially with the branching factor of the control-flow graph. High fan-in to a miss poses a challenge because it requires additional prefetch instructions to be injected (adding overhead), and each injection site has a smaller overall impact in reducing misses.

Figure 15 shows the fan-in for the top twenty instruction cache misses from a web search profile. In several cases, the number of paths leading in to a single miss exceeds 100 even when looking backward only 10 instructions. These are usually common library calls (for example, the top line is memcmp). For other hot misses, the fan-in is much less significant, even when looking backward over 50 instructions. Both situations can be addressed by prefetching, but low-fan-in targets will incur significantly smaller overheads. Prior approaches [24] had to aggressively insert prefetches in all paths to obtain high coverage, and then resort to filtering out superfluous prefetches in hardware, thus wasting hardware resources and instruction cache capacity. Our approach leverages profiling information to only insert helpful prefetches which increase coverage and minimize overhead and fan-in.

Fan-out. When selecting an injection site for a miss, landing in the window (see Figure 14) is necessary but not sufficient. The window could contain code that is running in a loop, or which is a hot library function that returns to a large number of callers. In these situations, adding prefetch injections will cause a large number of untimely or useless requests that waste resources. Instead, we want to insert prefetches in execution paths that are likely to lead to the miss, or in other words, which have low fan-out. We address this in Section 5.3 by pruning paths which exceed a maximum fan-out threshold.

Instruction overhead. Software prefetch instructions carry overheads, even if optimized. At the very least, they need to be stored in instruction caches, as well as decoded and issued by the pipeline. While these overheads should generally be minimal, overly-aggressive prefetching can end up causing performance degradation instead of improvement. Our injection algorithm selects prefetch sites for those that resolve the highest number of misses and reduce overall instruction overhead.

5.3 Ensuring timeliness

Prefetch timeliness critically requires that we initiate prefetch requests within a “sweet spot” window of time that is neither too early nor too late. Specifically, we need to determine the distance d and window size w (see Figure 14, measured in instructions). As outlined in Section 5.1, d and w are ultimately defined by the microarchitecture and memory system.

More formally, we define the distance d as the shortest amount of time before a miss that a prefetch for that cache line can be made without causing stalls. It is the sum of the latency of the prefetch instruction itself and the time required by the memory system to bring the block into the cache. We assume the prefetch instruction latency is minimal (due to lack of register dependencies and pipeline pruning) and focus on the delay of the memory system. Since the instruction working set easily fits in the L3 cache of a server processor (Section 1), d in instructions is simply the latency of the L3 cache in cycles, multiplied by the application specific average IPC. For this work, we use long-term IPC averages for each application. As IPC can vary further with program phase behavior, future work can leverage fine-grain IPC measurements from a local phase to vary the prefetch distance depending on the code location.

The window w allows for some leeway in choosing prefetch sites further than the minimum distance d. Normally we want to minimize to d in order to reduce the risk of high fan-in (see Figure 15). On the other hand, larger windows allow us to minimize the risk of fan-out. By considering all of the candidate injection sites within the window we can optimize for both. Consider the three scenarios in Figure 16, assuming $d = 1$:

Instruction D (at distance 1 from the miss) has minimal fan-out because it always leads to the miss. In addition, choosing D instead of its predecessors helps to minimize fan-in to the miss. Thus D is a good prefetch insertion candidate.

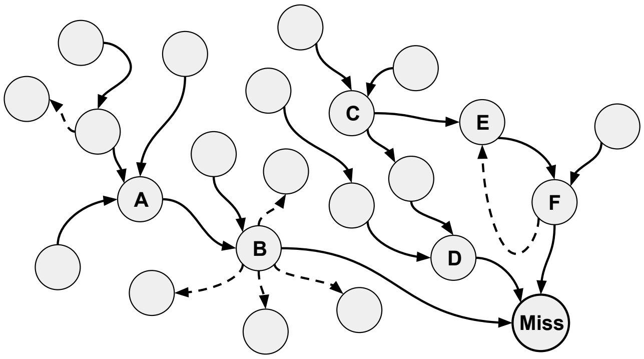

Figure 16: Example instruction execution history tree.

Instruction B is also at distance 1 from the miss. Unfortunately, B jumps to many other locations and only leads to the miss 5% of the time (due to high fan-out). However, if all of these paths from B to the miss are from a single predecessor instruction (for example, A) which is still within the window, then that is clearly a better prefetch location.

Instruction F is part of a loop. Inserting prefetches anywhere in the loop body (instructions F and E) would cause redundant prefetch requests and excessive instruction overhead. Thus even though F only leads to two instructions, it has high execution fan-out. If a window size of two or greater allows us to insert a prefetch at the earlier instruction C (distance 3), then we can maximize the covered execution paths and avoid the execution fan-out of F and E.

The maximum window size is the limit after which prefetch injections would lead to early misses, or would otherwise evict useful data to the detriment of performance. For a fully-associative LRU cache, the upper-bound maximum window size would be the cache capacity divided by the average instruction size, or 8,192 for a 32-KiB cache and 4-byte average instruction size. Empirically, we’ve found that window sizes larger than about 200 instructions cause enough evictions of useful data to reduce overall performance.

5.4 Prefetch injection procedure

Having a profile of miss instructions, the program execution sequence, and the window parameters d and w, we are equipped to compute and inject software prefetches into the execution stream. At a high level, the process is threefold: First, construct the execution history for each miss going back to the end of the window ($d + w$). Second, find prior instructions within the window with low enough fan-out. These instructions may be located on one or more execution paths. Third, insert prefetches for each miss target at the computed injection locations in the program.

This approach provides four key contributions over prior static approaches: First, we determine prefetch targets based on miss frequency instead of guessing viable candidates. Second, we leverage application-specific IPC to determine the optimal prefetch distance, instead of relying on a fixed heuristic. Third, we introduce a maximum window size and corresponding scan that allows us to discover non-intuitive prefetch optimizations (such as prefetching ahead of loops) without requiring detailed knowledge of program flow semantics. Fourth, we use profiling information to prune low probability prefetches in cases of high fan-in or fan-out, thus reducing the instruction execution and footprint overhead over more conservative static approaches. We next discuss the three steps in more detail.

Construct per-miss execution histories. The purpose of this step is to identify all possible prefetch injection sites for each miss. For each miss location (identified as described in Section 5.1), we start a bottom-up walk of the control-flow graph starting from the miss, adding basic-block counts at every level. We terminate the walk at every node where it reaches $d + w$, leaving us with an execution history graph for the miss, similar to Figure 16. If instead of the full control-flow graph, we have an instruction trace, we can similarly reconstruct the history graphs with a single pass over the trace.

Compute prefetch injection locations. The injection calculation step selects the best locations (if any) of the execution graphs in which to prefetch each miss. It begins by selecting all prior instructions at distance d, which is the minimum time required for a prefetch to be timely. Each of these priors is recursively compared against its own predecessors, up to the maximum window size ($d + w$). Selecting among them choses a node that leads to the target miss most often, with some minimum threshold percentage to limit fan-out. At the end of the injection computation, a (possibly empty) set of injection sites is available for each miss.

Inject prefetches. Finally, after identifying profitable prefetch sites, we can insert each computed (injection, target) address pair into the program. This process will vary for each production environment. Without loss of generality, we assume that this is done as post-link step of the compilation process, and only requires reassembly and relinking, not a full recompilation. Thus, we unburden the programmer from having to deal with the time and complexity of injecting prefetches manually in the source code.

Implementation. Our prefetch mechanism is software-based but relies on a "code prefetch" instruction to load cache blocks directly into the L1-I cache or adjacent prefetch buffer. In practice, these instructions often do not exist in commercial server-class processors (x86), or have inconsistent or implementation-defined semantics (`pli` for ARM), or don't provide the required level of control (`icbt` on POWER). From an architectural standpoint, a code prefetch instruction is simple to implement, and would require the following properties:

(1) Requests are loaded into the L1-I or prefetch cache (minimum) or additionally the L2 and L3 caches (optional).
(2) Requests utilize the instruction TLB, if any, and not the data TLB.
(3) Blocks are brought into the cache in the S state, not the E state (for MESI-like cache coherent systems).
(4) The instruction has no register dependencies. Target addresses are encoded in the instruction. If possible, prefetches are pruned from the back-end portions of the processor pipeline to minimize overheads and latency.

We imagine that a code prefetch instruction would be an extension of – or very similar to – existing data prefetch instructions such as `prefetcht*` and `prefetchnta` on x86, `pli` on ARM, and `icbt` on POWER.

Parameter	Value
CPU	Intel Haswell E5 18-core
L1 Instruction cache	32 KiB, 8-way
L2 Unified cache	256 KiB, 8-way
L3 Unified cache	Shared 2.5 MiB/core 22-way
All-core turbo frequency	2.8 GHz
L3 cache latency	27 ns / 76 cycles (measured)

Table 2: System Configuration

5.5 Evaluation methodology

While we envision an end-to-end system that uses fleetwide profile information and a compiler, we prototyped our proposal using memory traces and simulation. We require simulation because current server-class processors do not include a suitable prefetch instruction for code. In addition, replaying trace-driven simulation allows us to perform limit studies and compare our prefetch insertion approach against an ideal instruction cache.

Data collection. We use DynamoRIO's [4] memtrace client to capture instruction and data memory traces for our target applications. We limit traces to 2 billion instructions during steady-state execution which is more than sufficient for instruction cache studies.

We use the traces both to construct dynamic control-flow graphs of observed execution, as well as to identify instruction cache misses (after simulation). This is for prototyping convenience and reproducibility only. A real system can use dynamic CFGs collected from last-branch records (as reconstructed in `AsmDB`) and instruction miss profiles collected with Precise Event-Based Sampling (PEBS) [8], which allow us to identify individual instructions that miss in the L1 or L2 instruction caches.

Simulation. We use a modified version of the ZSim simulator [31]. We included a trace-driven execution mode, as well as models for our best guess of an Intel-Haswell-based server processor, with parameters described below. We model a single core and allow 10 MiB of L3 capacity in order to ensure that the instruction working set fits in the L3 cache (see Section 1). We also extended ZSim to include an access-driven next-2-line prefetcher for the instruction cache.

System parameters. We use parameters modeled against an Intel Haswell datacenter-scale server processor. It has 18 cores, each with a private 32-KiB L1 instruction cache and unified 256 KiB L2 cache. All 18 cores share a unified 45-MiB L3 cache. The detailed system parameters are summarized in Table 2. Based on these parameters and a per-application average IPC, we can estimate the minimum prefetch distance d. In the case of web search the average IPC is 0.67, leading to d = 76 * 0.67 = 51 instructions on average.

Workloads. We focus primarily on three WSC applications – a web search leaf node, an ads matching service, and a knowledge graph back-end. For each workload, we collect traces during a representative single-machine loadtest, which sends realistic loads to

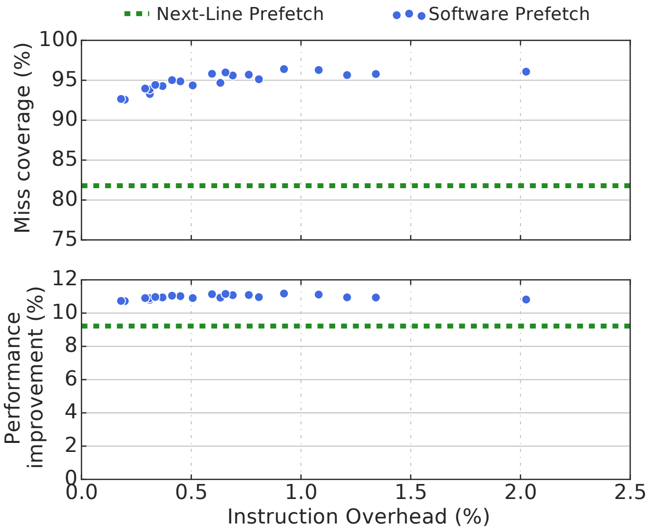

Figure 17: Overall performance improvement and miss coverage vs. instruction cost for several prefetch configurations on web search.

the server-under-test. We also include three SPEC CPU2006 applications (`400.perlbench`, `445.gobmk`, `471.omnetpp`) to demonstrate that the prefetching technique generalizes across workloads. Since most of SPEC CPU suite has a tiny instruction working set, we simulated `400.perlbench` and `471.omnetpp` with only 8KiB of private L1 i-caches. This results in MPKI rates of the same order of magnitude as our WSC workloads.

5.6 Prefetching results

We fixed the distance at 51 instructions and varied the injection threshold, window size, and miss coverage. Figure 17 shows detailed results for web search. It combines the multidimensional configurations into a single flattened view in terms of instruction overhead and the percentage of misses that were eliminated. Here, "instruction overhead" refers to the additional dynamic prefetches executed by the processor. It is not the time overhead or increase in the static program size (in all cases the static program size growth is less than 1%). All performance improvements measure IPC over a system without any prefetching.

Our goal is to prefetch the misses we observe in the fleet which are caused nearly universally by control-flow changing instructions. These misses lie in the gap between a perfect instruction cache and what an effective sequential prefetcher (NLP) can provide, as shown in the miss coverage subplot of Figure 17. In other words, while NLP covers over 80% of instruction cache misses for web search, the performance opportunity measured in Figure 1 comes from the gap.

Our prefetching technique is able to increase the overall miss coverage of web search up to 96% with no other front-end assumptions than a next-line prefetcher. Similarly, Figure 18 shows that we can achieve miss coverage between 91%-96% for all other workloads, with a performance improvement proportional to the front-end-boundedness of the application and the gap left from NLP. In all cases the dynamic instruction execution overhead due to prefetches is less than 2.5%, and the static program growth due to prefetches is between 0.01% and 1%.

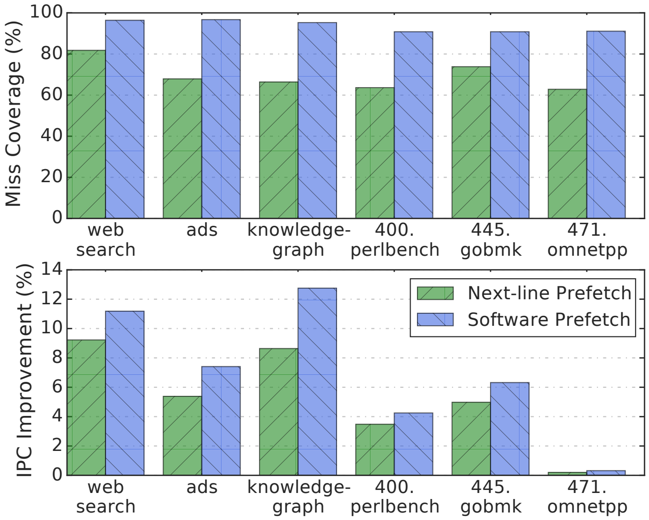

Figure 18: Miss coverage and performance improvement for the best-performing configuration for each workload.

6 RELATED WORK

Datacenter-wide profiling. Modern systems for always-on profiling trace their beginnings to DCPI [1]. Of these, AutoFDO [5] (built on top of Google-wide-profiling (GWP) [30]) is perhaps the most similar to `AsmDB`. Both AutoFDO and this work use continuously-collected LBR samples for compiler optimization. AutoFDO summarizes them into basic block counts and maps them back to source code for traditional feedback-directed optimization during compilation. In contrast, this work fully materializes a program's control-flow graph and uses it both for offline analysis and for post-link time prefetch insertion on the binary level.

Profiling efforts, both on production WSCs [16] and on isolated benchmarks [9, 15, 17], have previously identified i-cache misses as a significant performance bottleneck. This work stems from the same observations and dives into root-causes and solutions. Recently, BOLT [25] was inspired by i-cache fragmentation findings on benchmarks, confirmed by the fleetwide `AsmDB` results presented here.

Front-end prefetching. Researchers have proposed a number of prefetching techniques for reducing front-end stalls. Temporal streaming prefetchers capture and replay instruction sequences with high accuracy. However, they typically have enormous on-chip storage costs in the range of hundreds of kilobytes per core [10, 11, 20] or per chip [18]. Recent streaming prefetchers have reduced the required amount of on-chip storage [18–20]. However, they still require several megabytes of total chip storage, making them difficult to implement in commercial processors.

More recently, Boomerang [22] combines fetch-directed instruction prefetching [29] with BTB prefetching in a unified front-end solution that addresses branch misspeculation in addition to instruction cache misses. However, it is still limited by BTB capacity. Shotgun [21] addresses this limitation by optimizing BTB storage for macro-level control flow (i.e., unconditional branches) and leveraging spatial locality to capture micro-level conditional branch discontinuities. All these techniques rely on major hardware changes – adding a prefetch engine, fetch target queue, prefetch buffer, an instruction pre-decoder, and an entirely new BTB design (in the case of Shotgun). These have not made it into datacenter-scale processors so far, and each assumes significant complexity, cost, and risk to implement.

Luk and Mowry [24] have the most similar approach to reducing i-cache misses compared to this work. They also insert code prefetches with compiler help, after static control-flow analysis. Static analysis relies on heuristics for branch outcomes, indirect targets, and control flow which limits accuracy and significantly increases size overheads. We leverage empirical dynamic program behavior to target only paths that are performance-critical. More crucially, they rely on additional hardware to filter spurious prefetches – both an active prefetch filter and additional metadata bits in all i-caches. Our approach avoids spurious prefetches without the complexities of additional hardware with a combination of dynamic control-flow analysis and instruction miss profiles.

7 CONCLUSION

This paper focused on understanding and improving instruction cache behavior, which is a critical performance constraint for WSC applications. We developed `AsmDB`, a database for instruction and basic-block information across thousands of WSC production binaries, to characterize instruction cache miss working sets and miss-causing instructions. We used these insights to motivate fine-grain layout optimizations to split hot and cold code and better utilize limited instruction cache capacity. We also proposed a new, feedback-driven optimization that inserts software instructions for code prefetching based on the control-flow information and miss profiles in `AsmDB`. This prefetching optimization can cover up to 96% of instruction cache misses without significant changes to the processor and while requiring only very simple front-end fetch mechanisms.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and David Xinliang Li for their constructive feedback on drafts of this manuscript. We reserve our special thanks for our colleagues at Google: the platforms performance team for invaluable help with performance measurement, the memtrace team for the infrastructure capable of tracing large applications, and the GWP team for building and maintaining WSC-scale profiling infrastructure.

REFERENCES

[1] Jennifer Anderson, Lance Berc, George Chrysos, Jeffrey Dean, Sanjay Ghemawat, Jamey Hicks, Shun-Tak Leung, Mitch Lichtenberg, Mark Vandevoorde, Carl A Waldspurger, et al. 1998. Transparent, low-overhead profiling on modern processors. In *Workshop on Profile and Feedback-Directed Compilation*.

[2] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Memory hierarchy for web search. In *High Performance Computer Architecture (HPCA)*.

[3] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture* (2018).

[4] Derek Bruening. 2004. *Efficient, transparent, and comprehensive runtime code manipulation.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[5] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *Code Generation and Optimization (CGO)*.

[6] Robert Cohn and P Geoffrey Lowney. 1996. Hot cold optimization of large Windows/NT applications. In *Microarchitecture (MICRO)*.

[7] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: simplified data processing on large clusters. In *Operating Systems Design and Implementation (OSDI)*.

[8] Stéphane Eranian. 2008. What can performance counters do for memory subsystem analysis?. In *Workshop on memory systems performance and correctness*.

[9] Michael Ferdman, Babak Falsafi, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, and Anastasia Ailamaki. 2012. Clearing the clouds. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[10] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. 2011. Proactive instruction fetch. In *Microarchitecture (MICRO)*.

[11] Michael Ferdman, Thomas F Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal instruction fetch streaming. In *Microarchitecture (MICRO)*.

[12] Paul Havlak. 1997. Nesting of reducible and irreducible loops. *Transactions on Programming Languages and Systems (TOPLAS)* (1997).

[13] John L Hennessy and David A Patterson. 2012. *Computer architecture: a quantitative approach*.

[14] Robert Hundt. 2011. Loop recognition in C++/Java/Go/Scala. *Scala Days* (2011).

[15] Zhen Jia, Lei Wang, Jianfeng Zhan, Lixin Zhang, and Chunjie Luo. 2013. Characterizing data analysis workloads in data centers. In *Workload Characterization (IISWC)*.

[16] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Computer Architecture (ISCA)*.

[17] Harshad Kasture and Daniel Sanchez. 2016. TailBench: A benchmark suite and evaluation methodology for latency-critical applications. In *Workload Characterization (IISWC)*.

[18] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2013. Shift: Shared history instruction fetch for lean-core server processors. In *Microarchitecture (MICRO)*.

[19] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2015. Confluence: Unified instruction supply for scale-out servers. In *Microarchitecture (MICRO)*.

[20] Aasheesh Kolli, Ali Saidi, and Thomas F Wenisch. 2013. RDIP: Return-address-stack directed instruction prefetching. In *Microarchitecture (MICRO)*.

[21] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. 2018. Blasting through the front-end bottleneck with shotgun. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[22] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. 2017. Boomerang: A metadata-free architecture for control flow delivery. In *High Performance Computer Architecture (HPCA)*.

[23] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN*.

[24] Chi-Keung Luk and Todd C Mowry. 1998. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Microarchitecture (MICRO)*.

[25] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Code Generation and Optimization (CGO)*.

[26] David A Patterson. 2008. The data center is the computer. *Commun. ACM* (2008).

[27] Tribuvan Kumar Prakash and Lu Peng. 2008. Performance characterization of spec cpu2006 benchmarks on intel core 2 duo processor. *ISAST Transactions on Computer Software Engineering* (2008).

[28] Muhammad Yasir Qadri, Nadia N Qadri, Martin Fleury, and Klaus D McDonald-Maier. 2015. Software-Controlled Instruction Prefetch Buffering for Low-End Processors. *Journal of Circuits, Systems and Computers* (2015).

[29] Glenn Reinman, Brad Calder, and Todd Austin. 1999. Fetch directed instruction prefetching. In *Microarchitecture (MICRO)*.

[30] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro* (2010).

[31] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *Computer Architecture (ISCA)*.

[32] Tom Way and Lori Pollock. 2002. Evaluation of a region-based partial inlining algorithm for an ILP optimizing compiler. In *Parallel and Distributed Processing Techniques and Applications (PDPTA)*.

[33] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. (2014).