Instant Profiling: Instrumentation Sampling for Profiling Datacenter Applications

Hyoun Kyu Cho
University of Michigan
netforce@umich.edu

Tipp Moseley
Google
tipp@google.com

Richard Hank
Google
rhank@google.com

Derek Bruening
Google
bruening@google.com

Scott Mahlke
University of Michigan
mahlke@umich.edu

Abstract

Profile-guided optimization possesses huge potential to save costs for datacenters. Hardware performance monitoring units enable profiling with negligible overhead and they have been proven to be effective to help programmers find code regions to optimize by monitoring datacenter applications continuously on live traffic. However, these hardware features are inflexible and often buggy, limiting the types of data that can be gathered. Instrumentation-based profiling can complement or replace hardware functionality by providing more flexible and targeted information gathering. Unfortunately, the overhead of existing instrumentation mechanisms prevents their use in production runs. In order to be used in datacenters, we need a profiling mechanism to impose overheads of less than a few percent, in terms of both throughput and latency, while still generating meaningful profile data.

This paper presents *instant profiling*, an instrumentation sampling technique using dynamic binary translation. Instead of instrumenting the entire execution, instant profiling periodically interleaves native execution and instrumented execution according to configurable profiling duration and frequency parameters. It further reduces the latency degradation of initial profiling phases by pre-populating a software code cache. We evaluate the performance and effectiveness of this new profiling technique on the SPEC CINT2006 benchmark suite and two datacenter application benchmarks. We show that it is well-suited for deployment to datacenters by incurring less than 6% slowdown and 3% computational overhead on average.

CGO '13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Optimization, Run-time environments

General Terms Languages, Performance

Keywords Profiling, Instrumentation, Datacenters

1. Introduction

As cloud computing continues to expand, profile-guided optimization (PGO) on datacenter applications has the potential for huge cost savings. Single-digit performance gains from the compiler can yield tens of millions of dollors in savings. Isolating the execution of datacenter applications can be complex or even impossible. One challenge of PGO on datacenter applications is collecting profile data from the applications running on live traffic [23]. In order to monitor production runs, the profiling overhead in terms of both throughput and latency should be kept minimal for several reasons. First and foremost, datacenter application owners are not tolerant of latency degradations (even at the 99th percentile) of more than a few percent, unlike high performance computing or other throughput-oriented applications, because they hurt the quality of service. Second, excessive profiling overhead can cause observer distortion that thwarts meaningful analysis. Finally, profiling overhead might offset the cost savings gained with PGO.

One way to keep the profiling overhead minimal is to exploit hardware support. For instance, specialized profiling hardware such as Merten's hot spot identification [20], Vaswani's programmable hardware path profiler [27], and Conte's profile buffer [13] has been proposed for low overhead profiling. Furthermore, many recent microprocessor designs have included on-chip performance monitoring units (PMU) [16–18] containing configurable performance counters that can trigger software interrupts for sampling.

Google-Wide Profiling (GWP) [23] has shown that PMU-based profiling mechanisms can maintain small enough overhead to be deployed for large datacenters monitoring applications running on live traffic.

Although hardware profiling mechanisms incur low overhead, they suffer from limitations. First, the possible types of profile data are inherently defined by the features that the underlying microprocessor supports; thus, hardware profiling mechanisms are not as flexible as software-only mechanisms. In addition, PMU features are often very processor-specific, making profiling tools not portable. Lastly, as the top design priorities are hardware validation and processor performance, performance monitoring hardware tends to be considered as a second class feature with the increasing time-to-market pressures [25].

Such limitations of hardware profiling can significantly limit the potential of PGO for datacenter applications, since PGO systems must be aware of both what and how to optimize for effective optimization. Although PMUs implemented in recent microprocessors so far provide quite rich information on where to focus optimization efforts, deciding how to optimize is a considerably harder problem. For example, sampling the program counter (PC) at a high rate yields enough information to detect hot code, and current PMUs are even capable of giving finer information such as cache miss and branch mispredict PCs. However, PMU features so far give less attention on how to optimize.

While instrumentation-based profiling mechanisms can provide more useful information about how to optimize the target applications, they tend to impose higher overheads than hardware-based mechanisms. For instance, path profiling [4] is well-known to be effective for improving code layout and superblock formation, but incurs 30-40% overhead. Other techniques such as value profiling [8] and data stream profiling [12] not only achieve gains of over 20% but also cause ten- or hundred-times slowdowns during profiling. Such high overheads prevent these mechanisms from consideration for profiling even loadtests for datacenter applications.

In this work, we propose a novel instrumentation sampling technique, *instant profiling*, that uses dynamic binary translation. Instead of instrumenting the entire execution, instant profiling periodically interleaves native execution and instrumented execution. By adjusting profiling duration and frequency parameters, we can keep profiling overhead under a few percent, so that the framework can be used to continuously monitor cloud computing applications running in large scale datacenters with live traffic. We have implemented the prototype framework of instant profiling on top of DynamoRIO [6], and we evaluate the possibility of continuous profiling on real datacenter benchmarks.

Instant profiling offers the following features:

- Low computational overhead. Computational overhead includes the cycles consumed by the application as well as out-of-band computation like profiling and JITing. When target programs are running natively, instant profiling does not need to add any extra instructions to the programs, as opposed to previous techniques [3, 15] which need checking code even when not profiling. Also, we do not duplicate the original execution, unlike other prior work [21, 29]. For these reasons, instant profiling can keep the computational overhead minimized.
- Small latency degradation. Due to the overhead amortizing characteristics of dynamic translation techniques, end users might observe significant latency degradation for initial profiling phases even with low sampling rates. Instant profiling further reduces latency degradation by pre-populating a software code cache and jumping back to native after a predefined period.
- Eventual profiling accuracy. With sampling techniques, we cannot avoid making errors on profile data. Since our low overhead framework enables continuous profiling on production runs, however, the accuracy of instant profiling gets closer to full profiling with a long enough application lifetime or enough instances. Since the most important applications consume the most cycles, they will have the most instances, run the longest, and yield the most profiles.
- Flexibility. Instant profiling can be applied to any type of profiling or tracing as long as the entire execution does not need to be monitored, since it is an instrumentation-based profiling technique and does not rely on specialized hardware features. In addition, instant profiling is portable to other micro-architectures for the same reason.
- Tuning. The profiling duration and frequency are configurable, making it easy to adjust the tradeoff between information and overhead.

The remainder of this paper is organized as follows. Section 2 provides a brief explanation of dynamic instrumentation systems and DynamoRIO which we harness as a base platform. Section 3 then presents the design and implementation details of our instant profiling framework. Section 4 describes how the framework further reduces latency degradation by pre-populating its software code cache. Section 5 explores tuning tradeoffs and evaluates performance. Section 6 discusses related work, followed by Section 7 outlining future work. Finally, we summarize the contributions and conclude in Section 8.

2. Background

Before we delve into the details of instant profiling, we briefly describe dynamic binary instrumentation techniques and where extra overheads come from. Then we provide an overview of DynamoRIO upon which we implement the prototype framework of instant profiling.

2.1 Dynamic Binary Instrumentation

Dynamic binary instrumentation is a powerful technique for runtime program introspection, particularly collecting profile data for PGO. There are many dynamic binary instrumentation systems [6, 19, 22], sharing similar internal mechanisms. They intercept target applications' execution, instrument points of interest, place instrumented code in their software code cache, and execute it from the software code cache. Where and what to instrument are defined by users (client writers) via custom API's. One main benefit of instrumenting programs at runtime is the availability of a complete picture of programs' runtime behavior including shared libraries, plugins, and dynamically-generated code.

There are two major sources of overhead for dynamic binary instrumentation systems. One arises from the dynamic instrumentation systems themselves. Whenever the target program meets an unknown branch target, the dynamic instrumentation system must perform a code cache lookup, copy the original code to the software code cache and insert any necessary instrumentation. In order to make this process transparent to target programs, moreover, they have to save and restore program context. Although these costs are unavoidable, translation overheads can be amortized over long running time and there have been suggested many optimization techniques to reduce this type of overhead, e.g., direct and indirect branch linking, trace construction, register reallocation, etc.

The other source of overhead comes from the profiling client. For collecting profile data, dynamic instrumentation systems insert user-defined code into application code. As opposed to instrumentation overhead occurring only when new code comes into the software code cache, instrumented client code is executed every time the application code is executed. Thus, even fine-tuned profiling clients can impose large overheads, continuously throughout the target application's execution. Furthermore, while significant progress has been made in reducing the performance penalty of the dynamic instrumentation itself, less attention has been paid to user-defined profiling clients [30].

2.2 Overview of DynamoRIO

DynamoRIO [1, 6, 7] is an open source dynamic binary instrumentation system. DynamoRIO exports an interface for building a wide variety of dynamic tools (DynamoRIO clients) including program analysis, profiling, instrumentation, optimization, etc. It allows not just insertion of callouts/trampolines, but also arbitrary modifications to application instructions via a powerful instruction manipulation library and adaptive intermediate representation. DynamoRIO provides efficient, transparent, and comprehensive manipulation of an unmodified application running on stock operating systems (Windows and Linux) and commodity hardware (IA-32 and AMD64).

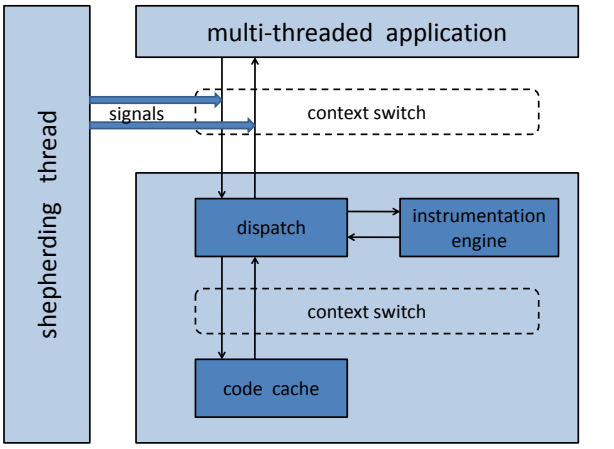

Figure 1. Control transfer for instrumentation sampling.

A thorough description of the internal design and implementation of DynamoRIO is outside the scope of this paper, but is described by Bruening [6].

3. Instrumentation Sampling

We modify DynamoRIO's control transfer for instrumentation sampling by interleaving native execution and instrumented execution. Unmodified, DynamoRIO initially takes over the control from native execution when DynamoRIO's shared library is loaded into the target program's address space, and never gives it back. On the other hand, our instant profiling framework gives back the control to native program execution right after initialization. During initialization, it sets up a signal handler for pre-defined profiling start/stop signals and creates a shepherding thread. After it starts executing the target program natively, the framework periodically takes over and gives back the control from and to native execution for sampling.

Figure 1 shows how control is transfered between native execution and instrumented execution in the instant profiling framework. The shepherding thread manages control transfers by periodically sending a profiling start/stop signal to each application thread, according to the profiling duration and frequency parameters. Then, the registered signal handler for the predefined signal transitions between native execution and instrumented execution. In order to make the instrumentation transparent to the target program, instant profiling needs to save and restore the target program's state every time the control is transfered via context switch.

If the profiling start signal is delivered when the thread is running natively, the signal handler saves the program state and hands over the control to the dispatch unit. iThe dispatch unit then checks whether the current program counter (PC) exists in the software code cache. If so, it restores the saved program state and executes the target code from the software code cache. If the current PC does not exist in the software code cache it invokes the instrumentation engine to instrument the target code region and place it in the software

code cache. Then the dispatch unit switches the context to the software code cache.

The transition from instrumented execution to native execution happens in a similar way. In this case, however, the context switch can only occur in between two instrumented fragments. A fragment is DynamoRIO's unit of translation and it can be either a basic block or a trace. Mapping the code cache state back to a native state is most easily done at the start or end of a code fragment. Thus, the signal handler delays the context switch until the current fragment in the software code cache finishes.

The rest of this section describes the technical issues involved in making the start/stop profiling transitions lightweight and transparent to the target program.

3.1 Context Switch

For a sampling mechanism to be effective, transitions of start/stop profiling should be very lightweight. Otherwise, the transitions would encroach on the overhead budget. In order to make the transitions lightweight, our instant profiling framework minimizes the operations needed for the context switch between native execution and instrumented execution.

The framework performs a context switch to start profiling as follows: when the start profiling signal handler gets a signal, the kernel hands over the machine context of native execution to the signal handler in the form of a *sigcontext* struct, which the handler passes to the dispatch unit after modifying a few fields To invoke the dispatch unit, the ip register in the sigcontext is set to the re-entry point of the dispatch unit. Then, when the signal handler returns, the kernel gives the control to the dispatch unit. The dispatch unit starts instrumented execution starting from the program counter value saved from the sigcontext struct.

3.2 Temporal Unlinking and Relinking of Fragments

One of DynamoRIO's optimizations that has a large impact on performance is direct and indirect branch linking. Since a context switch between the software code cache and the dispatch unit is expensive, DynamoRIO links branches instead of switching context whenever a branch target exists in the software code cache.

Although it is good for performance, the direct and indirect branch linking optimization can cause a problem for sampling control. Assume a thread is running inside a loop linked in the software code cache. When the stop profiling signal is delivered, the signal handler sets up the control transfer and continues running in the software code cache since it is in the middle of a fragment. In this case, however, the control transfer does not happen until the execution actually finishes running the loop and returns to the dispatch unit. For this reason, the direct and indirect branch linking optimization can cause unbounded profiling.

In order to prevent unbounded profiling, our instant profiling framework temporarily unlinks the outgoing branches of the currently running fragment when it gets the stop profiling signal. For better performance, the framework needs to re-link the branches afterward. So, it saves the unlinked branches in a scratch-pad data structure and re-links them when it restarts profiling.

3.3 Multi-threaded Programs

Although unmodified DynamoRIO seamlessly supports multi-threaded programs, we need several special treatments due to the structural difference between our instant profiling framework and DynamoRIO. The key issue is how to take over the control of all threads when we want to start profiling. This is not a problem for unmodified DynamoRIO since it takes over the control of the main thread before it spawns any other threads, observes every system call including thread creation, and never gives up the control of any thread. On the other hand, our framework only takes over the control when it is doing profiling, and does not keep supervision when the threads are running natively.

The basic strategy that our framework takes is to force its own signal handler for every thread and to send a profiling start signal to each thread. The shepherding thread can enumerate the thread IDs of every application thread even when they are not created and/or running under control of the framework, and send each thread a pre-defined signal that can be easily configured with a parameter. Since the kernel calls the registered signal handler when the signal is delivered, the framework can take over the control of every thread whenever it needs to in this way.

One problematic case is when the target program tries to mask the signal that we use or to register another handler for the signal. In this kind of conflict, the simplest circumvention is to use a different signal that is not touched by the application. For this purpose, the signal number we use as the start/stop profiling signal can be easily configured via a command-line parameter. Another solution is to intercept those tries by slipping in our wrapper functions for the library functions such as sigaction(), signal(), or sigprocmask(). In this case there still can be applications which directly call system calls (e.g., with assembly language), and they need to be handled with ptrace. They are extremely rare cases, however, especially for datacenter applications which mostly use standard libraries for portability. Finally, for the programs we have tested so far, changing the signal was enough.

3.4 Summarizing Profile Data

In order to enable profiling clients to summarize their results, our instant profiling framework extends DynamoRIO's API. DynamoRIO has various API functions to register customized instrumentation points and we add one more type of such event.

- dr_register_profiling_end_event(function)

The function registered with this API is called by the shepherding thread after every profiling phase. In this way profiling clients can manage profile data. The summarizing overhead can be hidden as it is performed in the shepherding thread and not included in an application's critical path.

4. Pre-populating Software Code Cache

Our instant profiling framework further reduces the latency degradation by pre-populating its software code cache. As mentioned in Section 1, minimizing latency degradation is extremely important for datacenter applications as it is directly related to the applications' quality of service. Many systems have expected 99th percentile latencies under 10ms. Meanwhile, using a software code cache technique amortizes its translation overhead over continued reuse of translated code. This means that end users may observe latency degradation for initial profiling phases even though we keep average overhead very small by setting a low profiling frequency. Instant profiling does not have to manifest instrumentation overhead to users, however, as it does not always run the programs from the software code cache. In other words, we can hide instrumentation overhead by instrumenting target code in parallel while the program is running natively. This can be understood in a similar way to prefetching into an instruction cache implemented in many modern micro-architectures, and we call this technique pre-populating a software code cache.

Our instant profiling framework decides which code regions to instrument for pre-populating its software code cache based on locality. When the target program is running natively, it uses hardware performance monitoring units to collect program counter samples. It is likely that those code regions with high sample counts will be executed again when the framework starts profiling. Therefore, it pre-populates its software code cache with the basic blocks containing the program counter whose counts counts exceed a threshold.

4.1 Finding Basic Block Headers

Finding code regions to instrument from program counter samples is not a trivial task, especially for processors with variable-length instructions like IA-32/AMD64. For DynamoRIO, code fragments are tagged and managed with the program counter values of their first instructions. Given a program counter, therefore, we need a mechanism to find the basic block header including that program counter.

One heuristic can be backward decoding. Starting from the target program counter, it decodes previous bytes until a valid instruction is found. The heuristic repeats this process until it meets a branch instruction, at which point it takes the post-branch program counter as the basic block header. With RISC architectures where instructions have fixed length, backward decoding works quite well. However, the overhead is too high for architectures with variable-length instructions. The overhead prohibits it from being used for datacenter applications, since our framework works on IA-32/AMD64 processors.

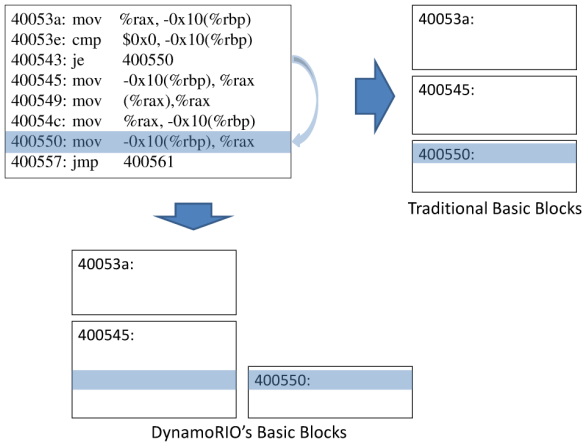

Figure 2. Traditional vs. DynamoRIO's basic blocks.

Instead of the backward decoding heuristic, our framework performs forward decoding. From the entry points of text segments, it decodes consecutive instructions in order and also records branch targets. After finishing this process, instructions following branch instructions and branch targets start new basic blocks. We save these basic block header addresses in sorted order. Then, we can identify the basic block header containing a given program counter with binary search. The overhead of initial basic block header calculation can be hidden by performing it before the start of profiling, or it can be done offline.

4.2 Affinity-based Pre-population

A given program counter sample can be in multiple basic blocks for DynamoRIO since its basic blocks are different from the traditional static analysis notion of basic blocks. The example in Figure 2 shows the difference between traditional basic blocks and DynamoRIO's basic blocks. DynamoRIO considers each entry point to begin a new basic block, and follows it until a control transfer is reached, even if it duplicates the tail of an existing basic block.

DynamoRIO uses this notion for simplicity of code discovery at runtime [6], but it can decrease the hit ratio of software code cache pre-population. For instance, suppose program counter 400550 in Figure 2 is sampled for pre-population. The basic block header found by the search in Section 4.1 will yield only 400550. For actual instrumented execution, however, both basic blocks starting from 400545 and 400550 can be encountered.

In order to solve this problem and exploit spatial locality in higher degree, our instant profiling framework adopts affinity-based pre-population. Instead of just pre-populating the software code cache with basic blocks containing sampled program counter, the framework also instruments additional basic blocks close to those basic blocks. Starting from the basic blocks found from program counter samples, it includes the branch targets of those basic blocks. It discovers

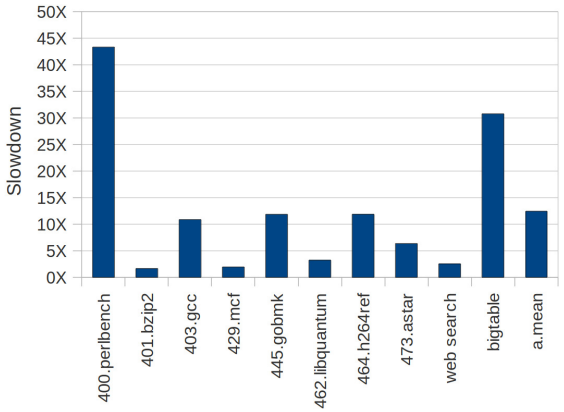

Figure 3. Overhead of edge profiling.

target basic blocks in a breadth-first-search-like manner to a pre-defined depth.

5. Performance Evaluation

Instant profiling balances a tradeoff between information and overhead. This balance can be controlled with two parameters. The first parameter, profiling duration, controls how long one profiling phase lasts. A longer profiling duration gives more information, but also incurs higher overhead. Moreover, it is possible that end users might feel intermittent latency degradation during profiling phases. So we limit profiling durations to a few milliseconds at maximum. Another parameter that affects the profiling overhead is profiling frequency. Considering datacenter applications' long running characteristics, our scheme of profiling a very small portion of execution can yield arbitrarily low average computational overhead, while still giving meaningful profile data. Since most of our benchmark workloads run only for a few tens to hundreds of seconds, however, we set profiling frequency relatively high – once in a few seconds at minimum. In these experiments, a pair of profiling duration and frequency parameters sets how long and how often profiling is performed. For example, the (2ms/4s) setting means profiling is conducted for 2 milliseconds for every 4 seconds. We compare results for (2ms/4s), (1ms/1s), (2ms/1s), (4ms/1s), and (2ms/250ms).

5.1 Experimental Configuration

All experiments are performed on a system with a 6-core Intel Xeon 2.67GHz processor with 12,288KB L3 cache. The system has 12GB of memory and is running Linux kernel version 2.6.32. We used gcc 4.4.3 to compile all binaries with -O3 optimization.

Instant profiling is evaluated using the SPEC CPU2006 integer benchmark suite and two proprietary datacenter application benchmarks. For the SPEC CPU2006 benchmark suite, the floating point benchmarks are omitted because they generally exhibit highly repetitive behavior that is not as interesting from the perspective of profiling. In addition,

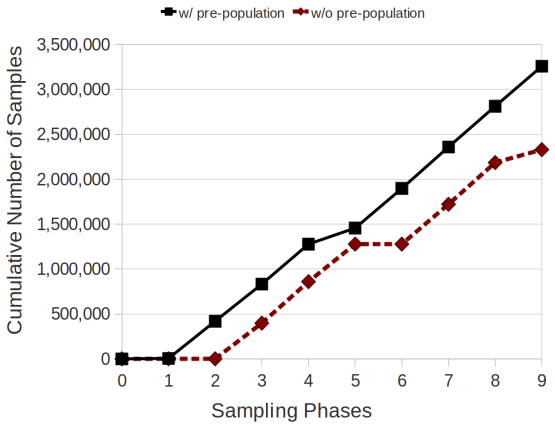

Figure 6. Effect of pre-populating a software code cache.

four integer benchmarks are omitted because our prototype framework does not yet work for them. The datacenter applications are web search and BigTable [9]. Although each experiment presented is the average of three repeated trials, there still exists some degree of variability in performance and accuracy due to the non-determinism caused by random starting points of profiling and thread interleaving.

5.2 Edge Profiling

We choose edge profiling as a profiling client to demonstrate the effectiveness of instant profiling, since it is widely used and relatively simple to implement, but incurs considerable overhead. Edge profiling is a traditional control flow profiling technique for profile-guided optimization. It measures how many times each edge (branch transition) in control flow graphs executes, and has been the basis of path-based optimizations that select hot paths. Although edge profiling collects strictly less information than path profiling, Ball[5] shows that various hot path selection algorithms based on edge profiles work extremely well in most cases.

Figure 3 presents the overhead of our edge profiling client, when it runs on original DynamoRIO without sampling. This naive implementation has little tuning or optimization, and its overheads are far larger than other optimized edge profiling techniques [14]. Although there are opportunities for optimizing the client itself, it is outside the scope of this paper and we demonstrate the effectiveness of instant profiling by showing how it performs even with a naively implemented experimental client. Since the tradeoff between information and overhead is tunable with sampling parameters, edge profiling makes a good test case because comparing edge profiles' quality is well studied.

5.3 Performance Overhead

The slowdowns caused by our instant profiling framework with the edge profiling client are shown in Figure 4. They are calculated as the profiled execution time (wall time) divided by the native execution time. Figure 5 also shows the computational overheads, which is calculated with CPU

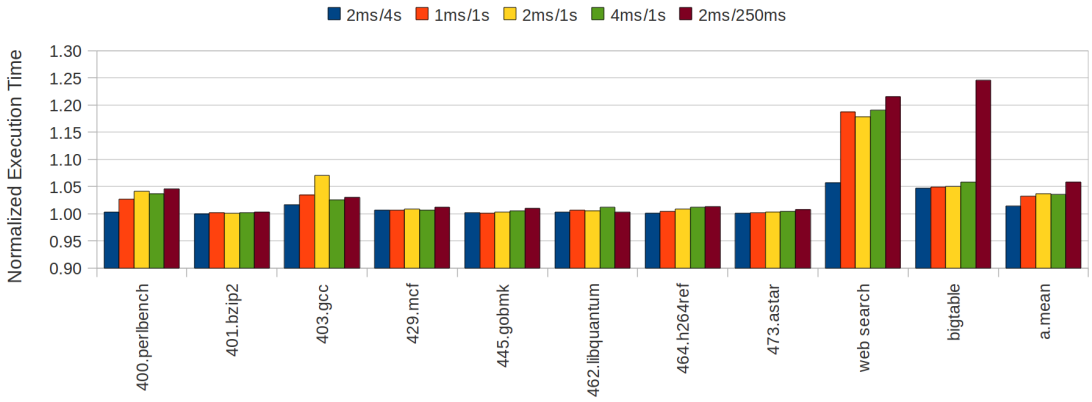

Figure 4. Execution time overhead of the instant profiling framework across five configurations of *(duration / frequency)*.

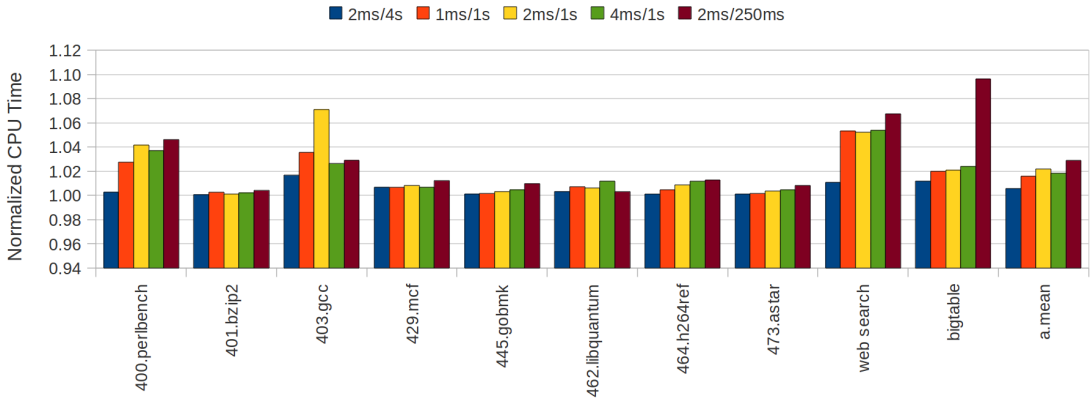

Figure 5. Computational overhead of the instant profiling framework across five configurations of *(duration / frequency)*.

time. For all configurations tested, the average slowdown ranges from 1.4% to 5.9%, and the average computational overhead ranges from 0.6% to 2.9%.

The main trend that can be observed is that increasing sampling rate either by increasing profiling duration or profiling frequency results in an increase in overhead. We chose profiling frequency once in every 4 seconds at least, since a few benchmarks only run about 30 seconds. For real datacenter environments, however, applications usually run much longer and there exist many instances of the same application running concurrently. In production environments, we can choose a much lower profiling frequency and expect commensurately lower overheads.

Although instant profiling can be tuned to impose very low average computational overhead, some of the configurations caused some benchmarks to slow down by up to 25%. There are two major locations where instant profiling adds extra instructions. One is profiling phases of every thread, but the durations of this type are controlled by the profiling duration parameter. The other location is the shepherding thread, especially the profile data summarizing phase. For the edge profiling client we used for the experiments, the shepherding thread summarizes and prints out profile data to disk after every profiling phase. While this overhead can be hidden for most benchmarks since it is not in the application's critical path, it can cause resource contention resulting in slowdowns. Although it is not yet clear, in our edge profiling case we think the resources that cause the slowdown are the data cache and load store queue. The two datacenter applications have larger working set size than SPEC benchmarks, and our edge profiling client traverses edge counters after every profiling phase. This increases the pressure on the data cache. Also, we use atomic increment instructions to modify edge counters for the datacenter benchmarks, since they are highly multi-threaded and non-atomic increments can cause data races on the counters in this case. This can impose substantial contention on the load store queue. As we can see with the bars where profiling frequency is 4 seconds, however, even the overhead caused by the resource contention of naively implemented profiling clients can be kept small with proper parameter settings. Moreover, we expect this overhead would go further down with practical profiling frequency in real datacenter environments.

We also examine how pre-populating a software code cache can reduce latency degradation. Figure 6 shows the cumulative number of samples with and without pre-population, for the web search benchmark with the (4ms/1s) setting. As can be observed in the graph with small slope phases, instrumentation overhead to populate the software code cache can result in a small number of samples, and thus more latency

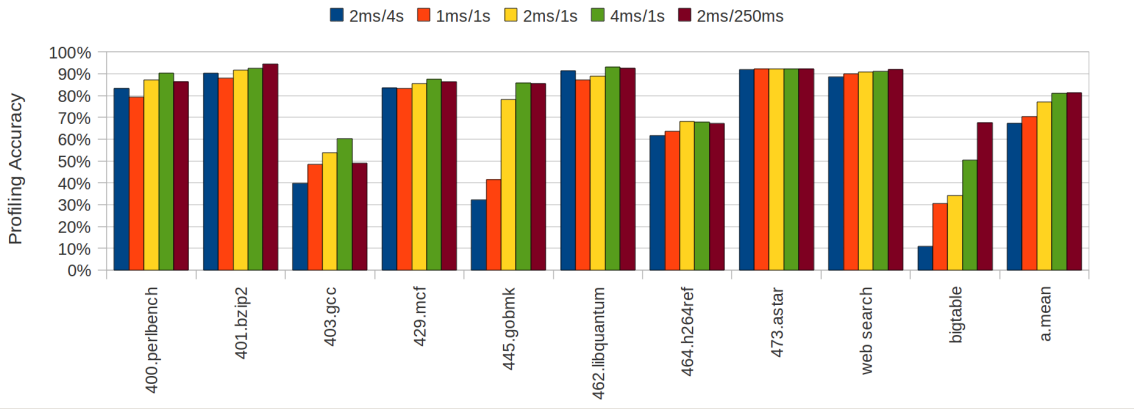

Figure 7. Edge profiling accuracy of the instant profiling framework across five configurations of *(duration / frequency)*.

degradation, for initial profiling phases. Pre-populating a software code cache reduces such degradation by decreasing the software code cache miss rate.

5.4 Profiling Accuracy

The accuracy of the edge profiling client can be ascertained by comparing the sampled profile with the profile collected with full instrumentation. We adopt a method similar to Wall's weight matching scheme [28]. We define edge profiling accuracy as

$$Accuracy = \frac{(MaxError - Error)}{MaxError} \times 100(\%) \quad (1)$$

$$Error = \sum_{e \in Edges} |freq_{full}(e) - freq_{sampled}(e)| \quad (2)$$

In the second equation, $freq_{full}(e)$ and $freq_{sampled}(e)$ represent relative frequencies of edge e in a fully instrumented profile and a sampled profile, respectively. Relative frequency is defined as the number of times that an edge is taken divided by the number of times any edge in the profile is taken. For the worst case results where edges are biased the opposite way, this error sums up to 2, defining $MaxError$ as 2.

The profiling accuracy of our instant profiling framework for edge profiling is shown in Figure 7. For all configurations tested, the average accuracy ranges from 67-81%, and many of the benchmarks achieve about 90% accuracy.

Despite many sources of noise, we can observe the general trend of increasing accuracy as profiling duration or profiling frequency increase. The more samples the framework collects, the closer the profile data gets to full instrumentation.

This can also be seen in Figure 8, which shows how the average accuracy changes as the number of profiling phases increases for two datacenter application benchmarks with (2ms/250ms) parameter setting. In the graph, although the curves are not strictly monotonic, we can see the accuracy generally goes up as more samples are collected.

Although the edge profile accuracy of our framework reaches 90% for many of the benchmarks, some benchmarks

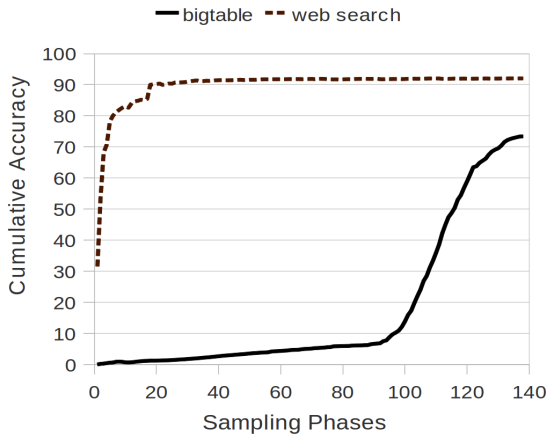

Figure 8. Asymptotic edge profiling accuracy.

such as BigTable and gcc show very low accuracy. The main reason for the low accuracy is that our framework could not collect enough samples as the execution time of these benchmarks is too short. For real datacenter environments, however, having low overhead is paramount and it can be tuned to collect profile data that would yield accuracy that is actionable with PGO.

6. Related Work

Inspired by the Digital Continuous Profiling Infrastructure (DCPI) [2], Google-Wide Profiling (GWP) [23] demonstrates continuous profiling is possible for datacenter applications running with live traffic. Although GWP also collects some lightweight callstack-based profiles through specialized libraries, it mainly relies on performance monitoring units (PMU) supported by recent microprocessors [18] to collect system-wide profiles with low overhead. The types of profiles GWP collects, therefore, are limited to the ones that either PMUs support or can be collected with specialized libraries. Our work tries to extend GWP for collecting more general profiles which can be gathered only through instrumentation. These types of profiles will enable more profile-guided optimizations (PGO) by providing profiles

that can help figure out not only ”what to optimize” but also ”how to optimize.”

Dynamic instrumentation tools such as DynamoRIO [6], Pin [19], and Valgrind [22] help instrument an application and collect general profiles of full execution. Even for simple profiles, however, the overhead of instrumenting an entire execution is prohibitive and infeasible to be deployed on datacenter applications running with live traffic.

One way to reduce the overhead of profiling is sampling, as several instrumentation approaches have demonstrated. Among them, the Arnold-Ryder instrumentation framework [3], implemented in the Jalapeno JVM, significantly lowers instrumentation overhead by sampling bursts of execution. It creates two versions of each procedure, one for checking and the other for actual profiling. The checking version counts how many times it is executed at procedure entries and loop back edges, and transitions to the profiling version if the counter reaches some pre-defined value. The profiling version collects an intra-procedural acyclic trace, resets the counter, and transitions back to the checking version. Bursty Tracing [15] extends the Arnold-Ryder framework for longer inter-procedural traces and further reduces the overhead with a few optimizations. In addition, Bursty Tracing is applied to IA32 binaries using the Vulcan binary rewriting tool, instead of Java bytecode.

Instant profiling is partially inspired by the Arnold-Ryder framework and Bursty Tracing. Instead of instrumenting all execution for checking, however, it does not instrument any code when it is not profiling. This is because even simple checking instrumentation imposes prohibitive overhead for datacenter applications. For example, even without any profiling client, the Arnold-Ryder framework results in instrumentation overhead of 6-35%, and Bursty Tracing lowers it to 3-18% [15]. On the other hand, instant profiling imposes less than 10% of computational overhead with a naive implementation of edge profiling. Unlike Arnold-Ryder or Bursty Tracing, moreover, instant profiling will incur neither latency degradation nor computational overhead while it is not profiling. Also, instead of managing a duplicate copy of every code region, instant profiling only JITs things it will likely need. Ephemeral Instrumentation [26] also takes a similar sampling approach to Bursty Tracing, but it is non-trivial to extend Ephemeral Instrumentation for many profile types because it uses branch patching; only information available at the branch can be recorded and it is difficult to find extra registers for architectures like x86. Conversely, instant profiling is flexible and not limited to any specific profile type. Finally, phase-guided profiling techniques [24] can help sampling-based profiling methods, including instant profiling, maintain higher accuracy while keeping the overhead low.

Another vein of previous work to reduce profiling overhead is to exploit parallelism for profiling. As the micro-architectural trends move toward massively multi-core processors, Shadow Profiling [21] and SuperPin [29] aim to leverage the abundance of extra hardware. Shadow Profiling runs the original program uninstrumented in parallel with instrumented slices to perform profiling. SuperPin uses a similar approach, but tries to deterministically replicate full execution by creating slices of execution between each system call. They both exploit modern kernels' copy-on-write mechanism by forking new processes for profiling. They significantly reduce the slowdown caused by profiling since the original process is not instrumented. However, SuperPin is not deployable for datacenters as it at least doubles resource contention, especially CPU utilization and memory bandwidth. Also, virtualizing fork for multi-threaded programs is very challenging to implement robustly and their initial implementations only support single-threaded programs.

PiPA [30] also exploits parallelism but in a different way. Instead of profiling in an extra process, it performs profiling in the same thread to produce compact profiles, and uses multiple threads to pipeline processing and analyzing of profile data. PiPA is particularly effective for the types of profiling that need complicated post-processing such as cache simulation.

There have been suggested many techniques specialized for other types of profiling. Ball [4] proposes techniques for path profiling. Calder [8] suggests an optimization to turn off profiling by realizing profile data is converging for value profiling. Chilimbi [11] proposes a compact representation for memory stream profiles. Instant profiling is orthogonal to these profile-specific techniques including PiPA, and they can be used to further improve the overhead.

7. Future Work

Since instant profiling does not need to run the target program from the software code cache all the time, further dynamic optimization for profiling code is possible in parallel. Bruening [7] has shown that dynamically applying optimizations such as redundant load removal, strength reduction, and indirect branch dispatch can yield substantial improvement, even for binaries already highly optimized with a static compiler. As optimizing instrumented code is difficult for client writers, we expect there would be potential to improve performance even more for dynamic optimization on profiling code.

Another realm of research we want to explore is how to leverage more detailed profile data for feedback-directed optimization (FDO) on datacenter applications. Although Chen [10] presented how hardware event samples can be used for FDO of general programs and GWP [23] profiles have provided performance insights for datacenter applications, not much has been studied about other applications of FDO on datacenter applications. As instant profiling enables various types of profiles to be collected continuously from datacenters, we expect this information to contribute to improving the performance of cloud computing.

8. Conclusion

We introduce instant profiling, a novel approach to reduce the overhead of instrumentation-based profiling for datacenter applications. The technique works by executing instrumented profiling code from a software code cache for only a short profiling duration. For normal execution phases, the original binary runs natively without any instrumentation. We further avoid possible latency degradation for initial profiling phases by pre-populating the code cache. The prototype framework of instant profiling is built on top of DynamoRIO, and it is evaluated on the SPEC CPU 2006 integer benchmark suite and two datacenter application benchmarks. We show that the overhead of profiling in terms of both throughput and latency can be kept to acceptable levels for continuous profiling of live datacenter applications. Furthermore, we have shown that sampling-based continuous profiling can yield asymptotically accurate profiling results with negligible overhead by collecting profile data over many instances or a long time period.

Acknowledgments

We would like to thank Google for providing the resources that made this study possible. The work was also supported in part by the National Science Foundation under grant CNS-0964478. Finally, we want to acknowledge our anonymous reviewers for their helpful feedback on the paper.

References

[1] Dynamorio: Dynamic instrumentation tool platform - http://dynamorio.org/home.html.

[2] J. M. Anderson et al. Continuous profiling: Where have all the cycles gone? In *Proc. 16th SOSP*, pages 1–14, 1997.

[3] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proc. '01 PLDI*, pages 168–179, 2001.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *Proc. 29th MICRO*, pages 46–57, 1996.

[5] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: The showdown. In *Conf. Record of the 25th POPL*, pages 134–148, 1998.

[6] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, M.I.T., Sept. 2004.

[7] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proc. 2003 CGO*, pages 265–275, 2003.

[8] B. Calder et al. Value profiling and optimization. *The Journal of Instruction-Level Parallelism*, 1, 1999.

[9] F. Chang et al. Bigtable: A distributed storage system for structured data. In *Proc. 7th OSDI*, pages 205–218, 2006.

[10] D. Chen et al. Taming hardware event samples for fdo compilation. In *Proc. 2010 CGO*, pages 42–52, 2010.

[11] T. M. Chilimbi. Efficient representation and abstractions for quantifying and exploiting data reference locality. In *Proc. '01 PLDI*, pages 191–202, 2001.

[12] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proc. '02 PLDI*, pages 199–209, 2002.

[13] T. M. Conte, B. A. Patel, and J. S. Cox. Using branch handling hardware to support profile-driven optimization. In *Proc. 27th MICRO*, pages 12–21, 1994.

[14] A. E. Eichenberger and S. M. Lobo. Efficient edge profiling for ilp-processors. In *Proc. 7th PACT*, pages 294–303, 1998.

[15] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Proc. 4th Workshop on Feedback-Directed and Dynamic Optimization*, pages 117–126, 2001.

[16] *PowerPC 740/PowerPC 750 RISC Microprocessor User's Manual*. IBM, 1999.

[17] *Intel Itanium 2 Processor Reference Manual: For Software Development and Optimization*. Intel Corporation, May 2004.

[18] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, Nov. 2006.

[19] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. '05 PLDI*, pages 190–200, 2005.

[20] M. C. Merten et al. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proc. 26th ISCA*, pages 136–147, 1999.

[21] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proc. 2007 CGO*, pages 198–208, 2007.

[22] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. '07 PLDI*, pages 89–100, 2007.

[23] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 30(4):65–79, July 2010.

[24] A. Sembrant, D. Black-Schaffer, and E. Hagersten. Phase guided profiling for fast cache modeling. In *Proc. 2012 CGO*, pages 175–185, 2012.

[25] B. Sprunt. Performance monitoring hardware will always be a low priority, second class feature in processor design until... In Workshop on Hardware Performance Monitors in conjunction with HPCA-11.

[26] O. Traub, S. Schechter, and M. D. Smith. Ephemeral instrumentation for lightweight program profiling, 2000.

[27] K. Vaswani, M. J. Thazhuthaveetil, and Y. N. Srikant. A programmable hardware path profiler. In *Proc. 2005 CGO*, pages 217–228, 2005.

[28] D. W. Wall. Predicting program behavior using real or estimated profiles. In *Proc. '91 PLDI*, pages 59–70, 1991.

[29] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proc. 2007 CGO*, pages 209–220, 2007.

[30] Q. Zhao, I. Cutcutache, and W.-F. Wong. Pipa: Pipelined profiling and analysis on multi-core systems. In *Proc. 2008 CGO*, pages 185–194, 2008.