Adaptive Correction of Sampling Bias in Dynamic Call Graphs

BYEONGCHEOL LEE, Gwangju Institute of Science and Technology

This article introduces a practical low-overhead adaptive technique of correcting sampling bias in profiling dynamic call graphs. Timer-based sampling keeps the overhead low but sampling bias lowers the accuracy when either observable call events or sampling actions are not equally spaced in time. To mitigate sampling bias, our adaptive correction technique weights each sample by monitoring time-varying spacing of call events and sampling actions. We implemented and evaluated our adaptive correction technique in Jikes RVM, a high-performance virtual machine. In our empirical evaluation, our technique significantly improved the sampling accuracy without measurable overhead and resulted in effective feedback directed inlining.

CCS Concepts:• **Software and its engineering → Runtime environments**; *Just-in-time compilers;* Dynamic compilers;

Additional Key Words and Phrases: Dynamic call graph, sampling, correction

ACM Reference Format:
Byeongcheol Lee. 2015. Adaptive correction of sampling bias in dynamic call graphs. ACM Trans. Archit. Code Optim. 12, 4, Article 45 (December 2015), 24 pages.
DOI: http://dx.doi.org/10.1145/2840806

1. INTRODUCTION

To develop and maintain large and complex software systems running on increasingly powerful hardware, programmers organize their programs as a collection of independent, interchangeable modules and combine them on demand. Programming languages, tools, and runtime systems feature modules, static and dynamic linking, and polymorphic dispatch. While this modular design practice is often desirable, the runtime overhead is significant when a program made up of many small methods spends a large fraction of its time on locating the precise target of the callees at polymorphic call sites and executing a sequence of instructions of entering and exiting methods. To analyze and optimize this runtime overhead around method boundaries, programming tools and runtime systems monitor the call events in a program run and summarize the frequency statistics as a *Dynamic Call Graph* (DCG) where nodes and edges represent methods and caller-callee pairs, and each edge is labeled by its execution frequency.

Profiling dynamic call graphs presents a trade-off between accuracy and overhead. Full instrumentation builds complete and accurate dynamic call graphs, but the overhead is not acceptable in some contexts. For instance, dynamic optimizers expect that the optimization benefit should significantly exceed the cost of profiling and

---

New article, not an extension of a conference paper.
This work is supported by Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korean government (MSIP) (No. R0190-15-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development). This work is supported in part by Korean government through IITP grant R0190-15-2012.
Author's address: B. Lee, School of Information and Communications, Gwangju Institute of Science and Technology, 123 Cheomdangwagi-ro, Buk-gu, Gwangju 61005, Korea; email: byeong@gist.ac.kr.

ACM 1544-3566/2015/12-ART45 $15.00
DOI: http://dx.doi.org/10.1145/2840806

optimizing the programs at runtime. Sampling approaches reduce the overhead dramatically by taking a subset of call events in a program run as samples at regular timer ticks [Arnold and Grove 2005; Grcevski et al. 2004; Zhuang et al. 2006] or stopping runtime monitoring once sufficient amounts of events are collected [Paleczny et al. 2001; Suganuma et al. 2001]. While sampling approaches keep the overhead within a desirable range (e.g., 0.1%), its accuracy is still far away from perfect accuracy due to sampling bias. Timer-based sampling assumes that both observable call events and sampling actions are equally spaced in time so that each call event has an equal chance of being a sample. Unfortunately, call events are generally not equally spaced in time in general and the operating system scheduler may distribute sampling actions unevenly in time.

This article analyzes and corrects the two major sources of sampling bias in timer-based sampling for dynamic call graphs: (1) unequally spaced call events and (2) unequally spaced sampling actions. To characterize unequally spaced call events and sampling actions, we introduce *call density* as the number of call events per unit time and *sampling latency* as the time from triggering a sampling action to taking a sample. We show that a timer-based sampling mechanism overcounts a sample when the call density and sampling latency at the point of sampling are low and high, respectively. As an antibias action, our adaptive sampling correction technique dynamically tracks both call density and sampling latency and weights samples proportionally and disproportionally to the current call density and sampling latency.

We implemented and evaluated our adaptive sampling correction technique in Jikes RVM [Alpern et al. 2005] running the 19 benchmarks from SPECjvm98 [Standard Performance Evaluation Corporation 1999] and DaCapo-2006-MR2 [Blackburn et al. 2006]. Our approach improved DCG accuracy over the default sampling configuration in Jikes RVM. Compared to the full instrumentation, default sampling attained 61% accuracy and our adaptive sampling correction technique increases accuracy to 72% without measurable overhead.

In summary, this article's contributions are (1) analysis of the sampling bias in a timer-based sampling for dynamic call graphs, (2) development of an adaptive sampling correction technique weighting samples based on call density and sampling latency, and (3) evaluation of our adaptive sampling correction technique.

This article is organized as follows. We first analyze sampling bias by the two motivating examples (Section 2). Next, we introduce our adaptive technique for correcting sampling bias with implementation details (Section 3). We then describe our experience with adjusting an optimizing compiler to exploit high-accuracy dynamic call graphs from correcting sampling bias (Section 4). Next, we describe the methodology used to support our claims regarding sampling bias and our antibiasing techniques (Section 5). Finally, we support our claims with experimental results (Section 6).

2. BACKGROUND AND ANALYSIS

This section first discusses how profilers collect call events in a program run and build aDCG as a summary of interprocedural call statistics. We next examine how timer-based sampling approaches trade accuracy for overhead. Finally, we analyze how unequally spaced call events and sampling actions introduce sampling bias and lower the accuracy of dynamic call graphs.

2.1. Collecting Dynamic Call Graphs

Programming tools and runtime systems collect dynamic call graphs to direct programmers and optimizing compilers to locate and eliminate highly frequent activations of subroutines at call sites [Arnold and Grove 2005; Suganuma et al. 2002; Chang and Hwu 1989; Graham et al. 1982]. A dynamic call graph maps caller and callee

relationships to their frequency values. Formally, a $DCG = (N, E, w)$ is a directed multigraph with a weight function. N is a set of nodes that represent a program's methods. Each edge in the set E is a triple $(caller, callsite, callee)$ that represents a set of the call events where the *caller* method in N calls the *callee* method in N at the *callsite* location. The weight function w maps a call edge in E to a real number to express the frequency of the call events from the caller to the callee in a program run.

To collect dynamic call graphs, compilers place a few instructions at the prologues of the methods in a program. These instructions trace the chain of activation stack frames based on a calling convention, identify the recent call event, and increment the weight value of the call edge in the DCG [Graham et al. 1982].

2.2. Timer-Based Sampling

Collecting all call events and updating the frequency values in the DCG adds substantial runtime overhead, making full instrumentation prohibitively expensive for dynamic optimizers. To keep the time overhead within a desirable bound (e.g., $\leq$1%), dynamic optimizers skip a large fraction of call events and take one or a few samples at periodic intervals (e.g., every 4ms). In the dynamic optimization context, compilers place yield points at method prologues, loop back edges, and optionally method epilogues so that application programs regularly yield to the runtime systems for a variety of services including garbage collection, profiling, biased lock revocation, fat lock deflation, deoptimizations, and debugging [Arnold and Grove 2005; Suganuma et al. 2002]. Runtime systems piggyback on these prologue yield points to sample call events and build an approximated dynamic call graph. For instance, the timer thread in Jikes RVM sleeps for some period (e.g., 4ms) and sets the thread local flag bit of each thread to direct the thread to take a yield point. A method prologue yield point in an application thread pools its thread local flag bit and branches to the system code of taking a call event sample when the flag bit is set.

Sampling produces a statistically representative result as long as it takes a sufficient number of samples, and each event in the population has equal chance of being a sample. Bursty counter-based sampling [Arnold and Grove 2005] takes multiple samples and skips N call events at each timer tick to increase the sampling rate and eliminate sampling bias in the call events that are aligned at timer ticks. `tprof` adds randomness to the timer ticks and eliminates sampling bias toward samples that are systematically aligned with timer ticks [Mytkowicz et al. 2010]. While all these approaches increase fairness in sampling, they miss some important classes of sampling bias in profiling dynamic call graphs.

2.3. Analysis of Sampling Bias

The fundamental problem in timer-based sampling is the mismatch between the timer-based triggering mechanism and frequency-oriented profiles [Arnold and Grove 2005]. This mismatch becomes significant due to unequal spacing in call events and sampling actions.

Unequally Spaced Call Events. Figure 1 illustrates a program that generates unequally spaced call events in a program run. The `main` entry method calls the `dense` and `sparse` method one by one. The `dense` method calls the `compute` method repeatedly with a small input size so that the call events from `dense` to `compute` are narrowly spaced in time. On the other hand, the `sparse` method doubles the input size to the `compute` method to space its call events sparsely. The dynamic call graph contains the four edges that are identified by their call sites: `A`, `B`, `C`, and `D`. `A` and `B` are executed only once from the `main` entry method, while `C` and `D` must have the same frequency value. Figure 2 lays out the call events and their samples in time while running the program

```
public class CallDensityBias {
    public static void main(String[] args) {
      int N = Integer.parse(args[0]);
A:    dense(N);
B:    sparse(N);
    }
    static void dense(int repeat) {
      for(int i = 0; i < repeat;i++)
C:      compute(1);
    }
    static void sparse(int repeat) {
      for(int i = 0; i < repeat;i++)
D:      compute(2);
    }
    static void compute(int size) {
      ... // The execution time is proportional to the "size" argument.
    }
}
```

Fig. 1. A program generating unequally spaced call events.

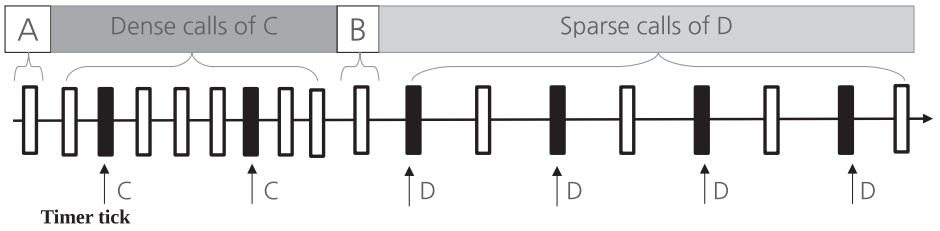

Fig. 2. Timer-based sampling under time-varying call density.

to illustrate how a timer-based sampling would attribute unequal weights to the call edges, which are supposed to have the same weight values in a full instrumentation. The program run has a phase behavior in the number of call events in a unit time. In the phase of dense calls of `C`, the average number of calls from old timer ticks (exclusive) to new timer ticks (inclusive) is 4. On the other hand, in the phase of sparse calls of `D`, the average number of calls between two successive timer ticks is 2. The time-based sampling attributes two and four samples to `C` and `D`, respectively, while an ideal sampling technique should attribute the same count numbers to these two call edges in the dynamic call graph. In short, the sampling bias in timer-based sampling is inversely proportional to the number of calls in a unit time. While we configured the iteration count N to be eight and the sampling interval to be hundreds of machine cycles for simplicity, we observed the same sampling bias in a realistic configuration with millions of iteration counts and the sampling interval of 4ms.

Unequally Spaced Sampling Actions. An ideal timer-based sampling would take samples at regular timer intervals, but, in reality, sampling intervals may not be equally spaced due to irregular sampling latency, from triggering a sampling action at a timer tick to taking a sample. When there is a long sampling latency in part due to operating system scheduling decisions and synchronizations, the call events right after a long sleeping period are more likely to be taken as a sample than the others. For instance, consider the multithreaded program with a contending lock in Figure 3. Each thread contends for the global lock (`lock`) and executes the `compute` method repeatedly while holding the global lock. While the two call sites in the `work` method execute the

```
public class LatencyBias {
  public static void main(String[] args) {
    final int NTHREAD= Integer.parseInt(args[0]);
    final int NITERATION = Integer.parseInt(args[1]);

    Thread[] workers = new Thread[NTHREAD];
    for(int i = 0; i < workers.length;i++) {
      workers[i] = new Thread() {
        public void run() { work(NITERATION);}};
      workers[i].start();
    }
    for(int i = 0; i < workers.length;i++)
      workers[i].join();
  }

  static final Object lock = new Object();

  static void work(int N) {
    for (int i = 0; i < N; i++) {
      synchronized (lock) {
        for (int j = 0; j < 50; j++)
A:        compute(1);
        for (int j = 0; j < 50; j++)
B:        compute(1);
      }
    }
  }

  static void compute(int size) {
    ... // The execution time is proportional to the "size" argument.
  }
}
```

Fig. 3. A program with a contending lock.

`compute` method the same number of times, the call events of `A` have a significantly higher chance of being taken as samples than the call events of `B`. Once one thread holds the lock for an extended period of time, the remaining threads get into a wait state. After a timer tick, these waiting threads in the blocked state cannot take a yield point and sample a call event. Once the lock owner thread releases the lock, one of the waiting threads would successfully acquire the lock, activate a yield point, and take the call events from `A` immediately. On the other hand, the call events from `B` are unlikely to be taken as a sample even though the call events from both `A` and `B` are supposed to have an equal chance of being taken as a sample. In short, all waiting time is credited to the call events that appear right after the waiting period.

Both unequally spaced call events and sampling actions contribute to sampling bias. To characterize unequally spaced call events and sampling actions, we introduced (1) call density, as the number of call events per unit time at a timer tick, and (2) sampling latency, as the time from triggering a sampling action to taking a sample. In our analysis with the two motivating programs, sampling bias is disproportional to the call density and proportional to sampling latency.

3. ADAPTIVE SAMPLING BIAS CORRECTION

This section presents our sampling bias correction technique and its implementation. Section 3.1 first discusses a technique for measuring call density and sampling latency, to characterize sampling bias in a sampling interval. Section 3.2 next presents how

our technique adjusts the weight value of samples to mitigate sampling bias at each sampling interval. Section 3.3 describes the implementation details of our adaptive sampling bias correction in the context of Jikes RVM, an adaptive optimization system.

3.1. Measuring Sampling Bias

We characterize sampling bias in terms of call density and sampling latency at each timer tick. The key constraint in measuring sampling bias is keeping measurement overhead as low as possible without perturbing observed program runs. Our lightweight measurement technique employs Time Stamp Counters (TSCs) and Hardware Performance Monitors (HPMs) that are available in modern processors.

Call Density. The call density at a timer tick is the number of call instructions normalized by the CPU cycles since the previous timer tick. To compute the call density at each timer tick, we rely on TSCs and HPMs to obtain CPU cycle times and call instruction counts, record their increments at each timer tick, and compute the call density as a ratio of the two increments. The call instruction counts and CPU cycle times are available directly from processors by executing special machine instructions (e.g., `rdtsc` in IA-32) and indirectly from operating systems through system calls (e.g,. `perf_event_open` and `read` in Linux). The thread startup routine records the current CPU cycle time and configures a HPM register to count retired call instruction events (e.g., `BR_INST_RETIRED.NEAR_CALL` in IA-32). The timer tick event handler obtains the increments in the CPU cycle time and call instructions count of the current thread and computes the call density. The thread completion routine releases the HPM resource by executing a system call (e.g., `close`).

Sampling Latency. Sampling latency at a timer tick is the number of CPU cycles from triggering the timer tick to taking a sample. We rely on TSCs for measurement. The thread startup routine initializes a thread local 64-bit unsigned integer variable `latency` that is large enough to hold the CPU cycle time. The timer tick generator temporarily stores the CPU cycle time at the timer tick into the `latency` variable. The timer tick handler subtracts the old CPU cycle time in `latency` from the CPU cycle time at the point of taking a sample and stores the result into the `latency` variable. `latency` holds the sampling latency at each timer tick.

3.2. Correcting Sampling Bias

Our adaptive sampling bias correction reduces the adversary effect of sampling bias due to time-varying call density and sampling latency at each timer tick. In a program run, the sampling bias at a timer tick is disproportional to the current call density and proportional to the current sampling latency. As an antibias action, our adaptive sampling bias correction weights a sample at a timer tick proportionally and disproportionally to the current call density and sampling latency, respectively. Consider a sequence of timer ticks in a program run, $t_1, t_2, \ldots, t_{i-1}, t_i, \ldots, t_n$, where they are designed to be separated by the constant *sampling period* as λ cycles. Let $calls(t_i)$ be the number of call events from the previous timer tick (t_{i-1}) to the current timer tick (t_i), and let $latency(t_i)$ be the sampling latency in cycles at the current timer tick (t_i). Let the current call density $density(t_i)$ equal $calls(t_i)/\lambda$. We define the *antibias weight* at a timer tick t_i:

$$weight(t_i) = \frac{density(t_i) \times 1{,}000}{(\lambda + \alpha \times latency(t_i))/\lambda},$$

where α is a constant *sampling latency factor* to control the effect of sampling latency on sampling bias. We design the antibias weight to hold the basic unit value when

the current call density is one call event per thousand cycles (0.001), and the current sampling latency is zero. The antibias weight ranges from 0 to 1,000 as long as the single processor core does not execute more than one call instruction in a cycle. To compute the weight value efficiently, we obtain a simplified formula:

$$weight(t_i) = \frac{calls(t_i)}{\lambda/1{,}000 + \alpha/1{,}000 \times latency(t_i)}.$$

$\lambda/1{,}000$ and α/λ are the constant parameters that model the effect of sampling period and latency on sampling bias.

For the prevalent machine environment where the typical sampling period is 4ms and processor frequency ranges from 1GHz to 4GHz, we set the sampling period (λ) to be 10,000,000 cycles and the sampling latency factor (α) to be 1,000. The sampling period (λ) ranges from 4 million cycles to 16 million cycles for the 1–4GHz processor operating frequency values, and our correction technique takes the average 10 million cycles as a representative one. We tried a variety of values for the sampling latency factor (α); we get a reasonable result when the sampling latency factor equals 1,000. Finally, we obtain a highly simplified formula:

$$weight(t_i) = \frac{calls(t_i)}{10{,}000 + latency(t_i)}.$$

Our sampling bias correction technique extends the dynamic call graph builder to increment the frequency of the call edge by the antibias weight rather than a constant value (e.g., 1).

3.3. Implementation in Jikes RVM

We extended Jikes RVM 3.1.3 to correct sampling bias in dynamic call graphs. The implementation is straightforward, and the scope of our extension was limited to adding three thread local variables and extending four methods.

The three thread local integer variables, `calls`, `latency`, and `fdhpm`, are used to keep track of call density and sampling latency. They are declared as instance field variables in the `RVMThread` class. `calls` holds the call instruction count of the previous timer tick. `latency` holds the sampling latency at the current timer tick. `fdhpm` holds the file descriptor of the HPM resource of counting call instructions.

The `startoff` method of the `RVMThread` class is the startup routine of Java threads. The startup routine initializes both `calls` and `latency` to hold all zeros. It next executes the `perf_event_open` system call to count the `BR_INST_RETIRED.NEAR_CALL` event and stores the file descriptor as the return value into `fdhpm`. Once the startup routine returns from the entry point method of the current thread, it releases the `fdhpm` resource.

The `run` method of the `TimerThread` class is the timer tick generator method containing a nonterminating loop of sleeping for some period (4ms in the default configuration) and turning on the yield point flag (the `takeYieldPoint` instance variable of the `RVMThread` class) of each thread. We extended the timer tick generator to store the current CPU cycle count at the point of triggering a sampling action into the `latency` thread local variable. To obtain the cycle count, the tick generator calls the `getTimeBase` method of the `Magic` class, which in turn executes the `rdtsc` instruction on IA32 processors.

The `yieldpoint` method of the `RVMThread` class is the timer tick handler method that is activated from the yield points in JIT compiled methods when the yield flag in the current thread is set by the timer tick generator. The timer tick handler examines the current call stack, identifies the current caller-callee pair, and inserts the pair into the global buffer. We extended the timer tick handler to compute the current antibias

weight value using the `calls` and `latency`. The global buffer was extended to annotate each pair of caller and callee with its antibias weight value. We completed the timer tick handler method by updating `calls` and `cycles` as the previous count values in the next timer tick handling.

The `thresholdReached` method of the `DynamicCallGraphOrganizer` class is the dynamic call graph builder method that aggregates caller-callee samples in the global buffer into the dynamic call graph. A background thread executes the build method whenever the global buffer is full. We extended the builder method to increment the frequency value of each sample in the global buffer by its antibias weight value rather than the default constant value (1).

The default counter-based sampling in Jikes RVM takes one or more samples at method prologues after a time tick. When taking N samples after a timer tick, our extension updates the call density and sampling latency only at the first sample and applies the same antibias weight to the $N - 1$ subsequent samples.

4. ADAPTING INLINING POLICY TO EXPLOIT HIGH-ACCURACY DYNAMIC CALL GRAPHS

One of the important clients of high-accuracy dynamic call graphs is profile-guided inlining, which replaces a frequently executed call site with the bodies of one or more callees. This section examines the evolution of profile-guided inline oracles and presents our inline oracle of exploiting the high-accuracy profiles from adaptive correction.

4.1. Evolving Inline Oracles to Exploit High-Accuracy Profiles

Inlining optimization replaces a call site with a copy of the callee's body for performance benefits: eliminating the instruction sequence of entering and exiting the callee method, specializing the callee's code in the context of the caller's code, and increasing spatial locality of the generated machine code. On the other hand, inefficient inlining decisions add significant time overhead to the compilation phases, increase memory pressure due to additional code and auxiliary metadata (e.g., garbage collection map), and decrease temporal locality of the generated machine code. To balance the cost of inlining against its benefits, profile-guided inline oracles examine dynamic call graphs where the high-frequency value of a call edge predicts the direct benefit, and the method size of a node estimates the cost.

Profile-guided inline oracles have coevolved with sampling-based profiling techniques through the iterative process of obtaining high-accuracy profiles, revising inlining heuristics, and generating high-quality inline decisions [Arnold and Grove 2005; Nakaike et al. 2014]. These inline oracles employ both static and dynamic heuristics to make up for the inaccuracy in the dynamic call graphs from sampling. Static heuristics inline small callee methods with zero or low-frequency values in the profiles. On the other hand, dynamic heuristics aggressively inline large callee methods with high-frequency values in the profiles. The early inline oracle of Jikes RVM [Arnold et al. 2000] employs static heuristics of inlining the callee methods whose inlined size is below a certain threshold as long as the normalized frequency of calls to them is above a certain threshold (e.g., 1%). In response to the unsatisfactory performance results with the poor inline decisions after applying the static heuristics to high-accuracy profiles from counter-based sampling, the inline oracle was revised to employ the dynamic heuristics of increasing the callee size threshold proportionally to the frequency values [Arnold and Grove 2005]. With highly accurate dynamic call graphs from the hardware-based sampling in the zEC12 processor [Shum et al. 2013], the cost-benefit-based inline oracle eliminates static heuristics completely and relies totally on dynamic heuristics [Nakaike et al. 2014].

Once highly accurate execution profiles are available, inline oracle designers weaken static heuristics and strengthen dynamic heuristics for cost-efficient inlining decisions.

Weakening static heuristics saves the cost of compiling and maintaining inactive copies of many small callees in a variety of compilation contexts. The runtime compilers then utilize their saved time for optimizing active copies of a small number of large callees. For instance, the cost-benefit-based inline oracle reduces compilation time, code size, and execution time in Java workloads on average [Nakaike et al. 2014].

4.2. An Inline Oracle for Adaptive Correction in Jikes RVM

To exploit high-accuracy profiles from our adaptive correction, we performed a series of experiments after removing the static heuristics in the default inline oracle in Jikes RVM completely, and we observed significant slowdown in some benchmarks. In particular, the purely dynamic heuristics excludes two critical inlining candidates: (1) inactive calls to trivial methods and (2) highly active calls to small methods without their samples due to omitted yield points.

Trivial methods have tiny bodies whose inlined instructions are smaller than their calling instruction sequences. Active calls to trivial methods are obviously desirable inlining candidates. On the other hand, inactive (dead) calls to trivial methods in a program run requires careful examination because inlining them could significantly increase the size of dead code in some cases. For instance, consider a high-frequency call site in a tiny wrapper method of calling its large implementation method. This wrapper method is a highly desirable trivial inlining candidate due to its tiny number of instructions of calling the implementation method. Inlining an inactive call to this wrapper method ends up with discovering the high-frequency call site, inlining the large implementation method transitively, and generating a large amount of dead machine code from the implementation method. Worse, the size increase in dead machine code is multiplied by the number of inlining decisions of this tiny wrapper method. A purely dynamic inlining oracle would reject all these inactive calls to trivial methods in favor of saving code space and compilation time, but a large fraction of these calls are desirable inlining candidates because inlining them reduces the execution time without increasing the size of compiled code significantly. For instance, the default constructor of the `Object` class in Java contains one instruction of returning to the caller method. Inlining this constructor method at an inactive call site eliminates the `return` instruction as well as the calling instruction sequence, deletes the false interference between the live variables and volatile registers at the call site, and reduces spill code instructions. In other words, inlining this constructor method improves the precision of dataflow analysis and reduces the execution time indirectly although the program does not execute this inactive call site directly. Our performance experiment revealed measurable runtime speedups from inlining inactive calls to trivial methods on the register-scarce IA32 architecture.

The yield points in software-based sampling techniques define the scope of profiling as a set of call events whose callee method contains a yield point at the prologue. Unfortunately, yield points have small but measurable time overhead [Lin et al. 2015], and dynamic optimizers omit the yield points in the small methods that do not need to set up stack frames [Arnold et al. 2004]. The call events without yield points in the callee methods are never taken as samples even if they are highly desirable inlining candidates. The purely dynamic inlining oracle rejects all these active calls to the small methods whose yield points are omitted. On the other hand, static heuristics would accept them based on the small code size.

To handle these two pathological cases and exploit the high-accuracy profiles, we revised the default inline oracle by reducing the strength of static heuristics instead of eliminating them completely. The default static inlining algorithm in Jikes RVM computes the cost of inlining a callee method as the number of its estimated inlined machine instructions. This inline logic decides to inline the callee method when its

estimated machine code size is below a certain size threshold. A small size threshold value effectively weakens the static heuristics. In our experimental setting, we reduce the size threshold for nontrivial methods without their frequency values in the dynamic call graph by 30% from 23 to 17, which is above the size threshold for trivial methods (11).

4.3. Discussion

While our revised inline oracle resorts to static heuristics to handle a few pathological cases, we strongly believe that profile-guided inlining oracles should weaken static heuristics as much as possible as low overhead sampling and correction techniques will evolve and mature [Arnold and Grove 2005; Lee et al. 2007; Chen et al. 2010; Wu et al. 2013; Nakaike et al. 2014]. The performance improvement from static inlining of inactive calls to trivial methods would be either marginal or unmeasurable in a register-rich environment (e.g., ×86-64). Asynchronous immediate sampling techniques without yield points [Buytaert et al. 2007; Mytkowicz et al. 2010; Nakaike et al. 2014] would capture the missing samples due to omitted yield points.

5. METHODOLOGY

This section describes our methodology for measuring accuracy, overhead, and performance of our adaptive sampling bias correction techniques compared to timer-based sampling and full instrumentation.

5.1. Experimental Environment

Benchmarks. We present results for the two motivating programs in Figure 1 and Figure 3 as microbenchmarks: `CallDensityBias` and `CallLatencyBias`. For macrobenchmarks, we run the 19 benchmarks from SPECjvm98 [Standard Performance Evaluation Corporation 1999] and the DaCapo 2006-10-MR2 [Blackburn et al. 2006]. We use the largest input size (`-s100`) for SPECjvm98 and the `default` input size for DaCapo 2006-10-MR2.

Jikes RVM. We use the default *production* configuration of Jikes RVM 3.1.3, a high-performance Java virtual machine, which precompiles the Virtual Machine (VM) methods and the libraries at the highest optimization level (O2) into a "boot image" [Alpern et al. 2005], manages Java heap using the default generational Immix garbage collector, and profiles applications using the timer-based sampling for adaptive optimization. Jikes RVM contains two compilers: the *baseline compiler* and the *optimizing compiler*, which has three optimization levels (O0, O1, and O2).

To account for variability due to nondeterministic microarchitectural events and operating system activities, we execute 40 trials for each measurement and take the median of measurement values to statistically tolerate experimental noise.

System Platform. We use an Intel Xeon E5-2665 2.4GHZ running 32-bit Ubuntu 12.04 LTS distribution. The Linux 3.2.0-48 kernel is configured to enable PAE paging to use 16GB of DDR3-1500 memory.

5.2. Evaluating Accuracy and Overhead

Opt0 Profiling Run. To collect dynamic call graphs, we follow the prior approach of turning off the adaptive optimization and optimizing each method at the lowest level (`O0`) at the first invocation [Arnold and Grove 2005]. This configuration reduces nondeterminism in profiling while collecting critical call events. The optimizing compiler at the lowest level inlines the trivial callees whose size is below a certain small threshold.

The instruction stream includes only the call events to the nontrivial callee methods that are subject to profile-guided inlining. In other words, profilers observe the critical call events that affect the decisions in an inlining optimization, which is a main client of dynamic call graphs.

Call Density and Sampling Latency. Our correction technique hypothesizes that variability in call density and sampling latency at timer ticks increases sampling bias and lowers sampling accuracy. Our adaptive correction technique keeps track of call density and sampling latency at each timer tick. To measure variability in call density and sampling latency, we configured the modified Jikes RVM to report a time series of call densities and sampling latencies for each thread at the end of a program run.

Full Instrumentation. Full instrumentation captures all call events and collects a complete and precise dynamic call graph. We modified Jikes RVM to allocate a table of representing dynamic call graphs during the initialization phase, take the yield points at every method prologue, and update the table. We experimentally find out the size of the table large enough for the 19 benchmarks. This preallocation allows the full instrumentation to handle yield points without allocating any objects and respect the `Unpreemptible` invariant of avoiding garbage collections in handling yield points. A coarse-grained lock protects the dynamic call graph table from concurrent accesses. At the end of the run, our modified Jikes RVM dumps the dynamic table as a dynamic call graph. In this full instrumentation, we favored simplicity while there must be room for reducing the full instrumentation overhead.

Accuracy. To measure how our sampling bias correction technique improves the quality of profiles, we follow the prior work [Arnold and Grove 2005] that measures how closely the normalized edge weights of the sampling-based DCG match the perfect DCG from full instrumentation. Given two DCGs g_1 and g_2, let E_1 and E_2 be a set of call edges and w_1 and w_2 be maps from call edges to their weights. The *overlap* measures similarity of the normalized edge weights by summing the minimum of normalized weights over the common call edges. Formally, $overlap(g_1, g_2)$ is defined to be

$$\sum_{e \in E_1 \cap E_2} min\left(\frac{w_1(e)}{\sum_{e_1 \in E_1} w_1(e_1)}, \frac{w_2(e)}{\sum_{e_2 \in E_2} w_2(e_2)}\right) \times 100.$$

The overlap ranges from 0 to 100%. When the two dynamic call graphs share the same normalized edge weights, their overlap value is 100%.

We define the accuracy of dynamic call graphs relative to the dynamic call graph based on full instrument. Given a DCG g from sampling or our correction and a DCG $g_{instrument}$ from full instrument, the accuracy of sampling or correction is the following:

$$accuracy(g) = overlap(g, g_{instrument}).$$

5.3. Evaluating Performance

Replay Performance Run. To measure how dynamic call graphs of different accuracy guide the optimizing compiler to make proper decisions, we use a *warmup replay* methodology. We iterate each benchmark 10 times to reach a well-optimized steady state and collect compilation decisions and edge profiles. In a performance run, we iterate a benchmark once without any optimization to load all classes and resolve all symbolic references. We next apply the optimization decisions with the edge profiles from the previous run and a number of the dynamic call graphs from sampling to

correction to full instrument. We finally measure and report the execution time of the subsequent iteration.

The Optimizing Compilers for Aggressive Inlining. The optimizing compiler in Jikes RVM has three optimization levels: `O0`, `O1`, and `O2`. The `O0` level favors the speed of the optimization over the quality of the generated code by applying local optimizations and inlining trivial callees such that inlining them never increases code size. The `O1` level extends the aggressiveness of inlining decisions by (1) predicting the targets of a call site analyzing static class hierachy and dynamic call graph profiles and (2) guarding the inlined callees for polymorphic call sites. The `O2` level used to include global optimizations on the top of Static Single Assignment (SSA) form, but these SSA global optimizations were disabled due to some implementation bugs.

To reduce unexpected slowdown from weak downstream optimizations after aggressive inlining, we enabled global SSA-based optimizations when we can run a benchmark successfully with these optimizations. For the `O1` level, we enabled SSA conversions and redundant branch elimination to improve dataflow analysis with inlining guards. The set of enabled optimizations at the `O2` level includes SSA conversions, redundant branch elimination, global common subexpression elimination, global expression folding, and coalescing move instructions. We could not apply load elimination and global code placement due to the failures in running some benchmarks.

While measuring and analyzing the performance with high-accuracy dynamic call graphs in Jikes RVM, we fixed a few negative interactions between aggressive inlining and linear scan register allocation due to immature preassignment of the live variables to physical registers at Potentially Excepting Instructions (PEIs). Inlining a callee method containing PEIs into the context of the caller containing exception handler increases the number of register preallocations and guides the register allocator to make poor decisions. Such negative interactions in Jikes RVM are not surprising because the optimizing compiler was engineered for the register rich PowerPC architecture for an extended period of time from the beginning [Arnold et al. 2000]. We eliminated this register preallocation by placing additional logic into the exception dispatcher for optimized methods.

Garbage Collection. To reduce the overhead of tracing a large amount of code and metadata objects, we increase the size of boundary nursery from 32MB to 128MB to reduce the chance of additional minor collections. In addition, we prevent the full heap garbage collection from being triggered at a user program's request through calling the `System.gc`. In the second iteration after the warmup replay run, we force a full heap collection to clean up the dead objects before the performance run.

6. RESULTS

This section evaluates the accuracy, overhead, and performance effects of our sampling correction approaches.

6.1. Sampling Bias

Our sampling bias correction hypothesizes that variability in call density and sampling latency reduces sampling fairness. Our measurement reveals that both call density and sampling latency vary their values at each timer tick.

6.1.1. Call Density. Figure 4 plots call density values over timer ticks in the `main` thread from the 14 single-threaded benchmarks running on Jikes RVM. Each benchmark exhibits its own phase behavior of varying call density over timer ticks. Call density ranges from 0.00001 to 0.1 over all benchmarks, and this amount of variation would lead a timer-based sampling to overcount or undercount some samples up to a factor

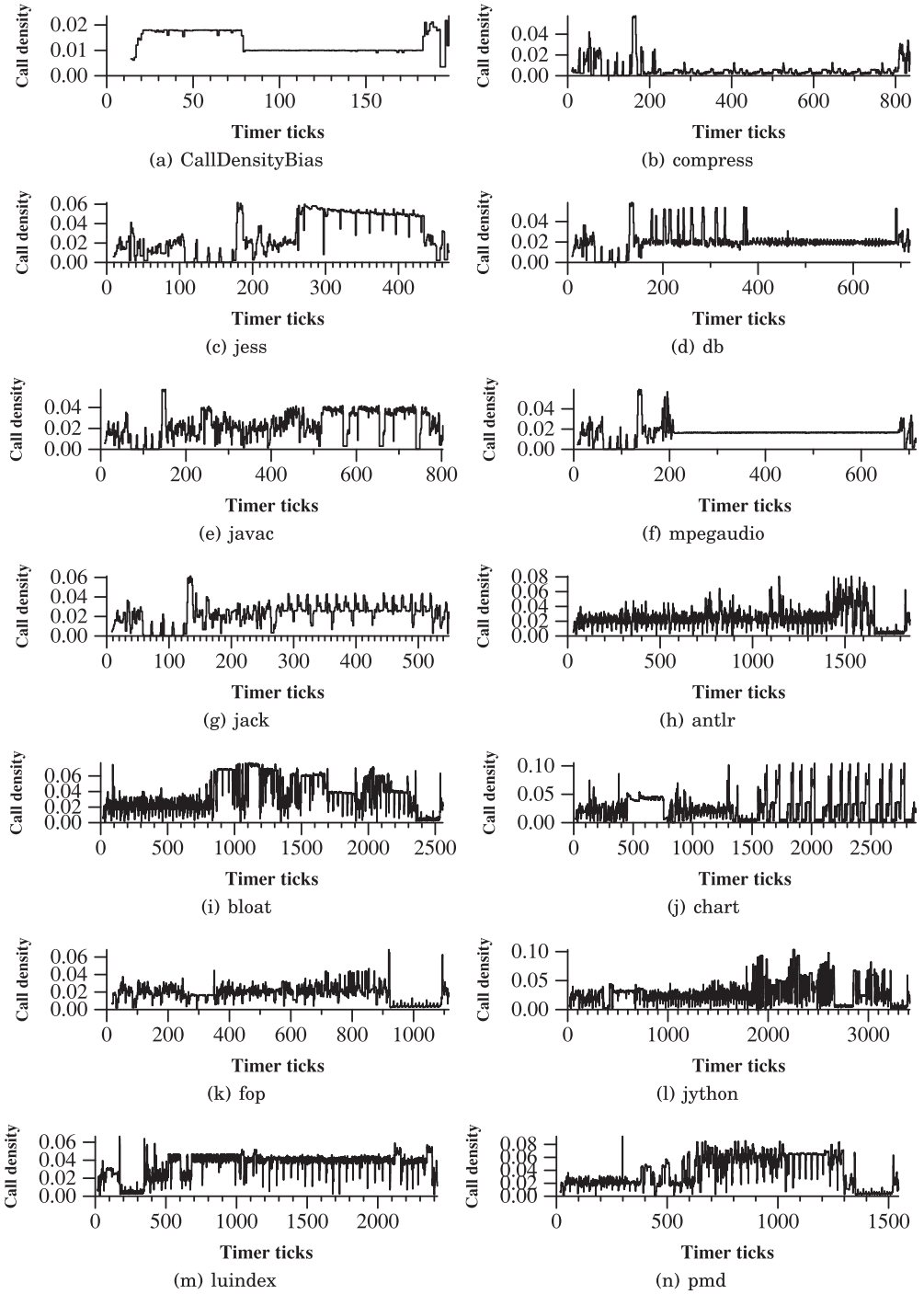

Fig. 4. Time-varying call density in the single-threaded benchmarks.

of 10,000. Figure 5 presents call density values in the six multithreaded benchmarks. We present the results in the `main` thread and one other thread for each benchmark. The `main` threads exhibit a sleeping period where no call density points are present in the plots. These main threads fork a number of worker threads for subtasks and wait for the completion of these tasks. When a thread is in a block state, the thread cannot take any yield point and compute the current call density.

6.1.2. Sampling Latency. Table I presents statistics on sampling latency cycles over timer ticks from running the 20 benchmarks. Sampling latency ranges from a few hundreds (`CallDensityBias`) to dozens of billions (`eclipse`). The averages are a few orders of magnitude larger than the medians in general, except for `lusearch`. This large gap is driven by the long sampling latency cycles that are close to the maximum sampling latency cycles ranging from dozens of millions to dozens of billions. Occasional stop-the-world generational garbage collections would pause the application threads for a few millions of CPU cycles.

Figure 6 plots the violin graphs to visualize distribution of sampling latency cycles for each benchmark. A large fraction of sampling latency cycles are concentrated around the average cycles except for the `LatencyBias` and `lusearch` benchmarks. Both benchmarks run multiple threads contending for a lock. Multiple worker threads in `LatencyBias` contends for the `lock` lock in Figure 3. The 32 worker threads in `lusearch` repeatedly execute the `intern` method of Java string objects that end up with searching for a cached string in a global hash table and contending for the lock of protecting the table. In the lock contention, the lock is promoted to be a fat lock, and a thread goes into a blocked state when it fails to acquire the lock. Given the 32 threads of spending more than dozens of thousands of cycles in the critical section, the average latency is a few hundreds of thousands of cycles. This long sampling latency in `lusearch` is consistent with the long Garbage Collection (GC) latency `lusearch` from the work of analyzing yield point behaviors [Lin et al. 2015].

6.1.3. Summary. Call density varies significantly at each timer tick in all threads of the 21 benchmarks, while sampling latency varies little in general except for the two multithreaded benchmarks contending for a lock. These results suggest that call-density-based correction will contribute significantly to all benchmarks, while the latency-based correction will improve sampling accuracy for `LatencyBias` and `lusearch`.

6.2. Sampling Bias Correction

The direct benefit from sampling bias correction is high-accuracy dynamic call graphs with small overhead. In our measurement, our correction technique improves accuracy significantly without any measurable overhead.

Accuracy. Figure 7 shows the accuracy of sampling, adaptive sampling bias correction with call density, and adaptive sampling bias correction with call density and sampling latency. The sampling uses a default counter-based sampling of taking eight samples per timer tick in two strides, and it achieves overlap accuracy of 30%–80%, averaging 61%. The call-density-based correction improves the overlap accuracy to 71% on average consistently over all benchmarks. While sampling latency-based correction increases the average accuracy by 1 % up to 72%, it increases the accuracy of `lusearch` from 38% to 67%. `lusearch` creates 32 worker threads that contend for a lock while running the `intern` method of Java string objects that end up with searching for a cached string in a global hash table. Sampling latency component eliminates the sampling bias due to this lock contention. All correction techniques never decrease the overlap accuracy for any benchmark. These results show that our correction

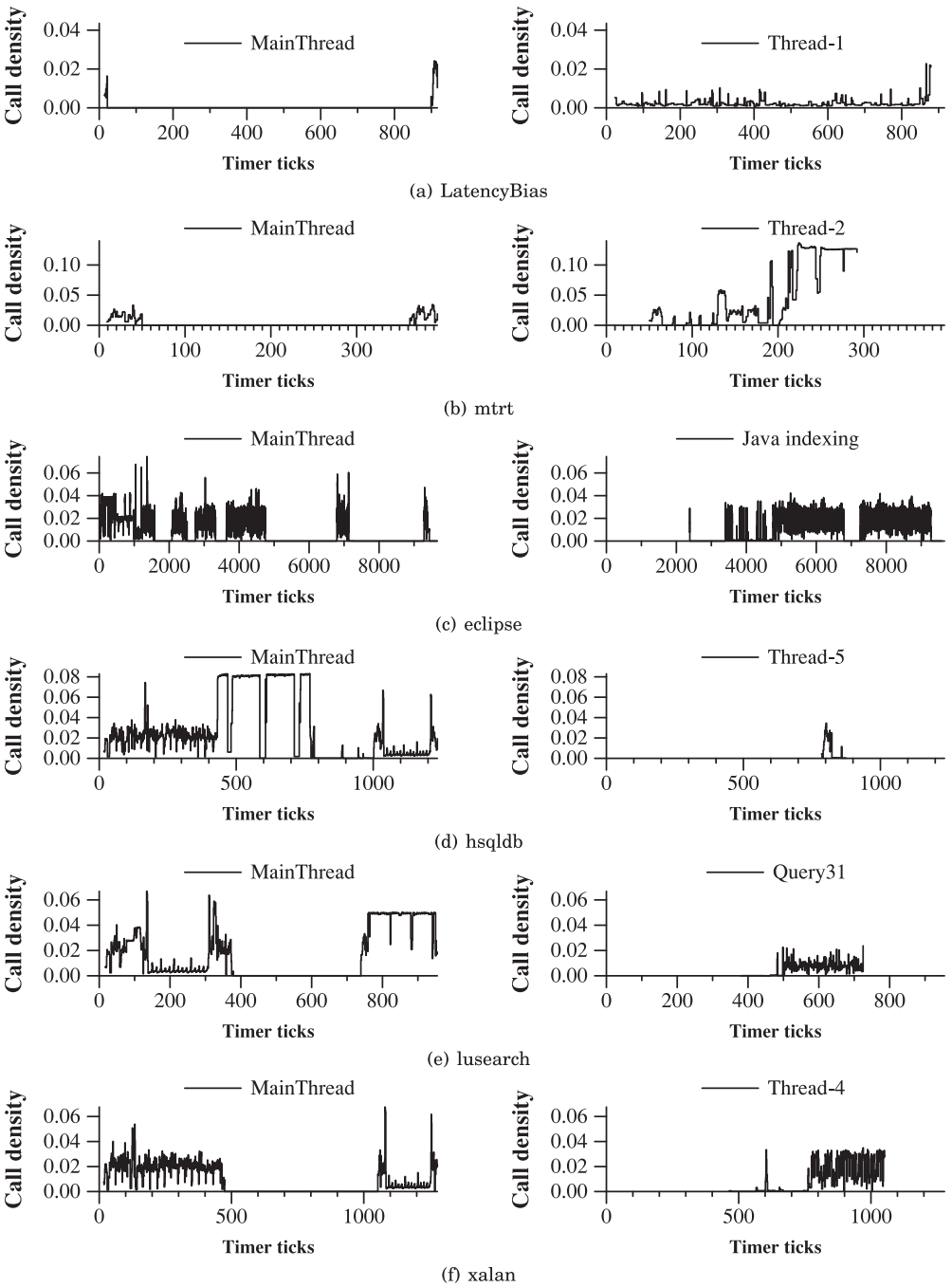

Fig. 5. Time-varying call density in the multithreaded benchmarks. `raytrace` is omitted due to it similiarity to mtrt.

Table I. Statistics on Sampling Latency Cycles

Benchmarks	Min.	Max.	Median	Average
CallDensityBias	567	26,328,513	693	146,115
LatencyBias	615	27,842,090	7,840	587,981
compress	840	36,064,212	1,107	147,325
jess	879	39,253,358	2,457	247,939
raytrace	771	38,325,786	1,620	298,017
javac	819	85,017,695	3,078	572,759
mpegaudio	612	47,015,096	1,317	141,262
mtrt	765	32,670,532	1,842	315,140
jack	861	36,776,840	2,727	281,644
antlr	807	52,337,041	3,048	363,452
bloat	798	51,491,470	2,781	266,009
chart	747	78,114,270	1,920	557,791
eclipse	570	22,082,664,980	3,168	3,224,272
fop	771	41,918,081	3,270	311,546
hsqldb	753	201,737,433	2,979	865,378
jython	699	33,083,743	2,715	200,251
luindex	753	46,589,800	2,643	133,585
lusearch	552	52,943,910	496,305	2,077,687
pmd	792	65,042,556	2,520	326,665
xalan	792	45,662,589	3,150	374,958

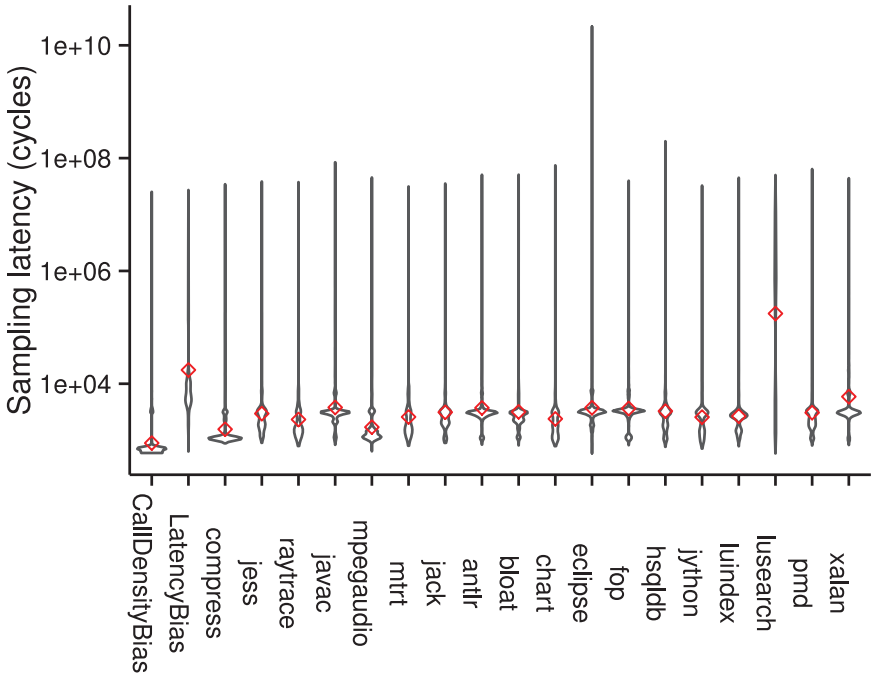

Fig. 6. Sampling latency measured in cycles (log scale y axis).

techniques improve the profiling accuracy over the sampled configurations significantly and rigorously.

Time Overhead. Figure 8 presents the time overhead. We do not observe any measurable time overhead on average. This is not surprising because our correction technique

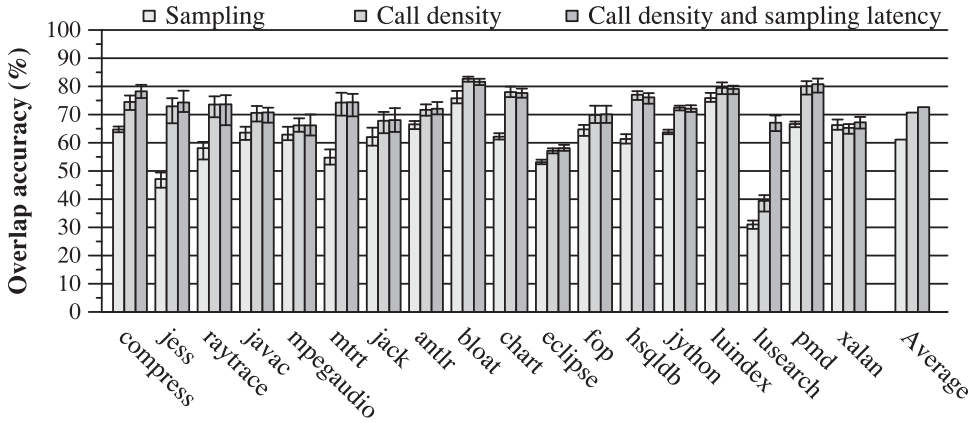

Fig. 7. Overlap accuracy of sampling and its adaptive correction.

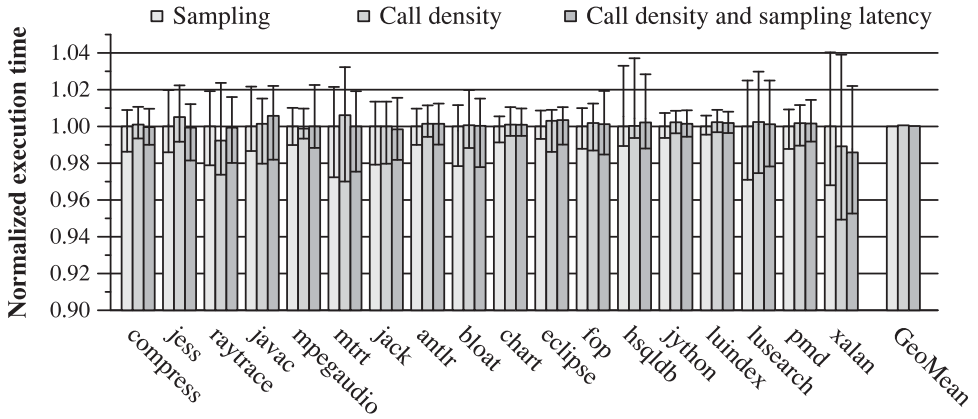

Fig. 8. Time overhead of sampling and its adaptive correction.

adds very little overhead to the runtime by design. At the beginning and ending of each application thread, our correction technique executes only dozens of additional instructions of running two system calls to turn on and off the hardware performance counter of counting retired call instructions. At each timer tick, our correction technique adds hundreds of machine instructions of computing the antibias weight of a sample based on the current call density and sampling latency. The instructions at timer ticks do not add much time overhead as long as timer ticks are designed to happen rarely (e.g., at every tens of millions of instructions).

Space Overhead. The space overhead is arguably small by design. Our technique consumes both thread local and global space. For each thread, our technique adds a few bytes of recording the call counts and CPU cycles. For each entry in the shared buffer of caller and callee pair to keep the incremental updates to the dynamic call graph, our technique adds one integer value containing the antibias weights associated with the caller-callee pairs. The size of the shared buffer is configurable, and the default size is 160. In short, our technique adds a few kilobytes space overhead, which is insignificant

in practice compared to dozens of megabytes or more for the metadata and optimizing compilers in Jikes RVM.

Trade-Off Between Accuracy and Overhead. Our adaptive correction technique is complementary to counter-based sampling in that it achieves a point beyond the accuracy-overhead trade-off space of counter-based sampling only approach. Counter-based sampling increases profiling accuracy at the cost of time overhead as it increases the number of samples per timer tick (samples) and the number of skipped events between two samples (stride). We denote `CBS(samples, stride)` as a counter-based sampling configuration. High values in these two parameters increases both accuracy and overhead. For instance, in our experiment, the canonical configuration `CBS(1,1)` of disabling bursty sampling results in 48% overlap accuracy, while our baseline configuration `CBS(8,2)` increases the accuracy to 61% at the cost of 0.2% time overhead. The `CBS(8,2)` configuration with our adaptive correction achieves 72% accuracy at the 0.2% time overhead with respect to the `CBS(1,1)` configuration. An aggressive configuration `CBS(32,3)` without correction increases the accuracy to 66% at the cost of 1% time overhead. To get close to the 72% accuracy, a highly aggressive configuration `CBS(256,1)` without correction increases the accuracy to 68% at the cost of 2.5% time overhead. Given the monotonically increasing overhead of counter-based sampling, none of the configuration values would subsume our adaptive correction technique. When we keep the time overhead below 0.2%, the default `CBS(8,2)` configuration with our adaptive correction produces the highest accuracy.

Discussion. While our adaptive technique requires modest implementation effort with almost zero overhead, our accuracy results show room for further improvement. In general, the approximated dynamic call graphs from sampling-based techniques have two classes of errors: (1) inaccurate weight values of call edges due to sampling bias and (2) missing call edges due to low sampling rate. Our adaptive technique reduces the sampling bias well, but it does not address the problem of missing call edges.

6.3. Performance

We evaluate the indirect benefits of using our adaptive DCG correction techniques to drive one client, inlining. We first present the performance impact of our adaptive correction on the inlining decisions under the default inline oracle in Jikes RVM. We then evaluate the performance results after reducing the aggressiveness of static heuristics in the default inline oracle to exploit high-accuracy dynamic call graphs.

6.3.1. Inlining with the Default Inline Oracle. We use the default inlining policy with dynamic call graph obtained from sampling (*Sampling*) as the baseline. Figure 9 shows the relative performance of eight SPECjvm98 benchmarks and 11 DaCapo 2006 benchmarks. The correction technique of using call density (*Call density*) shows the improvement of 0.5% on average, and the correction using both call density and sampling latency (*Call density and sampling latency*) improves the performance up to 1% on average. While this modest improvement is consistent with the prior work [Arnold and Grove 2005; Lee et al. 2007], some relatively large and complex benchmarks show potential of using high-accuracy dynamic call graphs. For instance, our corrected dynamic call graphs improve the performance of `bloat`, `jython`, `jack`, `lusearch`, and `xalan` by 6.1%, 1.8%, 1.5%, 1.5%, and 1.5%. On the other hand, our corrected dynamic call graphs slow down `mpegaudio` and `pmd` by 0.8% and 0.4%.

To investigate how high-accuracy dynamic call graphs guide the inlining optimization to speed up or slow down the execution time, we collect relevant hardware performance counts: (1) dynamic call instructions, (2) dynamic instructions, and (3) instruction cache

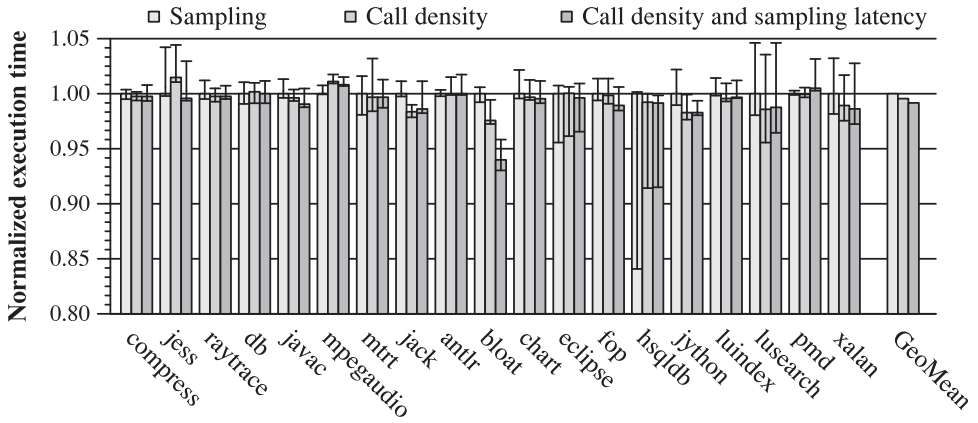

Fig. 9. Performance of inlining decisions using high-accuracy dynamic call graphs. The data is normalized relative to the run with the dynamic call graph from sampling without any correction.

misses. Figure 10 presents these sets of hardware performance counters relative to the baseline run after inlining using the dynamic call graphs from sampling. The corrected dynamic call graphs guide the optimizing compiler to inline methods aggressively and reduce the dynamic call instructions by 4% and 7%. This reduction in the call events is translated into a 1% reduction in dynamic instructions. The aggressive inlining replicates the inlined callees in several places and increases the instruction footprints. This increase in the working set creates additional instruction cache misses in general. The average increase in instruction cache misses is zero. The 32% outstanding increase in `jack` was offset by a 3% decrease in the dynamic instructions, which sped up the bottom-line performance by 1.5%.

6.3.2. Inlining with the Weak Static Heuristics. To exploit high-accuracy dynamic call graphs from our adaptive correction, we evaluate the inline performance after reducing the aggressiveness of static heuristics in the default inline oracle with and without our adaptive correction. Figure 11 presents the relative performance and compiled code size when the baseline configuration is the counter-based sampling with the default inline oracle. Weakening static heuristics does not change the execution time but saves the compiled code size by 7% on average. This result suggests that static heuristics inefficiently inline some inactive methods that do not contribute to the execution time. The high-accuracy profiles from adaptive correction lead the default inline oracle to inline aggressively frequently executed large callee methods and improve the execution time up to 1% on average at the cost of an 8% average increase in the compiled code size. This 8% increase in the compiled code size is reduced to 1% on average after turning on the modified inline oracle of reducing aggressiveness of static heuristics. In particular, `bloat` shows quite promising results of improving both cost and benefit in the inlining decisions. In short, the results suggest that high-accuracy call graph profiling techniques should evolve together with inlining heuristics for efficient inlining decisions.

7. RELATED WORK

This section first discusses profiling and reconciliation technique for dynamic call graphs. It then compares our sampling bias correction approach to other correction approaches. It finally discusses the evolutionary process of revising inline oracles to exploit high-accuracy profiles.

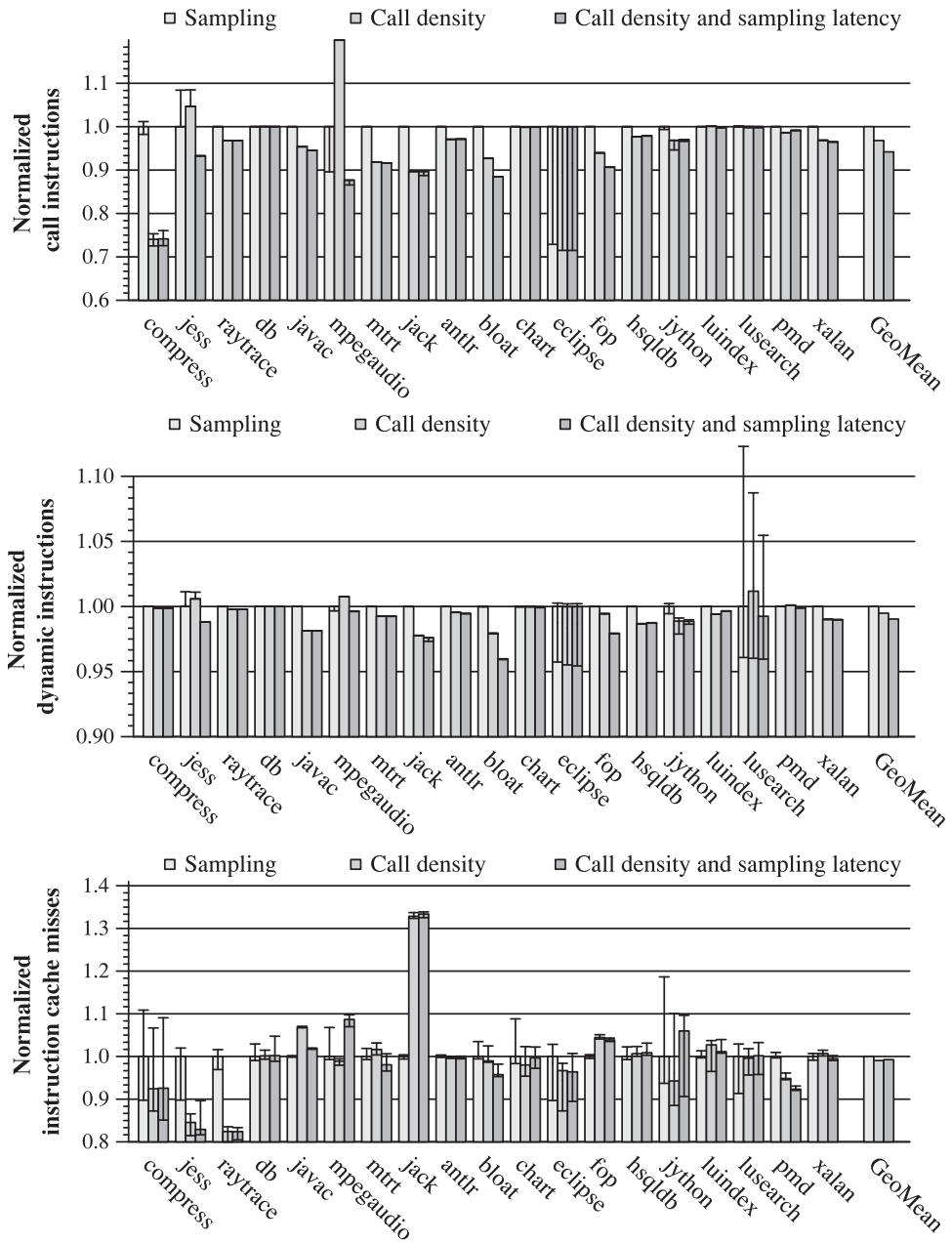

Fig. 10. Hardware performance counter statistics from inlining optimization using high-accuracy dynamic call graphs. Hardware performance counts relative to the run with the dynamic call graph from the default timer-based sampling.

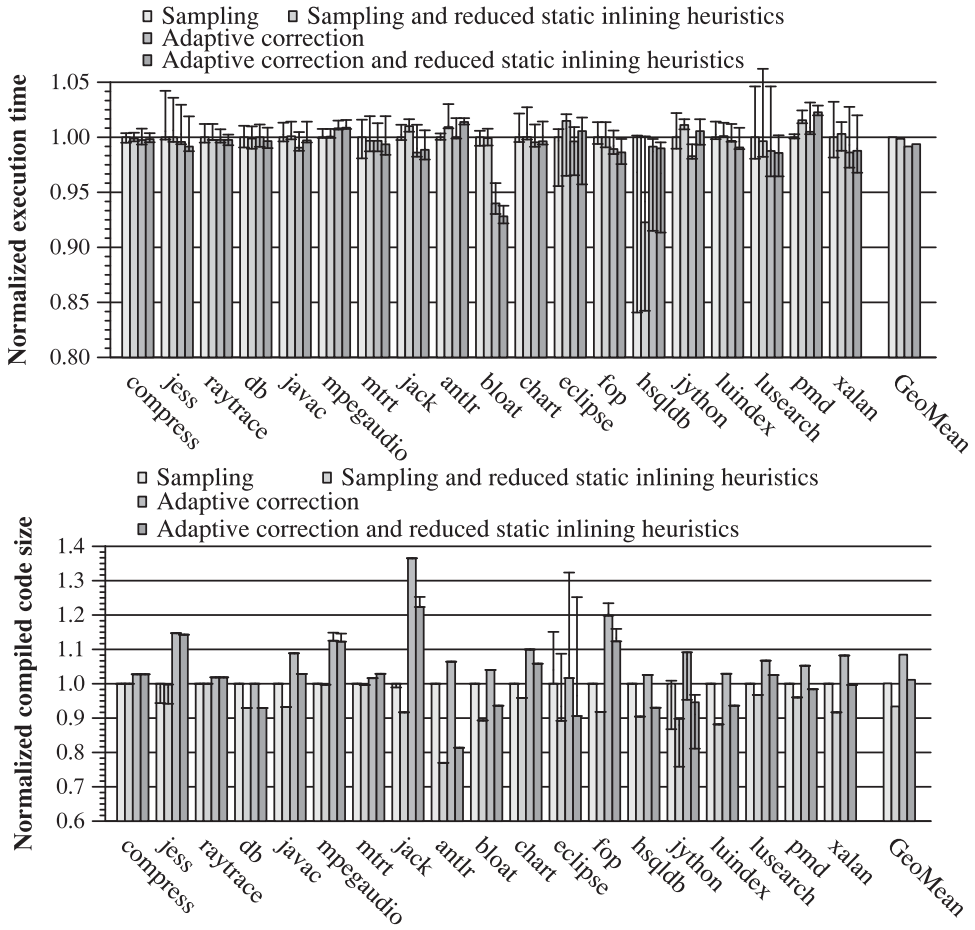

Fig. 11. Performance and code size of inlining decisions with and without adaptive correction and static inlining heuristics. The data are normalized relative to the run with dynamic call graph sampling without any correction. Adaptive correction increases the accuracy of dynamic call graphs, and by reducing static inlining heuristics the inlining decision relies more on dynamic call graphs.

7.1. Collecting Dynamic Call Graphs

`gprof` [Graham et al. 1982] pioneered the idea of profiling dynamic call graphs to summarize and analyze performance bottlenecks due to many small routines in large complex programs. In the context of dynamic optimization [Paleczny et al. 2001; Suganuma et al. 2001], the profiling overhead is significant, and software-based sampling approaches reduce the overhead dramatically by taking a tiny fraction of the call events at the yield points in a program run [Arnold and Sweeney 2000; Arnold et al. 2000] or turning on and off profiling for some period [Paleczny et al. 2001; Suganuma et al. 2001; Zhuang et al. 2006]. Increasing the number of samples improves the profiling accuracy [Arnold and Grove 2005], but the overhead increases significantly for higher accuracy at some point. In addition, this software-based sampling does not sample some of the call events in the blind spots where yield points are omitted to reduce their runtime overhead [Arnold et al. 2004; Lin et al. 2015]. Hardware-based immediate sampling techniques collect events samples without using yield points and increase the coverage of the profiled code including the blind spot [Buytaert et al.

2007; Nakaike et al. 2014]. Our adaptive approach corrects the sampling bias in these sampling techniques. Our experimental results demonstrate the effectiveness of our antibias techniques on software-based sampling techniques, and we believe that our techniques would also work well for the hardware-based approaches as long as their sampling actions are triggered by wall clock timers or CPU cycle counters.

7.2. Reconciling Sampling Errors with Additional Information

Lee et al. [2007] discover static and dynamic control-flow constraints and reconcile the sampling errors in the dynamic call graph with these constraints. Chen et al. [2010] apply supervised learning techniques to hardware-based sampling to improve the accuracy of basic-block-level sampled profiles. Wu et al. [2013] identify classes of sampling errors in basic block profiles, study how these errors influence feedback-driven program optimizations, and suggest a few rectification approaches using dynamic profiles in training runs. While our adaptive sampling bias correction technique shares the idea of reconciling sampling errors by taking additional information into account, our adaptive technique corrects the sampling error early at the point of taking samples. In addition, our approach does not require a relatively sophisticated analysis.

7.3. Reducing Sampling Bias with Randomization

Arnold and Grove [2005] identifies sampling bias to the mismatch between the timer-based sampling mechanism and frequency-oriented profiles and illustrates. To reduce the impact of oversampling the call events that are systematically aligned with timer ticks, counter-based sampling [Arnold and Grove 2005] skips a number of call events immediately after each timer tick. Autocorrelation analysis reveals that yield-point-based sampling could systematically bias some events that are aligned with some program or system activity [Mytkowicz et al. 2010]. These randomization approaches are different from our adaptive correction in that our approach monitors the causes of systematic bias due to unequal spacing of events and sampling actions and mitigates sampling bias. In addition, our analysis classifies and quantifies the two classes of sampling bias using the call density and sampling latency metric.

7.4. Exploiting High-Accuracy Profiles

In the context of dynamic optimization, sampling techniques and profile-guided inline oracles have coevolved together to exploit increasingly accurate profiles. The early inline oracle of Jikes RVM [Arnold et al. 2000] relies totally on the static inline heuristics of equally treating high- and low-frequency values as long as they are above a certain threshold (e.g., 1%). The high-accuracy sampling techniques encourage inline oracle designers to add dynamic inline heuristics or remove static inline heuristics [Arnold and Grove 2005; Nakaike et al. 2014]. As our adaptive correction increases the sampling accuracy, our work also reduces static inline heuristics for efficient inline decisions.

8. CONCLUSION

This article presents a practical technique for increasing the accuracy of dynamic call graphs from sampling by correcting the sampling bias based on call density and sampling latency at the point of taking samples. To keep track of call density and sampling latency, our technique uses hardware performance monitors and CPU time stamp counters that are available on modern processors. We implemented our adaptive sampling bias correction in Jikes RVM and report significant improvement in the overlap accuracy from 61% to 72% without any measurable overhead. We carefully configured the optimizing compilers in Jikes RVM to take advantage of profile-guided aggressive inlining using a high-accuracy call graph. While the average performance improvement is modest, we observe a measurable speedup in some large benchmarks

including `jack` and `bloat` in SpecJVM98 and DaCapo-2006-MR2. The average increase in the compiled code size is 1% after we reduce the aggressiveness of static heuristics in the default oracle in Jikes RVM. In short, our adaptive approach of correcting the sampling bias in dynamic call graphs effectively improves profiling accuracy and feedback directed optimization.

ACKNOWLEDGMENTS

We thank the anonymous referees for the feedback and suggestions that helped us to improve the presentation of this work. We also thank Michael D. Bond and Kathryn S. McKinley for their comments and suggestions.

REFERENCES

Bowen Alpern, Steven Augart, Stephen M. Blackburn, Maria Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark Mergen, J. Eliot B. Moss, Ton Ngo, and Vivek Sarkar. 2005. The Jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal* 44, 2 (Jan. 2005), 399–417.

Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2000. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*. ACM, New York, NY, 47–65.

Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2004. *Architecture and Policy for Adaptive Optimization in Virtual Machines*. Technical Report RC23429. IBM Research, Yorktown Heights, NY.

Matthew Arnold and David Grove. 2005. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*. IEEE Computer Society, Washington, DC, 51–62.

Matthew Arnold and Peter F. Sweeney. 2000. *Approximating the Calling Context Tree via Sampling*. Technical Report RC 21789. IBM T.J. Watson Research Center.

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*. ACM, New York, NY, 169–190.

Dries Buytaert, Andy Georges, Michael Hind, Matthew Arnold, Lieven Eeckhout, and Koen De Bosschere. 2007. Using HPM-sampling to drive dynamic compilation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA'07)*. ACM, New York, NY, 553–568.

Pohua P. Chang and Wen-Mei W. Hwu. 1989. Inline function expansion for compiling C programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI'89)*. ACM, New York, NY, 246–257.

Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. 2010. Taming hardware event samples for FDO compilation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*. ACM, New York, NY, 42–52.

Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (SIGPLAN'82)*. ACM, New York, NY, 120–126.

Nikola Grcevski, Allan Kielstra, Kevin Stoodley, Mark Stoodley, and Vijay Sundaresan. 2004. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium—Volume 3 (VM'04)*. USENIX Association, Berkeley, CA, 12–12.

Byeongcheol Lee, Kevin Resnick, Michael D. Bond, and Kathryn S. McKinley. 2007. Correcting the dynamic call graph using control-flow constraints. In *Proceedings of the 16th International Conference on Compiler Construction (CC'07)*. Springer-Verlag, Berlin, 80–95.

Yi Lin, Kunshan Wang, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 2015. Stop and go: Understanding yieldpoint behavior. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management (ISMM'15)*. ACM, New York, NY, 70–80.

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the accuracy of Java profilers. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, New York, NY, 187–197.

Takuya Nakaike, Hiroshi Inoue, Toshio Suganuma, and Moriyoshi Ohara. 2014. Characterization of call-graph profiles in Java workloads. In *Proceedings of the 2014 IEEE International Symposium on Workload Characterization (IISWC'14)*. 161–170.

Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java hotspot server compiler. In *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium—Volume 1 (JVM'01)*. USENIX Association, Berkeley, CA, 1–12.

Chung-Lung Shum, Fadi Busaba, and Christian Jacobi. 2013. IBM zEC12: The third-generation high-frequency mainframe microprocessor. *IEEE Micro* 33, 2 (March 2013), 38–47.

Standard Performance Evaluation Corporation 1999. *SPECjvm98 Documentation* (release 1.03 ed.). Standard Performance Evaluation Corporation. Retrieved from http://www.spec.org/osg/jvm98/.

Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. 2001. A dynamic optimization framework for a Java just-in-time compiler. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*. ACM, New York, NY, 180–195.

Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. 2002. An empirical study of method in-lining for a Java just-in-time compiler. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*. USENIX Association, Berkeley, CA, 91–104.

Bo Wu, Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, Raul Silvera, and Graham Yiu. 2013. Simple profile rectifications go a long way. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13)*. Springer-Verlag, Berlin, 654–678.

Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. 2006. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*. ACM, New York, NY, 263–271.

Received June 2015; revised October 2015; accepted October 2015