Computer Performance Microscopy with SHIM

Xi Yang[‡] Stephen M. Blackburn[‡] Kathryn S. McKinley[*]
[‡]Australian National University [*]Microsoft Research
{xi.yang, steve.blackburn}@anu.edu.au mckinley@microsoft.com

Abstract

Developers and architects spend a lot of time trying to understand and eliminate performance problems. Unfortunately, the root causes of many problems occur at a fine granularity that existing continuous profiling and direct measurement approaches cannot observe. This paper presents the design and implementation of SHIM, *a continuous profiler that samples at resolutions as fine as 15 cycles; three to five orders of magnitude finer than current continuous profilers.* SHIM*'s fine-grain measurements reveal new behaviors, such as variations in instructions per cycle (IPC) within the execution of a single function. A* SHIM *observer thread executes and samples autonomously on unutilized hardware. To sample, it reads hardware performance counters and memory locations that store software state.* SHIM *improves its accuracy by automatically detecting and discarding samples affected by measurement skew. We measure* SHIM*'s observer effects and show how to analyze them. When on a separate core,* SHIM *can continuously observe one software signal with a 2% overhead at a ~1200 cycle resolution. At an overhead of 61%,* SHIM *samples one software signal on the same core with SMT at a ~15 cycle resolution. Modest hardware changes could significantly reduce overheads and add greater analytical capability to* SHIM. *We vary prefetching and DVFS policies in case studies that show the diagnostic power of fine-grain IPC and memory bandwidth results. By repurposing existing hardware, we deliver a practical tool for fine-grain performance microscopy for developers and architects.*

1. Introduction

Understanding the complex interactions of software and hardware remains a daunting challenge. Developers currently use two main approaches: direct measurement and sample-based continuous profiling. They pose hypotheses and configure these tools to instrument software events and read hardware performance counters. Next they attempt to understand and improve programs by correlating code with performance events, e.g., code to bandwidth consumption, low Instructions Per Cycle (IPC) to branch behavior, and loops to data cache misses. State-of-the-art continuous profiling tools, such as Intel VTune and Linux perf, take an interrupt and then sample hardware performance counter events [16, 18]. The possibility of overwhelming the kernel's capacity to service interrupts places practical limits on their maximum resolution [19]. Consequently, their default sample rate is 1 KHz and their maximum sample rate is 100 KHz, giving profile resolutions of around 30 K to 3 M cycles on a modern core.

Unfortunately, sampling at a period of 30 K cycles misses high frequency events. Statistical analysis sometimes mitigates this problem, for example, when a single small portion of the code dominates performance. However, even for simple rate-based measures such as IPC, infrequent samples are inadequate because they report the mean of the period, obscuring meaningful fine-grain variations such as those due to small but ubiquitous code fragments, as we show in Section 2.

Consequently, developers must currently resort to microbenchmarks, simulation, or direct measurement to examine these effects. However, microbenchmarks miss interactions with the application context. Simulation is costly and hard to make accurate with respect to real hardware. Industry builds proprietary hardware to examine performance events at fine granularities at great expense and thus such results are scarce.

The alternative approach directly measures code by adding instrumentation automatically or by hand [12, 13, 18, 34]. For instance, developers may insert instrumentation that directly reads hardware performance counters. Software profilers such as PiPA and CAB instrument code automatically to record events such as path profiles. A consuming profiler thread analyzes the buffer offline or online, sampling or reading it exhaustively. In principal, developers may perform direct measurement and fine-grain analysis with these tools [2]. However, inserting code and the ~30 cycles it takes to read a single hardware performance counter both induce observer effects. Observer effects are inherent to code instrumentation and are a function of the number of measurements—the finer the granularity and the greater the coverage, the more observer effect. In summary, no current solution delivers accurate continuous profiling of hardware and software events with low observer effects at resolutions of 10s, 100s, or even 1000s of cycles.

This paper introduces a new high resolution approach to performance analysis and implements it in a tool called SHIM.[1] SHIM efficiently observes events in a separate thread by ex-

ISCA'15, June 13-17, 2015, Portland, OR USA
ACM 978-1-4503-3402-0/15/06.
http://dx.doi.org/10.1145/2749469.2750401

[1]A shim is a small piece of material that fills a space between two things to support, level, or adjust them.

ploiting unutilized hardware on a different core or on the same core using Simultaneous Multithreading (SMT) hardware. A SHIM observer thread executes simultaneously with the application thread it observes, but in a separate hardware context.

Signals We view time-varying software and hardware events as *signals*. A SHIM observer thread reads hardware performance counters and memory locations that store software signals (e.g., method and loop identifiers). A compiler or other tool configures software signals and communicates memory locations to SHIM. SHIM treats software and hardware data uniformly by reading (sampling) memory locations and performance counters together at very high frequencies, e.g., 10s to 1000s of cycles. The observer thread logs and aggregates signals. Further online or offline analysis acts on this data.

Measurement fidelity and observer effects The fidelity of continuous sampling is subject to at least three threats: (i) skew in measurement of rate metrics, (ii) observer effects, and (iii) low sample rates. SHIM reduces these effects.

To improve the fidelity of rate metrics, we introduce *double-time error correction* (DTE), which automatically identifies and discards noisy samples by taking redundant timing measurements. DTE separately measures the period between the start of two consecutive samples and the period between the end of the samples. If the periods differ, the measurement was perturbed and DTE discards it. By only using timing consistency, DTE correctly discards samples of rate measurements that seem obviously wrong (e.g., IPC values of > 10), without explicitly testing the sample itself.

A minimal SHIM configuration that only reads hardware performance counters or software signals inherent to the code does not instrument the application, so has no direct observer effect. SHIM induces secondary effects by contending for hardware resources with the application. This effect is largest when SHIM shares a single core with the application using Simultaneous Multithreading (SMT) and contends for instruction decoding resources, local caches, etc. We find that the SHIM observer thread offers a constant load, which does not obscure application behavior and makes it possible to reason about SHIM's effect on application measurements. When SHIM executes on a separate core, it interferes much less, but it still induces effects, such as cache coherence traffic when it reads a memory location in the application's local cache. This effect is a function of sampling and memory mutation rates. SHIM does not address observer effects due to invasive instrumentation, such as path profiles.

Randomization of sample periods is essential to avoiding bias [3, 21]. We show high frequency samples are subject to many perturbations and their sample periods vary widely.

We measure a variety of configurations and show that SHIM delivers continuous profiles with a rich level of detail at very high sample rates. On one hand, SHIM delivers ~15 cycle resolution profiling of one software signal in memory on the same core with SMT at a 61% overhead. Placing these results in context, popular Java profilers, which add instrumentation to applications, incur typical overheads from 10% to 200% at 100 Hz [21], with sample periods six orders of magnitude longer than SHIM. Because SHIM offers a constant load on SMT, SHIM observes application signals with reasonable accuracy despite its overhead. To fully validate SHIM's fine-grain accuracy would require ground truth from proprietary hardware-specific manufacturer tools, not available to us.

On a separate core, SHIM delivers ~1200 cycle resolution when profiling the same software signal with just 2% overhead.

Case Studies The possibility of high fidelity, high frequency continuous profiling invites hardware innovations such as ultra low latency control of dynamic voltage and frequency scaling (DVFS) and prefetching to tune policies to fine-grained program phases. We analyze the ILP and bandwidth effects of DVFS and turning off and on prefetching on two examples of performance-sensitive code, showing that fine-grain policies have the potential to improve efficiency.

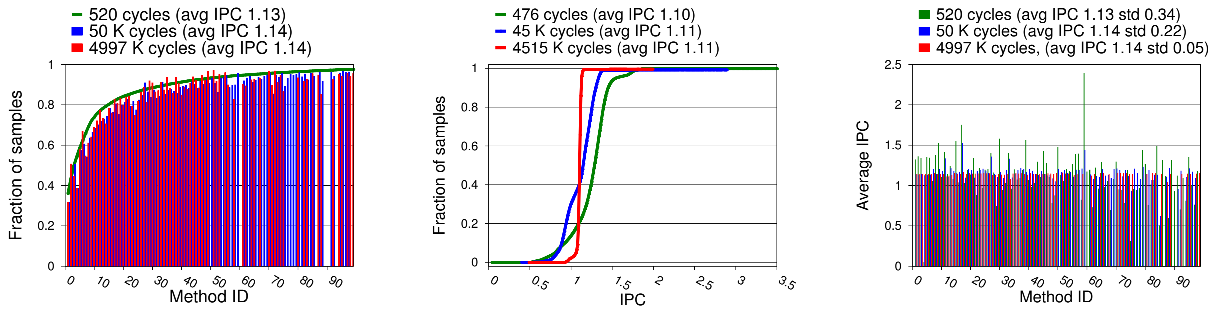
Hardware to improve accuracy and capabilities Modest hardware changes could significantly reduce SHIM's observer effects, improve its capabilities, and make it a powerful tool for architects as well as developers. When SHIM shares SMT execution resources, a thread priority mechanism, such as in MIPS and IBM's Power series [9, 26], would reduce its observer effect. Executing SHIM with low priority could limit the SHIM thread to a single issue slot and only issue instructions from it when the application thread has no ready instruction. When SHIM executes on a separate core, a no-caching read, such as on ARM hardware [29], would reduce observer effects when sampling memory locations. A no caching read simply transfers the value without invoking the cache coherence protocol. On a heterogeneous architecture, the simplicity of the SHIM observer thread is highly amenable to a small, low power core which would reduce its impact. If hardware were to expose all performance counters to other cores, such as in IBM's Blue Gene/Q systems [5], SHIM could capture all events while executing on a separate core. We show that SHIM in this configuration would incur essentially no overhead and experience very few observer effects.

By repurposing existing hardware, SHIM reports for the first time fine-grain continuous sampling of hardware and software events for performance microscopy. We leave to future work how to translate fine-grain profiling into concrete software and hardware improvements, and make SHIM publicly available [32] to encourage this line of research and development.

2. Motivation

This section motivates fine-grain profiling by comparing coarse-grain and fine-grain sampling for hot methods, instructions per cycle (IPC), and IPC for hot methods.

Identifying hotspots Figure 1(a) lists the 100 most frequently executed (hot) methods of the Java application lusearch for three sampling rates, ranging from ∼500 cycles to ∼50 K cycles to ∼5 M cycles. The medium and low frequency data are subsamples of the high frequency data. These results



(a) Hot methods (b) Instructions Per Cycle (IPC) (c) IPC of hot methods

Figure 1: The impact of sample rate on lusearch. (a) Varying the sample rates identifies similar hot methods. The green curve is the cumulative frequency distribution of samples at high frequency. Medium (red) and low (blue) frequency sampling cumulative frequency histograms are permuted to align with the green. (b) Sample rate significantly affects measures such as IPC. (c) Sample rate significantly affects IPC of hot methods. Each bar shows average IPC for a given hot method at one of three sample rates.

support the conventional wisdom that sample rate is not a significant limitation when identifying hot methods. The green curve is the cumulative frequency distribution of the number of samples taken for the hottest 100 methods when sampling at the highest frequency. The leftmost point of the green curve indicates that the most heavily sampled method accounts for 36% of all samples, while the rightmost point of the curve reveals that the 100 most sampled methods account for 97% of all samples. The blue and red bars are the cumulative frequency histograms for medium and low frequency sampling respectively, reordered to align with the green high frequency curve. A gap appears in the blue and red histograms whenever a method in the green hottest 100 does not appear in their top 100 method histogram. The red histogram is noisy because many methods attract only two samples. However, the blue histogram is very well sampled, with the least sampled method attracting 142 samples. Most bars fall below the green line, indicating that they are relatively under sampled at lower frequencies, although a few are over sampled. While sample rate does not greatly affect which methods fall in the top 50 hottest methods, Mytkowicz et al. [21] show that randomization of samples is important to accurately identifying hot methods.

Revealing high frequency behavior On the other hand, Figures 1(b) and 1(c) show that sampling at lower frequencies masks key information for metrics that vary at high frequencies, using IPC as the example. Figure 1(b) presents IPC for retired instructions. We measure cycles and instructions retired since the last sample to calculate IPC. The figure shows the cumulative frequency distribution for IPC over all samples. The green curve shows that when the sample rate is higher, SHIM observes a wider range of IPCs. About 10% of samples observe IPCs of less than 0.93 and 10% observe IPCs of over 1.45, with IPC values as low as 0.04 and as high as 3.5. By contrast, when the sample period grows, observed IPCs fall in a very narrow band between 1.0 and 1.3, with most at 1.11. As the sample period grows, each observation asymptotically approaches the mean for the program.

Figure 1(c) illustrates a second source of error due to coarse-grain sampling of rate-based metrics. In Figure 1(c), we calculate IPC, attribute it to the executing method, and then plot the average for each of the hottest 100 methods. Lowest frequency sampling (red) suggests that IPC is very uniform, at around 1.14, whereas high frequency sampling (green) shows large variations in average IPC among methods, from 0.04 (#4), to 2.39 (#59). The lower IPC variation at lower sample rates is largely due to the fact that IPC is measured over a period. As that period grows, the IPC reflects an average over an increasingly large part of the running program. This period is typically much larger than the method to which it is attributed. When the period is orders of magnitude longer than the method's execution, sampling loses an enormous amount of information, quantified by the standard deviations at the top of Figure 1(c). This problem occurs whenever a rate-based measure is determined by the period of the sample.

Direct measurement avoids this problem [2, 12, 18, 25], but requires instrumenting the begin and end of each method in this example. When just a few methods are instrumented, this method can work well, but when many methods are instrumented, methods are highly recursive, or methods execute only for a few hundred cycles, taking measurements (the observer effect) will dominate, obscuring the context in which the method executes.

3. Design and Implementation

Viewing time varying events as signals motivates our design. A SHIM observer thread executes continuously in a separate hardware context, observing events from an application thread executing on neighbor hardware. The observer samples hardware and software signals at extremely high frequencies, logging or analyzing samples depending on the configuration. SHIM samples signals that the hardware automatically generates in performance counters and memory locations that the application intensionally generates in software. SHIM consists of three subsystems: a coordinator, a sampling system, and a software signal generating system.

Signals SHIM observes signals from either hardware or software for three kinds of events: *tags*, *counters*, and *rates*. An event *tag* is an arbitrary value, such as a method identifier, program counter, or stack pointer. An event *counter* increments a value each time the event occurs. Our various analyses detect counter overflow. Hardware counters include all performance events supported by the processor, including cycles, instructions retired, cache misses, prefetches, and branches taken. Software similarly may count some events, such as allocations, method invocations, and loop iterations. Software signals may be implicit in the code already (e.g., a method identifier or parameter on the stack) or a tool may explicitly add them. For example, the compiler may insert path profiling code. SHIM computes *rates* by reading a given counter X and the clock, C, at the start and end of a sample period and then computing the change in X over change in C for that period. Section 4 describes how SHIM correctly reports rates by detecting and eliminating noise in this process.

Coordinator The coordinator configures the hardware and software signals, sampling frequency, analysis, and the observer thread(s) and the location(s) on a different core on a Chip Multiprocessor (CMP) or the same core with Simultaneous Multithreading (SMT) as the application thread(s). The coordinator configures hardware performance counters by invoking the appropriate OS system calls. It invokes the software signal generation system to determine memory addresses of software signals. The coordinator communicates the type (counter or tag) of each signal, hardware performance counters, and memory locations for software signals to the observer thread. The coordinator binds each SHIM observer thread to a single hardware context. For each observer thread, we assign a *paired neighbor* hardware context for application thread(s). The coordinator executes one or more application threads on this paired neighbor. The coordinator starts the application and observer execution. We add to the OS a software signal that identifies application threads, such that SHIM can differentiate multiple threads executing on its paired neighbor, attributing each sample correctly. SHIM thus observes multithreaded applications that time-share cores.

Sampling system The SHIM observer thread implements the sampling system. It observes some number of hardware and software signals at a given sampling rate, as configured by the coordinator. The sampling system observes hardware signals by reading performance counters and the software signals by reading memory locations. The coordinator initializes the sampling system by creating a buffer for samples. The observer thread reads the values of the performance counters and software addresses in a busy loop and writes them in this buffer, as shown in Figure 2.

We divide the observer into two parts, one for counters (lines 4 to 9) and another for tags (lines 13 to 16). Software or hardware may generate counters, rates, or tag signals.

```
1  void shimObserver() {
2    while(1) {
3      index = 0;
4      buf[index++] = rdtscp();    // counters start marker
5      foreach counter (counters){ // hardware or software counter
6        rdtscp(); //serializing instruction
7        buf[index++] = rdpmc(counter) or read_signal(counter);
8      }
9      buf[index++] = rdtscp();    // counters end marker
10     // which application thread is executing the paired neighbor?
11     pid_and_ttid = *pidsignal;
12     buf[index++] = pid_and_ttid;
13     if (tags){
14       foreach tag (tags)        // hardware or software tag
15         buf[index++] = rdpmc(tag) or read_signal(tag);
16     }
17     // online analysis here, if any
18   }
19 }
```

Figure 2: SHIM observer loop.

Recording counters and rates requires high fidelity in-order measurements. We use the rdtscp() instruction, which returns the number of cycles since it has been reset. It forces a synchronization point, such that no read or write may issue out of order with it. It requires about 20 cycles to execute.

Each time SHIM takes one or more counter samples, it first stores the current clock (line 4 in Figure 2). It then synchronously reads every counter event from either hardware performance counters or a software specified memory location and then stores the clock again (line 9). We can measure rates by comparing values read at one period to those read at the previous one. The difference in the clock tells us the period precisely. If the time to read the counters (lines 4 to 9) varies from period to period, the validity of the rate calculation may be jeopardized. As we explain in Section 4, we can precisely quantify such variation and filter out affected measurements. Because it is correct to sample any tag value within the period, we do not read tags synchronously (lines 13 to 16).

The simple observer in Figure 2 stores samples in a buffer. Realistic observers will use bounded buffers which are periodically consumed or written to disk, or they may perform lightweight online analysis such as incrementing a histogram counter. Although we do not explore it in this paper, a feedback directed optimizer could process and act on such data.

Signal system The signal system generates software signals and selects hardware and software signals.

For hardware signals, we choose and configure hardware performance counter events. These configurations depend on the analysis and on which core the observer executes. For example, when observing the number of cycles the application thread executes, we only need the elapsed cycles event counter, when all hardware threads execute at the same frequency. To measure how many instructions the application executes, we need two counters on a two-way SMT processors to derive what happens on the neighbor thread. One counter counts instructions retired by the whole core and another one counts instructions retired by the SHIM thread. The difference is due to the application thread.

For software signals, we record the address where the application writes the software signal. Applications and runtimes already generate many interesting software signals automatically. For example, consider recording the memory allocation throughput to correlate it with cache misses. Many runtimes use bump pointers which are a software signal reflecting memory allocation. As we explain in Section 6.1, some JVMs also naturally produce a signal that reflects execution state at a 15-150 cycle resolution. Of course if the address of a software signal changes, it needs to be communicated to SHIM. Note that updating a memory address after say, every garbage collection, will be less intrusive than instrumenting every write of this address for frequently written variables.

For software signals that require instrumentation, we modify the Jikes RVM compiler. For each signal, the program simply writes the value in the same memory address repeatedly. We experiment below with method and loop identifier signals and show that even though they occur frequently, they incur very low overheads on CMP. Because software signals write the same location, they exhibit good locality. Furthermore, most modern architectures use write-back polices, which coalesce and buffer writes, executing them in very few cycles.

Adding the instrumentation for software signals for highly mutating values in the SHIM framework incurs less observer effect than the direct measurement approach, which both mutates the value and then typically writes it to a distinct buffer location, rather than overwriting a single memory location.

This same software instrumentation mechanism may communicate hardware register state that is only architecturally visible to the application thread, e.g., the program counter and stack pointer could be written to memory as a software signal, but we leave that exploration to future work.

4. Observation Fidelity

This section examines SHIM's observer effects and shows how SHIM manages them to improve its accuracy. This section describes and illustrates SHIM's (i) *double-time error correction* (DTE) of samples of rate metrics; (ii) sample period randomization; and (iii) some of its observer effects.

4.1. Sampling Correction for Rate Metrics

Many performance events are *rate-based*. For example, IPC relates retired instructions and clock ticks with respect to a time interval. The two major considerations for rate metrics are: (1) attributing a rate to one tag in the interval from possibly many tags for discrete semantic events, and (2) ensuring fidelity of the measure in the face of noise and timing skew.

Attribution of Tags to Sample Periods Although tags, such as method identifiers, occur at discrete moments in time, counting metrics are calculated with respect to a *period* (e.g., $C_{t_n} - C_{t_{n-1}}$ in Figure 3 explained below). SHIM reads each tag signal once during a sample period, and then attributes it to counters in that same period. Tags correspond to the correct period, but not to any particular point within that period. SHIM reads all hardware and software tags immediately after it has completed reading each counter value (i.e., after $t_{n-1} + \varepsilon'$).

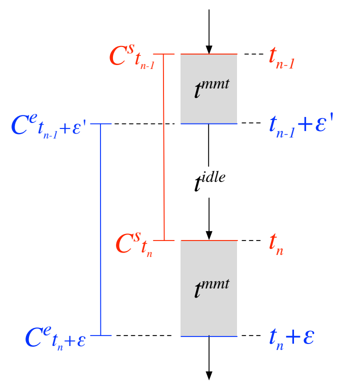


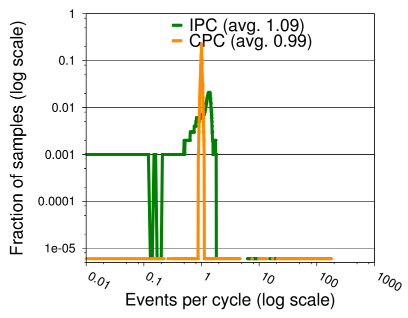
Figure 3: Four clock readings ensure fidelity of rate measurements. Grey regions depict two measurements, t^{mmt}_n and t^{mmt}_{n-1}, in which SHIM reads all counters. The sample period is, C^s (red) to C^e (blue). If the ratio of red and blue periods is one, then t^{mmt}_n = t^{mmt}_{n-1} and SHIM does not induce noise. DTE discards noisy measurements of rate metrics based on this ratio.

Rate Metrics Rates depend on *four* measurements. For example, IPC requires two instructions retired (IR) and two clock (C) measurements: $IR_{t_n} - IR_{t_{n-1}} / C_{t_n} - C_{t_{n-1}}$, ostensibly at times t_n and t_{n-1}. Since hardware can not read both simultaneously, SHIM takes each measurement at a slightly different time, t_n and $t_n + \varepsilon$, resulting in: $IR_{t_n+\varepsilon} - IR_{t_{n-1}+\varepsilon'} / C_{t_n} - C_{t_{n-1}}$, at times t_n, $t_n + \varepsilon$, t_{n-1}, and $t_{n-1} + \varepsilon'$. Figure 3 shows two overlapping intervals in red and blue. For accurate rate-based measurements, we must bound the skew between the intervals $[t_n, t_{n-1}]$ and $[t_n + \varepsilon, t_{n-1} + \varepsilon']$.

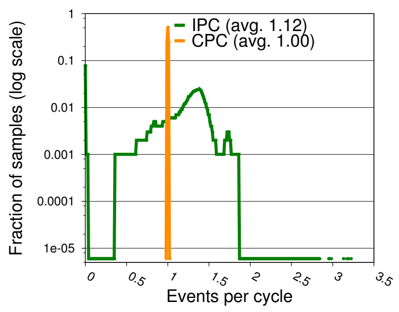
The time to take the measurements, t^{mmt} plus the time spent idle, t^{idle} defines the sample period. Variance in t^{idle} is not problematic, in fact, it helps remove bias. The intervals $[t_n, t_{n-1}]$ and $[t_n + \varepsilon, t_{n-1} + \varepsilon']$ both cover a *single* t^{idle}, so variation in t^{idle} cannot introduce skew between $[t_n, t_{n-1}]$ and $[t_n + \varepsilon, t_{n-1} + \varepsilon']$. On the other hand, the intervals $[t_n, t_{n-1}]$ and $[t_n + \varepsilon, t_{n-1} + \varepsilon']$ encompass *two* measurement periods, t^{mmt}_n and t^{mmt}_{n-1} of the exact same measurements, so variation in either measurement period introduces skew. When the sample rate is high t^{idle} becomes small, and variation in t^{mmt} may dominate. Variation in t^{mmt} will thus be exposed as t^{idle} approaches t^{mmt} and can introduce skew, which as we show next undermines the fidelity of rate-based measures.

DTE Filtering of Rate Metrics We introduce *double-time error correction* (DTE) to correct skew in rate metrics. DTE takes the *two* clock measures, C^s and C^e for each measurement period t^m, one at the start, and one at the end (lines 4 and 9 of Figure 2). The rate-based measure $C^e_{t_n+\varepsilon} - C^e_{t_{n-1}+\varepsilon'} / C^s_{t_n} - C^s_{t_{n-1}}$ precisely identifies measurement skew.

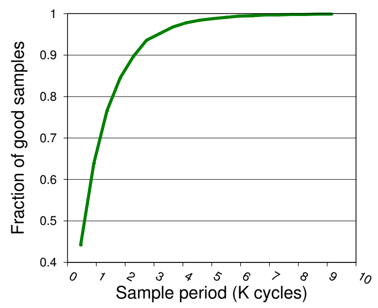
Note that CPC=$C^e_{t_n+\varepsilon} - C^e_{t_{n-1}+\varepsilon'} / C^s_{t_n} - C^s_{t_{n-1}}$ and will be 1.0 when the two clock readings are not distorted. Since they measure the same idle period, if CPC=1 the time to take the two distinct measurements t^{mmt} is the same. DTE uses this measure to identify statistically significant variation in t^{mmt} and discards affected samples. DTE therefore automatically discards



(a) Unfiltered IPC and CPC distortions, e.g., IPC> 4 on a 4-way superscalar processor.



(b) DTE correction with ±1% CPC error eliminates all distorted IPC values.



(c) Fraction of good samples with ±1% CPC error as a function of sampling frequency.

Figure 4: DTE filtering on SMT keeps samples for which ground truth CPC is 1.±0.01, eliminating impossible IPC values. At small sample periods, DTE discards over half the samples. At sample periods >2000, DTE discards 10% or fewer samples.

noisy samples with significant variations in ε and ε' since they cannot correctly compute rate metrics. Figures 4(a) and (b) show the effect of DTE. (Section 5 describes methodology.) The graphs plot on log scales IPC in green and cycles-per-cycles (CPC) in orange. CPC=1 is ground truth. Figure 4(a) shows that before filtering values are clearly distorted—CPC is as high as 175 and IPC is 37 on a 4-way superscalar processor. Figure 4(b) shows IPC samples after DTE discards all samples with CPC values outside a ±1% error margin. DTE filtering transforms CPC to an impulse at 1.0 (by design) and eliminates all IPC values greater than 4 (which we assume are wrong). All these wrong rates were introduced by sampling skew, which DTE detects and eliminates.

Figure 4(c) shows the fraction of good samples with DTE. At the highest frequency, DTE discards over 50% of samples, but at periods of ≥2500, 90% or more of samples are valid.

4.2. Randomizing Sample Periods

Prior work shows regular sample intervals in performance profiling, compared to random intervals, introduces observation bias [3, 21]. Figure 5 plots SHIM's variation in sample period and gap on a log/log scale for SMT. We plot lusearch, but the results are similar for the other DaCapo benchmarks. The figure plots the frequency distribution histogram of sample periods ($C^s_{t_n} - C^s_{t_{n-1}}$) in green and the gap between samples in red. Figure 5 shows that there is enormous variation in sample period and the gap. The most common sample period, ~500 cycles, reflects only one percent of samples, and the gap between samples ranges from ~350 to ~49,000 cycles. Both SMT and CMP show a wide degree of variation, although different. This result gives us confidence that the hardware is naturally inducing large amounts of randomness in SHIM's sampling, and thus SHIM avoids the sampling bias problem due to regular intervals.

4.3. Other Observer Effects

The sampling thread has observer effects because it executes instructions, competing and sharing hardware resources with the application. Many of SHIM's effects depend on how many and often SHIM samples memory locations and counters, and

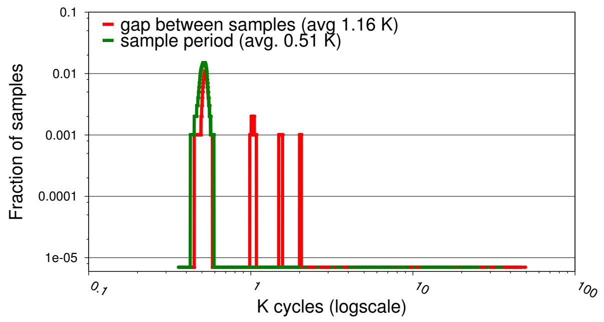


Figure 5: SHIM has large variation in sample period and between samples with DTE filtering. The green curve shows variation in the period of good samples. The red curve shows variation in the period between consecutive good samples.

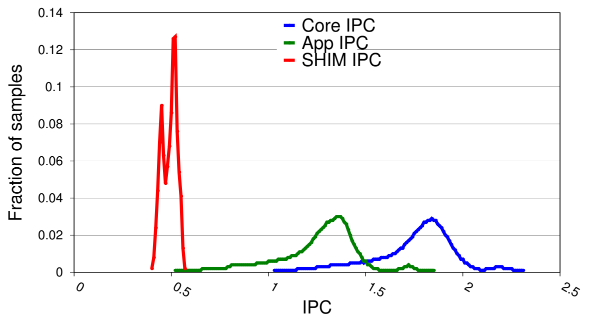


Figure 6: SHIM SMT observer effect on IPC for 476 cycle sample period with lusearch. The green curve shows lusearch IPC, red shows SHIM's IPC, and blue shows IPC for the whole core.

the hardware configuration (SMT or CMP). Section 6 systematically quantifies many of these effects.

This section examines the behavior of the sampling thread itself to determine whether we can reason about it more directly. Figure 6 plots the SMT effect of SHIM on IPC, on a log scale. The blue curve shows IPC for the whole core while the red curve shows the IPC just for SHIM. The IPC of the SHIM thread is extremely stable. The underlying data reveals that when operating with a 476 cycle period, 67% of all SHIM samples are attributed to IPCs of 0.48 to 0.51 and 99% to IPCs of 0.54 to 0.43. By contrast, the IPC of the workload is broadly spread. The uniformity of SHIM's time-varying effect on the core's resources makes it easier to factor out SHIM's contribution to whole core measurements by simply subtracting its private counter value. For each hardware metric, developers can plot SHIM, the total, and the difference to reason about observer effects.

Summary SHIM corrects skewed samples of rates, randomizes sample periods, and offers a constant instruction execution observer effect on SMT. These features reduce, especially when compared to interrupt-driven and direct measurement profiling, but do not eliminate, observer effects.

5. Methodology

The evaluation in this paper uses the following methodologies.

Software implementation We implement SHIM in Jikes RVM [1], release 3.1.3 + hg r10718, a Java-in-Java high performance Virtual Machine. We implement all of the functionality described in Section 3 by adding coordinator functionality, by modifying the VM scheduler, and by inserting signals with the compiler and VM. All measurements follow Blackburn et al.'s best practices for Java performance analysis [7]. We use the default generational Immix collector [6] with a heap size of six times the minimum for each benchmark and a 32 MB fixed size nursery to limit full heap collections to focus on application code in this paper. We measure an optimized version of the code using *replay compilation*. Jikes RVM does not have an interpreter: it uses a baseline compiler to JIT code upon first execution and then recompiles at higher levels of optimization when a cost model predicts the optimized code will amortize compilation cost in the future [4]. We record the best performing optimization plan, replay it, execute the resulting code once to warm up caches, then we iterate and only report measurements from this third iteration. We run each experiment 20 times and report the 95% confidence interval.

Benchmarks We draw benchmarks from DaCapo [7], SPECjvm98 [27], and pjbb2005 [8] (a fixed workload version of SPECjbb2005 [28] with 8 warehouses and 10,000 transactions per warehouse.) The DaCapo and pjbb2005 benchmarks are non-trivial real-world open source Java programs under active development [7]. In Sections 4 and 7, we use lusearch, a search application from the industrial-strength Lucene framework. The lusearch benchmark behaves similar to commercial web search engines [14]. We confirmed that profiling findings for lusearch generalize to other DaCapo benchmarks.

Hardware & OS We use a 3.4 GHz Intel i7-4700 Haswell processor [15] with 4 Chip Multiprocessor (CMP) cores, each with 2 way Simultaneous Multithreading (SMT) for 8 hardware contexts; Turbo Boost maximum frequency is 3.9 Ghz, 84 W TDP; 8 GB memory, 8 MB shared L3, four 256 KB shared L2s, and four private 32 KB L1 data caches, 32 KB L1 instruction caches, and 1.5 K μop caches for each core.

We use Linux kernel version 3.17.0 with the perf subsystem to access the hardware performance counters. We add to Linux a software signal that identifies threads, allowing thread switches to be identified by SHIM on SMT.

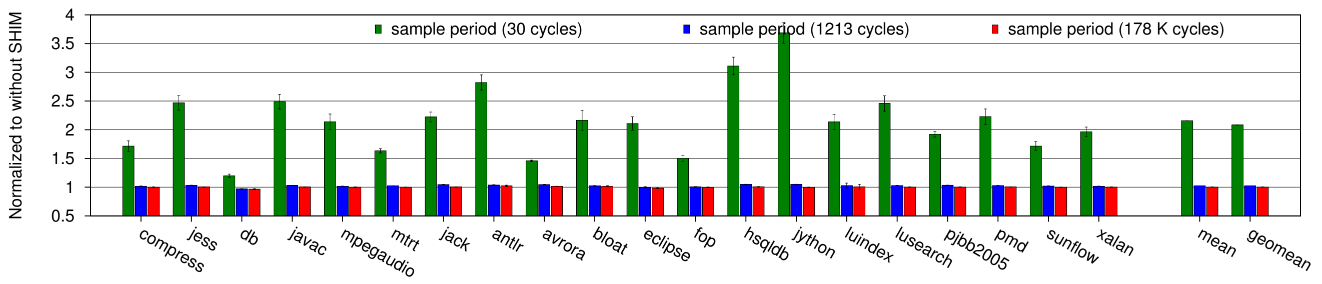
6. Evaluation

This section evaluates the strengths, limitations, and overheads of a variety of SHIM configurations and sampling rates. We start with simple, but realistic profiling scenarios, and build up more sophisticated ones that correlate software and hardware events. (1) We first compare SHIM sampling a highly mutating software signal that stores loop and method identifiers in a single memory location on the same core in an SMT context and on a different CMP core. Both exhibit high overheads at very fine (<30 cycle) resolutions due to execution resource competition on SMT (~60%) and caching effects on CMP (~100%). However, CMP overheads are negligible for coarser (~1200 cycle) resolutions. (2) When SHIM computes IPC, a rate metric, on SMT with hardware performance counters (the relevant counters are not accessible on another CMP core), high frequency sampling overheads are 47%, similar to sampling a software signal on SMT. (3) We then configure SHIM to correlate method and loop identifiers with IPC on SMT and show that the overheads remain similar. (4) Finally, we show that if hardware vendors made local performance counters visible to the other cores, SHIM overhead on CMP for correlating IPC with a highly mutating software signal would drop to essentially nothing at a resolution of ~1200 cycles.

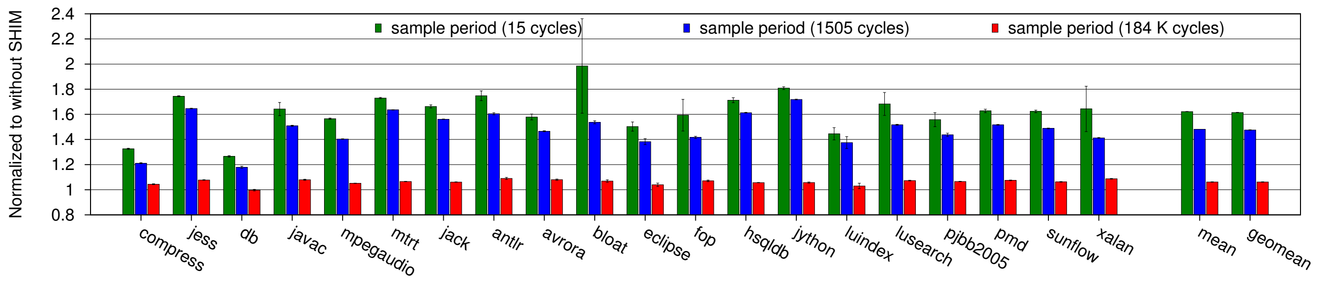
6.1. Observing Software Signals

This section evaluates SHIM overheads in configurations on the same core in an SMT hardware context and on a separate core in a CMP hardware context when it samples a software signal by simply reading a memory location. Comparing these configurations shows the effects of sharing execution resources with SMT versus inducing cache traffic with CMP. We control sample rate and contention by idling the SHIM thread.

Method and loop identifiers We repurpose *yield points* to identify methods and loops as a fine-grain software signal. JVMs use yield points to synchronize threads for activities such as garbage collection and locking. The compiler injects yield points into every method prologue, epilogue, and loop back edge. A very efficient yield point implementation performs a single write to a guard page [17]. When the VM needs to synchronize threads, it protects the guard page, causing all threads to yield as they fault on their next write to the protected page. Jikes RVM implements yield points with an explicit test rather than a guard page, so for this experiment we simply add to each yield point a write of its method or loop identifier, adding an average of 1% execution time overhead. We measure yield point frequency and find that the average period ranges from 14 (jython) to 140 (avrora) cycles. The evaluation uses as the baseline a JVM configuration with the additional store, but without SHIM. SHIM increments a bucket in a method and loop identifier a histogram. We pre-size this histogram based on an earlier profile execution. These changes produce a low cost, high frequency software signal



(a) SHIM CMP overheads are ~100% at a 300 cycle period, but drop at a 1213 cycle sample period to only 2%.



(b) SHIM SMT overheads range from 61% to 26%.

Figure 7: SHIM observing on SMT and CMP method and loop identifiers—a highly mutating software signal

that identifies fine-grain execution state.

Overheads Figures 7(a) and (b) show SHIM sampling yield points with three sampling periods from 15 cycles to 184 K cycles on CMP and SMT hardware. A period of 184 K cycles (18.5 KHz) approximates the sample rate of tools such as VTune and Linux perf (1-100 KHz). We throttle SHIM to slow its sampling rate using nop instructions or by calling clock_nanosleep(), depending on the rate. The error bars on these and other figures report 95% confidence interval.

The green bars in Figure 7(a) show SHIM in a CMP configuration sampling a memory location as fast as it can, resulting in an average sample period of 30 cycles over all the benchmarks, incurring an overhead of around 100%. This sampling frequency imposes a high overhead on applications because each application frequently writes the yield point value, which dirties the line. Every time SHIM's observer thread reads it, the subsequent write by the application must invalidate the line in the observer's cache. This invalidation stalls the application frequently at high sample rates. Section 6.2 examines this effect in more detail. Decreasing the sample period to ~1200 cycles reduces these invalidations sufficiently to eliminate most of the performance overhead.

The cost of observing software signals would be substantially reduced by a special read instruction that returns the value without installing the line in the local cache and where the cache coherence protocol is modified to ignore the read, as implemented in some ARM processors [29].

Figure 7(b) shows the cost of observing a single software signal on the same core with SMT as a function of sampling rate. On the same core, SHIM can sample memory locations at a higher rate compared to sampling from another core (every 15 vs 30 cycles). The rate on SMT compared to CMP is faster because SHIM on SMT is not limited by cache coherence traffic, only by competition for execution resources. Note that restricting the observer thread to fewer CPU resources, for example one issue slot, using priorities (such as those on MIPS and the IBM Power series [9, 26]) or some other mechanism, could significantly reduce this overhead.

Comparing the two, note that because the memory location mutates frequently, it is cheaper to sample on SMT than CMP at the highest sampling rates, but SMT is still relatively expensive for a ~15 cycle period, at 61%. However, with sample periods as low as ~1500 cycles, SMT overheads remain high, whereas CMP sampling overhead drops to a negligible amount. The next section studies these effects in more detail.

6.2. Software Signal Breakdown Analysis

Software signal overhead has two components: (1) application instrumentation, and (2) the SHIM observer thread competing either for the cache line on a separate CMP core or for hardware resources on the same SMT core.

We use the microbenchmark in Figure 8 to understand how the rates of producing software signals and consuming them impacts overheads. The producer on the left generates events by writing into one memory location. The write to the local variable dummy forces the machine to make the write to flag architecturally visible. On the right, the consumer reads the flag memory location and increments a counter.

The inner for loops control producer and consumer rates, which we adjust by varying the number of rdtscp instructions. In Figure 9(a), the blue line (increasing p-wait) shows the consumer observing as fast as it can, while the producer slows its rate of event production on the x-axis. Conversely, the red line shows the producer writing as fast as possible, while the consumer slows its rate of consumption on the x-axis.

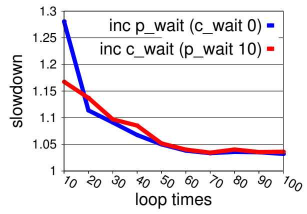
```
extern int flag;
void  producer() {
  int dummy;
  for(j=0; j<100000; j++){
    for (i=0; i < p_wait; i++){
      rdtsc();
    }
    write flag;
    // force visibility of
    // write to flag
    write dummy;
}  }
```

(a) Producer

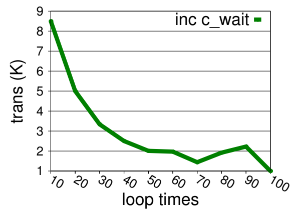
```
extern int flag;
extern int counter;

void consumer() {
  while(1){
    for(i=0; i< c_wait; i++){
      rdtsc();
    }
    read flag;
    counter++;
  }
}
```

(b) Consumer

Figure 8: Microbenchmarks explore software overheads.

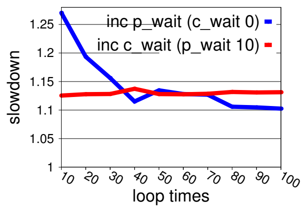


(a) Overheads as a function of producer and consumer rates.

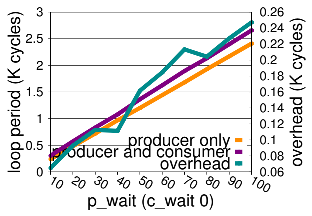


(b) Producer generated write-invalidations as a function of consumer sampling rate.

Figure 9: CMP overheads. Write-invalidates induce observer effects and overheads. Increasing the producer or consumer periods drop the overheads to $<$ 5%.



(a) Producer is degraded by high consumption rate.



(b) Sharing resources uniformly incurs overhead on SMT.

Figure 10: SMT overheads. SMT observer effects are highest as a function of producer, but then relatively constant at ~10%.

Except when both operate at their highest rate, both rates impact overhead similarly. Figure 9(b) reports the number of write-invalidations as a function of consumer (read) sample rate when the producer writes most frequently. More samples induce more invalidations. With highest frequency production and consumption rates, write-invalidation traffic dominates overhead, inducing observer effects. Increasing the sampling or production period drops overheads to less than 5%.

Figure 10 shows the same experiment on SMT hardware where overhead is dominated by the consumer. The more often the consumer reads the tag, the more it interferes with the producer. When the consumer is sampling as fast as it can, and the producer is writing at a high rate (the blue line in Figure 10(a)), they compete for execution resources. Figure 10(b) shows that the consumer adds a relatively constant number of cycles as function of sampling rate, and thus observer effects come mostly from competition for execution resources.

6.3. Observing Hardware Signals

Figure 11 illustrates the overheads of SHIM observing IPC, a rate-based hardware signal, on SMT at three sample rates. IPC cannot be evaluated on CMP because the instructions retired performance counter is not visible to other cores. In this experiment, SHIM reads two retired instruction counters (one for the core, one for SHIM itself), reads the cycle counter, computes application IPC and CPC, performs DTE, and builds an IPC histogram with 500 buckets for IPC values from 0 to 4. Because SHIM consumes execution resources, it incurs overhead of around 47% at sample periods of ~400 and ~1900 cycles. Sampling every 185 K cycles incurs a penalty of 6.3%. Although overhead is relatively high, because we discard perturbed samples and the SHIM observer thread offers a constant load, we believe that the signals are not obscured (recall the analysis in Section 4.3 and of Figure 6). Hardware manufacturers could validate our results with ground truth using their proprietary hardware measurement tools.

6.4. Correlating Hardware and Software Signals

This experiment measures overheads when we configure SHIM to correlate method and loop identifiers with IPC and with data cache misses. These are practical configurations that will help developers identify poorly performing low IPC loops and methods, and whether cache misses are responsible. Because SHIM needs two performance counters to correctly compute rate metrics, the cache miss configuration consumes five hardware performance counters.

Figure 12 compares SHIM sampling as fast as it can when it samples method and loop identifiers and IPC, with sampling them plus cache misses. Adding another performance counter makes SHIM sample more slowly (729 versus 495 cycles) because it must read both core and SHIM counters and it is limited by the 30 to 40 cycle latency of reading each counter and executing a rdtscp instruction. However, slowing SHIM to gather more hardware information incurs less overhead because it stalls more often, inducing fewer observer effects on the application. Section 7 shows two detailed case studies on critical methods using similar configurations that reveal how this fine-grain information generates hardware and software insights that prior work cannot and that suggest future directions for optimizations and mechanisms.

6.5. Negligible Overhead Fine-Grain Profiling

This section shows that if all profiling work could be performed on the separate CMP core, overheads and observer effects would be extremely low. Figure 13 shows SHIM reading three hardware performance counters on a separate CMP core, sampling as fast as it can, which results in a period of ~300 cycles on average. We use three counters because this is the minimum required to compute a rate. The time to read one performance counter ranges from 30 to 40 cycles, limiting the sample rate. SHIM executes the reads in sequence with the synchronous rdtscp() instruction (line 6 of Figure 2), because all reads must be performed in order to correctly correlate and

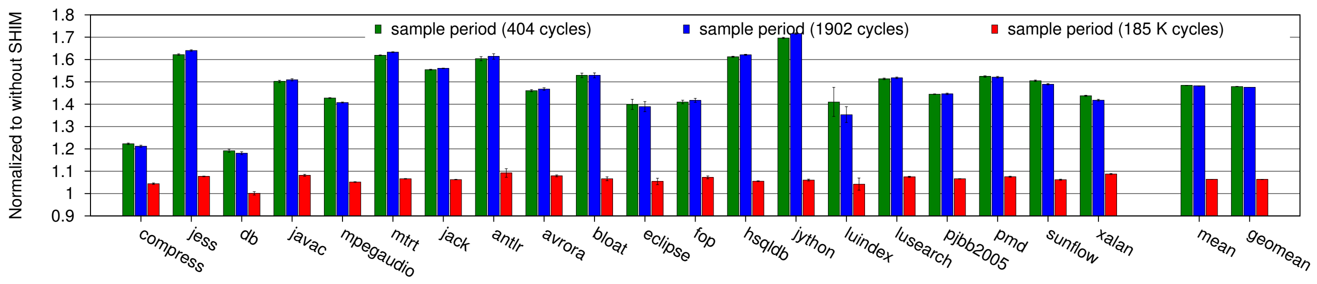


Figure 11: SHIM on SMT observing IPC as a function of sample rate. Overheads range from 47% to 19%.

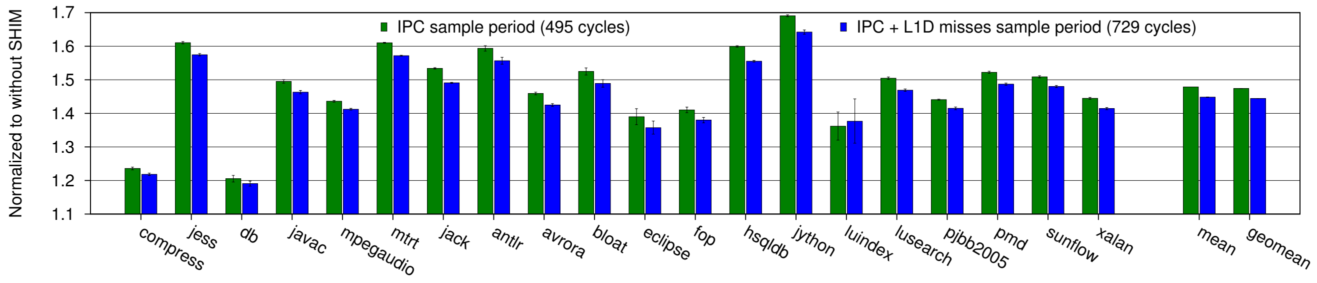


Figure 12: SHIM on SMT correlating method and loop identifiers with IPC and cache misses.

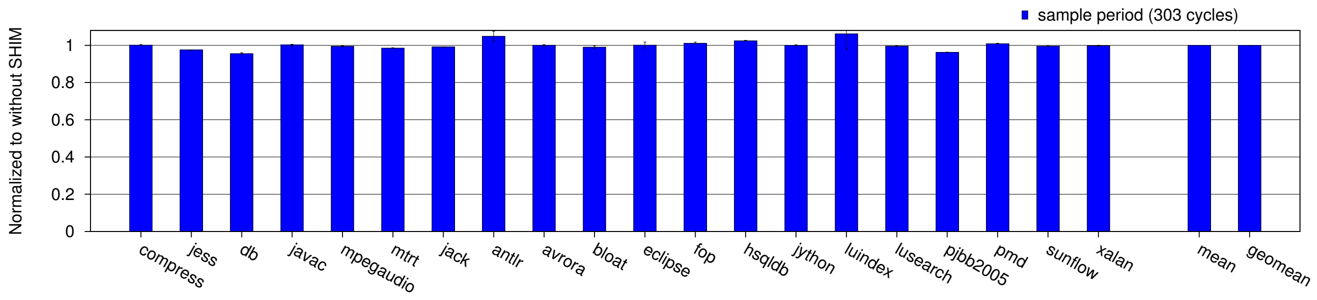


Figure 13: The overhead of SHIM is essentially zero when observing IPC from a remote core.

count events. This analysis shows that SHIM can operate at a high sample rate with no statistically significant overhead when reading hardware signals from another core. This result motivates increasing the visibility of core-private hardware performance counters to other cores.

7. Case Studies

This section shows examples of the diagnostic power of fine-grain program observations and compares it to the *average* results (reported in the top of each figure in this section) that previous tools must report for these same metrics because their sampling period is much longer.

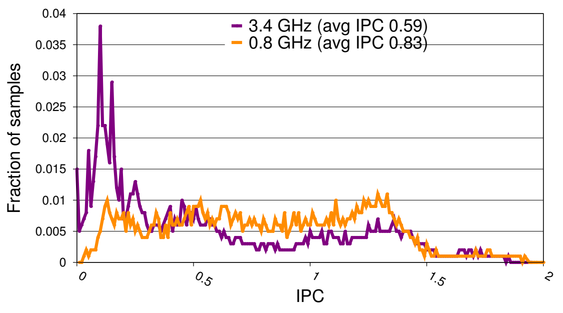
We consider two phases of performance-critical garbage collection in Jikes RVM. We first examine the response of the two phases when using DVFS to change the frequency from 3.4 to 0.8 GHz. Then we examine their response to turning on and off the hardware prefetcher. We instrument the collector with software signals that identify the phases. SHIM reads the software and hardware signals to compute IPC and memory bandwidth and attributes them to the appropriate phase.

We choose two phases on the critical path of garbage collection: (1) *stack scanning* (stacks), where the collector walks the stacks of each executing thread, identifying all references and placing them in a buffer for later processing, and (2) *global scanning* (globals), where the collector walks a large table that contains all global (static) variables, identifies references and places each reference in a buffer. Superficially, the phases are similar: the collector walks large pieces of contiguous memory identifying and buffering references which it processes in a later phase. In the case of globals, a single contiguous byte map straightforwardly identifies the location of each reference. Global scanning performs a simple linear scan of the map. On the other hand, stack scanning requires unwinding each stack one frame at a time and dereferencing a context-specific stack map for every frame to find the references within it. This highly irregular behavior leads to poor locality.

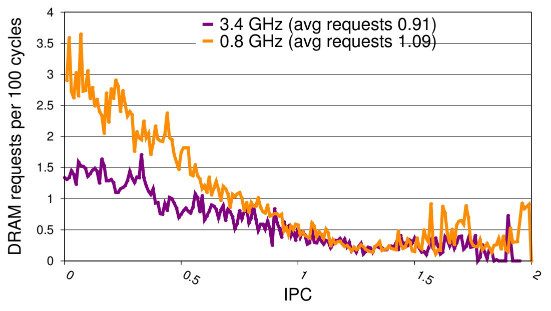
In a modern generational garbage collector, these phases can dominate the critical path of frequent ‘nursery’ collections, particularly in the frequent case where object survival is low. Therefore, VM developers are concerned with their performance. The good and poor locality of these two phases also serves as a pedagogical convenience for our analysis of DVFS and prefetching.

7.1. DVFS of Garbage Collection Phases

This section evaluates IPC and memory bandwidth at two clock speeds: 3.4 GHz (default) and 0.8 GHz on the Haswell

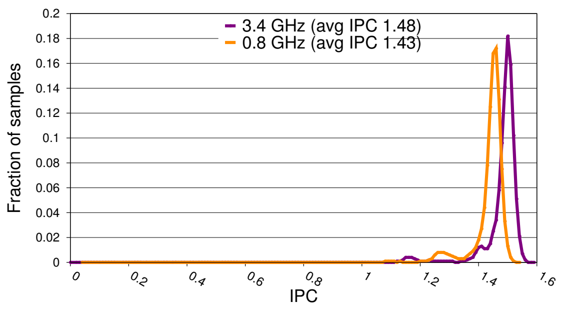


(a) IPC frequency distribution

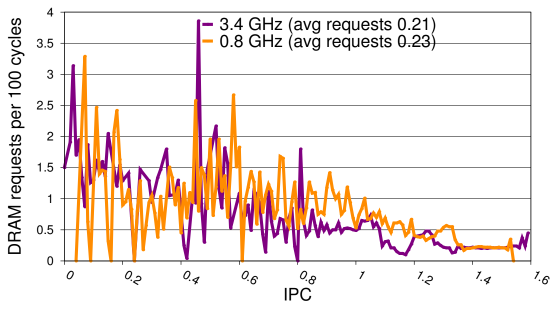


(b) Average memory bandwidth relative to IPC

Figure 14: DVFS effect at 3.4 and 0.8 GHz on IPC and memory bandwidth for stacks (poor locality).



(a) IPC frequency distribution



(b) Average memory bandwidth relative to IPC

Figure 15: DVFS effect at 3.4 and 0.8 GHz on IPC and memory bandwidth for globals (good locality).

processor. Figure 14 plots IPC and memory bandwidth for the stack phase (poor locality) and Figure 15 plots the global phase (good locality). Figure 14(a) plots the distribution of sampled IPC values at 3.4 GHz (purple) and 0.8 GHz (orange). The slower clock improves the IPC from 0.59 to 0.83, which is unsurprising for a memory-bound workload because memory accesses are relatively lower latency at lower clock rates. The purple line shows a large spike where many samples observe an IPC of ~0.10 at 3.4 GHz. This spike disappears when at 0.8 GHz (orange line), instead the IPC distribution is quite uniform. However, the slower clock speed has almost no affect on the distribution of samples above 1.3, which presumably reflect program points that are not memory bound.

Figure 14(b) shows the difference in memory bandwidth consumption between the two clock rates on the stacks phase. The histograms bucket samples according to IPC (x-axis) and for each bucket plots the average number of memory requests per 100 cycles for 3.4 GHz (purple) and 0.8GHz (orange). The lower clock rate increases memory bandwidth by 20% from 0.91 to 1.09 memory requests per 100 cycles. When IPC is low, the memory bandwidth increases by a factor of two from about 1.4 requests per 100 cycles to about 3.

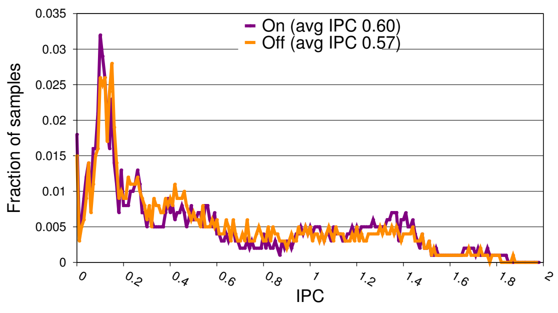
The memory-bound stack phase may benefit from a DVFS-reduced clock rate because the relatively more effective use of memory bandwidth leads to a 40% improvement in IPC. This fine-grained analysis also shows that the phase is not homogenous, and many samples show little response to DVFS.

Figure 15(a) plots the distribution of sampled IPC values for the globals phase (good locality) at 3.4 GHz (purple) and 0.8 GHz (orange). The graph shows a strikingly more focussed and homogenous distribution than the stacks phase. Interestingly, we see a counter-intuitive IPC *reduction* for the lower clock speed. Figure 15(b) shows that there is no clear change in memory bandwidth. The data to the left of Figure 15(b) is very noisy, but this noise is due to a paucity of samples—95% of all DRAM requests are due to samples with IPCs greater than 1.2. A slightly lower IPC at a slower clock is non-intuitive, but we hypothesized that the hardware prefetcher was responsible and examine this hypothesis next.

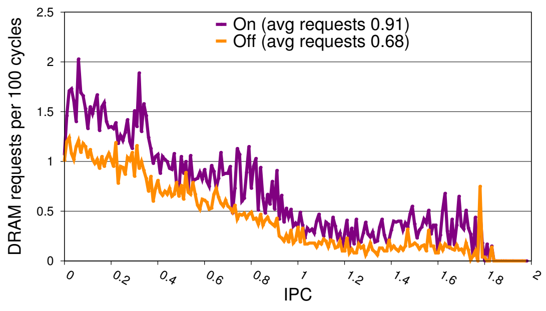
Before we continue, note that fine-grain sampling reveals the unique behavior of global scanning in the context of the entire garbage collection, which is memory bound on average and thus more resembles stack scanning. As the granularity of sampling increases it will tend toward the average for the whole of garbage collection, obscuring the distinct behavior of globals and stacks. Even if the two behaviors were equally representative of garbage collection, coarse-grain sampling may still miss the drop in IPC, because the magnitude of the response to DVFS is so much higher for the stacks.

7.2. Hardware Prefetching of Garbage Collection Phases

This section compares the effect of enabling and disabling the hardware prefetcher on IPC and memory bandwidth on the two phases. Figure 16 plots the stacks (poor locality). Figure 16(a) plots the distribution of sampled IPC values with (purple) and without (orange) prefetching. The differences are modest; on average turning off the prefetcher reduces IPC from 0.60 to 0.57; a 5% reduction. On the other hand, Figure 16(b) shows a more substantial reduction in memory bandwidth,



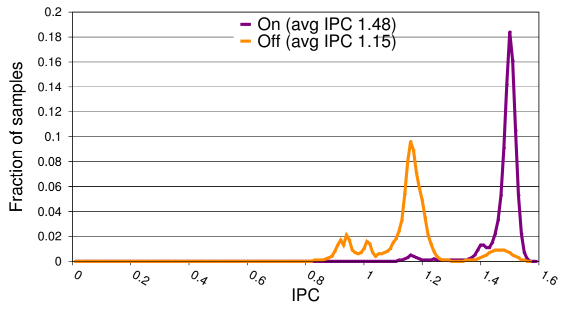
(a) IPC frequency distribution



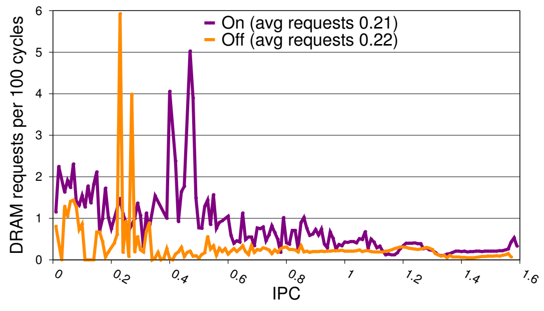
(b) Average memory bandwidth relative to IPC

Figure 16: Prefetching effect (on/off) on IPC and memory bandwidth for stacks (poor locality).

from 0.91 requests per 100 cycles to 0.68; a 25% reduction. Together these graphs suggest that the hardware prefetcher is not effective at compensating for poor locality since the reduction in memory traffic outstrips the IPC decrease by a factor of 5.



(a) IPC frequency distribution



(b) Memory bandwidth relative to IPC

Figure 17: Prefetching effect (on/off) on IPC and memory bandwidth for globals (good locality).

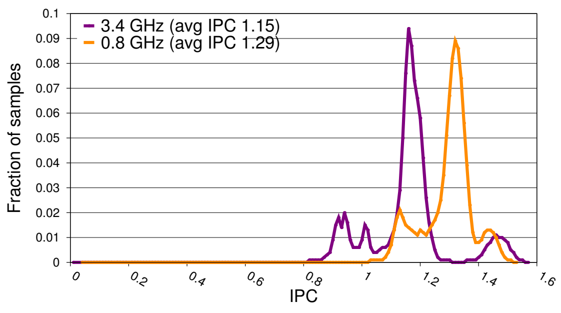


Figure 18: DVFS effect on IPC for globals with prefetching off.

Figure 17 considers the hardware prefetcher and the globals phase (good locality). Figure 17(a) shows a very clear reduction in IPC when the prefetcher is disabled, from 1.48 to 1.15. Figure 17(b) shows that unlike the stack roots phase, the average memory bandwidth is unaffected (values less than 1.2 IPC with prefetching on contribute only 5% of traffic and are noisy due to a paucity of samples).

In the stack phase (poor locality), the prefetcher is not effective and it consumes additional memory bandwidth compared to not prefetching at all, reshaping the data in the caches, and to no effect. Whereas the prefetcher for the globals (good spatial locality) is so accurate that it not only delivers the correct data in a timely fashion, it actually reduces memory bandwidth. These results suggest that if hardware vendors provide low latency ways to adjust DVFS and prefetching, a dynamic optimization could improve efficiency and perhaps performance by adjusting DVFS and prefetching at a fine granularity.

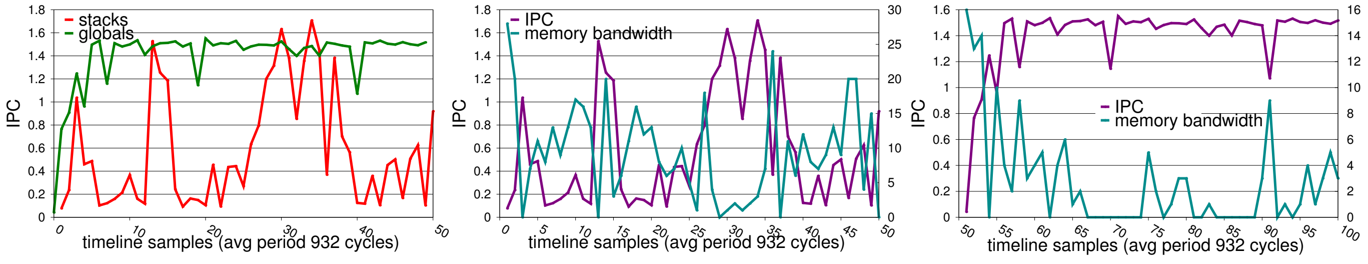
Figure 18 reconsiders the effect of DVFS on globals, this time with prefetching disabled. We find that IPC increases from 1.15 to 1.29, matching intuition and confirming our hypothesis (compare to Figure 15(a)).

Figure 19 plots time line series for IPC and memory bandwidth. These figures further illustrate the different IPC behaviors of the two phases and that memory bandwidth consumption is highly correlated with low IPC, explaining their behaviors. Note that simply examining the averages in these results that coarse-grain tools would produce does not lend itself to these insights.

8. Related Work

Four themes in profiling and performance analysis are most related to our work: profilers that sample application behavior using interrupts, software profilers, direct measurement, simulators and emulators, and feedback-directed optimization.

Interrupt-Based Sampling Interrupt-based samplers have proved invaluable at helping systems builders and application writers by performing low overhead sampling, identifying software hotspots, and attributing performance pathologies to code locations. DCPI [3] is the progenitor of today's profiling tools such as VTune, OProfile, Linux perf, and top-down analysis [16, 18, 24, 30, 33]. These systems use interrupts to sample system state and build a profile of software/hard-



(a) IPC for 50 consecutive samples (b) stacks (c) globals

Figure 19: Strong correlation between IPC and memory bandwidth revealed in time line series for stacks and globals.

ware interactions. DCPI introduced random sample intervals to avoid the sampling bias suffered by prior timer-based systems [3]. Elapsed time or a count of hardware events may define the sample period. Nowak and Bitzes [23] overview and thoroughly evaluate Linux perf on contemporary hardware. New hardware and software support, such as Yasin's top-down analysis [33], PEBS, and *Processor Tracing* [30] add rich information about the hardware and software context at each sample, but are still limited by sample rate.

Because the OS services interrupts on critical code paths within the kernel, interrupt driven systems must throttle their sample rate to avoid system lockup [3, 16, 19]. The dependence on interrupts limits these tools' ability to resolve performance behavior to events visible at sampling periods of around 30 K cycles. In contrast, SHIM exploits unutilized hardware contexts to continuously monitor performance, instead of using interrupts, and thus operates at resolutions as fine as 15 cycles. At a period of around 1 K cycles, many configurations have very low overhead. SHIM avoids sample bias as a result of its natural variation in sample period.

Direct Measurement Directly reading hardware performance counters is also a widely used approach [2, 12, 13, 18, 25, 34]. Unfortunately perturbing the code with performance counter reads that take ~30 to 40 cycles each induces observer effects–shorter periods increase coverage, but increase observer effects. In contrast, SHIM reads hardware counters without perturbing the application code itself.

Software Profilers Software profiling tools such as PiPA and CAB insert code into applications that records software events of interest in a buffer, such as paths and method identifiers, for online or offline processing [13, 34]. They however did not correlate hardware and software events, although their frameworks could support it. Ammons et al. [2] combine path profiling with hardware events, but suffer substantial performance overheads. SHIM profiles software events at very low overhead, either by injecting code that emits software signals, or by observing existing software signals. Software profilers such as PiPA and CAB can produce complete traces or samples. Sampling profilers such as SHIM cannot guarantee a complete trace, so are unsuitable when completeness is a requirement. Mytkowicz et al. show that despite the problem being identified a decade earlier [3], many software profilers using interrupt-driven timers still suffer bias [21].

Simulators and Emulators Simulators and emulators profile software at instruction and even cycle resolution. Shade [10], Valgrind [22], and PIN [20, 31] are examples of popular tools that use binary re-writing and/or interpretation to instrument and profile application code. Although they profile at an instruction level and emulate unimplemented hardware features, these tools are heavyweight. Instead of measuring hardware performance counters, they instrument code and emulate hardware. They are therefore unsuitable for correlating fine-grain hardware and software events.

Feedback-Directed Optimization Feedback-directed optimization is an essential element of systems that dynamically compile, including managed language runtimes and dynamic code translation software such as Transmeta's code morphing software [11]. These systems use periodic sampling to identify and then target hot code for aggressive dynamic optimization. SHIM provides a high resolution, low overhead profiling mechanism that lends itself to feedback directed optimization.

9. Conclusion

Performance analysis is a critical part of computer system design and use. To optimize systems, we need tools that observe hardware and software events at granularities that can reveal their behavior. This paper presents the design and implementation of SHIM, the first fine-grain profiling tool. SHIM repurposes existing hardware to execute a profiling thread that simply reads performance counters and memory locations and then logs or aggregates them. We show that configurations of SHIM offer a range of overheads, sampling frequencies, and observer effects. We show how to correct for noise and control for some observer effects. We propose modest hardware changes that would further reduce SHIM's overheads and observer effects. We present case studies that demonstrate how this performance microscope delivers a new level of analysis and potential optimizations.

Acknowledgements This work is supported in part by NSF SHF-0910818 and ARC DP140103878 grants. We thank Karin Strauss, Luis Ceze, Todd Mytkowicz, Konrad Lai, Andi Kleen, and our ISCA reviewers for interesting discussions.

References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. J. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes RVM Project: Building an open source research community. *IBM System Journal*, 44(2):399–418, 2005.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 85–96, 1997. URL `http://doi.acm.org/10.1145/258915.258924`.

[3] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems (TOCS)*, 15(4):357–390, Nov. 1997.

[4] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 47–65, Minneapolis, MN, Oct. 2000.

[5] R. Bertran, Y. Sugawara, H. M. Jacobson, A. Buyuktosunoglu, and P. Bose. Application-level power and performance characterization and optimization on IBM Blue Gene/Q systems. *IBM Journal of Research and Development*, 57(1/2):4:1–4:17, Jan 2013. ISSN 0018-8646. URL `http://dx.doi.org/10.1147/JRD.2012.2227580`.

[6] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator locality. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 22–32, Tuscon, AZ, June 2008.

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 83–89, Portland, OR, Oct. 2006.

[8] S. M. Blackburn, M. Hirzel, R. Garner, and D. Stefanović. pjbb2005: The pseudojbb benchmark, 2006. URL `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`.

[9] C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C.-Y. Cher, and M. Valero. Software-controlled priority characterization of POWER5 processor. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 415–426, 2008.

[10] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137, 1994. URL `http://doi.acm.org/10.1145/183018.183032`.

[11] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing^{TM} software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*, pages 15–24, 2003.

[12] J. Demme and S. Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *IEEE/ACM Annual International Symposium on Computer Architecture*, pages 353–364, 2011. URL `http://doi.acm.org/10.1145/2000064.2000107`.

[13] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 155–174, 2009.

[14] M. E. Haque, Y. h. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ACM International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, pages 161–175, 2015. URL `http://doi.acm.org/10.1145/2694344.2694384`.

[15] Intel. Intel Core i7-4770 processor, 8m cache, 3.90 GHz, 2013. URL `http://ark.intel.com/products/75122/Intel-Core-i7-4770-Processor-8M-Cache-up-to-3_90-GHz`.

[16] Intel. VTune Amplifier, accessed 11/2014, 2014. URL `https://software.intel.com/en-us/intel-vtune-amplifier-xe/details`.

[17] Y. Lin, S. M. Blackburn, A. L. Hosking, M. Norish, and K. Wang. Stop and Go: Understanding Yieldpoint Behavior. In *Proceedings of the 14th International Symposium on Memory Management, ISMM'15, Portland, OR, June 14, 2015*. ACM, 2015.

[18] Linux. Linux kernel profiling with perf, accessed 11/2014, 2014. URL https://perf.wiki.kernel.org/index.php/Tutorial\#Event-based_sampling_overview.

[19] Linux. Perf core.c perf_duration_warn, accessed 11/2014, 2014. URL http://lxr.free-electrons.com/source/kernel/events/core.c#L229.

[20] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. PIN: Building customized program analysis tools with dynamic instrumentation. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[21] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of Java profilers. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 187–197, 2010.

[22] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 89–100, San Diego, CA, 2007.

[23] A. Nowak and G. Bitzes. The overhead of profiling using PMU hardware counters, July 2014. URL http://dx.doi.org/10.5281/zenodo.10800.

[24] OProfile. OProfile, accessed 11/2014, 2014. URL http://oprofile.sourceforge.net.

[25] M. Pettersson. Linux Intel/x86 performance counters, 2003. URL http://user.it.uu.se/mikpe/linux/perfctr/.

[26] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreading processor. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 66–76, 2002.

[27] *SPECjvm98 Documentation*. Standard Performance Evaluation Corporation, release 1.03 edition, March 1999.

[28] SPEC. *SPECjbb2005 (Java Server Benchmark), Release 1.07*. Standard Performance Evaluation Corporation, 2006. URL http://www.spec.org/jbb2005.

[29] A. Stevens. Introduction to AMBA 4 ACE^{TM} and big.LITTLE^{TM} Processing Technology, July 2013.

[30] B. Strong, Sr. Debug and Fine-grain Profiling with Intel Processor Trace. In *Intel IDF14, San Francisco*, Mar. 2014.

[31] S. Wallace and K. Hazelwood. SuperPin: Parallelizing dynamic instrumentation for real-time performance. In *International Symposium on Code Generation and Optimization*, pages 209–220, 2007.

[32] X. Yang, S. M. Blackburn, and K. S. McKinley. SHIM open source implementation, June 2015. URL https://github.com/ShimProfiler/SHIM.

[33] A. Yasin. A top-down method for performance analysis and counters architecture. In *IEEE Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, March 2014.

[34] Q. Zhao, I. Cutcutache, and W.-F. Wong. PiPA: Pipelined profiling and analysis on multi-core systems. In *International Symposium on Code Generation and Optimization*, pages 185–194, Boston, MA, 2008.