

Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines

Matthew Arnold David Grove
IBM T.J. Watson Research Center
{marnold, groved}@us.ibm.com

Abstract

Due to the high dynamic frequency of virtual method calls in typical object-oriented programs, feedback-directed devirtualization and inlining is one of the most important optimizations performed by high-performance virtual machines. A critical input to effective feedback-directed inlining is an accurate dynamic call graph. In a virtual machine, the dynamic call graph is computed online during program execution. Therefore, to maximize overall system performance, the profiling mechanism must strike a balance between profile accuracy, the speed at which the profile becomes available to the optimizer, and profiling overhead.

This paper introduces a new low-overhead sampling-based technique that rapidly converges on a high-accuracy dynamic call graph. We have implemented the technique in two high-performance virtual machines: Jikes RVM and J9.

We empirically assess our profiling technique by reporting on the accuracy of the dynamic call graphs it computes and by demonstrating that increasing the accuracy of the dynamic call graph results in more effective feedback-directed inlining.

1 Introduction

One of the challenges to achieving high performance for programs written in an object-oriented style is to construct large enough units of compilation to enable effective optimization. Good object-oriented programming style tends to decompose the program into a large number of relatively small methods linked together via frequent virtual function calls. Without aggressive procedure inlining, this style can severely degrade performance due to both the direct costs of frequent procedure calls and the indirect costs of missed optimization opportunities caused by restricting the scope of optimization to such small program units.

Inlining is typically one of the more expensive optimizations that can be employed in an optimizing compiler: it

can result in large increases in compiled code space and compilation time (as downstream optimizations process the large compilation units created by inlining). Therefore, even though aggressive inlining is quite effective at removing both the direct and indirect costs of frequent procedure calls, it must be applied selectively. This is especially true in virtual machines, where compilation occurs during program execution. One commonly used technique to achieve a more favorably cost/benefit ratio for inlining is to utilize profile information to identify frequently executed call edges [6, 16, 12]. An additional benefit of profile-directed inlining is that it can enable guarded inlining of the dominant target(s) of a virtual dispatch when the distribution of receiver classes is skewed [5, 16, 12].

Although making high-quality inlining decisions can have a large impact on overall performance, many virtual machines make inlining decisions using relatively inaccurate profile data. Because the profiling is performed online, there is a tendency to implement simple techniques with very low overhead, even if the information they collect is relatively inaccurate. The relatively simplicity of most standard benchmarks (e.g., SPECjvm98, SPECjbb) encourages this tendency since even fairly inaccurate profiles can yield impressive performance improvements on these programs. Some virtual machines have developed more sophisticated profiling techniques to improve the accuracy of the profiles collected [25], but these techniques are more complex, require more engineering effort to implement and maintain, and the overall accuracy of the profile data is unclear.

The primary contributions of this paper are

- A new technique for collecting dynamic call graph profiles online in virtual machines that
 - rapidly collects fairly accurate profiles
 - continuously collects profiles, rather than only profiling a particular time window
 - is easily implementable with essentially zero overhead in many virtual machines
 - is easily extensible to context-sensitive profiling

- Implementations of the technique in two high-performance virtual machines: Jikes RVM and J9.
- Experiments that evaluate the technique’s overhead and accuracy on a wide range of benchmarks and that demonstrate that improvements in profile accuracy result in more effective profile-directed inlining.

Section 2 continues with a problem statement. Section 3 describes existing techniques for online collection of the dynamic call graph. Section 4 describes our approach and Section 5 describes its implementation in Jikes RVM and J9. Section 6 presents an empirical evaluation, reporting on the technique’s accuracy, overhead, and impact on overall system performance. Finally Section 7 discusses other related work and Section 8 concludes.

2 Background and Problem Statement

A common technique used to optimize object-oriented languages is to profile the call sites in the application to observe the dynamic frequency of the call site and the distribution of virtual targets.

A *call graph* $CG = (N, E)$ is a multigraph where N is a set of nodes, and E is a set of edges. Each node in N represents a distinct method. Each edge in E is a triple (N_1, CS, N_2) that represents a call from call site CS in method N_1 to method N_2 . In the case of virtual calls, a single call site can have multiple target methods.

A *dynamic call graph* (DCG) is a call graph that has frequencies associated with the call edges, and contains only those edges that are observed at runtime; therefore the edges of a DCG are a subgraph of the complete static call graph.

The problem addressed in this paper is how to collect a DCG in a virtual machine, with the following constraints:

1. The runtime overhead imposed by the technique must be low, ideally not visible above the noise.
2. The DCG should be as accurate as possible with respect to an exhaustively profiled DCG . Furthermore, the accuracy of the DCG should rapidly converge to facilitate its use by online optimizations.

The first constraint is important because it reduces the risk of incorporating the technique into a high-performance virtual machine. If the technique imposes measurable overhead, the VM must rely on client optimizations being sufficiently effective that they buy back the overhead of the profiler to avoid a net loss in performance. This is a non-issue if the technique already has insignificant overhead.

The second constraint is important because feedback-directed optimizations are effective only if the profile data is available in a timely fashion and is accurate enough to enable the proper optimization decisions to be made. For

simple benchmarks that have small hot spots and easily detectable patterns, relatively inaccurate profile data may be sufficient; however, as application complexity increases, so does the likelihood that inaccurate profile data may limit optimization effectiveness. Moreover, inaccurate profile data has the potential to *degrade* performance, causing a previously effective feedback-directed optimization to become a liability. Having reliably accurate profile data helps increase the stability of feedback-directed optimizations on challenging (possibly never seen before) applications.

3 Previous Approaches

This section describes previous approaches used to profile call edges in virtual machines. We divide the existing techniques into three categories: exhaustive instrumentation, code patching, and timer-based sampling.

3.1 Exhaustive Instrumentation

One of the earliest systems to incorporate fully automatic online profile-directed inlining was Self-93 [16]. Self is a dynamically typed pure object-oriented language. Self implementations used inline caches and polymorphic inline caches (PICs) to implement message sends [15]. An insight of the Self-93 system was that the dispatch information contained in the PICs in effect encoded most of the edges in the DCG . When a method is optimized, the compiler used the DCG information encoded in the PICs to perform profile-directed guarded inlining of virtual dispatches.

As implemented in Self-93, using PICs to gather profile data imposed no additional overhead. However, the information encoded in Self-93’s PICs only sufficed to discover edges in the DCG , not their relative weights. The Vortex compiler later augmented PICs with optional counters to collect edge weights [12]. However, instrumenting PICs with counters degraded performance by 15% to 50% depending on the benchmark.

One approach for reducing the overhead of exhaustive instrumentation is to include it in unoptimized code only. This technique ensures low overhead because frequently executed code will be identified and moved to a higher level of optimization by the adaptive optimization system. The main disadvantage of this approach is that profiling can take place only during the early stages of execution, and this behavior may not be representative of overall program behavior. This approach is used by HotSpot [21].

3.2 Code patching

Suganama et al. [25] presents a study of online profile directed method inlining in the IBM DK 1.3.1. Their system collects call site frequencies and distributions using dy-

namic instrumentation (code patching). To avoid profiling initialization code, their system does not profile any methods during their initial execution. Once a method reaches a certain level of optimization, profiling is enabled by installing a listener in the method's prologue; this listener executes each time the method is invoked and records the caller–callee relationship in a profile repository.

After a fixed number of samples are recorded, the listener uninstalls itself by patching the method prologue sequence. Their system also measures the time needed to collect the fixed number of samples, providing an estimate of the callee's invocation frequency. Their adaptive optimization system contains a component that decides which methods should be instrumented, at what times.

This technique has been proven to be successful; it is implemented in a production JVM and has been carefully evaluated with respect to compile time, code space, and steady-state performance. They validated the accuracy of their technique by comparing the performance of their on-line system to a system that used perfect offline profile data; for most benchmarks their system matched the performance of the system with offline data, but there were substantial differences for some of the benchmarks.

Although their technique was effective for the benchmarks used in their evaluation (SPECjvm98 and SPECjbb2000) we believe it has two potential disadvantages. First, the system is fairly complex to design and implement. It requires changes to the optimizing compiler to generate the listener, a code patching mechanism to uninstall the listener, and extensions to the governing adaptive optimization system to decide when listeners should be installed, uninstalled, and possibly re-installed. The engineering decisions made for any of these components could affect the accuracy of the profile collected.

A second potential disadvantage is the coarse grained nature of the profile collected. A method is not instrumented until it reaches a certain level of optimization; while this delay may be an advantage if non-representative initialization code is skipped, it can also be a disadvantage because it decreases responsiveness. Similarly, once a method is instrumented, the entire set of call samples is collected in rapid succession. Using such a short profiling window is dangerous because it increases the probability that the profile captures a short burst of non-representative behavior.

3.3 Timer-based Sampling

Some systems profile the DCG by sampling the call stack based on a timer interrupt. Whaley [26] uses a timer-based sampler to construct a calling context tree [1]. Jikes RVM uses a timer-based mechanism to construct a DCG that is used to guide inlining decisions [2, 13].

While both of these systems use a timer interrupt to trig-

ger samples, their mechanisms for performing the sampling is different. Whaley's system has a sampling thread that periodically scans the other executing threads, observing their program counters and stack pointers and recording them in a profile repository. The program threads do not perform profiling work, and are not aware that they have been sampled. In Jikes RVM, a timer interrupt is used to periodically set a *threadswitch* flag. All compiled methods contain *yield-point* sequences in their prologues, epilogues, and on loop backedges that check this flag to determine whether they should yield to the thread scheduler. The Jikes RVM adaptive optimization system [2, 3] exploits this mechanism by intercepting threads as they yield to the scheduler; the system identifies the method that was executing prior to the yield, and updates the profile repository. To approximate hot call edges, the system records the caller–callee relationship each time a yield occurs in a prologue or epilogue.

Timer-based sampling mechanisms have several advantages. They have low overhead and are relatively easy to implement because they don't require modifications to the compiler (instrumentation, code patching, etc.) or adding logic in the adaptive optimization system (deciding which methods to profile, etc.). Timer-based mechanisms also can be performed continuously throughout program execution, rather than being turned on and off periodically.

However, timer-based sampling mechanisms have the serious disadvantage that they do not collect certain kinds of profiles accurately. There are two major sources of inaccuracy. First, timer interrupts are infrequent relative to the speed of the processor,¹ so for short running programs not enough samples are taken to produce an accurate profile. As processor speed increases, and thus more instruction execute per millisecond, this becomes an increasingly large source of error. Second, there is a fundamental mismatch between the trigger mechanism and the desired profiling information. A timer-based trigger will report where *time* is being spent. This works well for determining the hot methods in the program, but is not correct for approximating the execution *frequency* of edges in the DCG.

Figure 1 presents a simple code sequence demonstrating the pitfall of using a timer-mechanism to approximate call edge frequency. The loop contains a long sequence of non-call instructions (the choice of `getfield` and `putfield` was arbitrary) followed by two calls to short running methods. The majority of timer interrupts will occur during the sequence of `getfield`'s and `putfield`'s. As a result, neither of the previously discussed timer-based techniques will accurately profile the DCG. Whaley's scheme will observe that method `M()` is repeatedly at the top the stack, and the majority of the calls to `call_1()` and `call_2()` will be missed. In Jikes RVM, the *threadswitch* flag will be set

¹Stock Linux systems restrict the minimum timer interrupt granularity available to non-privileged processes to 10ms, i.e., 100 samples a second

```

void M() {
  while (...) {
    getfield // Long sequence of non-calls
    putfield
    ...
    getfield
    putfield

    call_1(); // Two short calls
    call_2();
  }
}

```

Figure 1. Example program demonstrating the problem of using timer-based sampling to approximate call frequency.

during the sequence of non-calls, and will be read by the yieldpoint in the prologue of `call_1()`, thus `call_1()` will appear hot and `call_2()` will appear cold.

Clearly, neither of these timer-based schemes properly handles this code sequence, and the problem worsens as the number of non-call instructions increases. In fact, any time-consuming operation, such as an I/O operation, can create similar inaccuracies. Our empirical evaluation in Section 6 confirms that these sources of error are not limited to contrived examples; the observed accuracy of profiles collected using the timer-based mechanism is quite poor.

4 Our Approach

Our approach uses a combination of timer-based and counter-based sampling, as illustrated in Figure 2. Sampling is initially triggered by a timer interrupt, but instead of taking a single sample per timer interrupt, it takes a sequence of N samples. These N samples are selected using a count-down mechanism to sample every i^{th} method call. We refer to the value of i as the sampling *stride*, and the value of N as the *samples-per-timer-interrupt*.

To achieve this sampling behavior, the semantics of the pseudocode in Figure 3 is logically executed on all method entries. The flag `profilingEnabledByTimer` is `false` in the common case, so most of the logic is bypassed. Sampling begins when the timer interrupt mechanism sets the flag `profilingEnabledByTimer` to be `true`. The logic in Figure 3 then ensures that `SAMPLES_PER_TIMER_INTERRUPT` samples will be taken, each separated by `STRIDE` calls.

This hybrid approach combines timer-based and counter-based sampling in a way that exploits their strengths and reduces their weaknesses. Using a timer interrupt to initiate a profiling window removes the counter decrement and check from the common path, resulting in a lightweight mecha-

nism that can be used throughout program execution. Increasing the number of samples per timer interrupt helps overcome the coarse-grained nature of the timer, allowing faster convergence on an accurate profile. Increasing the value of *stride* increases the population of calls from which the samples are chosen by extending the length of the profiling window; this minimizes the impact of the skew that can occur from samples taken immediately after a timer interrupt, as described Section 3.3.

To ensure that all calls in the profiling window have an equal chance of being profiled, the timer mechanism can select the initial value of `skippedInvocations` from the interval `[1..STRIDE]` via either a pseudo-random number generator or a round-robin approach.

For any fixed values of the parameters `STRIDE` and `SAMPLES_PER_TIMER_INTERRUPT`, an adversary program can be constructed for which our technique will collect an inaccurate profile; however, increasing the values of these parameters reduces the likelihood that such an adversary will occur in practice. The observed impact of varying these parameters is presented in Section 6.

Implementation Options To check the value of `profilingEnabledByTimer` on each method entry, a straightforward implementation would execute three extra instructions: a load, compare, and (correctly predicted) conditional branch. The rest of the code sequence is easily moved out of line to a runtime system routine.

However, in most virtual machines it is possible to eliminate the check entirely, resulting in zero overhead when samples are not being taken. Virtual machines such as Self-93 [16], Hotspot [7], Jikes RVM [19], and J9 [11] already execute at least one conditional test during method entry to check for stack overflow or a request to interrupt normal execution to perform some system service. In such a VM, the check can easily be modified to test a new flag that represents the logical OR of the original flag and our sampling flag (see Figure 4). The runtime test now checks `overloadedFlag` which encodes whether *either* flag is true; if so, control jumps to an out of line routine that tests both flags and performs the appropriate action. When the overload flag is true, additional work is required to determine what action(s) should be taken. However, this overhead should be insignificant since both the original flag and the sampling flag have the property that they are rarely true.

Using this approach, our technique can potentially be incorporated into a virtual machine without *any* changes to the code generated by the compiler; as a result, there is no additional runtime overhead while samples are not being triggered, and no increase in compiled code size because all profiling code can be placed out of line. The only scenario in which our technique would introduce unavoidable overhead is if the virtual machine does not already execute any

mentation of inlining in Jikes RVM. Our work uses a vanilla Jikes RVM 2.3.3 that does not include their extensions for context-sensitive profiling.

Jikes RVM's compilers generate yieldpoints in method prologues, epilogues, and on loop backedges. The primary purpose of these yieldpoints is to support quasi-preemptive m-to-n threading, but they are overloaded to support other system services such as the adaptive system's sampling mechanism, transition to garbage collection, and on-stack-replacement.

Implementation of counter-based sampling To implement our new profiling technique in Jikes RVM, we made a minor modification to the yieldpoint code sequence generated by its compilers. Previously, a yieldpoint checked to see if a thread-local word was non-zero, if so it invoked a runtime routine, `VM_Thread.yieldpoint`. We modified the compilers so that prologue and epilogue yieldpoints check for non-zero, but backedge yieldpoints check for greater than zero. This enables us to efficiently encode three possible states in the single existing flag: yieldpoints should not be taken (0), prologue/epilogue yieldpoints should be taken (-1), and all yieldpoints should be taken (1).

When a yieldpoint is taken, execution transfers to `VM_Thread.yieldpoint`. We modified this method so that when a yieldpoint is taken in response to a timer interrupt (which sets the yieldpoint control word to 1), our counter based sampling mechanism described by Figure 3 is enabled (by setting the yieldpoint control word to -1). The system then enters a mode in which all executed prologue and epilogue yieldpoints are taken until the requisite number of samples have been gathered. Then the control word is set to 0, the thread switch is allowed to occur and the next scheduled thread begins executing normally.

To gather our samples, the adaptive system was extended to optionally register its *listeners* [2, 13] (profile-gathering agents) on counter events instead of on timer-interrupt events. The overall logic of the adaptive system, including the *organizers* that process the raw profile data were unchanged: they simply process samples without needing to know if the samples came from a listener that was responding to time-based or counter-based events.

Inlining Jikes RVM's existing profile-directed inliner was able to consume the more accurate profile data produced by our technique without modifications. We performed a series of experiments to assess the effectiveness of our technique and were unpleasantly surprised to discover that improvements in profile accuracy did not lead to improved inlining decisions. We eventually realized that this was happening because the existing inliner had been designed to compensate for inaccurate profile data by being fairly conservative in deciding what to inline. It only used profile data to determine whether or not a given call graph edge was hot (accounted for more than 1% of the total weight of the DCG). If

an edge was hot, the inliner would increase its space thresholds and allow larger callee methods to be inlined. Profile data for non-hot edges was completely ignored. As a result, it was missing many opportunities where the profile data indicated that a non-hot (but still profiled) virtual call site had only been seen to invoke a single small target method.

Therefore, we implemented a new profile-directed inliner that is able to more effectively exploit higher accuracy profile data. Our new inliner differs from the previous one in two main ways. First, it does not make a sharp distinction between hot and non-hot edges. Instead, edge weight is used as an input to a linear function that computes a size threshold for a call site; the hotter a call site is, the larger the size threshold for callee methods. This function is bounded by a maximum allowable size to avoid observed performance degradations when inlining truly massive methods. Second, the new inliner considers the shape of the distribution for call sites that (dynamically) have multiple targets. Only callees that account for more than 40% of the distribution are considered for guarded inlining at that call site. We also fixed several oversights in the static inlining logic that was resulting in additional missed opportunities.

After implementing and tuning the new inliner, it became clear that the old inliner had been overly conservative. Even using the default timer-based profiling mechanisms, the new inliner improved performance on SPECjvm98 by an average of 3%, with gains of 6% on `javac` and 5% on `jess`, two of the more object-oriented benchmarks in SPECjvm98. Our new inliner has been incorporated in Jikes RVM and became the default in October 2004.

5.2 The J9 Virtual Machine

J9 [11] is one of IBM's production Java virtual machines. J9 contains an interpreter, a JIT compiler with multiple optimization levels, and an adaptive optimization component. The performance of J9 is competitive with other leading product virtual machines.

Implementation of counter-based sampling To implement our call graph profiler in J9, we made a trivial change to the method entry sequence. J9 already executes a check on method entry to see if runtime services need to be performed; we overloaded this check (as in Figure 4) to check a new flag that encodes both our sampling flag and the original flag conditions. This approach enabled us to avoid any significant changes to the method entry sequence (and thus J9's JIT compiler). The flag disambiguation and sampling logic from Figure 3 is implemented entirely in J9's runtime system, and is not inlined into the compiled code. Thread-local variables are used for the counters to avoid potential scalability issues or race conditions.

The call stack sampler uses existing J9 stack-walking routines to collect a call stack sample. We considered writ-

ing specialized routines, optimized to extract only the information needed build the dynamic call graph, but decided that we should reuse existing code to gain the benefits of a simpler and easier to maintain system. Optimizing the sampling mechanism could potentially allow a higher sample rate to be used (given the cheaper cost per sample), but thus far does not appear to be necessary.

Inlining J9’s inliner relies on both static and dynamic heuristics for guiding inlining; however, J9 static inlining heuristics are much more aggressive than those in Jikes RVM. For this work, we modified J9’s dynamic heuristics to use the call graph collected by our profiler. If a call site is cold, the static heuristics are overridden and inlining is not performed; if a call site is hot, the original static size thresholds are increased. Similar to Jikes RVM, the profiling threshold required for inlining is a linear function of method size.

6 Experimental Results

This section presents an empirical evaluation of our technique by answering two primary questions: 1) Can our technique improve the accuracy of dynamic call graphs without increasing runtime overhead? 2) Will improved accuracy improve the effectiveness of feedback-directed inlining?

6.1 Benchmarks

Table 1 describes the benchmark programs used in this study. The first seven comprise the SPECjvm98 benchmark suite [24].² SPECjbb2000 is an emulator of a typical Java business application [23]. `Ipsixql` is a benchmark of persistent XML database services [8]. The `xerces` benchmark measures a simple XML Parser exercise, using the Apache Xerces Java2 parser [27]. `Daikon` is a dynamic invariant detector from MIT [9]. `Kawa` exercises a Java-based Scheme system [20]. `Soot` is a Java bytecode analysis and transformation framework from McGill University [22].

To produce a wide range of running times for the benchmark suite, each benchmark is run with two different sized inputs: “small” and “large”. For each input size, Table 1 reports the following three quantities: running time of the benchmark (seconds) when executed on a production build of Jikes RVM, the number of methods executed, and the total size of executed bytecodes (in kilobytes).

The experiments were performed using IBM Intellistations running Red Hat Linux version 9.0. The machine used for the Jikes RVM experiments contained two 2.8 GHz Intel Xeon processor and 2 gigabytes of RAM; the machine used for the J9 experiments contained a 3.0GHz Intel Xeon processor and 1 gigabyte of RAM.

²None of the results reported conform to the official SPEC run rules, so our results do not directly or indirectly represent a SPEC metric.

Benchmark	Small Input			Large Input		
	Time (sec)	Meth exe	Size (K)	Time (sec)	Meth exe	Size (K)
compress	1.38	243	22	7.96	243	22
jess	0.92	662	42	5.17	675	43
db	0.46	258	24	17.17	262	24
javac	0.80	939	86	10.56	967	87
mpegaudio	1.90	416	67	7.56	415	67
mtrt	0.91	368	31	5.98	369	31
jack	0.85	477	52	5.51	478	52
ipsixql	1.34	459	31	8.68	490	33
xerces	3.28	719	64	7.87	822	71
daikon	4.51	1671	140	29.82	1673	141
kawa	0.95	1794	96	5.07	3496	161
jbb	2.00	597	51	28.08	597	51
soot	1.67	1215	111	82.45	1734	126

Table 1. Benchmarks used in this study

6.2 Evaluating Accuracy and Overhead

This section explores the accuracy and overhead of our technique. Evaluating the overhead of a profiling technique is fairly straightforward, however evaluating profiling accuracy can be more complex; there are two approaches most commonly used in the literature:

1. **Accuracy metric (direct measure):** Compare the approximate profile to an perfect profile and compute a similarity function, or *accuracy metric*.
2. **Impact on client (indirect measure):** Use the approximate profiles as input to some client optimization(s) and compare the resulting performance.

Accuracy metrics are a more direct measure of profile accuracy and avoid complications with additional variables that may be introduced by a client optimization. However, the bottom-line impact of accuracy can be determined by examining its impact on a client. Therefore, we use both kinds of metrics. The remainder of this section evaluates accuracy using an accuracy metric and Section 6.3 reports the impact of accuracy on a client: profile-directed inlining.

Accuracy Metric To evaluate profile accuracy we compare the sampled profiles to a perfect profile and compute an accuracy score using the *overlap* metric. The overlap metric has been used in previous studies of profile accuracy [4, 10] and is defined as:

$$overlap(DCG_1, DCG_2) = \sum_{e \in CallEdges} \min(Weight(e, DCG_1), Weight(e, DCG_2))$$

where *CallEdges* is the set of call edges contained in $(DCG_1 \cap DCG_2)$ and $Weight(e, DCG_x)$ is the percentage of samples associated with call edge e in DCG_x . Intuitively, the overlap percentage represents the percentage of

profiled information that exists in both profiles. The result will always be in the range from 0—100, where 0 represents two profiles that contain no common information, and 100 represents two identical profiles. The *accuracy* of a sampled profile is computed by comparing it to a perfect profile using the overlap metric:

$$accuracy(DCG_{sample}) = overlap(DCG_{sample}, DCG_{perfect})$$

We chose the overlap percentage as our metric because it is fairly intuitive to understand and interpret the results. We used this metric while debugging our profiling implementations and found it to be very helpful; the accuracy reported by the overlap metric was a reasonable predictor of what we would discover if we examined the profiles by hand. If overlap is in the range of 10-20%, it is clear that the profiles vary substantially; if the overlap is 80-90%, the profiles are quite similar. Of course, the magnitude of difference in overlap that should be considered significant is client-dependent.

Benchmarking setup To evaluate accuracy, we disabled the adaptive optimization systems and ran the VM’s in a “JIT-only” configuration, where all methods are compiled at the same optimization level upon first execution. We chose this configuration because the actions of an adaptive optimization system make it difficult to compare profile accuracy. In an adaptive configuration, code is initially unoptimized (no inlining has occurred) so all calls execute explicitly and would be profiled. As hot code is optimized, methods will be inlined and no longer be profiled. Furthermore, both VM’s use some degree of timer based sampling to select optimization candidates which results in run-to-run variations in optimization decisions. Small variations in these decisions can have a drastic effect on the calling behavior. Using a non-adaptive system produces deterministic calling behavior that can be used as a consistent basis for evaluating our profiling technique.

We chose a low level of optimization (Level “0” for Jikes RVM, and J9’s lowest optimization level) so that trivial methods (methods whose inlined bodies are smaller than a calling sequence) would be inlined, but all other calls remain, and thus have the potential to be profiled. We believe this is a reasonable baseline, particularly since systems often rely on profiling for inlining all non-trivial methods [25].

Results Table 2A reports the overhead and accuracy of our technique in Jikes RVM, when run with a wide range of values for the *Stride* (columns) and number of *Samples-per-timer-tick* (rows). For ease of exposition, *Samples-per-timer-tick* will furthermore be referred to as *Samples*. All numbers reported are the average over all benchmarks, and the data for each benchmark is the median of 10 runs to reduce noise. Each cell in the table reports two values:

1. **Overhead:** The first number is the percent overhead relative to the system that does not construct a DCG. Higher numbers represent higher overhead (worse performance). Negative numbers are indicative of the magnitude of run-to-run variation.
2. **Accuracy:** The second number is the accuracy, reported as overlap percentage. Accuracy ranges from 0–100. Higher numbers represent higher accuracy.

The first interesting observation is the accuracy of the original Jikes RVM sampling mechanism, which is reported in grid location *Stride=1, Samples=1*; the accuracy for this system was 38%. This relatively low accuracy confirms that the previously discussed weaknesses of sampling based on timer interrupts. As the value of either parameter (*Stride* or *Samples*) increase, the accuracy improves, helping confirm that *both* of the accuracy anomalies discussed in Section 3.3 occur in real programs. Note that Jikes RVM is already requesting timer interrupts at the finest granularity supported for normal user programs by the host operating system, therefore it is not feasible to increase the accuracy of the original mechanism simply by increasing timer interrupt frequency.

A large range of values in the upper left portion of the table report what we would consider “low” overhead, meaning that it is mostly indistinguishable from the noise. A configuration such as *Stride=3, Samples=32* results in overhead of 0.3% with accuracy 66, which is 1.7 times the accuracy of the original system.

Table 2B reports the same information as Table 2A, but was collected using our J9 implementation. Despite the differences between the two virtual machines, the performance trends of the J9 system are very similar to that of Jikes RVM. The accuracy of the default system (*Stride=1, Samples=1*) is 37%. A configuration such as *Stride=7, Samples=32* produces an accuracy 69% with 0.5% overhead.

Table 3 reports overhead and accuracy data for each benchmark for both the timer-based and counter-based profilers. J9 does not normally use a timer-based call graph profiler, so our technique with parameters *Stride=1, Samples=1* is used as the base. For the counter-based profiler, the values of *Stride* and *Samples* were selected to achieve a reasonable tradeoff between overhead and accuracy. This data is the median of 10 runs.

This data confirms that our technique has low average overhead (<0.3% for both VMs), but more importantly demonstrates there are no significant overhead spikes for any of the benchmarks. The accuracy of our system is always equal or better than the default system, except for compress-large, for which the base system surprisingly outperforms our technique in both VM’s. However, our technique is substantially more accurate than the base system for almost all other benchmarks.

Table 2A: Jikes RVM

Samples Per Timer Tick	Stride													
	1	3	7	15	31	63	127	255	511	1023	2047	4095	8191	
1	-0.1/38	-0.1/37	0.1/42	0.0/44	0.1/47	0.1/49	0.0/49	0.1/50	0.3/50	0.2/51	0.2/49	0.3/49	0.2/49	
2	-0.1/43	-0.1/45	-0.1/49	0.0/52	-0.1/54	0.1/55	0.2/55	0.5/56	1.0/57	1.6/57	3.4/57	6.3/59	13.1/60	
4	0.1/48	0.0/52	-0.1/56	0.1/58	0.1/59	0.2/60	0.5/61	1.3/62	2.5/63	5.0/63	9.8/65	21.1/66	53.0/68	
8	0.0/54	-0.1/58	0.0/61	0.1/62	0.3/64	0.7/64	1.4/65	2.9/66	5.6/66	11.5/69	25.4/71	68.6/73	512.8/79	
16	0.1/59	0.2/63	0.3/65	0.5/65	0.8/67	1.7/67	3.0/68	6.0/69	12.3/72	27.7/74	76.8/76	224.4/82	920.9/84	
32	0.0/62	0.3/66	0.4/67	0.7/68	1.6/69	3.1/70	6.3/71	12.9/74	28.6/76	81.0/77	814.0/84	922.6/85	920.4/84	
64	0.6/66	0.8/68	1.2/68	2.0/70	3.5/70	6.7/72	13.4/74	29.9/77	85.1/78	854.6/85	927.7/86	924.9/85	922.4/84	
128	0.6/68	1.0/69	1.9/71	3.4/71	6.5/73	13.4/75	30.0/77	86.1/79	876.9/86	932.4/86	925.7/86	922.5/85	920.7/80	
256	0.9/70	1.7/70	3.4/71	6.6/74	13.5/76	30.5/78	87.8/79	903.0/86	946.1/86	932.8/86	925.7/85	922.2/80	-/-	
512	1.9/70	4.3/71	7.7/74	14.8/76	32.2/78	91.2/79	943.2/86	974.6/87	949.7/86	935.2/86	928.3/79	-/-	-/-	
1024	4.6/72	8.0/73	15.2/76	32.6/78	92.0/79	1007.9/86	1016.5/86	971.4/86	944.8/86	931.4/79	-/-	-/-	-/-	
2048	9.0/73	16.7/75	34.3/77	95.8/80	1101.9/85	1096.1/87	1019.2/86	970.1/86	944.8/86	-/-	-/-	-/-	-/-	
4096	17.5/75	36.5/77	99.2/80	1182.6/85	1227.1/87	1098.2/87	1015.3/86	969.1/80	944.8/80	-/-	-/-	-/-	-/-	
8192	36.6/77	99.2/80	1035.1/81	1404.4/87	1232.6/86	1094.5/85	1012.8/81	-/-	-/-	-/-	-/-	-/-	-/-	

Stride

Samples Per Timer Tick	Stride													
	1	3	7	15	31	63	127	255	511	1023	2047	4095	8191	
1	0.1/37	0.3/42	0.2/44	0.2/45	0.1/46	0.2/46	-0.1/46	0.1/47	0.2/48	0.2/49	0.5/50	0.6/50	1.2/50	
2	0.3/47	-0.1/49	0.2/50	0.3/51	0.2/52	0.2/52	0.0/54	0.2/54	0.2/55	0.4/55	1.0/57	1.1/57	2.5/58	
4	0.3/53	0.1/54	0.2/56	0.2/56	0.2/57	0.1/57	0.2/59	0.4/60	0.4/61	0.8/62	1.5/64	2.2/65	4.8/66	
8	0.4/58	0.3/59	0.1/60	0.2/60	0.3/62	0.3/63	0.4/65	0.5/66	0.8/68	1.2/69	2.5/70	4.5/71	9.5/72	
16	0.5/62	0.2/63	0.3/64	0.5/65	0.2/66	0.3/67	0.5/70	1.1/71	1.4/74	2.5/75	5.0/76	9.7/77	21.1/78	
32	0.4/65	0.5/66	0.5/69	0.4/69	0.8/72	0.5/72	1.0/75	1.9/77	2.9/79	5.1/80	9.8/82	21.3/82	49.1/84	
64	0.9/68	0.8/69	0.7/71	0.7/73	0.9/74	1.4/76	2.0/79	3.0/81	5.5/83	10.4/84	21.9/86	50.5/87	75.8/87	
128	1.2/70	1.0/72	1.1/73	1.2/75	1.6/79	2.4/80	3.3/83	5.9/85	10.9/87	24.6/88	49.7/89	75.9/89	78.4/87	
256	2.0/72	1.7/74	1.9/76	2.1/78	3.0/82	4.4/84	7.1/87	12.2/87	24.7/90	53.0/91	79.1/92	79.2/89	78.8/87	
512	3.2/74	3.3/76	3.6/80	4.4/82	5.7/86	8.7/87	14.3/89	28.1/90	59.0/93	83.0/93	80.9/92	79.7/89	78.4/87	
1024	6.0/76	6.2/79	7.0/83	8.6/85	11.5/88	18.2/89	33.1/92	67.7/93	88.0/95	84.1/93	81.6/92	79.8/89	78.8/87	
2048	11.6/78	12.6/82	14.8/85	18.0/88	27.1/91	45.9/92	86.7/95	96.6/95	88.5/95	84.4/93	81.6/92	79.9/89	-/-	
4096	27.1/81	29.5/86	35.5/89	47.8/91	80.2/93	121.7/96	112.5/97	97.3/95	89.4/95	84.4/93	81.9/92	-/-	-/-	
8192	88.8/86	106.4/89	153.4/92	215.0/94	183.0/98	139.0/97	112.4/97	97.5/95	88.8/95	84.6/93	-/-	-/-	-/-	
16384	1593.9/94	841.5/97	446.7/98	271.3/97	184.0/98	138.0/97	112.6/97	97.9/95	89.1/95	-/-	-/-	-/-	-/-	
32768	2037.4/100	895.2/98	458.0/98	272.0/97	184.2/98	137.5/97	112.6/97	97.9/95	-/-	-/-	-/-	-/-	-/-	
65536	2043.1/100	894.3/98	457.6/98	271.8/98	184.1/98	137.4/97	112.4/97	-/-	-/-	-/-	-/-	-/-	-/-	
131072	2037.7/100	893.5/98	456.8/98	272.7/97	184.3/98	138.2/96	-/-	-/-	-/-	-/-	-/-	-/-	-/-	

Table 2. Overhead and Accuracy of counter-based sampling for a wide range of values for parameters Stride and Samples per timer interrupt. Table 2A presents the results for Jikes RVM, and Table 2B presents the result for J9.

Benchmarks	Jikes RVM		J9	
	Base	Stride=3 Samp=16	Stride=1 Samp=1	Stride=7 Samp=16
compress-small	0.0/9	-0.7/27	-0.4/73	0.1/80
jess-small	0.0/25	0.2/53	-0.1/25	0.6/34
db-small	0.0/20	0.1/46	0.5/55	0.2/54
javac-small	0.0/29	0.1/48	-0.9/5	-1.1/28
mpegaudio-small	0.0/25	0.2/43	-1.0/21	-0.8/43
mtrt-small	0.0/21	-0.5/56	-0.1/7	-0.2/34
jack-small	0.0/30	-0.9/65	-0.1/12	0.8/50
ipsixql-small	0.0/22	2.8/52	0.9/31	1.0/59
xerces-small	0.0/22	1.7/68	1.0/20	1.1/63
daikon-small	0.0/31	-0.2/63	0.4/31	0.7/66
kawa-small	0.0/28	0.3/49	0.5/10	0.4/18
jbb-small	0.0/44	-0.4/82	0.1/37	0.0/80
soot-small	0.0/29	0.2/57	0.4/20	0.5/52
Average Small	0.0/26	0.3/55	0.0/27	0.2/51
compress-large	0.0/83	-0.9/81	0.9/91	1.0/86
jess-large	0.0/55	0.3/77	0.7/46	0.6/66
db-large	0.0/25	0.7/25	0.2/76	0.2/77
javac-large	0.0/44	-0.4/71	-0.2/35	1.1/69
mpegaudio-large	0.0/47	0.1/63	-0.9/38	-1.2/59
mtrt-large	0.0/48	-0.5/84	-0.3/37	-0.1/83
jack-large	0.0/52	-0.6/75	0.4/42	0.8/82
ipsixql-large	0.0/60	1.3/78	0.1/52	0.3/88
xerces-large	0.0/41	-1.4/57	0.8/45	1.4/80
daikon-large	0.0/57	1.4/82	0.1/49	0.3/72
kawa-large	0.0/32	0.4/56	0.9/11	-0.2/44
jbb-large	0.0/52	0.4/82	0.5/39	0.4/82
soot-large	0.0/49	0.4/70	0.1/41	0.6/80
Average Large	0.0/50	0.0/69	0.2/46	0.4/74
Average All	0.0/38	0.2/62	0.1/37	0.3/64

Table 3. Overhead and accuracy breakdown

For the small running benchmarks, our system averaged 55% accuracy for Jikes RVM and 51% for J9. While this accuracy is not particularly high, we believe it is primarily due to the short running nature of the applications. The base system averaged only 26% and 27% for Jikes RVM and J9, respectively. Code-patching approaches (Section 3.2) would also perform poorly because they delay profiling until code has reached the first level of optimization; many of these benchmarks are so short running that very few methods are optimized before the program exits. Accuracy on the long running benchmarks was higher, averaging 69% for Jikes RVM and 74% for J9 with our technique, but only 50% and 46% respectively for the base system.

6.3 Inlining Performance

Figure 5 shows the impact of profile-guided inlining on steady-state performance using both the original profiling mechanism (timer-only) and our new technique (cbs). To evaluate steady-state performance, the benchmarks were configured to iterate for 2 minutes total, and performance was evaluated using the second minute only. Our perfor-

mance results include only those benchmarks that could be configured for this execution scenario.

For the counter-based profiler, we kept the same values of *Stride* and *Samples* used previously in Table 3. We found that within the “reasonable” space of parameters that maximize accuracy while holding overhead to less than 0.5%, the exact choice of parameters did not result in significant performance differences.

The left graph in Figure 5 shows the speedups of profile-guided inlining in a production build of Jikes RVM. Both configurations (timer-only and cbs) use the new inliner described in Section 5. The importance of profile-directed inlining varies from program to program. It makes the largest difference for `mtrt`, `jess` and `compress`. The more accurate profile data only makes a significant difference on `javac`, although it also yields small improvements on `jack` and `mpegaudio`. The result on `javac` is encouraging since it is one of the more complex benchmarks and this suggests that profile accuracy may be more important as program complexity increases. We explored different combinations of *Stride* and *Samples* for `javac` and found that the performance gain was mostly (but not entirely) due to increasing the value of *Samples*. It is hard to reliably quantify the relative importance of *Stride* and *Samples* for `javac` as the performance differences between different parameter combinations are small (1–2%).

The right graph in Figure 5 shows the speedup obtained in J9 from profile-directed inlining. Performance is normalized to a configuration of J9 that uses static inlining heuristics only. Enabling dynamic heuristics based on the cbs profiler produces a 8.7% speedup for `mtrt`, and roughly a 1% speedup for 5 of the other 8 benchmarks. The impact of using profiling information is smaller than in Jikes RVM because J9’s static heuristics are significantly more aggressive. However, an additional benefit of the dynamic heuristics is that they actually reduce the total amount of inlining; compilation time was reduced by an average of 9% over all benchmarks, helping improve startup performance.

When using the timer-only profiling strategy, the dynamic heuristics hurt performance for 6 of the 8 benchmarks. Our dynamic heuristics reduce inlining at cold call sites, allowing more aggressive inlining to be performed at hot sites where it is likely to be more effective. With inaccurate profile data, inlining can be restricted at too many call sites, causing a nontrivial reduction in performance.

The two inliners used in this study vary substantially, with J9’s inliner relying more heavily on static heuristics, yet our profiling technique produces measurable improvements in both systems. This result helps confirm the general applicability of our technique, and suggests the potential for improving performance in other systems as well.

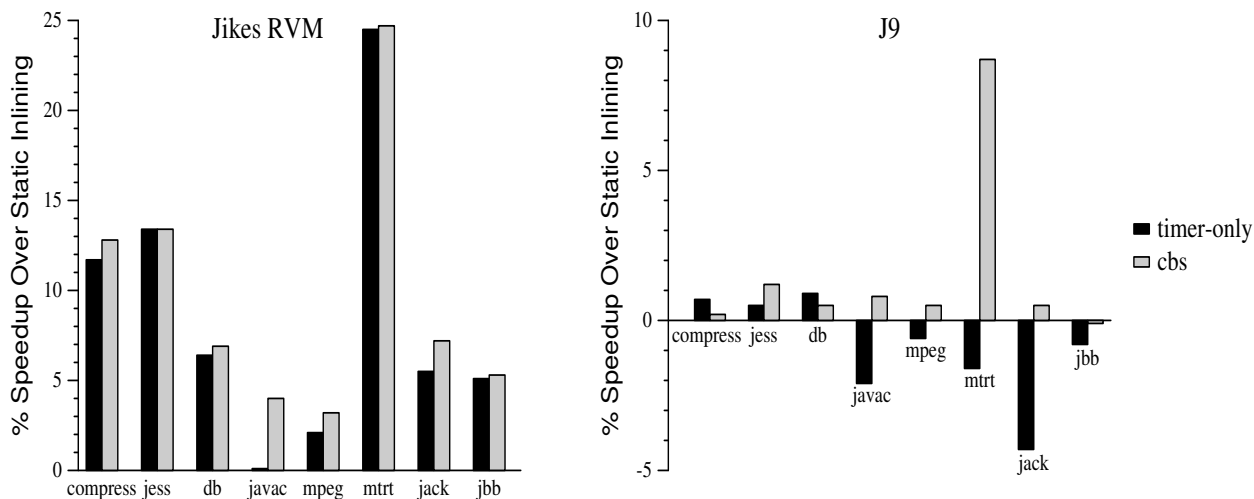


Figure 5. Percentage speedup obtained by profile directed inlining in Jikes RVM (left) and J9 (right) using two sampling mechanisms: timer-only and our new mechanism (cbs).

7 Additional Related Work

Arnold and Ryder [4] describe a technique for reducing the overhead of executing instrumented code. Like our technique, their work uses a count-down timer to trigger samples. However, their technique is part of the compiler and is used for inserting instrumentation into compiled code; our technique is part of the runtime system and does not require modification to the compiler. Hirzel and Chilimbi [14] extended this technique to allow collecting samples in bursts. After a counter is tripped execution remains in the instrumented version of the code until a fixed number of checks are executed; this is similar to our technique, which takes a fixed number of samples after each timer interrupt.

Yasue et al. [28] use a hierarchical structure for enabling and disabling the counters to collect more accurate intraprocedural path profiles. This technique solves a different problem than our technique, but has some similarities in they both collect profiles in controlled bursts.

Timer-based sampling of the dynamic call graph suffers from two main sources of inaccuracy: low sampling rates and the inherent mismatch between a time-based trigger and the desired frequency-based data. Attempts to directly gather the desired profile data in software suffer from large overheads. An alternative approach is to build hardware mechanisms that can efficiently and directly gather the desired profile data. For example, to construct an accurate context-insensitive dynamic call graph it would be sufficient to sample call instructions, capturing the PC of the call instruction and its target PC. Current hardware performance monitoring mechanisms come very close to providing the necessary functionality, but still have some deficiencies. For example, the Pentium 4 contains multiple hardware perfor-

mance monitoring mechanisms that could almost be used to efficiently capture this information [17, 18]. The Pentium 4 supports both low overhead, but somewhat imprecise, and precise, but high overhead, sampling of executed call instructions. An ideal hardware mechanism would be simultaneously low overhead *and* precise. However, with some additional work to compensate for the imprecise nature of the low-overhead sampling mechanism it should be possible to use it to collect a dynamic call graph. An additional issue with the hardware-assisted approach is that the necessary hardware mechanisms are typically specific to a particular micro-architecture. Thus, exploiting them in a virtual machine that must ship on several architectures (and an even larger set of micro-architectures) represents a significant engineering investment.

8 Conclusions and Discussion

This paper presents a new low-overhead sampling technique for collecting high-accuracy dynamic call graph profiles in virtual machines. We implemented and evaluated the technique in two VMs, confirming that it can improve profile accuracy without measurably increasing overhead. We used the collected profiles to drive profile-directed inlining and demonstrated that for both VMs, improvements in profile accuracy resulted in modest, but measurable, performance gains on some programs. On Jikes RVM, the largest improvement was 4% on `javac` and no program was degraded. On J9, the largest improvement was 9% on `mtrt`, and the largest degradation was under 1% on `compress`.

In general, we strongly believe that VM's should exploit the best quality profile information obtainable with low run-

time overhead and reasonable software engineering effort even if it does not immediately produce large speedups. Using inaccurate profile data and tuning on simplistic benchmarks is a dangerous practice, particularly since feedback-directed optimizations have the potential to degrade performance. Having a reliably accurate profiler helps ensure the stability of feedback-directed optimizations on challenging, possibly never seen before applications.

Although this paper focused on the use of the new mechanism for collecting a dynamic call graph, the sampling technique is fairly general. It could be applied any time it is desirable to use low overhead timer-based sampling to collect frequency-based profile data.

9 Acknowledgments

We thank Stephen Fink for technical discussions. We thank Michael Hind, C.J. Newburn, and the anonymous reviewers for their feedback and detailed comments.

References

- [1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, pages 85–96, Las Vegas, Nevada, 15–18 June 1997. *SIGPLAN Notices* 32(5), May 1997.
- [2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, Oct. 2000. Proceedings of the 2000 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'00).
- [3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Architecture and policy for adaptive optimization in virtual machines. Technical Report RC23429, IBM Research, Nov. 2004.
- [4] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 168–179, June 2001.
- [5] B. Calder and D. Grunwald. Reducing branch costs via branch alignment. *ACM SIGPLAN Notices*, 29(11):242–251, Nov. 1994.
- [6] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W.-M. W. Hwu. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience*, 22(5):349–369, May 1992.
- [7] C. Cliff. Personal Communication.
- [8] Colorado Bench. http://www-plan.cs.colorado.edu/henkel/projects/colorado_bench.
- [9] The Daikon dynamic invariant detector. <http://pag.csail.mit.edu/daikon>.
- [10] P. T. Feller. Value profiling for instructions and memory locations. Masters Thesis CS98-581, University of California, San Diego, Apr. 1998.
- [11] N. Grcevski, A. Kilstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Usenix 3rd Virtual Machine Research and Technology Symposium (VM'04)*, May 2004.
- [12] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. *ACM SIGPLAN Notices*, 30(10):108–123, Oct. 1995. Published as part of the proceedings of OOPSLA'95.
- [13] K. Hazelwood and D. Grove. Adaptive online context-sensitive inlining. In *International Symposium on Code Generation and Optimization*, pages 253–264. IEEE Computer Society, 2003.
- [14] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, pages 117–126, Dec. 2001.
- [15] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *5th European Conference on Object-Oriented Programming*, pages 21–38, July 1991.
- [16] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. *ACM SIGPLAN Notices*, 29(6):326–336, June 1994. Published as part of the proceedings of PLDI'94.
- [17] Intel. *IA32 Intel Architecture Optimization Manual*. 2004.
- [18] Intel. *IA32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*. 2004.
- [19] Jikes Research Virtual Machine (RVM). <http://www.ibm.com/developerworks/oss/jikesrvm>.
- [20] Kawa, the Java-based Scheme system. <http://www.gnu.org/software/kawa>.
- [21] M. Paleczny, C. Vick, and C. Click. The Java Hotspot server compiler. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 1–12, Apr. 2001.
- [22] Soot, a Java Bytecode Analysis and Transformation Framework. <http://www.sable.mcgill.ca/software/#soot>.
- [23] Standard Performance Evaluation Corporation. SPECjbb2000 Java Business Benchmark. <http://www.spec.org/jbb2000>.
- [24] Standard Performance Evaluation Corporation. SPECjvm98 Benchmarks. <http://www.spec.org/jvm98>.
- [25] T. Suganuma, T. Yasue, and T. Nakatani. An empirical study of method in-lining for a Java just-in-time compiler. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM'02)*, pages 91–104, Aug. 2002.
- [26] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *ACM 2000 Java Grande Conference*, June 2000.
- [27] Xerces2 Java Parser Readme. <http://xml.apache.org/xerces2-j/index.html>.
- [28] T. Yasue, T. Suganuma, H. Komatsu, and T. Nakatani. An efficient online path profiling framework for Java Just-In-Time compilers. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 148–158, Sept. 2003.