A Framework for Reducing the Cost of Instrumented Code*

Matthew Arnold Barbara G. Ryder

Department of Computer Science
Rutgers University, Piscataway, NJ 08854
{marnold,ryder}@cs.rutgers.edu

Abstract

Instrumenting code to collect profiling information can cause substantial execution overhead. This overhead makes instrumentation difficult to perform at runtime, often preventing many known *offline* feedback-directed optimizations from being used in online systems. We present a general framework for instrumenting code that uses fine-grained sampling to allow previously expensive instrumentation to be performed accurately with low overhead. Our framework does not rely on any hardware or operating system support and is fully tunable; the sample rate can be adjusted at any time to match the type of instrumentation being performed. By reducing the overhead of instrumentation, our framework eliminates one of the biggest obstacles to performing feedback-directed optimizations at runtime. We present experimental results validating the low overhead and high accuracy of our technique.

1 Introduction

The first wave of virtual machines with JIT compilation relied on simple static strategies for choosing compilation targets, typically compiling each method with a fixed set of optimizations the first time it was invoked. Examples of such virtual machines include [1, 14, 19, 25, 33, 39]. A second wave of more sophisticated virtual machines [5, 23, 30, 31, 34] moved beyond this simple strategy by adaptively selecting a subset of all methods for optimization, attempting to focus optimization effort on program hot spots. This *selective optimization* approach avoids the overhead of optimizing all methods, yielding larger performance improvements for shorter running programs [7].

Long running applications, such as server applications, will easily amortize the cost of optimizing all methods, for any reasonable level of optimization. For these applications the most substantial performance improvements will come from *feedback-directed* optimizations, where profiling information is used to decide not only *what* to optimize, but *how* to optimize.

There exists a large body of work on collecting *offline* profiles [3, 11, 12, 16, 27], as well as optimizations based on offline profiles [6, 17, 18, 20, 21, 28]. Some systems [5, 10, 22, 23, 32] apply limited forms of online feedback-directed optimizations, however most of the offline work mentioned above has not yet been applied in fully automated online systems. The main difficulty in applying these optimizations online is that they often rely on instrumenting the code to collect detailed information about program execution, and instrumentation can cause substantial performance degradation. Overheads in the range of 30%–1,000% above non-instrumented code is not uncommon [3, 11, 12, 17, 18, 28], and overheads in the range of 10,000% (100 times slower) has been reported [17].

An online system needs to execute instrumented code for some period of time, prior to performing optimization. The overhead introduced by instrumentation makes this task difficult to perform for several reasons. First, when instrumentation is expensive, the instrumented code must be run for only a short amount of time to keep overhead to a minimum; however, being forced to profile for a very short time period is not desirable because the profile collected may not be representative of overall program behavior. A second problem is that there must be a way to stop the instrumented code from executing, in order to prevent the program from running indefinitely with poor performance. If an instrumented method never exits, on-stack replacement [29] is needed to hot-swap execution back to the non-instrumented version *while the method is running*. Without on-stack replacement, it is dangerous to execute high-overhead instrumented code, because it may degrade performance for an undetermined amount of time. Unfortunately, on-stack replacement is difficult for optimized code.[1]

We solve these problems by presenting a general framework for instrumenting code that allows previously expensive instrumentation to be performed accurately with low overhead, making it practical to perform instrumentation at runtime. Our framework reduces the cost of instrumentation by using *compiler-inserted counter-based sampling* to allow fine-grained switching between instrumented and non-instrumented code. To the best of our knowledge, this is the first general framework that allows arbitrary instrumentation to be performed with low overhead by using fine-grained sampling.

*Funded, in part, by IBM T.J. Watson Research Center and NSF grant CCR-9808607. Currently, Matthew Arnold is an intern at Watson and Barbara Ryder is on sabbatical leave at Watson.

[1] A JVM will most likely perform instrumentation in optimized code, as the execution patterns of optimized code can differ substantially from unoptimized code.

Rutgers University Technical Report
DCS-TR-424, November 2000

Our framework offers the following advantages:

- Instrumentation can be performed for a longer period of time while causing only minimal performance degradation, allowing previously expensive instrumentation techniques to be used at runtime, even without the ability to perform on-stack replacement.
- The framework is tunable, allowing the tradeoff between overhead and accuracy to be adjusted easily at runtime.
- Most instrumentation techniques can be plugged into our framework without needing any modification. Overhead is controlled entirely by the framework, allowing implementors of instrumentation techniques to concentrate on developing new techniques quickly and correctly, rather than focusing on minimizing overhead.
- Multiple types of instrumentation can be used together simultaneously, without the normal concern for overhead. This allows an adaptive system to perform several forms of instrumentation while recompiling the method only once.
- The framework does not rely on any hardware or operating system support.
- The framework is deterministic, which simplifies debugging. Running an application twice will result in identical profiles, assuming the application itself is deterministic.

We validate the framework using the Jalapeño JVM [2] by providing experimental evidence of the overhead and accuracy when applied to two types of instrumentation. Our results show that high accuracy can be achieved (95–99% overlap with a perfect profile) with low overhead, (averaging ∼4% with a naive implementation).

Section 2 describes the instrumentation framework in detail. Section 3 describes two variations designed to reduce the space required by the framework. Section 4 describes an experimental evaluation of the overhead and accuracy of our technique. Sections 5 and 6 discuss related work and conclusions, respectively.

2 Sampling Framework

This section describes our sampling framework. Section 2.1 discusses existing sampling mechanisms, and Section 2.2 describes counter-based sampling, the mechanism used to trigger samples in our framework.

Our framework essentially transforms an instrumented method that has high overhead, into a modified instrumented method that has low overhead. This is accomplished by introducing a second version of the code, called the *checking* code, within the instrumented method, as shown in Figure 1. The checking code is almost identical to the original code, but is modified slightly to allow execution to swap back and forth between the checking code and instrumented code in a fine-grained, controlled manner. On regular sample intervals, execution moves into the instrumented code for a small, bounded amount of time. Overhead is kept to a minimum by ensuring that the majority of execution occurs in the checking code.

The switching between the checking and instrumented code is illustrated in Figure 2. The checking code has conditional branches inserted (which we refer to as *checks*) that

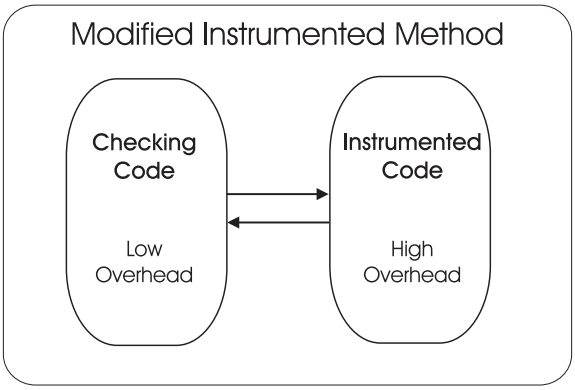

Figure 1: A high level view of an instrumented method generated by our framework. A second version of the code is introduced, called the *checking code*, which is minimally instrumented to allow control to transfer in and out of the instrumented code in fine-grained and controlled manner.

monitor a *sample condition*. When the sample condition is true, a sample is *triggered* and control jumps to instrumented code, rather than continuing in the checking code. Checks are placed on all method entries and backward branches (which will be referred to as *backedges*) in the checking code. This placement of the checks ensures that (a) only a bounded amount of execution occurs between checks, and (b) all code has the opportunity to be sampled.

The instrumented code is also modified so that there are no backedges within the instrumented code (also shown in Figure 2). Instead, all backedges in the instrumented code transfer control back to the checking code, ensuring that only a bounded amount of time is spent in the instrumented code during each sample.

This framework has several desirable properties. First, the ratio of time spent in instrumented code vs. checking code can be controlled by changing the rate at which the sample condition is true. The sample rate can be adjusted, depending on the cost of the particular instrumentation being applied, to keep overhead to a minimum.

An important property of our framework is that the checking code performs checks on method entries and backedges only; thus the number of checks executed is *independent of the instrumentation being performed*. We refer to this property as Invariant 1 for ease of reference.

Invariant 1 *The number of checks executed in the checking code is less than or equal to the number of backedges and methods entries executed, independent of the instrumentation being performed by the instrumented code.*

Our framework allows expensive instrumentation to be performed at runtime even without the ability to perform on-stack replacement. If an instrumented method gets stuck on the stack, setting the sample condition permanently to false will ensure that execution remains in the checking code. Execution will not switch back to a totally non-instrumented version of the code until the method exits (nor can a newly optimized version of the code execute); however, the overhead of the checking code is small enough to be of little concern, especially compared to the cost of instrumentation.

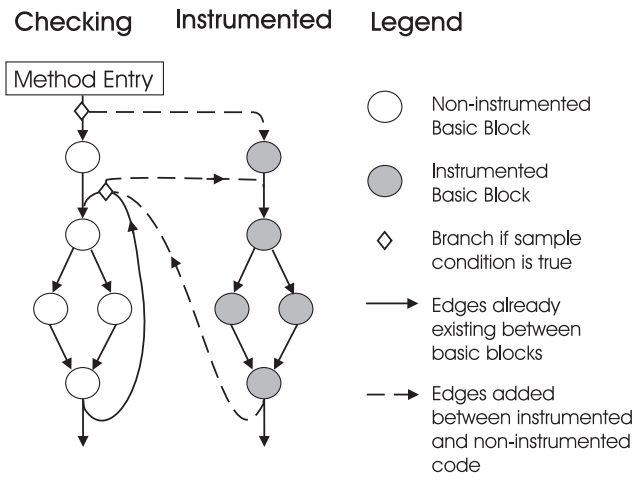

Figure 2: Illustration of the flow of control between the checking code and instrumented code. All method entries and backedges in the checking code contain a conditional branch that jumps to the instrumented code when a sample condition is true. All backedges in the instrumented code are modified to return to the checking code.

Of course, there are some differences between exhaustive and sampled instrumentation. For example, exhaustive instrumentation can be used to establish that a particular event *never* occured during the profiled interval, whereas this is not possible with a sampled profile. Nevertheless, it is unlikely that this type of functionality is useful for an adaptive JVM; even when an event will not occur during the profiling interval, this does not guarantee that the event does not occur in the future. Most common profiling techniques (such as basic block profiling, intraprocedural path profiling, field access profiling, value profiling, etc.) are designed to count some sort of event with the goal of identifying events that occur most frequently. These types of instrumentation need no modification in our sampling framework.

More complex profiling techniques may assume that events are observed exhaustively, such as [3], which updates a context-sensitive data structure on all method entries and exits. Profiling techniques such as these will need to be modified to produce accurate results in our sampling framework; work such as [8, 38] are examples of how this can be achieved.

2.1 Trigger mechanisms

The framework relies on a trigger to determine when execution should transfer into the instrumented code. To keep overhead low, samples must be taken infrequently enough to ensure that the majority of execution remains in the checking code. However, to ensure accuracy, samples must be taken frequently enough to allow a reasonable sample set to be collected. Even more importantly, samples must be triggered in a statistically accurate manner; that is, the basic blocks in the instrumented code must be executed proportionally to their execution frequency in the non-instrumented code.

One approach for triggering samples is to use some type of hardware or operating system timer interrupt. In our framework, timer interrupts could be used to set a "trigger bit" that is monitored by the checks in the checking code. This approach — of checking a timer-set bit — has been used previously (Self-91 [19], Jalapeño [2]) to determine when system services should be performed

One drawback of relying on a timer interrupt is that the sample rate is limited by the frequency of the interrupt, which may be a problem when sampling on the level of basic blocks or instructions. A more serious drawback is that when used in our framework, this technique would not produce a proper distribution of execution in instrumented code. Our framework does not take a sample immediately upon receiving the timer interrupt, but instead jumps to instrumented code only after the next check in the checking code is reached. Any sequence of instructions that executes for a long time (due to an I/O operation, etc.) has a high probability of having a timer interrupt issued during its execution, which, in turn, causes the *next* sequence of instructions to be sampled. Section 4.2 confirms that this improper attribution of samples, as well as the low sample frequency, substantially reduces the accuracy of our framework.

DCPI [4] describes a sampling system that uses interrupts generated by the performance counters on the ALPHA processor, allowing a very high sample rate (5200 samples/sec on a 333-MHz processor). This technique could be incorporated into our framework by using the high frequency interrupt to set the trigger bit that is monitored by the checking code. However, similar to the timer-interrupt, this technique would improperly attribute samples in our framework. In addition, this technique requires hardware performance counters that signal interrupts upon overflow, a feature not available on all architectures.

To obtain an accurate distribution of samples in our framework, the number of times each check (in the checking code) triggers a sample should be proportional to the number of times that particular check is executed. Since we do not know of a performance counter that counts backedges and method entries, our framework performs the counting in software, as described in the next section.

2.2 Compiler-inserted counter-based sampling

Counting a particular event and sampling when the counter reaches a threshold (which we refer to as *counter-based sampling*) is an effective way of triggering samples proportionally to the frequency of that event. This is the fundamental principle behind the accuracy of DCPI [4], where performance counters count instruction cycles, thus sampling instructions proportionally to their execution frequency. However, it may be desirable to sample events for which there is no counter-based hardware interrupt available, as is the case with our framework, which needs to count backedges and method entries. DCPI approximated frequencies of non-counted events (intraprocedural edges) using flow constraints, and showed the accuracy obtained to be inferior to the accuracy of counted events.

To obtain high accuracy when no hardware counting support is available, we propose implementing a counter-based trigger in software by having the compiler insert code to decrement and check a global counter as shown in Figure 3. The counter variable is named `globalCounter` to emphasize that there is only one counter for the whole program. We call this technique *compiler-inserted counter-based sampling*. There are several options for implementing such an approach; the simplest is to execute the code exactly as shown in Figure 3 each time an event occurs. The counter

```
globalCounter--;
if (globalCounter == 0) {
   takeSample();
   globalCounter = resetValue;
}
```

Figure 3: Code inserted for a counter-based check

variable (`globalCounter`) will most likely be in a register, or in the cache, and the branch will be predicted (not taken), therefore the performance overhead should be low. Such an approach was implemented in Jalapeño, placing the code in Figure 3 on all backedges and method entries; the checking overhead averaged 3.6% (for executing the checks only, when no samples were taken). A detailed description of the overhead is included in Section 4.2.

As long as the overhead of the counting and checking is kept to a minimum, the advantages of compiler-controlled counter-based sampling are numerous. First, it is extremely simple to implement, and allows high frequency sample rates that can be adjusted at runtime. Second, it does not rely on any hardware or operating system support.[2] Finally, counter-based checks trigger samples deterministically, creating reproducible sampling results, aiding in debugging.

Certain architectures may have instructions that can be used to reduce the cost of counter-based checks. For example, the PowerPC architecture has a decrement-and-check instruction that decrements a count register, compares it against zero, and performs a conditional branch, allowing the code in Figure 3 to be executed in one instruction. VLIW and superscalar architectures may also be able to hide the cost of the checks in unused cycles.

There may also be compiler specific techniques that can further reduce the overhead of the checks. For example, in Jalapeño, all backedges and method entries contain *yield-points* that check whether it's time for the executing code to yield to the thread scheduler. The PowerPC decrement-and-check instruction described above can be used to implement the semantics of both the counter-based check *and* the yield point, without increasing the number of instructions executed.[3] Although this example is specific to Jalapeño, the counter-based checks are simple enough that, by using any available hardware instructions and compiler-specific techniques, it may be possible to implement the checks with extremely low, or even no overhead.

3 Space saving variations

The framework described in Section 2 uses two versions (checking and instrumented) of each instrumented method, thus requiring twice as much space. The instrumentation should not affect locality, because the instrumented code is executed infrequently and can be placed somewhere out of the common path; however, doubling the size of the method will double the space consumed by the instrumented code, and also increase (but not double[4]) compile time.

[2] Although hardware and O.S. techniques may be used to lower the overhead of the checks, no support from either is required.

[3] Some existing uses of the PowerPC count register would need to be modified, which could potentially introduce overhead indirectly.

[4] If the instrumentation is performed on optimized code, the "doubling" of all nodes could be performed after optimization has taken place. Moreover, instrumentation is likely to be performed in code that is optimized at the highest level, thus the cost of doubling all nodes would be small compared to optimization time.

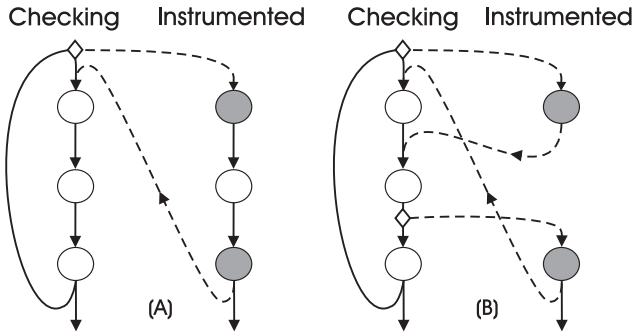

Figure 4: Removing non-instrumented nodes from the instrumented version can increase the number of checks executed. Dark nodes represent basic blocks containing instrumentation. (A) represents an instrumented version with all basic blocks duplicated. (B) shows an instrumented version where the non-instrumented node is not duplicated. Note the extra check (diamond) in the checking version of (B).

In scenarios where instrumentation is sparse, ideally only those nodes containing instrumentation would need to be duplicated. Unfortunately, it is not always possible to remove a non-instrumented node from the instrumented code without violating Invariant 1. As shown in Figure 4, when the non-instrumented node is removed from the instrumented code, an additional check must be added to allow the second instrumented node to be sampled.

There are many possible space saving variations of our framework. Two such variations, Variation-1 and Variation-2, are presented below. The original framework described in Section 2 will hereafter be referred to as Variation-0.

3.1 Variation-1

The goal of Variation-1 is to remove as many non-instrumented basic blocks from the instrumented code as possible without violating Invariant 1. Two types of nodes in the instrumented code are defined: *top-nodes* and *bottom-nodes*, both of which can be removed from the instrumented code without invalidating Invariant 1. Both types of nodes are defined on the *instrumented code DAG*, which is the instrumented code with all backedges removed.

A **bottom-node** is defined to be any non-instrumented node, n, in the instrumented code DAG such that no instrumented nodes are reachable from n.

All bottom-nodes can be removed from the instrumented code without violating Invariant 1, because once a bottom-node is executed, no further instrumentation will be performed without returning to the checking code first. Any edge in the instrumented code that previously connected an instrumented node to a bottom-node needs to be adjusted to branch to the corresponding node in the checking code.

A **top-node** is defined to be any non-instrumented node, n, in the instrumented code DAG such that no path from entry to n contains an instrumented node. All top-nodes can be removed from the instrumented code without violating Invariant 1; however, the following two adjustments must be made:

1. In the checking code, all checks that branch to a top-node should be removed.

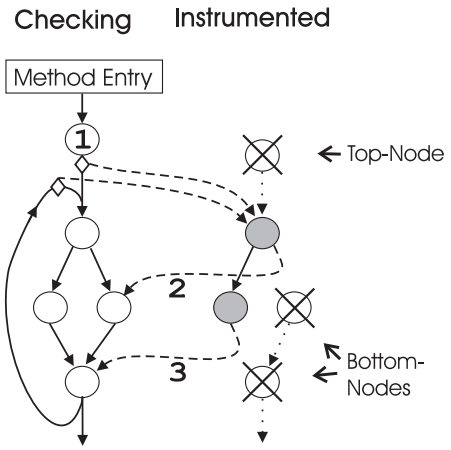

Figure 5: Example of Variation-1 after top-nodes and bottom-nodes are removed. The check after method entry is removed because it branched to a top-node. A check is added to the edge exiting the basic block labeled "1" because the corresponding edge in instrumented code connected a top-node to an instrumented node. The edges labeled "2" and "3" now lead back to checking code because they previously connected an instrumented node and a bottom-node.

2. In the instrumented code, for every edge that previously connected a top-node to an instrumented node, the corresponding edge in the checking code should have a check added.

Figure 5 revisits the code from Figure 2, but with Variation-1 updates applied, assuming that only the two shaded nodes contain instrumentation. The check after method entry is removed because it would have branched to a top-node. A check is added to the edge exiting the basic block labeled "1" because the corresponding edge in the instrumented code connected a top-node to an instrumented node. In this particular example, the two checks can be combined into one. The edges labeled "2" and "3" now lead back to checking code because they previously connected an instrumented node and a bottom-node.

This technique eliminates as many nodes as possible without violating Invariant 1. Although the static number of checks may increase or decrease, the dynamic number of checks executed is less than or equal to the number executed with Variation-0. Instrumentation is performed identically to Variation-0.

3.2 Variation-2

If Invariant 1 can be weakened, there are many other alternatives for reducing code duplication; any non-instrumented node can be removed from the instrumented code, as long as the appropriate checks are added in the checking code. In fact, by guarding all instrumentation operations with checks, there is no need to duplicate any code. Such an approach will be referred to as *Variation-2*, and is shown in Figure 6, which illustrates one basic block with two instrumented instructions. Although none of the instructions themselves are duplicated, all instructions with associated instrumentation must check the sample condition before executing the instrumentation.

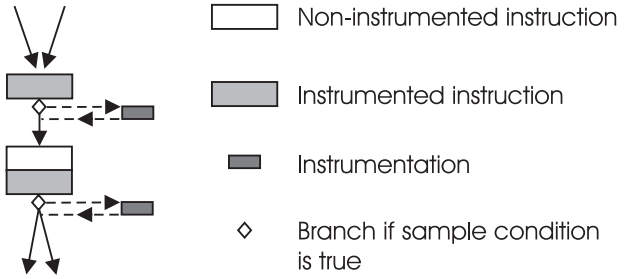

Figure 6: Example of Variation-2, showing one basic block containing two instrumented instructions. No code is duplicated, but checks are placed on all instrumentation operations. Invariant 1 is violated since there could be more than one check per loop iteration. However, the number of checks executed could also be *less* than with Variation-0 if instrumentation is sparse.

Unlike Variation-1, Variation-2 will *not* perform instrumentation identical to Variation-0. With Variation-0, a sample causes execution in that method to remain in instrumented code until the next backedge is reached, whereas in Variation-2, a sample triggers only one instrumentation operation to be performed. Although they perform the instrumentation in a slightly different manner, they both execute the instrumented instructions proportionally to their execution frequency, resulting in accurate sampling results, as demonstrated empirically in Section 4.2.

The only drawback of Variation-2 is that it may execute more checks at runtime than the previous variations. The overhead for executing the additional checks introduced by this technique may be significant if there are a substantial number of instrumentation operations per loop iteration. However, the number of checks executed could also be reduced if instrumentation operations occur less frequently than backedges and method calls. In any case, the overhead is likely to be less than the overhead of full instrumentation (as shown in Section 4.2), making Variation-2 useful when a reduction in space is important. Variation-2 is also the easiest to implement, as it involves no code duplication.

Combining Variation-1 and Variation-2 is also a possibility, allowing some code to be duplicated, while executing some additional checks at runtime. Exactly which variation (or a combination thereof) should be used depends on the type of instrumentation being performed and the time and space constraints that must be satisfied. A reasonable approach for a JVM would be to fill the instrumented version with all desired instrumentation, and let it run for a while at a low sample rate. Variation-0 would probably be the best choice for this scenario, since most of the basic blocks would contain instrumentation.

4 Experimental Results

To assess the feasibility of our framework, experiments were performed to measure both the accuracy and runtime overhead, when applied to two example instrumentations.

4.1 Methodology

Our experiments were performed using the Jalapeño JVM [2, 5] being developed at IBM T.J. Watson Research

Center. Currently Jalapeño contains two fully operational compilers, a non-optimizing *baseline* compiler and an *optimizing compiler* [14]. Jalapeño is written in Java, and begins execution by reading from a *boot image* file, which contains the core services of Jalapeño precompiled to machine code.

Our benchmark suite consists of the SPECjvm98 [24] benchmarks with input size 10, the Jalapeño optimizing compiler [14] run on a small subset of itself, the Volano benchmark [37], and the pBOB benchmark [13]. The running times of the benchmarks ranged from 1.1 to 4.8 seconds. To show that our framework can collect accurate profiles in a short amount of time, short running benchmarks were chosen and all profiles used for accuracy comparisons were collected using a single run. All overhead timings were collected using the median of 10 runs to eliminate noise from external processes. The benchmarks range in cumulative class file sizes from 10,156 (`209_db`) to 1,516,932 (`opt-cmp`) bytes. All results were gathered on a 333MHz IBM RS/6000 PowerPC 604e with 1048MB RAM, running AIX 4.3.

Our sampling framework is evaluated using the following two examples of instrumentation:

1. **Call-edge instrumentation** All calls are instrumented to record the call edge, which consists of the caller method, the callee method, and the call-site within the caller method (specified by a bytecode offset). A counter is maintained for each call edge, and the counter is incremented on each call.

2. **Field-access instrumentation**[5] A counter is maintained for each field of all classes. All field accesses (generated from a `get_field` or `put_field` bytecode instruction) are instrumented to increment the counter for the field they are accessing. This type of profile is useful for cache-conscious data layout [17, 18, 21].

All overhead and accuracy data reported (both exhaustive and sampled) were collected by instrumenting all methods in the benchmark, including library methods, however methods in the Jalapeño boot-image were not instrumented. An adaptive JVM would most likely instrument just a few of the hottest methods, so instrumenting all methods represents a worst case scenario regarding overhead. All code (both instrumented and non-instrumented) was optimized pre-execution at level *O2*, which is currently Jalapeño's highest optimization level [5].

4.2 Results

Table 1 characterizes the two profiling techniques used in this study by showing their overhead when applied exhaustively (not using our framework) so that all methods spend 100% of the time in the instrumented code. The first column lists the benchmarks, while the second and third columns show exhaustive instrumentation overhead for call-edge and field-access instrumentation respectively. The call-edge instrumentation averages 77.2% overhead, and the field-access averages 77.6% overhead. Clearly, these instrumentations are too expensive to execute unnoticed at runtime.

Checking code overhead The overhead of the checking code was measured by comparing the running time of the checking code only (the trigger was set permanently to false, so execution never moved into the instrumented code) to the running time of the original code, thus capturing the overhead of the checks. A full breakdown of the checking overhead is shown in Table 2. Column 1 lists the benchmarks, and columns 2–4 show the execution time overhead when using Variation-0.[6] The column labeled *All Checks* shows the overhead when checks are inserted on both backedges *and* method entries, therefore representing the total cost of the checking code. The average combined overhead is 3.6%, which is quite low, especially compared to the cost of full instrumentation. The columns labeled *Backedges only* and *Method entry only* report a breakdown of the overhead when checks are placed only on backedges or method entries, respectively. Although column 2 is roughly the sum of columns 3 and 4, the overhead is not exactly additive. This could be due to complex downstream effects of the inserted checks (such as increased cache misses), or simply due to noise in the timing data. It is difficult to accurately measure such small overheads; an example of noise in the timing is shown by `compress`, which yielded a negative method entry overhead.

Benchmark	Call-edge (%)	Field-access (%)
201_compress	76.2	288.2
202_jess	142.4	90.1
209_db	5.3	15.3
213_javac	75.9	15.7
222_mpegaudio	133.5	58.9
227_mtrt	121.8	77.1
228_jack	28.4	115.2
opt-compiler	79.3	59.6
pBOB	75.3	49.3
Volano	33.9	6.3
Average	77.2	77.6

Table 1: Time overhead of exhaustive instrumentation without our framework (all execution occurs in instrumented code) compared to the original, non-instrumented code.

Although averaging ∼4% overhead is very reasonable, it does *not* represent a lower bound, as it could easily be improved upon by using a number of techniques. First, our checking implementation does *not* use any of the PowerPC- or Jalapeño-specific techniques mentioned in Section 2.2. Furthermore, Jalapeño does not currently perform loop unrolling, which would substantially reduce the backedge overhead of the loop intensive applications (such as `compress` and `mpegaudio`). Similarly, method entry overhead could be reduced by performing more aggressive inlining before instrumentation occurs (these experiments were run using the default, non-aggressive static inlining heuristics).[7]

The fifth column, labeled *Maximum space overhead*, shows the approximate space overhead of Variation-0 when all methods are instrumented. Variation-0 doubles the size of each instrumented method, so the maximum space overhead is computed by summing the sizes of the final optimized code for all methods. However, a JVM would need less than this space because it would most likely instrument only a few of the hottest methods, not necessarily all at the same time. If space is limited, Variation-0 can be used by limiting the number of methods that are instrumented

[5]We would like to thank Sharad Singhai and Peter Sweeney from IBM Research for the use of their field-access profiling implementation.

[6]Recall that the checking overhead of Variation-1 is less than or equal to the checking overhead of Variation-0.

[7]It is reasonable to expect an adaptive system to apply expensive instrumentation only after simpler optimizations have been applied.

Benchmarks	Variation-0				Variation-2	
	All Checks (%)	Backedges Only (%)	Method entry Only (%)	Maximum space Overhead (KB)	Call-edge (%)	Field-access (%)
201_compress	5.9	3.1	-2.5	40	-2.5	102.1
202_jess	6.3	4.2	2.3	75	2.3	55.7
209_db	*	*	*	45	*	3.5
213_javac	1.3	0.6	2.1	128	2.1	14.2
222_mpegaudio	8.4	7.9	0.9	157	0.9	52.7
227_mtrt	0.9	0.6	*	57	*	60.1
228_jack	6.1	4.3	*	87	*	43.2
opt-compiler	2.6	1.6	1.5	103	1.5	48.3
pBOB	2.4	*	2.7	300	2.7	39.1
Volano	2.7	0.6	1.4	36	1.4	1.4
Average	3.6	2.3	0.8	84	0.8	41.2

Table 2: Time time overhead of the checking code (execution never leaves checking code – overhead is for the checks only) compared to the original, non-instrumented code, for both Variation-0 and Variation-2. A "*" is shown for overheads whose absolute value is less than 0.5%. The fifth column, *Maximum space overhead*, represents the space increase when all methods are duplicated.

simultaneously. When used selectively, the space requirements of Variation-0 could easily be less of a concern than other space consuming transformations, such as inlining and specialization, which can cause potentially exponential code growth.

The last two columns of Table 2 show the execution time overhead of the checking code when using Variation-2, for both call-edge and field-access instrumentations. Recall that Variation-2 does not guarantee to maintain Invariant 1, and thus the overhead in the checking code may be higher or lower than the checking overhead of Variation-0. For the call edge instrumentation, the average overhead is 0.8% – less than the checking overhead of Variation-0. This is because for the call-edge instrumentation, Variation-2 performs checks on all call edges only (i.e., method entries), explaining why columns 4 and 6 are identical. However, the average overhead of the field-access instrumentation is 41.2%. Although this is less than the overhead of the exhaustive field-access instrumentation, it is significantly higher than the overhead for Variation-0, because unlike Variations 0 and 1, the checking overhead of Variation-2 increases as more instrumentation is added to the instrumented code.

Instrumentation overhead and accuracy Next, the overhead and accuracy of the actual instrumentation (as opposed to just the checking code) are evaluated when sampled by our framework. We implemented Variation-2 and a prototype simulated version of Variation-0, that does not actually duplicate the code as described in Section 2. Instead the checking code turns on an instrumenting flag that is checked by all instrumentation operations. A separate flag is maintained for each method, to properly simulate the behavior of Variation-0. This approach produces instrumentation results identical to Variation-0, but at the cost of increased checking overhead.[8]

To compute the overhead of the sampled instrumentation, the running time of the sampling version is compared to the running time of the checking code only (where no samples are triggered). The performance difference represents the time spent performing sampled instrumentation, and does not include the overhead of the checks. The checking overhead is not included in these numbers because it is independent of the instrumentation overhead, and can vary depending on the implementation (as discussed previously).

To assess accuracy, profiles collected at different sample rates are compared against a perfect profile (which is collected by allowing 100% of the execution to remain in instrumented code) and an accuracy metric is computed. An *overlap percentage* metric is used, similar to that used in [27]. Informally, the overlap of two profiles represents the percent of profiled information weighted by execution frequency, that exists in both profiles. The following example defines more specifically how the metric is computed.

Figure 7 helps describe the overlap metric by visually representing the call-edge profile for `javac`. Each bar represents a call edge, and the y-axis represents the *hotness* of the edge, defined as the percentage of all samples attributed to that edge. The height of each bar shows the hotness of the edge according to the perfect profile, while each circle (either within or above each bar) shows the hotness of that edge according to a sampled profile. The *overlap* for an edge is simply the minimum of the two hotness values. The *overlap percentage* of the benchmark is the sum of the overlaps for all edges. A sampled profile that is identical to the perfect profile yields an overlap percentage of 100%. Any variation in the sampled profile from the perfect profile yields an overlap percentage of less than 100%.[9] Overlap for the field-access profile was computed in the same way, but using field hotness rather than call-edge hotness. The `javac` profile shown in Figure 7 yields an overlap of 93.8%, showing that an overlap of this value corresponds to very accurate profile.

Table 3 shows the overhead and accuracy of the sampled instrumentation, averaged over all benchmarks, for several different sample rates. Because samples are triggered by counter-based sampling, sample rates are reported as *sample intervals*, as shown in the first column of the table. The

[8] This simulated version may cause the overhead numbers to be inaccurate to a certain degree, but this is not significant because the overhead drops close to zero very quickly for both Variation-0 and Variation-2. More importantly, the accuracy results are identical to those that would be obtained by an actual implementation of Variation-0.

[9] Edge hotness in the perfect profile sums to 100%. Likewise, edge hotness in the sampled profile sums to 100%. Therefore, if the sampled profile overestimates the hotness of some edge, it must also underestimate the hotness of another edge.

Sample Interval	Call-edge					Field Accesses				
	Variation-0		Variation-2			Variation-0		Variation-2		
	Overhd (%)	Accur. (%)	Overhd (%)	Accur. (%)	Num Samples	Overhd (%)	Accur. (%)	Overhd (%)	Accur. (%)	Num Samples
1	67.0	100	69.7	100	$2.0x10^6$	69.7	100	78.6	100	$3.8x10^7$
10	8.7	99	8.8	98	$2.0x10^5$	6.9	99	10.2	98	$3.9x10^6$
100	1.7	98	0.9	97	$2.0x10^4$	1.9	99	2.3	96	$3.9x10^5$
1,000	1.0	96	*	95	2000	*	97	1.7	95	$3.9x10^4$
10,000	*	89	*	87	200	*	94	*	93	3900
100,000	*	72	*	69	22	*	83	*	81	354

Table 3: Time overhead and accuracy of the sampled instrumentation. *Overhd* shows overhead normalized with respect to the checking code, and thus does not include the checking overhead. A "*" is shown for overheads whose absolute value is less than 0.5%. *Accur.* shows accuracy as an *overlap percentage*. *Num Samples* shows the average number of samples collected at that sample interval when using Variation-2; the number of samples taken when using Variation-0 are almost identical.

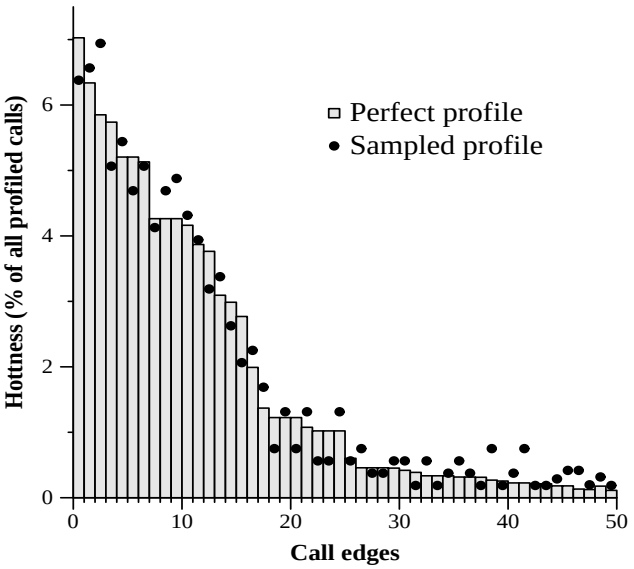

Figure 7: A graphical representation of the `javac` call-edge profile, illustrating an accuracy of 93.8% using the *overlap percentage* metric.

sample interval represents the number of checks in the checking code that are executed before each sample is triggered. A sample interval of 1000 means that roughly $1/1000^{th}$ of the execution will occur in instrumented code. The next five columns show data for call-edge profiling, and the last five show data for field-access profiling. Each profiling type contains a breakdown of the overhead for both Variation-0 and Variation-2. Variation-2 for each instrumentation type also includes a column labeled *Num Samples*, which shows the average number of samples taken for each sample interval. The number of samples taken is almost identical for both Variation-0 and Variation-2, thus data is reported for Variation-2 only.

As would be expected, increasing the sample interval reduces both overhead and accuracy. At sample interval 1,000 there is practically no overhead for the instrumentation (excluding checking overhead), and the accuracy is at or above 95% for both variations and both instrumentations. Even at sample interval 10,000, the accuracy is good, even though only $1/10,000^{th}$ of the execution is spent in instrumented code. The accuracy finally degrades at sample interval 100,000 where there are simply not enough samples collected, given the short running time of the benchmarks. When examining this table, one must keep in mind that the sample intervals are increasing exponentially, and that there is actually a huge range of sample intervals (from 100 to 10,000) that offer extremely high accuracy with almost no overhead. An area of future work is to investigate whether adding a small random factor to the sample interval can increase accuracy.

Trigger Mechanisms As discussed in Section 2.2, it is possible to use triggers different than a counter-based trigger. To show the advantages of the counter-based trigger, its accuracy is compared against a time-based trigger. Jalapeño has a *threadswitch* bit that is set every 10ms by a hardware interrupt, and is used to determine when the executing code should yield to the thread scheduler. This threadswitch code was modified to also trigger a sample, using Variation-0 for the field-access instrumentation. To make a fair comparison, a sample interval of 30,000 was used for the counter-based sampling, because it resulted in approximately the same number of samples as the time-based trigger for these benchmarks.

Table 4 compares the accuracy of both techniques (when compared against a perfect profile using overlap percentage) for all benchmarks. Clearly, counter-based sampling is more accurate, averaging 84% accuracy, as opposed to time-based sampling, which averaged 63%; this difference in accuracy is most likely due to the inaccurate attribution of samples by the time-based trigger, as discussed in Section 2.1. Another accuracy advantage of the counter-based trigger is that it allows the sample interval to be increased. As previously shown in Table 3, a shorter sample interval of 100 or 1000 achieves a much higher accuracy (95–99%), while overhead remains near zero.

5 Additional Related Work

We do not know of any framework based on sampling that allows general instrumentation to be performed at low overhead. The work closest to ours is that of Traub et al. [35] which describes using *ephemeral instrumentation* to collect intraprocedural edge profiles. Their technique allows efficient edge profiling by dynamically rewriting conditional branches in the executing code, causing execution to jump to instrumentation code that increments the frequency of

Benchmark	Time-based (%)	Counter-based (%)
201_compress	88	98
202_jess	91	95
209_db	66	95
213_javac	59	73
222_mpegaudio	69	95
227_mtrt	51	67
228_jack	45	94
opt-compiler	58	65
pBOB	75	87
Volano	27	71
Average	63	84

Table 4: Comparing the accuracy (overlap percentage) of field access instrumentation when samples are driven by a time-based and counter-based trigger.

the executed edge. When used to guide superblock scheduling, the edge profiles collected by this technique produced speedups similar those from a perfect profile, although a few benchmarks showed substantial differences.

Unlike our framework, it is not clear how this technique can be used to perform general instrumentation. An interesting approach would be to combine both techniques, using branch rewriting to trigger the jump from checking code to instrumented code, thus eliminating the overhead of the checking code when samples are not being taken. However, there are other sources of overhead, such as maintaining cache consistency, that are introduced by the complexity of binary rewriting.

Dynamo [10] employs a technique named *NET (Next Executing Tail)* [26] to reduce the profiling overhead of identifying hot paths. Using NET, when counters on *start-of-trace* points exceed a threshold, the next executing trace is recorded and assumed to be hot. Although NET potentially introduces more noise [26] than traditional path profiling techniques, it identifies hot paths quickly, making it an effective choice for Dynamo, where all code is interpreted until identified as part of a hot path, and then optimized.

There are several fundamental differences between our framework and NET. First, our framework assumes the execution environment of a JVM, where code can be optimized either with or without profiling information. Because reasonable performance can be expected prior to collecting instrumentation, there is less need to base optimization decisions on a single sample. Instead, multiple samples can be collected over a longer time period to increase accuracy. Second, NET is specific to identifying hot paths, while our framework allows a variety of instrumentation to be performed. Finally, NET uses multiple counters to exhaustively record the execution frequency of traces, whereas our counter-based sampling uses one counter to distribute samples across all sample points.

Self-93 [30] used method invocation counters to determine when the call stack should be sampled. Similar to NET, Self-93 used multiple counters (one per method) and made optimization decisions based on a single sample.

Previous work [17, 27, 36] has used the idea of wrapping each instrumentation operation inside a conditional branch (as is done in our Variation-2). Calder and Feller [17] describe *convergent value profiling*, where profiling is turned off once the profiled values appear to have converged; their technique is compared against a random sampling, where (exhaustive) sampling is turned off for periods of random length. Viswanthan and Liang [36] describe a *Java virtual machine profiler interface*, a general purpose mechanism for obtaining information from a JVM, supporting both sampling and instrumentation, but not combining the two. However, unlike our work, these approaches do not use the conditional checks to perform fine-grained sampling; instead, the conditional is used as a switch to turn exhaustive profiling on and off for a given period of time.

There has been work on reducing the cost of specific types of instrumentation [11, 12, 16, 27]; however, these techniques are specific to one kind of instrumentation, and it is unclear whether they reduce overhead enough to allow the instrumentation to run unnoticed in an adaptive system.

Recent work [8, 38] has used sampling to reduce the cost of building the *calling context tree* [3], while [15] uses sampling to reduce the cost of profiling program values. These techniques are a specialized examples of the general technique proposed by our framework, that is, using sampling to reduce the overhead of a previously exhaustive instrumentation.

Aron et al. [9] describes *soft timers*, an operating system facility that allows efficient processing of high frequency software events, by strategically scheduling them at times when overhead will be minimized. This is orthogonal to the goal of counter-based sampling, which triggers high frequency samples to achieve accurate sampling.

6 Conclusions

We have presented a tunable framework for instrumenting code that allows previously expensive instrumentation to be performed with low overhead. The framework uses compiler-inserted counter-based sampling to switch between instrumented and non-instrumented code in a controlled, fine-grained manner. Our sampling technique does not rely on any hardware or operating system support, yet provides a flexible, high frequency sample rate, ensuring accurate and deterministic sampling results.

The reduction in overhead provided by our sampling framework allows instrumentation to be performed for a longer period of time, while causing only minimal performance degradation, allowing a system to utilize expensive instrumentation techniques even without the ability to perform on-stack replacement. Our framework also makes it possible for multiple types of instrumentation to be combined together at once, without the normal concern for overhead. This is a very attractive approach for an adaptive JVM, as it would allow several forms of instrumentation to be performed at the same time while requiring the method to be recompiled only once. Because overhead is controlled completely by the framework, implementors of instrumentation techniques are no longer required to focus on minimizing overhead, but instead can concentrate on other issues surrounding online feedback-directed optimization.

Using the Jalapeño JVM with two different types of instrumentation, we have shown experimentally that our technique can achieve excellent accuracy (achieving a 95–99% overlap with a perfect profile) with low overhead (on average ~4%).

Acknowledgments We would like to thank David Grove, Michael Hind, and Peter Sweeney for comments on an earlier draft of this work. We would also like to thank Michael Hind for his support of this work, and all of the Jalapeño members who helped facilitate our research.

References

[1] A.-R. Adl-Tabatabai, M. Cierniak, C.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a Just-in-Time Java compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 280–290, Montreal, Canada, 17–19 June 1998. *SIGPLAN Notices* 33(5), May 1998.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[3] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN '97 Conf. on Programming Language Design and Implementation*, 1997.

[4] J. M. Andersen, 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? Technical Note 1997-016a, Digital Systems Research Center, www.research.digital.com/SRC, Sept. 1997.

[5] M. Arnold, S. 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*, Oct. 2000.

[6] M. Arnold, S. Fink, V. Sarkar, and P. Sweeney. A comparative study of static and dynamic heuristics for inlining. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, 2000.

[7] M. Arnold, M. Hind, and B. G. Ryder. An empirical study of selective optimization. In *13th International Workshop on Languages and Compilers for Parallel Computing*, Aug. 2000.

[8] M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. Technical Report RC 21789, IBM T.J. Watson Research Center, July 2000.

[9] M. Aron and P. Druschel. Soft timers: efficient microsecond software timer support for network processing. In *In Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 232–246, 1999.

[10] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[11] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[12] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57. ACM Press, 1996.

[13] S. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. Munroe. Java server benchmarks. *IBM Systems Journal*, 39(1), 2000.

[14] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.

[15] M. Burrows, U. Erlingson, S.-T. A. Leung, M. T. Vandevoorde, C. A. Waldspurger, K. Walker, and W. E. Weihl. Efficient and flexible value sampling. In *To Appear: Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[16] B. Calder, P. Feller, and A. Eustace. Value profiling. In *the 30th International Symposium on Microarchitecture*, pages 259–269, Dec. 1997.

[17] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, Vol 1, Mar. 1999.

[18] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, San Jose, California, Oct. 1998. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society.

[19] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, Nov. 1991. *SIGPLAN Notices* 26(11).

[20] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software – Practice and Experience*, 22(5):349–369, May 1992.

[21] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 13–24, Atlanta, May 1999. ACM Press.

[22] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the International Symposium on Memory Management (ISMM-98)*, volume 34, 3 of *ACM SIGPLAN Notices*, pages 37–48, New York, Oct. 17–19 1999. ACM Press.

[23] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[24] T. S. P. E. Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1998.

[25] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *11th Annual ACM Symposium on the Principles of Programming Languages*, pages 297–302, Jan. 1984.

[26] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *To Appear: Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[27] P. T. Feller. Value profiling for instructions and memory locations. Masters Thesis CS98-581, University of California, San Diego, Apr. 1998.

[28] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–123, Oct. 1995.

[29] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 32–43, San Francisco, California, 17–19 June 1992. *SIGPLAN Notices* 27(7), July 1992.

[30] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems*, 18(4):355–400, July 1996.

[31] The Java Hotspot performance engine architecture. White paper available at http://java.sun.com/products/hotspot/whitepaper.html, Apr. 1999.

[32] T. P. Kistler. *Continuous Program Optimization.* PhD thesis, University of California, Irvine, 1999.

[33] A. Krall. Efficient JavaVM Just-in-Time compilation. In J.-L. Gaudiot, editor, *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, Oct. 1998.

[34] T. Suganama, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time compiler. *IBM Systems Journal*, 39(1), 2000.

[35] O. Traub, S. Schechter, and M. D. Smith. Ephemeral instrumentation for lightweight program profiling. Technical report, Harvard University, 1999.

[36] D. Viswanathan and S. Liang. Java Virtual Machine Profiler Interface. *IBM Systems Journal*, 39(1):82–95, 2000.

[37] VolanoMark 2.1. `http://www.volano.com/benchmarks.html`.

[38] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *ACM 2000 Java Grande Conference*, June 2000.

[39] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman. LaTTe: A Java VM Just-in-Time compiler with fast and efficient register allocation. In *International Conference on Parallel Architectures and Compilation Techniques*, Oct. 1999.