A Framework for Reducing the Cost of Instrumented Code

Matthew Arnold* Barbara G. Ryder
Rutgers University
{marnold,ryder}@cs.rutgers.edu

ABSTRACT

Instrumenting code to collect runtime profiling information can substantially degrade performance, making instrumentation difficult to perform at runtime in an adaptive system. We present a general framework for instrumenting code that uses fine grained sampling to allow 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 adjusted, at any time, to match the type of instrumentation being performed. We present experimental results validating the overhead and 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 [1,13,17,22,28,32]. A second wave of more sophisticated virtual machines [5,21,26,27,29] 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 [6].

However, 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. Instrumentation is a common way of figuring out how to optimize. The overhead of instrumented code can make it difficult, or even impossible, to perform instrumentation at runtime in an adaptive system.

In this work we present a general framework for instrumenting code that allows previously expensive instrumentation to be performed accurately with low overhead, averaging ~6% using our naive implementation. This is accomplished by using *compiler controlled counter-based sampling* to allow fine-grained switching between instrumented and non-instrumented code. To the best of our knowledge, this is the first framework that allows arbitrary instrumentation to be performed with low overhead by using fine grained sampling.

The reduction in overhead provided by our sampling framework has the following advantages:

- The framework does not rely on any hardware or operating system support.
- The framework is flexible and tunable. It allows the sampling rate to be modified at any time, including being varied at runtime during the same execution.
- Instrumentation can be performed for a longer period of time, while causing only minimal performance degredation.
- Expensive instrumentation techniques can be used at runtime even without the ability to perform stack frame rewriting.
- Overhead is controlled completely by the framework, allowing implementors of instrumentation techniques to concentrate on developing new techniques quickly and correctly, rather than trying to minimize overhead.
- Multiple types of instrumentation may be combined together at once, without the normal concern for overhead. This is a very attractive approach for an adaptive system, as it would allow several forms of instrumentation to be performed at the same time while requiring the method to be recompiled only once.
- Most instrumentation techniques do not need to be modified, and are simply "plugged into" the framework.

We validate the framework by providing experimental evidence of the overhead and accuracy when applied to two examples of instrumentation. Our results show that high accuracy can be achieved (98–99% overlap with a perfect profile) with low overhead (~6% even with a naive implementation).

*Thank IBM for their support here. RU grants also.

Section 3 describes the instrumentation framework in detail. Section 4 describes two variations designed to reduce the space required by the framework. Section 5 describes an experimental evaluation of the overhead and acuracy of our technique. Sections 6 and 7 discuss related work and conclusions, respectively.

2. BACKGROUND

There exists a large body of work on collecting *offline* profiles [3,11,12,14,23], as well as optimizations based on offline profiles [15, 16, 18, 19, 24]. Most of these techniques instrument the code to collect detailed information about program execution. Unfortunately, instrumentation can cause substantial performance degredation. Overhead in the range of 30%–1,000% over non-instrumented code is not uncommon [3, 11, 12, 15, 16, 24], and sometimes overhead in the range of 10,000% (100 times slower) is reported [15].

The overhead of executing instrumented code can make it difficult, or even impossible, to perform instrumentation at runtime in an adaptive system. Some online systems [5, 20, 21] apply limited forms of feedback-directed optimizations, although most of the offline work mentioned previously has not yet been applied in fully automated online systems. Work has been done for reducing the cost of specific types of instrumentation [11, 14, 23], however it is unclear whether these techniques reduce overhead enough to allow it to run unnoticed in an adaptive system.

A more general solution for performing instrumentation at runtime is to execute the instrumented code for only a short period of time to keep overhead to a minimum. This can be acheived by compiling an instrumented version of a particular method, and having the next invocation of that method call the instrumented version. After collecting instrumentation for the desired duration, the next invocation calls the original, non-instrumented version. Although this technique works when the instrumented method does not run for longer than the desired instrumentation period, if the instrumented method does not return for a long period of time, execution would be stuck in the (slow) instrumented version. When this occurs, switching back to non-instrumented code requires the ability to perform stack frame rewriting [25], which can be a difficult problem for optimized code.[1]

Even when stack-frame rewriting is possible, an expensive instrumentation would have to be used for only a short amount of time, to keep overhead low; however, this is still undesirable because profiling for a short period of time may not result in a profile that is representative of overall program behavior.

3. FRAMEWORK

This section describes our sampling framework. Section 3.1 describes existing trigger mechanisms, and Section 3.2 describes counter-based sampling, the trigger mechanism used by our framework.

Prior to our work, a system performing instrumentation would have two versions of a method, the original and the instrumented. Our framework introduces a third version of the code, called the *checking* code, 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 very fine grained, but controlled manner.

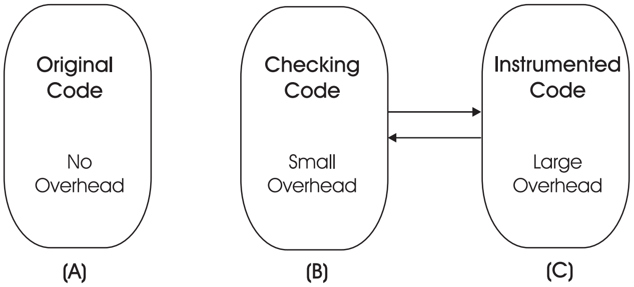

Figure 1: The three versions of the code in our framework. (A) represents the orignal code. (B) represents a minimally instrumented version containing checks that allow control to transfer into (C) in a fine-grained and controlled manner. (C) represents the fully instrumented code.

The total overhead of the instrumentation can be kept to a minimum by ensuring that most of the time is spent in the checking code. On regular sample intervals, execution moves into the instrumented code for a small, bounded amount of time.

The switching between the checking and instrumented code is accomplished as follows. The checking has conditional branches inserted (which we refer to as *checks*) that monitor a *sample condition*. When the sample condition is true, control jumps to instrumented code, rather than continuing in the checking code. Checks are currently placed on all method entries and backward branches (which will be referred to as *backedges*) in the checking code, as shown by Figure 2. This placement of the checks guarantees that (a) only a bounded amount of execution will occur 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 (as 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 overhead of the particular instrumentation being applied, to keep overhead to a minimum. Techniques for controlling the sample rate are discussed next in Section 3.1.

This framework also has the property that the checking code performs checks on method entries and backedges only; thus

[1] Because the execution patterns of optimized code can vary substantially from unoptimized code, an adaptive system will most likely perform instrumentation in optimized code.

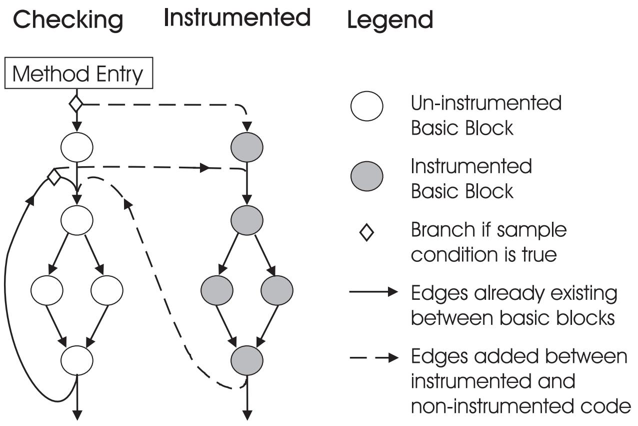
Checking
Instrumented
Legend
Method Entry
Un-instrumented Basic Block
Instrumented Basic Block
Branch if sample condition is true
Edges already existing between basic blocks
Edges added between instrumented and non-instrumented code

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

the overhead of the checking code is kept to a minimum, and 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 equal to the number of backedges and methods entries executed, independent of the instrumentation being performed by the instrumented code.*

Although our sampling framework does not totally eliminate all benefits of stack-frame rewriting, it greatly reduces the need for it. Even 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 the original 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.

Of course, there are some differences between executing instrumentation exhaustively and executing it only on sample intervals. For example, with exhaustive instrumentation it is possible to determine that a particular event *never* occured during the profiled interval, whereas this is not possible with a sampled profile. However, this is not a serious restriction because is unlikely that this type of functionality would be useful for an adaptive JVM; even when an event doesn't occur during a period of exhaustive instrumentation, it doesn't guarantee it won't happen in the future. Most of the common profiling techniques (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 the events that occur most frequently. These types of profiling work unmodified in our sampling framework.

More complex profiling techniques may assume that events are observed exhaustively, such as [11], 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; however, work such as [8, 31] are examples of how this can be achieved.

3.1 Trigger mechanisms

The framework described in Section 3 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; i.e., 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 — is already being used in Jalapeño JVM [2] to determine when an executing method should yield to the thread scheduler.

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 and 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 would not take a sample immediately upon recieving the timer interrupt, but instead would jump to instrumented code 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) would have a high probability of having a timer interrupt issued during its execution, which, in turn, would cause the *next* sequence of instructions to be sampled. Section 5 confirms that this improper attribution of samples, as well as the low sample frequency, substantially reduce 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 encorporated 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. Another drawback is that 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

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

Figure 3: Code for a CBS check

know of a performance counter that counts backedges and method entries, our framework simply performs the counting in software, as described in the next section.

3.2 Compiler controlled counter-based sampling

Counting a particular event and sampling when the counter reaches a threshold (which we refer to as *counter-based sampling*, or CBS for short) is an effective way of triggering samples proportionally to the frequence of that event, and is the fundamental principle behind the accuracy of DCPI [4]. Their performance counters count instruction cycles, and thus instructions are sampled proportionally to their execution frequency. However, it may be desirable to sample events for which there is no counter-based interrupt available, as is likely 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 to be inferior to that of the accuracy of counted events.

To ensure accuracy, we propose implementing the trigger mechanism in software; the compiler simply inserts code that decrements and checks a counter, as shown in Figure 3. 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. We call this technique *compiler inserted counter-based sampling*. The counter variable (`eventCounter`) 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 fairly low. We implemented such an approach in Jalapeño, placing the code in Figure 3 on all backedges and method entries, and the overhead averaged 5.94%. A detailed description of the overhead is included in Section 5.

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 entirely in software, without relying on any features of the hardware or operating system.[2] A more subtle advantage is that the CBS checks will trigger samples deterministically, allowing sampling results to be reproducable, which aids in debugging.

Certain architectures may even have instructions that can aid in the efficient implementation of CBS 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 – all in one cycle, allowing the code in Figure 3 to be executed efficiently on a powerPC.

> *I'm not really sure if this is true. I need to check whether it really executes in one cycle?*

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

4. SPACE SAVING VARIATIONS

The framework described in Section 3 uses two versions (checking and instrumented) of each instrumented method, and thus requires twice as much space. The instrumentation should not affect locality, since 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 compile time.[4]

In scenarios where instrumentation is sparse, ideally only those nodes containing instrumentation would need to be duplicated. However, it is not always possible to remove a non-instrumented node from the instrumented code without violating Invariant 1, as shown by 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.

The framework described in Section 3 will furthermore be referred to as Variation-0. Below, two variations for reducing the space of Variation-0 are discussed. Variation-1 (Section 4.1) identifies non-instrumented basic blocks that can be removed from the instrumened code without violating Invariant 1. Variation-2 (Section 4.2) describes techniques that modifies the placement of the checks to reduce the space requirements, but may violate Invariant 1.

4.1 Variation-1

[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 count register would need to be modified, which could potentially introduce overhead.

[4] Compile time will not be doubled. 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 doubling all nodes would be a small compared to optimization time.

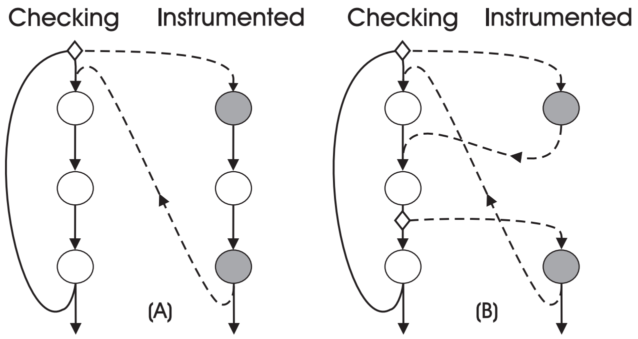

Figure 4: An example illustrating how removing non-instrumented nodes from the instrumented version can increase the number of checks executed in the checking code. Dark nodes represent basic blocks containing instrumentation. (A) represents an instrumented version with all basic blocks duplicated. (B)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

Variation-1 simply removes as many non-instrumented basic blocks as possible from the instrumented code 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 n is executed, no furthur instrumentation will be performed without returning to the checking code first. Any edge in the instrumented code that previously connected a non-bottom-node to a bottom-node is simply 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, some adjustments must be made to the checks in the checking code, because they may have previously pointed to top-nodes that are being removed. The adjustment is as follows.

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

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

This technique eliminates as many nodes as is possible without violating Invariant 1. Although the static number of checks may increase, the number of checks executed is the same as Variation-0, and instrumentation in an identical manner to Variation-0. A proof of correctness is available in [7].

4.2 Variation-2

If Invariant 1 can be violated, there are many other alternatives for reducing code duplication. 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*. Although none of the instructions themselves are duplicated, all instructions with associated instrumentation must check the sample condition before executing the instrumentation.

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 5.

The only drawback of Variation-2 is that it may execute more checks at runtime than the previous variations. This 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 5), making Variation-2 useful in situations where a reduction in space is important. Variation-2 is also the easiest to implement since 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 approach should be used (Variation-0, Variation-1, Variation-2, or a combination thereof) depends on the type of instrumentation being performed, and the time and space constraints that must be satisfied. A reasonable approach for an adaptive system would be to fill the instrumented version with every type of instrumentation it knows how to perform, 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.

5. EXPERIMENTAL RESULTS

To assess the feasibility of using our framework in an adaptive JVM, we performed experiments to measure both the runtime overhead and the accuracy of the technique when applied to two example instrumentations.

5.1 Methodology

Our experimentation was performed using the Jalapeño JVM [2, 5] being developed at IBM T.J. Watson Research. Currently Jalapeño contains two fully operational compilers, a nonoptimizing *baseline* compiler and an *optimizing compiler* [13]. 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 benchmark suite with input size 10. The running times of the benchmarks ranged from 1.1 to 4.2 seconds. Overhead numbers were collected using the minimum of 10 runs to eliminate noise; however, the instrumentated profiles used for accuracy comparisons were collected using a single run. Benchmarks with short running times were chosen specifically to show that our framework can collect accurate profiles in a short amount of time. 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 concious data layout [15, 16, 19].

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.

5.2 Results

Table 1 characterizes the two profiling techniques used in this study by showing their overhead when applied exhaustively (not using our framework, where all methods spend 100% of the time in the instrumented code). The first column lists the benchmarks, while the second and third columns show the overhead of exhaustive instrumentation for call-edge and field-access instrumentation respectively. The call edge instrumentation averages 82.8% overhead, and the field access averages 94.1% overhead. Clearly, these instrumentations are too expensive to execute unnoticed at runtime.

Benchmark	Call edge	Field accesse
201_compress	77.1	286.5
202_jess	140.7	90.2
209_db	4.5	15.3
213_javac	75.3	15.4
222_mpegaudio	132.5	59.2
227_mtrt	121.6	76.4
228_jack	28.3	113.2
opt-compiler		
pBOB		
Volano		
Geom. Mean	82.8	94.1

Table 1: Percent overhead of exhaustive instrumentation (100% of time in instrumented code, no checking code)

Next, the overhead of the checking code was computed by comparing the running time of the checking code only (the trigger is permanently false, so execution never moves 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 overhead when using Variation-0 or Variation-1.[6] The column labeled “Both” shows the overhead when the checks are inserted on both backedges *and* method entries, therefore representing the total cost of the checking code. The average combined overhead is 5.94%, which is very low compared to the cost of the full instrumentation. The columns labeled “Backedges” and “Method entries” provide furthur insight into the cost of the checking code by reporting a breakdown of the overhead when checks are placed only on backedges or method entries, respectively.

These measurements were obtained *without* using any of the powerPC or Jalapeño specific techniques mentioned in Section 3.2, so these numbers act as a “worst case” overhead for a naive implementation. The natural development of Jalapeño should also lower the overhead of the checks. Jalapeño currently does not perform loop unrolling, which would lower the cost of backedge checks. Secondly, the timings were collected using the default static inlining heuristics. Performing instrumentation after more aggressive inlining has taken place[7] could substantially reduce the number of method calls, further reducing checking overhead.

The last two columns of Table 2 show the 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

[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 overhead of the checking code is the same for both Variation-0 and Variation-1.

[7] which is not an unreasonable thing to do, as an adaptive system would likely resort to expensive instrumentation only after all the simpler optimizations are applied.

	Variations 0 & 1			Variation-2		
Benchmark	Both	Backedges	Method entry	Call edges	Field accesses	Space Reduction (K)
201_compress	11.20	9.36	5.62	5.62	101.9	
202_jess	5.65	4.32	5.23	5.23	31.8	
209_db	1.29	1.65	1.10	1.10	4.2	
213_javac	1.97	1.22	4.51	4.51	3.2	
222_mpegaudio	10.28	7.40	3.94	3.94	22.8	
227_mtrt	5.83	0.73	5.00	5.00	59.0	
228_jack	5.40	3.20	3.30	3.30	29.3	
opt-compiler						
pBOB						
Volano						
Geom. Mean	5.94	3.98	4.10	4.10	36.0	

Table 2: Overhead of the checking code reported as a percentage (execution never leaves checking code – overhead is for the checks only).

checking overhead of Variation-0. For the call edge instrumentation, the average overhead is 3.98% – less than the checking overhead of Variation-0. This is because for the call-edge instrumentation, Variation-2 performs checks on all call edges (i.e., method entries), explaining why columns 4 and 5 are identical. However, the average overhead of the field access instrumentation is 36.0%. Although this is less than the overhead of the exhaustive field access instrumentation, it is significantly higher than the overhead for Variation-0. The Variation-2 overhead would increase if more instrumentation were added to the instrumented code (whereas the overhead of Variation-0 would not).

Here we mention an advantage of Variation-2 and show how it can save space, and referencing the last column of 2.

5.2.1 Instrumentation overhead and accuracy

The overhead and accuracy of the actual instrumentation (as opposed to just the checking code) is evaluated when sampled by our framework. We implemented Variation-2 and a prototype "simulated" version of Variation-0. The simulated version of Variation-0 does not actually duplicate the code as described in Section 3. 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 actions of Variation-0. This technique 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, not including the overhead of the checks.

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* metric is used, similar to that used in [23]. 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 for call edges and field accesses.

Figure 5 helps describe the overlap metric by visually representing the call-edge profile for the `javac`. Each bar represents a call edge, and the y-axis represents the "hotness" of the edge, where an edge's hotness is defined as the percentage of all samples that were attributed to that edge. The height of each bar is 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 each edge is simply the minimum of the two hotness values. The total overlap of the benchmark is the sum of the overlaps for all edges. Identical profiles yield an overlap of 100%, while any variation from perfect will produce an overlap of less than 100%. 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 5 yeilds an overlap of 93.8%, showing that an overlap in the mid to low 90's is still very accurate.

Table 3 shows the overhead and accuracy of the sampled instrumentation, averaged over all benchmarks for several different sample rates. Because samples are driven by counter-based sampling, the sample rate is given as a *sample interval*. The sample interval represents the number of checks (in the checking code) that are executed before each sample is triggered. Therefore a sample interval of 1000 means that roughly $1/1000^{th}$ of the execution will occur in instrumented code. The first column of Table 3 shows the sample interval. The next four columns show data for call-edge profiling, and the last four show data for field access profiling. Each profiling type contains a breakdown of the overhead for both Variation-0 and Variation-2.

As would be expected, increasing the sample interval re-

[8] This simulated version may cause the overhead numbers to be inaccurate to a certain degree, but this is of very little significance 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 a true implementation of Variation-0.

	Call Edge				Field Accesses			
	Variation-0		Variation-2		Variation-0		Variation-2	
Sample Interval	Overhead	Accuracy	Overhead	Accuracy	Overhead	Accuracy	Overhead	Accuracy
1	89.03	100	108.90	100	80.12	100	77.32	100
10	8.10	99	11.24	99	8.22	100	7.80	99
100	0.72	99	1.33	98	0.91	99	0.75	99
1,000	0.05	95	0.18	97	0.05	98	0.14	98
10,000	0.07	89	0.07	89	0.03	96	0.01	95
100,000	0.03	62	0.09	67	0.04	85	0.03	84

Table 3: Sampled Instrumentation – Overhead and Accuracy. Accuracy is reported as an "overlap percentage", as described in the text. Overhead is reported as percent overhead, not includeing checking overhead.

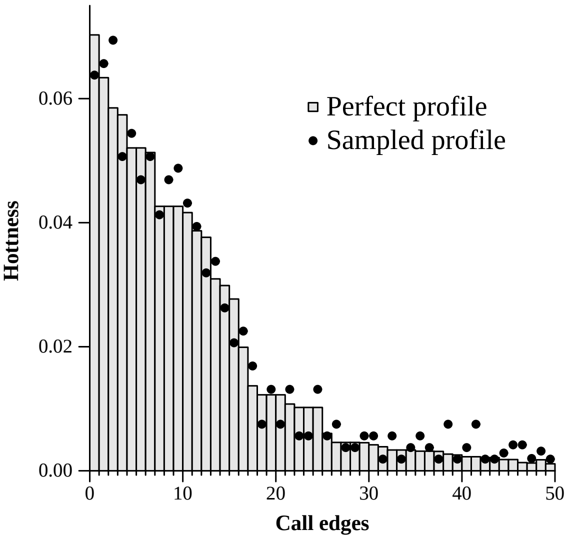

Figure 5: A graphical representation of the `javac` call-edge profile, illustrating the overlap accuracy metric.

duces the overhead, and also reduces the accuracy. At sample interval 1,000 there is practically no overhead (for the instrumentation, excluding checking overhead), and the accuracy is still very near 100% for both instrumentation types and both variations. 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 observing this table, keep in mind that the sample intervals are increasing exponentially, and that there is actually a huge range of sample intervals (anywhere from 100 to 10,000) that offer extremely high accuracy with almost no overhead.

5.2.2 Trigger Mechanisms

As discussed in Section 3.1, it is possible to use triggers other than a counter-based trigger. To show the advantages of the counter-based trigger, we compared its accuracy 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. We used this threadswitch bit to also trigger a sample, using Variation-0 for the field-access instrumentation. To make a fair comparison, we used a sample interval of 30,000 for the counter-based sampling, because it results in approximately the same number of samples as the time-based trigger for these benchmarks.

Benchmark	Time-based	Counter-based
201_compress	89	99
202_jess	88	94
209_db	51	94
213_javac	66	72
222_mpegaudio	73	94
227_mtrt	83	78
228_jack	85	91
opt-compiler		
pBOB		
Volano		
Geom. Mean	76	89

Table 4: Accuracy of a counter-based trigger vs. a time-based trigger for field access instrumentation.

Figure 4 compares the accuracy of both techniques for all benchmarks. Clearly, counter-based sampling is more accurate, averaging 89.7% accuracy, as opposed to time-based sampling, which averaged 74.8%. This is most likely due to the time-based trigger's inaccurate attribution of samples, as discussed in Section 3.1. Another advantage of the counter-based trigger is that it allows the sample interval to be increased. As previously shown in Table 3, a faster sample interval of 100 achieves a much higher accuracy (98–99%), while overhead remains at near zero. The time-based technique, however, does not allow the sample interval to be increased, because it is limited by the frequency of the hardware interrupt.

6. RELATED WORK

We do not know of any framework based on sampling designed to allow general instrumentation to be performed at low overhead. Previous work [15, 23, 30] has used the idea of wrapping each instrumentation operation inside a conditional branch (as is done in our "Variation-

Update the TR reference to the journal version

2") to allow the instrumentation to be turned on and off. However, unlike our work, none of these approaches use the conditional checking to perform fine-grained sampling; instead, the conditional was used as a switch to turn exhaustive profiling on and off for a given period of time. [15, 23] 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 et al. [30] describes a *Java virtual machine profiler interface* (JVMPI), which also wraps instrumentation operations inside a conditional branch to turn exhaustive instrumentation on and off. JVMPI is a general purpose mechanism for obtaining information from a JVM, supporting both sampling and instrumentation, but not combining the two. Another difference from our work is that JVMPI is designed to provide feedback to a programmer, as opposed to providing feedback to an adaptive system; the finest grained profiling supported by the interface is at the method level (excluding basic block profiling, etc). The overhead of the checks using all of these these approachs ([15, 23, 30]) will increase as more instrumentation is added; this is not the case with our "Variation-0".

Previous systems have counted the frequency of events to determine when optimization should occur. Self-93 [26] compares method counters against a threshold to determine when a method should be optimized. Dynamo [10] uses counters on *start-of-trace* points to determine when a trace should be optimized. There are several fundamental differences between these uses of counters and our counter-based sampling. First, these approaches use multiple counters (Self-93 uses one per method, Dynamo uses one per trace) to count the frequency of events, whereas our counter-based sampling uses one counter to distribute samples in a statistically relavant manner. Second, when a counter reaches a threshold in their systems, the current state is observed,[9] and optimization takes place based on this observation. Thus, their optimization decisions are essentially based on a sample set of size one, whereas with our approach, many samples can be collected over a long period of time before action is taken. Lastly, none of these approaches solved the problem of providing a general framework to allow general instrumentation to be performed efficiently.

There has been work in reducing the cost of specific types of instrumentation [11, 14, 23] but these techniques do not use sampling. Although these techniques reduce the overhead of instrumentation, they most likely do not reduce the overhead enough to allow it to run unnoticed in an adaptive system.

Recent work [8, 31] has used sampling to reduce the cost of building the *calling context tree* [11]. Their techniques are an example of the general technique proposed by our framework, i.e., using sampling to collect profiling information that is traditionally collected by exhaustive instrumentation.

Morph [33], and *DCPI* [4] use low overhead sampling to collect information about a program at runtime, however they do not address the problem of allowing general instrumentation to be performed.

Aron et al. [9] describes *soft timers*, an operating system facility that allows the efficient scheduling of software events at a granularity down to tens of microseconds; however, the goal of soft timers is somewhat orthogonal to our counter-based sampling. The goal of counter-based sampling is to trigger high frequency samples on a regular basis to achieve a statistically accurate sampling, whereas the goal of soft timers is to process high frequency events with low overhead by strategically scheduling them at strategic times when overhead will be minimized.

7. CONCLUSIONS

We have presented a tunable framework for instrumenting code that allows expensive instrumentation to be performed with extremely low overhead, averaging ∼6% in our experience. Our framework uses compiler controlled counter-based sampling to switch between instrumented and non-instrumented code in a controlled, fine-grained manner. Our sampling techniques does not rely on any hardware or operating system support, yet allows a high enough sample rate to achieve accurate sampling results.

The reduction in overhead provided by our sampling framework allows instrumentation to be performed for a longer period of time at runtime, while causing only minimal performance degredation, allowing a system to utilize expensive instrumentation techniques even without the ability to do stack frame rewriting. Because total overhead is controlled completely by the framework, implementors of instrumentation techniques are no longer required to concentrate on minimizing overhead, but instead can concentrate on developing new techniques quickely and correctly. 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.

We have shown experimentally, using two different types of instrumentation, that our technique can achieve excellent accuracy (achieving a 98–99% overlap with a perfect profile) with very low overhead (averaging ∼6% with a very naive implementation).

8. 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.

[9] Self-93 observes the call stack; Dynamo observes the next trace that executes.

[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, D. Grove, M. Hind, S. Fink, 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, 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.

[7] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. Technical Report In preparation, Rutgers University.

[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] 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.

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

[15] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. Technical Report CS98-592, UCSD, July 1998.

[16] 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*, San Jose, California.

[17] 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).

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

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

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

[32] 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.

[33] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automated profiling and optimization. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, Operating Systems Review, 31(5), pages 15–26, Oct. 5–8 1997.