A Framework for Reducing the Cost of Instrumented Code*

Matthew Arnold Barbara G. Ryder

Rutgers University, Piscataway, NJ, 08854
{marnold,ryder}@cs.rutgers.edu

†IBM T.J. Watson Research Center, Hawthorne, NY, 10532

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. This paper presents a general framework for performing *instrumentation sampling* to reduce the overhead of previously expensive instrumentation. The framework is simple and effective, using code-duplication and *counter-based sampling* to allow switching between instrumented and non-instrumented code.

Our framework does not rely on any hardware or operating system support, yet provides a high frequency sample rate that is tunable, allowing the tradeoff between overhead and accuracy to be adjusted easily at runtime. Experimental results are presented to validate that our technique can collect accurate profiles (93–98% overlap with a perfect profile) with low overhead (averaging ~6% total overhead with a naive implementation). A Jalapeño-specific optimization is also presented that reduces overhead further, resulting in an average total overhead of ~3%.

1 Introduction

Early 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, 18, 24, 33, 40]. More advanced *adaptive systems* [5,22,30,31,35] moved beyond this simple strategy by dynamically 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, 10, 11, 15, 26], as well as optimizations based on offline profiles [6,16,17,19,20,27]. Although some systems [5, 9,21,22,32] apply limited forms of online feedback-directed optimizations, 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, 10, 11, 16, 17, 27], and overheads in the range of 10,000% (100 times slower) have been reported [16].

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 small time interval 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, to prevent the program from running indefinitely with poor performance. This could be achieved by using dynamic instrumentation [28,36] to insert and remove instrumentation without recompiling the method, or by performing on-stack replacement [29] to hot-swap execution back to the non-instrumented version while the method is running. However, dynamic instrumentation is architecture-specific, and may introduce complexities such as maintaining cache consistency. On-stack replacement can be difficult for optimized code because a method can be swapped only at program points where full program state is known. A common solution (used by [29,31]) for implementing on-stack replacement is to restrict optimizations by introducing *interrupt points* where state is required to be consistent.

We present a general framework for performing *instrumentation sampling*, allowing previously expensive instrumentation to be performed accurately with low overhead. The framework using code-duplication combined with *compiler-inserted counter-based sampling* to allow fine-grained switching between instrumented and non-instrumented code. Previous work [8,14,25,36,39] has used

*Funded,in part, by NSF grant CCR-9808607.

†Matthew Arnold is an intern at Watson; Barbara Ryder was on sabbatical leave at Watson from August 2000 – March 2001.

sampling to reduce the cost of instrumentation, but these techniques are specific to one kind of instrumentation. To the best of our knowledge, this is the first technique that allows general instrumentation to be performed with low overhead.

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 dynamic instrumentation or 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 incorporated into our framework without 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 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 a deterministic application twice will result in identical profiles.

The framework is implemented and evaluated using the Jalapeño JVM [2], providing experimental evidence of the overhead and accuracy when applied to two types of instrumentation. The results show that high accuracy can be achieved (93–98% overlap with a perfect profile) with low overhead, (averaging ~6% total overhead with a naive implementation). A Jalapeño-specific implementation is also presented as an example of how hardware- or compiler-specific optimizations can be used to further reduce overhead, resulting in an average total overhead of ~3%.

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

2 Instrumentation 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 *duplicated* code, within the instrumented method, as shown in Figure 1. The original version of the code is now referred to as the *checking code* because it is modified slightly to allow execution to swap back and forth between the checking code and duplicated code in a fine-grained, controlled manner. On regular sample intervals, execution moves into the duplicated code for a small, bounded amount of time, so expensive instrumentation can be inserted in the duplicated code. Overhead may be kept to a minimum by ensuring that the majority of execution occurs in the checking code.

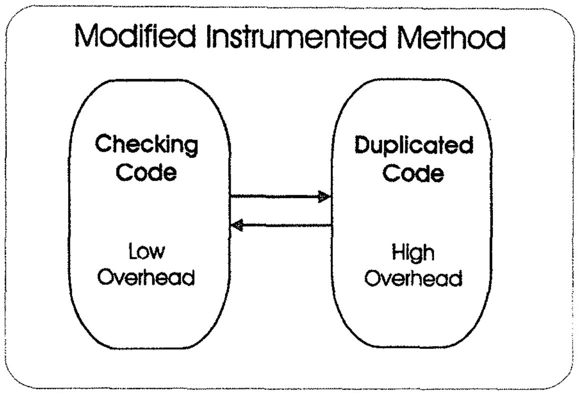


Figure 1: A high-level view of an instrumented method generated by our framework. A second version of the code is introduced, called the *duplicated code*, which contains all instrumentation. The original code becomes the *checking code*, which is minimally instrumented to allow control to transfer in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

The switching between the checking and duplicated code is illustrated in Figure 2. The checking code has conditional branches inserted (which we refer to as *checks*) that monitor a *sample condition*. When the sample condition is true, a sample is *triggered* and control jumps to duplicated 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 duplicated code is also modified so that there are no backedges within the duplicated code (also shown in Figure 2). Instead, all backedges in the duplicated code transfer control back to the checking code, ensuring that only a bounded amount of time is spent in the duplicated code during each sample. The ratio of time spent in duplicated code vs. checking code can be controlled by changing the rate at which the sample condition is true. If the instrumented method is no longer needed, but the method continues to execute (does not return), 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.

This version of the framework will be referred to as `Full-Duplication`, since all of the code in the method is duplicated. `Full-Duplication` has the desirable property that the checking code performs checks on method entries and backedges only. For ease of reference this will be referred to as Property 1, defined as follows:

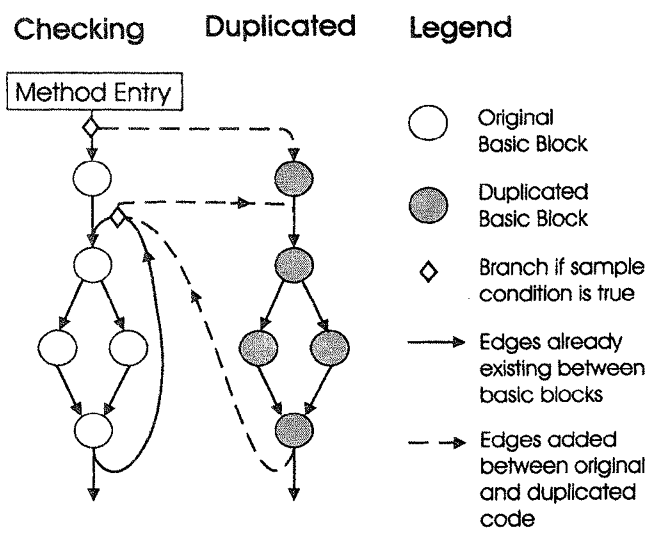

Checking
Duplicated
Legend
Method Entry
Original Basic Block
Duplicated Basic Block
Branch if sample condition is true
Edges already existing between basic blocks
Edges added between original and duplicated code

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

Property 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.*

Applicability to various types of instrumentation Many of the common profiling techniques can be used without modification in our framework. For example, any instrumentation designed to perform event counting (such as intraprocedural edge or path profiling, field access profiling, value profiling, etc.) will work effectively when inserted as-is into the duplicated code. Instrumentation that counts events associated with backedges is not a problem because the instrumentation can be attached to the edge transferring control from the duplicated code to the checking code (this edge was previously a backedge in the duplicated code).

There are some profiling techniques that require special treatment to be sampled correctly in our framework. For example, some profiling techniques rely on observing events in succession, 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 a sampling context; work such as [8,39] are examples of how this can be achieved. Another example would be monitoring the behavior of multiple (N) consecutive loop iterations. This could be achieved in our framework by adding a counted backedge within the duplicated code. After N iterations are profiled, control would return to the checking code.

There are also some profiles that are impossible to collect via sampling. For example, exhaustive instrumentation can be used to establish that a particular event *never* occured during the profiled interval; this is not possible with a sampled profile. However, it is not clear that this functionality is particularly useful for the purpose of an adaptive JVM, because even with perfect knowledge of the past, no guarantees can be made about future behavior. The goal of profile-directed optimization is to predict the likelihood of future behavior, which can be achieved with both sampled and exhaustive profiles.

2.1 Trigger mechanisms

Our instrumentation sampling 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 manner that is statistically meaningful; 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. The approach of checking a timer-set bit has been used previously (Self-91 [18], 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 4.6 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 any hardware 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

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

Figure 3: Code inserted for a counter-based check

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. 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 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 without using a dedicated register, placing the code in Figure 3 on all backedges and method entries, and the overhead averaged 4.9% (for executing the checks only, when no samples were taken). A detailed description of the overhead is included in Section 4.2.

For multi-threaded applications, the global counter may raise some concerns. First, access to the global counter is not synchronized for performance reasons, so data races may occur. Fortunately, it is not necessary to maintain 100% accuracy of the global counter, as it is simply a means of triggering samples on a semi-regular basis. Having the counter value off-by-one occasionally would have little affect on the resulting accuracy. A more serious problem is that access to a single global counter could become a performance bottleneck as the number of threads and processors increases. In this case, the global counter could be replaced by thread- or processor-specific counters, allowing unsynchronized access to the counter, with no resource contention.

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 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.[1] Finally, counter-based checks trigger samples deterministically, creating reproducible sampling results for deterministic applications.

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

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

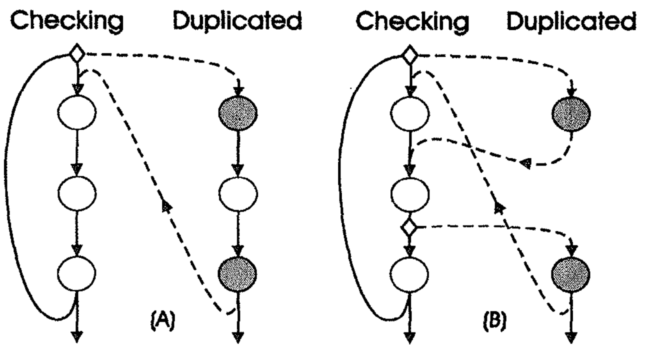


Figure 4: Removing non-instrumented nodes from the duplicated code can increase the number of checks executed. Dark nodes represent basic blocks containing instrumentation. (A) represents an instrumented method with all basic blocks duplicated. (B) shows an instrumented method where the non-instrumented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

the code in Figure 3 to be executed in one instruction. VLIW and superscalar architectures may also 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 *yieldpoints* that check whether or not the executing code should yield to the thread scheduler. These yieldpoints can be exploited to eliminate much of the overhead of the checking code (described further in Section 4.3). Although this example is specific to Jalapeño, it serves as an example of how hardware- or compiler-specific techniques can be used to reduce the overhead of the counter-based checks.

3 Space-saving variations

`Full-Duplication`, the algorithm described in Section 2, uses two versions (checking and duplicated) of each instrumented method, thus increasing both space and compile time. Although this is clearly not desirable, it may not be as bad as it first appears. Duplicating the code should have a minimal effect on locality because the duplicated code is executed infrequently and can be placed somewhere out of the common path. Although compile time is increased, it is not necessarily doubled. As discussed further in Section 4.3, our implementation performs the doubling of all code after most of the compilation has taken place, increasing compile time by 34%. Finally, an adaptive system will likely instrument only the hot methods, and not necessarily at the same time. If space is limited, the number of methods instrumented simultaneously can be limited.

Nevertheless, it is undesirable to increase space when unnecessary. 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 duplicated code without violating Property 1. As shown in Figure 4, when the non-instrumented node is removed from the duplicated code, an additional check must be added to allow the second instrumented node to be sampled.

There are many space-saving variations of our framework. Two such variations, Partial-Duplication and No-Duplication, are presented below.

3.1 Variation 1: Partial-Duplication

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

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

All bottom-nodes can be removed from the duplicated code without violating Property 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 duplicated 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 duplicated code DAG such that no path from entry to n contains an instrumented node. All top-nodes can be removed from the duplicated code without violating Property 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.
2. In the duplicated 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 Partial-Duplication 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 duplicated 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 duplicated nodes without violating Property 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 Full-Duplication. Instrumentation is performed identically to Full-Duplication.

3.2 Variation 2: No-Duplication

If Property 1 can be weakened, allowing more than one check to be executed per loop iteration or method call, there are several other alternatives for reducing code duplication; any non-instrumented node can be removed from the duplicated 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 No-Duplication and is shown in Figure 6, which illustrates one basic block with two instrumented instructions. Although none of the

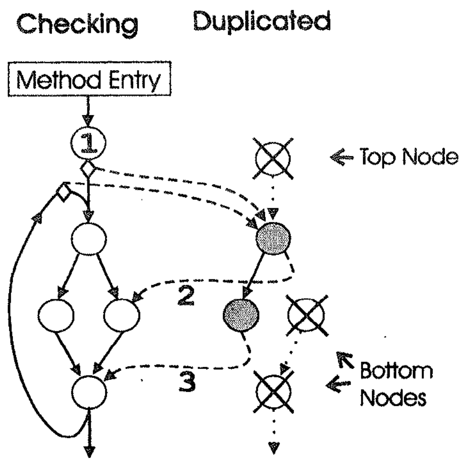
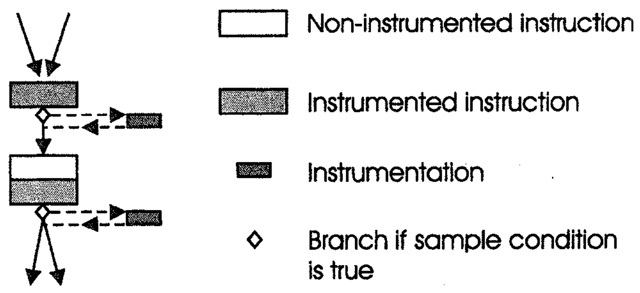


Figure 5: Example of Partial-Duplication after top-nodes and bottom-nodes are removed. The check after method entry is removed because it branched to a top-node. A check is added to the edge exiting the basic block labeled "1" because the corresponding edge in duplicated code connected a top-node to 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.



Non-instrumented instruction
Instrumented instruction
Instrumentation
Branch if sample condition is true

Figure 6: Example of No-Duplication, showing one basic block containing two instrumented instructions. No code is duplicated, but checks are placed on all instrumentation operations. Property 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 Full-Duplication if instrumentation is sparse.

instructions themselves are duplicated, all instructions with associated instrumentation must check the sample condition before executing the instrumentation.

No-Duplication will *not* perform instrumentation identical to Full-Duplication. With Full-Duplication, a sample causes execution in that method to remain in duplicated code until the next backedge is reached, whereas in No-Duplication, 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 4.4.

The only drawback of the No-Duplication approach 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, making `No-Duplication` a good option in these situations.

Combining both variations (`Partial-Duplication` and `No-Duplication`) is also possible, 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 as well as the time and space constraints that must be satisfied. A reasonable approach for a JVM would be to selectively instrument only the hot methods, but to apply many types of instrumentation at once. `Full-Duplication` is probably the best choice for this scenario, because the checking overhead does not increase as more instrumentation is added.

4 Experimental Results

The following section presents an implementation and evaluation of our framework, demonstrating its low overhead and high accuracy. Both the `Full-Duplication` and `No-Duplication` algorithms were implemented and applied to two example instrumentations.

4.1 Methodology

Our framework was implemented 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* [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 [23] benchmarks with input size 10, the Jalapeño optimizing compiler [13] run on a small subset of itself, the `Volano` benchmark [38], and the `pBOB` benchmark [12]. 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 20 runs to eliminate as much noise as possible. 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 2096MB RAM, running AIX 4.3.

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 compiled prior to execution at level *O2*, which is currently Jalapeño's highest optimization level [5].

4.2 Instrumentation examples

The framework is evaluated using the following two examples of instrumentation:

Benchmark	Call-edge (%)	Field-access (%)
`201_compress`	72.4	204.8
`202_jess`	133.2	60.9
`209_db`	8.3	7.7
`213_javac`	75.7	14.2
`222_mpegaudio`	129.6	99.8
`227_mtrt`	122.2	46.0
`228_jack`	34.3	108.7
`opt-compiler`	189	34.9
`pBOB`	72.3	20.2
`Volano`	46.6	7.6
Average	88.3	60.4

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

1. **Call-edge instrumentation** All method entries are instrumented to examine the call stack. The caller method, the callee method, and the call-site within the caller method (specified by a bytecode offset) are recorded as a call edge. A counter is maintained for each call edge, and the counter is incremented on each call.

2. **Field-access instrumentation** 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 data layout optimizations, such as [16,17,20].

Table 1 characterizes these two types of instrumentation by showing their overhead when applied exhaustively (not using our framework) to all methods. 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 88.3% overhead, and the field-access averages 60.4% overhead. Clearly, these instrumentations as implemented here are too expensive to execute unnoticed at runtime.

These are not meant to represent the most efficient implementations for collecting call-edge and field-access profiles. Favoring simplicity over efficiency was an intentional design choice. One of the goals of our sampling framework is to allow new instrumentation techniques to be developed quickly and correctly, without requiring the implementor to spend time manually reducing overhead. These implementations are simply two examples of the numerous instrumentation techniques that could be sampled by our framework.

4.3 Framework overhead

This section describes the overhead of the framework itself, when no samples are taken. The global counter was set sufficiently high so that execution never left the checking code, and no instrumentation was inserted in the duplicated code.

Full-Duplication algorithm Table 2 presents the overhead of a simple, non-optimized implementation of `Full-Duplication`. The second column, labeled "Total Framework Overhead" shows the running time of

Benchmarks	Time Overhead			Space Related Overhead	
	Total Framework Overhead (%)	Checking overhead breakdown		Maximum space Increase (KB)	Compile Time Increase (%)
		Backedges (%)	Method Entry (%)		
201_compress	8.7	8.3	0.9	106	37
202_jess	3.3	2.9	0.1	244	37
209_db	2.1	1.8	0.2	123	34
213_javac	2.7	0.2	1.4	442	38
222_mpegaudio	9.9	9.0	0.8	156	31
227_mtrt	3.4	2.0	2.4	163	31
228_jack	8.4	6.6	1.2	258	18
opt-compiler	6.2	2.1	4.4	976	48
pBOB	3.8	2.6	0.9	306	37
Volano	1.4	0.3	1.0	75	32
Average	4.9	3.5	1.3	285	34

Table 2: Framework overhead of `Full-Duplication` compared to the original, non-instrumented code. No samples are taken (execution never leaves checking code) so all figures reflect the overhead of the framework itself, without performing any instrumentation.

`Full-Duplication` (with no instrumentation being sampled) relative to the original, non-instrumented code. This overhead includes the direct overhead of executing the counter-based checks on method entries and backedges, and also any indirect overhead associated with doubling the size of all methods. For example, the increase in code size could increase the number of instruction cache misses, even if the duplicated code is never executed. The total overhead of the framework ranged from 1.4% to 9.9% and averaged 4.9% for all benchmarks.

The third and fourth columns, labeled "Backedges" and "Method Entry" respectively, break down the direct overhead of executing the counter-based checks. These figures were obtained by inserting the backedge and method entry checks independently, but without actually duplicating any code,[2] therefore eliminating any of the indirect overhead associated with code duplication. It is difficult to accurately measure such small overheads, so these figures should be treated as an approximate breakdown only. Nevertheless, the sum of the backedge and method entry checking overhead is roughly equivalent to the total framework overhead, suggesting the indirect cost is minimal.

Although averaging 4.9% overhead is reasonable, it does *not* represent a lower bound, as it could be improved by using a number of techniques. First, backedge checks introduce significant overhead in `_201_compress` and `_222_mpegaudio` because their execution is dominated by tight loops. Loop unrolling, which is not currently implemented in Jalapeño, would significantly reduce this overhead by reducing the number of backedges executed.[3] Second, these experiments were run using the default, non-aggressive static inlining heuristics. The method-entry overhead would be reduced if more aggressive inlining were performed before instrumentation occurs, which is likely to be the case when used online in an adaptive system. Third, this checking implementation is naive, and does not use any hardware, operating system, or JVM specific techniques. For example, no effort was made to keep the global counter in a register; each check performs a memory load, compare, branch, decrement, and store. Section 4.5 presents the overhead of our Jalapeño-specific implementation.

[2]This configuration cannot be used to sample instrumentation. It is included solely to provide an approximate breakdown of the direct checking overhead.

[3]Only loops with a small number of instructions would need to be unrolled, because as the number of instructions executed per iteration increases, the overhead of inserting a backedge check decreases; therefore no substantial increase in code size should be necessary.

The last two columns of Table 2 report other sources of overhead introduced by the code duplication. The first of these columns, labeled "Maximum Space Increase", shows a rough approximate of the space overhead when `Full-Duplication` is applied to all methods. Since all methods are doubled in size, 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, the number of methods that are instrumented simultaneously can be restricted. When used selectively, the space requirements of `Full-Duplication` could be less of a concern than other space consuming transformations, such as inlining and specialization, which can potentially cause exponential code growth.

The last column, labeled "Compile Time Increase", shows the total increase in compile time for each benchmark when using `Full-Duplication`. Our implementation performs the doubling of all code relatively late in the compilation process,[4] and therefore the majority of the compilation process is unaffected. The average increase in compile time was 34%. This compile time increase is attributed mostly to instruction selection, instruction scheduling, and register allocation, all of which occur after code duplication.

No-Duplication algorithm `No-Duplication` inserts checks on all instrumentation operations, therefore the overhead introduced by the checks depends heavily on the type of instrumentation being performed. Table 3 shows the checking overhead of `No-Duplication` when applied to call-edge and field-access instrumentation. No samples are taken, so the only overhead is the the cost of executing the checks. Recall that `No-Duplication` does not guarantee to maintain Property 1, and thus the overhead may be higher or lower than that of `Full-Duplication`.

For field-access instrumentation, the checking overhead averages 51.1%, which is only slightly less than the cost of the exhaustive instrumentation. For each field access,

[4]Performed in the last phase of the Jalapeño's low-level IR (LIR)

Benchmarks	Call-edge (%)	Field-access (%)
201_compress	0.9	151.5
202_jess	0.1	36.6
209_db	0.2	6.9
213_javac	1.4	21.3
222_mpegaudio	0.8	100.7
227_mtrt	2.4	49.1
228_jack	1.2	72.1
opt-compiler	4.4	41.1
pBOB	2.3	21.3
Volano	1.0	10.4
Average	1.3	51.1

Table 3: Framework overhead of `No-Duplication` compared to the original, non-instrumented code. No samples were taken, so overhead is for executing the checks only.

the instrumentation performs two loads, an increment, and a store, which is similar to the cost of a counter-based check, making the insertion of checks completely ineffective. `No-Duplication` is beneficial only when the cost of executing a check is cheaper than executing the instrumentation itself.

For the call edge instrumentation, however, the average checking overhead is 1.3%. Checks are placed on all call edges only (i.e., method entries), explaining why column 2 of Table 3 is identical to column 4 of Table 2. Clearly `No-Duplication` is beneficial for our implementation of call-edge instrumentation, which spends time examining the stack to determine the call-edge information. There are also other types of profile information available at method entry, such as parameter values that can be used to guide specialization. An effective approach would be to use a counter-based check to guard a call to a method `profileEntry()` that would collect all of the desired profile information about the method.

4.4 Sampled instrumentation overhead and accuracy

In this section overhead and accuracy of the actual instrumentation (as opposed to just the checking code) are evaluated when sampled by our framework. The overhead introduced by sampled instrumentation depends on the instrumentation being performed, so call-edge and field-access instrumentation are used as examples; both were applied together during the same run, and sampling was performed using either `Full-Duplication` or `No-Duplication`. Table 4 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 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 the duplicated code. The column labeled "Num Samples" shows the average number of samples that were taken during a single run of each benchmark.[5] The overhead and accuracy results are described in detail below.

[5]Recall that profiles used for accuracy comparison were collected using a single run of the benchmark, whereas timings for overhead comparison were collected using the median of 20 runs.

Overhead The overhead of sampling both call-edge and field-access instrumentation (at the same time) is reported in Table 4, averaged over all benchmarks and normalized in two different ways. First, the columns labeled "Sampled Instrum." report the overhead *without* including the framework overhead, thus representing the additional overhead (above the framework overhead) that is introduced by taking samples. This includes both the overhead of executing instrumentation, and in the case of `Full-Duplication`, the cost of jumping to duplicated code, which will most likely incur one or more instruction cache misses. As would be expected, for both `Full-Duplication` and `No-Duplication`, a sample interval of 1000 or greater results in low overhead (1% or less), since over 99.9% the time is spent in the checking code.

The columns labeled "Total" report the total overhead of performing sampled instrumentation, compared to the original, non-instrumented code, and includes all possible overhead introduced by the sampling framework. As expected, a small sample interval (such as 1 or 10) cause the overhead to be dominated by the overhead of taking samples,[6] whereas a large sample interval (such 10,000 or 100,000) cause the overhead to be dominated by the overhead of the framework itself. Since the framework overhead of `No-Duplication` is fairly high (averaging ~50%, mostly from the field-access profiling, as shown in Table 2), the total overhead is also high even when few samples are being taken. The average framework overhead of `Full-Duplication` is around 5% (see Table 2) so at sample rate 1000, the total sampling overhead was 6.3%.

Accuracy To assess accuracy, profiles collected at different sample rates are compared against a perfect profile (collected by setting the sample interval to 1, causing all execution to occur in duplicated code) and an accuracy metric is computed. An *overlap percentage* metric is used, similar to that used in [26]. 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 most frequently sampled edges of the call-edge profile for `javac`. Each bar along the x-axis represents a call edge, and the y-axis represents the *sample-percentage*, defined as the percentage of samples attributed to that edge (sample-percentage(edge) = num_samples(edge) / total_samples * 100). The height of each bar shows the sample-percentage of an edge according to the perfect profile, while each circle (either within or above each bar) shows the sample-percentage of that edge according to a sampled profile. The *overlap* for an edge is simply the minimum of the two sample-percentages. 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%.[7] 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. Overlap for the

[6]Total overhead with sample interval 1 is higher than exhaustive instrumentation (Table 1) due to the overhead of jumping back and forth between as original and duplicated code.

[7]The y-axis is reported as a percentage of all samples, so if the sampled profile overestimates the sample-percentage of some edge, it must also underestimate the sample-percentage of another edge.

Sample Interval	Full-Duplication Num Samples	Full-Duplication Overhead (%) Sampled Instrum.	Full-Duplication Overhead (%) Total	Full-Duplication Accuracy (%) Call-Edge	Full-Duplication Accuracy (%) Field-Access	No-Duplication Num Samples	No-Duplication Overhead (%) Sampled Instrum.	No-Duplication Overhead (%) Total	No-Duplication Accuracy (%) Call-Edge	No-Duplication Accuracy (%) Field-Access
1	1.1x10^7	167.2	182.2	100	100	6.7x10^7	118.2	269.1	100	100
10	1.1x10^6	26.4	29.3	99	100	6.7x10^6	22.8	79.5	98	100
100	1.1x10^5	4.2	10.3	98	99	6.7x10^5	3.6	61.3	97	99
1,000	1.1x10^4	0.8	6.3	94	97	6.7x10^4	1.0	57.2	93	98
10,000	1,137	0.1	5.1	82	94	6736	0.2	55.7	81	96
100,000	109	0.1	5.0	71	83	662	0.2	55.2	70	87

Table 4: Time overhead and accuracy of the sampled instrumentation averaged over all benchmarks, for a variety of sample intervals. Both call-edge and field-access instrumentation were applied together during the same run, and sampled using either `Full-Duplication` or `No-Duplication`. Columns labeled "Sampled Instrum." do not include the framework overhead, and thus represent the additional overhead introduced by taking samples. The columns labeled "Total" include the framework overhead, and thus represent the total overhead above the original, non-instrumented code.

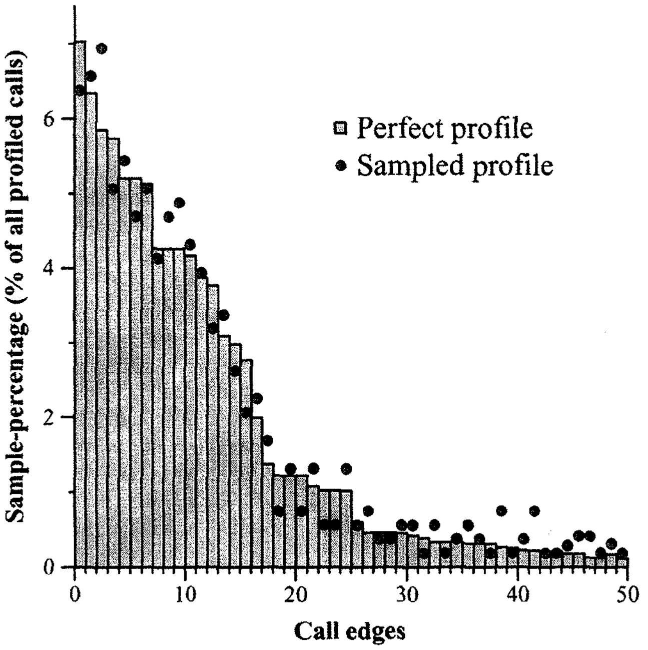


Figure 7: A graphical representation of the `javac` call-edge profile, illustrating an accuracy of 93.8% using the *overlap percentage* metric.

field-access profile was computed in the same way, but using sample-percentages for field-accesses rather than call-edges.

Table 4 reports the accuracy of the sampled profiles for both `Full-Duplication` and `No-Duplication`. As would be expected, increasing the sample interval reduces accuracy. At sample interval 1,000 the accuracy remained quite high, ranging from 93–98% for the both variations and both instrumentations, while the overhead remained low (excluding framework overhead, columns labeled "Sampled Instrum"). Even at sample interval 10,000, the accuracy is reasonably high, even though only $1/10,000^{th}$ of the execution is spent in the duplicated 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. However, the sample intervals in this table are increasing exponentially, so there is actually a large range of sample intervals (from 100 to 10,000) that offer high accuracy with low overhead.

These results confirm the low overhead and high accuracy of our technique, suggesting its effectiveness for driving online optimizations. The high accuracy also suggests that our technique could be useful for collecting offline profiles as well. One possible concern is that is possible for program behavior to correlate with our deterministic sampling mechanism, resulting in an inaccurate profile. For example, if a program performs some uncommon behavior every 1000th loop iteration, any sample interval that is a multiple of 1000 could result in the uncommon behavior being observed on every sample. Our experimental results suggest that this did not occur for benchmarks used in this study, however it could be a concern if accuracy is critical. Adding a small random factor to the sample interval (as done in [4]) could be used to reduce the probability of this worst case behavior, and could possibly even increase the accuracy in the expected case as well.

4.5 Jalapeño-specific optimization

This section presents one example of a JVM-specific optimization that can be applied to lower the overhead the `Full-Duplication` framework. This `Jalapeño-Specific` implementation reduces checking overhead by taking advantage of the fact that Jalapeño implements thread scheduling using *yieldpoints* (and therefore also applies to other systems that use yieldpoints, such as [18]). A yieldpoint is a sequence of instructions that checks whether it is time for the current thread to stop executing and give control back to the thread scheduler. Jalapeño currently places yieldpoints on all method entries and backedges to guarantee that there is a finite amount of time between yieldpoints. Once the code is duplicated using `Full-Duplication`, the yieldpoints can be moved to the duplicated code and removed from the checking code. As long as the sample-interval is always finite, the distance between yieldpoints is guaranteed to remain finite. This optimization does not affect the sampling accuracy, so the accuracy figures reported for `Full-Duplication` still apply.

The overhead of `Jalapeño-Specific` is reported in the two tables shown in Figure 8. Table (A) reports the framework overhead (full code duplication) while no samples are being taken. Because the sequence of instructions to imple-

Benchmark	Framework Overhead (%)
201_compress	1.4
202_jess	-0.5
209_db	1.6
213_javac	2.2
222_mpegaudio	-2.1
227_mtrt	1.9
228_jack	0.8
opt-compiler	4.8
pBOB	1.4
Volano	0.5
Average	1.4

Table (A) Framework Only

Sample Interval	Total Sampling Overhead(%)
1	179.9
10	27.6
100	8.1
1,000	3.0
10,000	1.5
100,0000	1.5

Table (B) Sampled Instrumentation

Figure 8: Overhead of the `Jalapeño-Specific` framework, with yieldpoint optimization performed. Table (A) presents the overhead of the framework only, with no sampling. Table (B) presents the total overhead of performing sampling (compared to non-instrumented code) averaged over all benchmarks for several sample rates.

ment a counter-based check is similar, but slightly different than that of a yieldpoint, removing yieldpoints and inserting counter-based checks introduces almost no overhead, and in some cases (_202_jess and _222_mpegaudio) performance is actually improved. The average overhead was 1.4%, as opposed to `Full-Duplication` which averaged 4.9% overhead. We have not yet determined why the yieldpoint optimization does not help much for `opt-compiler` and _213_javac.

Table (B) of Figure 8 presents the total overhead of sampling both instrumentations using the `Jalapeño-Specific` framework, relative to the non-instrumented code. This total overhead is similar to the total overhead of `Full-Duplication` (column 4 of Table 4), but converges on ~1.5% overhead rather than ~5%, due to the lower framework overhead. This Jalapeño-specific technique allows us to sample at intervals 1,000 and 10,000 with a total overhead (including sampled instrumentation) of 3.0% and 1.5%, respectively. This result is encouraging as it will allow the Jalapeño adaptive system [5] to perform online instrumentation while introducing overhead so small that in many cases it is hardly visible above the noise from one run to the next.

4.6 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; this bit is read by the yieldpoint instructions to determine when the executing code should yield to the thread scheduler. This threadswitch mechanism was modified to also set a *sample-bit* that is monitored by the checking code to allow samples to be triggered based on a timer-set bit. 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 5 compares the accuracy of both techniques (compared against a perfect profile using an overlap percentage) using `Full-Duplication` with field-access instrumentation. Clearly our framework is more accurate when driven by a counter-based trigger, which averaged 84% accuracy, as opposed to the time-based trigger, 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. The inaccurate attribution of samples is specific to our framework, so these results do not imply that time-based sampling is ineffective in general — only that it is less effective than counter-based sampling when used in our framework.

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

However, there is another advantage of the counter-based trigger that is independent of our framework; the counter-based trigger allows a flexible sample rate that is not restricted by any hardware or operating system limitations. As previously shown in Table 4, a smaller sample interval of 1000 or 10,000 achieves a much higher accuracy (94–99% for field-access profiling), while the overhead of sampling remains near zero; a faster time-based trigger is not available in our system.

5 Related Work

The contribution of our work is a simple and effective technique that uses instrumentation sampling to allow general instrumentation to be performed accurately with low overhead. The most novel aspect of our work is the `Full-Duplication` algorithm, which allows multiple instrumentation operations to be guarded by a single check, thus amortizing the cost of the check over multiple instrumentation operations. Previous work has touched on many of the techniques used by our framework, such as sampling and event counting; however, we do not know of any system that combines these techniques in a way that allows general instrumentation to be performed accurately with low overhead.

Traub et al. [36] 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 the executed edge. When used to guide superblock scheduling, the edge profiles collected by this technique produced speedups similar to 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. Our framework is also simpler to implement because it does not require the ability to perform binary rewriting.

Dynamo [9] employs a technique named *NET (Next Executing Tail)* [25] 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 [25] 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 performing 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] uses method invocation counters to determine when the call stack should be sampled. Similar to NET, Self-93 uses multiple counters (one per method) and made optimization decisions based on a single sample.

Previous work [16, 26, 37] has used the idea of wrapping each instrumentation operation inside a conditional branch (as is done in our `No-Duplication`). Calder et al. [16] and Feller [26] 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 [37] describe a *Java virtual machine profiler interface*, a general purpose mechanism for obtaining information from a JVM, which also uses boolean flag to conditionalize instrumentation operations. However, unlike our work, none of these approaches provide a mechanism to turn the flag on and off quickly to perform fine-grained sampling. Instead, the flag is used as a switch to turn exhaustive profiling off when it is not being used; while the flag is on, the full cost of instrumentation is incurred.

There has been work on reducing the cost of specific types of instrumentation [10, 11, 15, 26]; 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.

Previous work [8, 39] has used sampling to reduce the cost of building the *calling context tree* [3]. Burrows et al., [14] uses sampling to reduce the cost of profiling program values. These techniques are specialized examples of the general technique proposed by our framework, that is, using sampling to reduce the overhead of a previously exhaustive instrumentation.

Recent work [34] proposes new hardware support for reducing profiling overhead. In their approach, profiling events are collected and compressed in hardware before being fed to a software profiler. Their simulation shows accuracy and overhead results similar to that of our work for a variety of profiling types, although the hardware support for their technique does not exist in today's processors. Another fundamental difference between our work is the level of abstraction at which profiling is performed; their technique processes machine level information, whereas our approach can incorporate existing software profiling implementations.

6 Conclusions

This paper presents a general framework for performing instrumentation sampling, allowing previously expensive instrumentation to be performed with low overhead. The framework performs code duplication and uses compiler-inserted counter-based sampling to switch between instrumented and non-instrumented code in a controlled, fine-grained manner. The 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 previously expensive instrumentation techniques at runtime even without the ability to perform dynamic instrumentation or on-stack replacement. Our framework also makes it possible for many types of instrumentation to be collected at once, without the normal concern for overhead. 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.

An implementation and evaluation of the framework is presented using the Jalapeño JVM with two different examples of instrumentation, demonstrating that high accuracy can be achieved (93–98% overlap with a perfect profile) with low overhead (averaging $\sim$6% total overhead with a naive implementation). A `Jalapeño-Specific` implementation is also presented as an example of how hardware- or compiler-specific optimizations can be used to further reduce overhead, resulting in an average total overhead of $\sim$3%.

Acknowledgments We are grateful to Stephen Fink, David Grove, Michael Hind, Nasko Rountev, and Peter Sweeney for their helpful discussions and comments on an earlier draft of this work, as well as the anonymous reviewers for their helpful feedback. We also thank Sharad Singhai for the use of his field-access profiling implementation, as well as all of the Jalapeño members who helped facilitate this research. Finally, we thank Michael Hind for his support of this work.

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] 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.

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

[11] 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.

[12] 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.

[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] 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 *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

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

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

[17] 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.

[18] 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).

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

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

[27] 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.

[28] J. K. Hollingsworth, B. P. Miller, M. J. R. Goncalves, O. Naim, Z. Xu, and L. Zheng. MDL: A language and compiler for dynamic program instrumentation. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques (PACT '97)*, pages 201–212, San Francisco, California, Nov. 10–14, 1997. IEEE Computer Society Press.

[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] S. Sastry, R. Bodik, and J. E. Smith. Rapid profiling via stratified sampling. In *To appear in 28th International Symposium on Computer Architecture*, 2001.

[35] 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.

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

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

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

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

[40] 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.