

Leveraging Disposable Instrumentation to Reduce Coverage Collection Overhead

Kalyan-Ram Chilakamarri and Sebastian Elbaum
Department of Computer Science and Engineering
University of Nebraska - Lincoln
{chilaka, elbaum}@cse.unl.edu

Abstract

Testers use coverage data for test suite quality assessment, stopping criteria definition, and effort allocation. However, as the complexity of products and testing processes increases, the cost of coverage data collection may grow significantly, jeopardizing its potential application. To mitigate this problem this paper presents the concept of “disposable coverage instrumentation” – coverage instrumentation that is removed after its execution – through two techniques: local disposal and collective disposal. A Java virtual machine was extended to support these techniques, and their potential is shown through two studies utilizing the Specjvm98 and Specjbb2000 benchmarks. The results indicate that the techniques can reduce coverage collection overhead by an order of magnitude over state of the art techniques.

Keywords: software instrumentation, test coverage, empirical study.

1 Introduction

Coverage measures are important for engineers to quantify the level of completeness and identify weaknesses in their test suites. These measures represent the percentage of program entities executed by a test suite, where an entity can range from simple code blocks, to more complex targets obtained from representation models of the code (e.g., data flow graph model and coverage of d-u type entities [11]).

Independent of the coverage entity type, obtaining coverage data for a program P and test suite T often follows a common procedure: 1) identify target coverage entities E in P , 2) build P' by inserting coverage probes H into P to enable the capture of E during execution, 3) execute test suite T on P' to generate data trace D , and 4) process D to obtain E coverage information.

In spite of its deceptive simplicity, this process can be challenging when applied to a complex program and testing environment. As more entities are profiled, the coverage probes in program P' associated with those entities can generate an unacceptable execution overhead (reported to range from 10% to 390% [4, 15]). This overhead becomes more noticeable in the presence of large test suites where test cases repeatedly execute the same probes.

This paper presents an approach to address these challenges based on the removal of coverage probes after they are executed. This approach is called *disposable coverage instrumentation*. Since the computation of software testing coverage measures only requires to discern whether an entity is executed, coverage probes can often be removed after the first time their corresponding entity is exercised without loss of information

(Section 3.4 discusses the applicability of this approach to various coverage criteria). Pavlopoulou et al. introduced an instance of this approach called *residual testing*, where the probes in a program are removed (and the program re-built) after each test case execution [22]. When applied to four small applications, residual testing showed that instrumentation execution overhead to collect coverage data becomes negligible after the execution of a few test cases.

In spite of the potential benefits of disposable instrumentation to reduce execution overhead, its application remains challenging in at least two aspects:

- It normally requires that the target program to be stopped, re-built, and re-executed. (Some exceptional approaches that support run-time instrumentation removal such as Dynainst and JFluid are discussed in Section 2.) The potential savings due to removed probes is then overshadowed by the costs of long-running tests that need to be re-run from scratch after every build, or by long build processes that must be repeated after each test case execution.
- It cannot take advantage of multiple instances of the program that may run in parallel. This is important when testing is performed in parallel configurations, and it is particularly relevant for recent efforts that attempt to leverage multiple field instances of a released software to capture the execution of entities not exercised in-house [9, 19].

This paper introduces two disposable instrumentation techniques that address those challenges. The first technique, **local disposal (LD)**, disposes of each coverage probe h in program P as soon as h is executed, without requiring for P to stop. The second technique, **collective disposal (CD)**, adds a cooperative flavor to LD by disposing of probes executed by any running instance of P . Both techniques are implemented for the Java language by altering a Java Virtual Machine (Section 2.3 provides the rationale and more details on the targeted environment). Last, the paper presents two studies showing the techniques' benefits as well as identifying potential scenarios where they are likely to perform well.

This paper is organized as follows. Section 2 provides the necessary background information and related work. Section 3 introduces the LD algorithm and implementation, and assesses the LD performance under the Specjvm98 benchmark suite [26]. Section 4 introduces the CD architecture and implementation, and evaluates its performance under various scenarios within the Specjbb2000 server benchmark [27]. Section 5 discusses the threats to the validity of our findings. Section 6 explores the use of disposable instrumentation within more efficient virtual machines architectures. Section 7 summarizes the findings and presents future research avenues.

2 Background and Related Work

This section begins by introducing a sample of coverage tools for Java programs in Section 2.1. Section 2.2 presents the related efforts in reducing coverage profiling overhead. Section 2.3 describes some of the Java Virtual Machine (JVM) characteristics that are relevant to this work.

2.1 Coverage Tools

There are many commercial and freely available tools that help to collect coverage data. For the Java programming language for example, Clover [6], PureCoverage [23], and Jcoverage [21] are some of the most popular commercial tools, while Emma [10], JVMDI [12], and GroboCodeCoverage [17] are some of the open-source coverage tools the authors have used. The coverage data most commonly provided by these tools is class, method, and statement coverage. These tools differ primarily in the presentation of the coverage information and the capabilities to integrate with other tools and environments. For example, Clover was designed to easily integrate with Ant (Java build manager), while Emma provides facilities to visually and interactively explore coverage data at different granularities (class, method, block, line).

Another differentiating factor between these tools is how they obtain coverage information. The three most common mechanisms are: through the Java debugging interface, through the instrumentation of the Java class files with the assistance of bytecode manipulation libraries (e.g., BCEL [7] or Soot [14]), and through the modification of bytecodes at loading time (when they are being setup in the Java virtual machine). As it will be shown, the main distinction between the instrumentation techniques employed by these tools and the approach introduced in this paper is that the later one includes *run-time probe removal* capabilities as a mechanism to reduce profiling overhead.

2.2 Reducing profiling overhead

There have been many research efforts to reduce profiling overhead by more precisely determining where probes are needed. The idea is to identify what locations in the program must be instrumented to get an accurate description of the program behavior in terms of coverage. For example, Agrawal et al. introduced various techniques based on domination relationships to reduce the number of probes in the target program without loss of coverage information [2], Ball et al. introduced a technique to reduce the number of probes to capture path coverage through the identification of key edges with predetermined weights [5], and Bowring et al. sampled across the potential set of probes within the same sub-tasks [19]. Although the techniques presented in this paper use some of the previously proposed techniques to reduce the required initial number of instrumentation probes, the proposed techniques added capability of instrumentation removal is aimed at continuously reducing the number of probes during program execution.

The second relevant related research thrust is the reduction of profiling overhead by removing instrumentation probes after a certain condition is met. Pavlopoulou and Young introduced residual test coverage monitoring where software is deployed with the residual probes that have not been covered by the in-house testing process [22]. The prototype tool for residual test coverage [3] re-instrumented programs at the beginning of each execution. The techniques introduced in this paper also aims at reducing coverage overhead, but the removal of instrumentation happens at run-time, without the need to stop the program.

Tikir and Hollingsworth did introduce a set of techniques and a tool-set that enabled probe removal at run-time [28], which inspired part of the work in LD. Their approach replaces instrumentation points in a program's executable with pointers to memory addresses containing the original code and also additional code to collect coverage information (this pointer-type structure used to redirect execution flow is often

known as a trampoline). When the code is executed, the pointer to the trampoline is removed and the original code reallocated, disposing of the coverage related instrumentation. The techniques presented in this paper are different from Tikir’s in two aspects. First, they target the java programming language. This language introduces a distinct scenario since programs are executed in the context of virtual execution environments. Second, one of the new techniques performs *distributed* instrumentation disposal (CD), which takes into account the coverage obtained from multiple program instances. In general, tool-sets like Tikir’s that modify executables need to be continually updated and expanded in order to remain useful as compilers and architectures change and diversify. As virtual machines evolve, the implementations presented here may suffer similar challenges. However, with the current migration of instrumentation facilities into the APIs of existing virtual machines the efforts to address such challenges are likely to be reduced [8].

Orso et al. introduced a mechanism to provide run-time update capabilities for Java classes. They proposed to wrap classes to allow a transparent update through class-level swapping in the JVM reloading mechanism [20]. The same approach could be used in the context of collecting coverage data by dynamically manipulating instrumentation at the class level, periodically re-uploading a class with less instrumentation in each upload. However, the addition of wrapper classes and the repeated class loading to dispose of probes is not an efficient mechanism to reduce coverage collection overhead (e.g., class uploading includes checking the class, loading its binary into memory, defining it, resolving it). The Jfluid toolkit takes this approach a step further by providing method-level reloading [8] through a specialized API introduced into the JVM. Although more efficient, Jfluid still requires reloading (at a smaller granularity) and it currently presents restrictions on what methods can be reloaded.

2.3 JVM

The JVM is a platform to run Java programs. Java programs consist of classes implemented in the Java programming language. These programs are compiled to produce class files containing instructions that are understandable by a JVM. The instructions in a class file are called bytecodes, which resemble traditional machine code. The JVM instruction set has 256 instructions and the JVM’s job is to perform the functionality expected out of each bytecode it encounters during execution. The simplicity of this instruction set, its potential for direct mapping to source code, and the availability of tools for bytecode manipulation, make the bytecode level an ideal candidate to insert coverage probes. For the studies included in this paper, the Byte Code Engineering Library (BCEL) was utilized to analyze and manipulate the class files containing bytecodes [7]. The BCEL offers the facility to place a probe at a particular point in the program, while transparently handling the necessary adjustments for redirections and stack size adjustments.

A JVM implementation can perform the actions specified in the class files in various ways as long as it respects the JVM specifications [16]. JVM implementations vary in memory organization, execution mechanism (e.g., interpretation, compilation, or selective compilation), and communication to each platform. To demonstrate the feasibility of the techniques introduced in this paper, one of the authors modified the execution engine of the open source implementation of JVM by the Kaffe group (version 1.1.4) [13]. Kaffe’s JVM provides configurations to run it as an interpreter or as a just-in-time compiler. The interpreter converts

bytecodes to native code during every execution, whereas a Just-In-Time (JIT) compiler converts bytecodes to native code once and retains the native code for subsequent executions. Retaining native code implies additional initial setups but avoids repeated code conversions, often leading to better performance. Hybrid mechanisms such as Hotspot [1] aim for a compromise, performing in interpreted mode except for those methods with high execution frequency, for which it retains the native code.

The flexibility and opportunities of manipulation offered by the Java execution environment was an important reason for focusing on it. Java's popularity in the development of servers to support non-stop web applications (e.g., IBMs Websphere, BEAs WebLogic, Oracles Application Server, GNUs JBOSS) also influenced the direction of this work which targets the Java programming language.

3 Local Disposal

Local Disposal (LD) removes coverage probes as soon as they are executed. Its objective is to reduce the overhead associated with the repeated execution of coverage probes with no added value. The technique is especially appealing for programs with:

- linear execution, where repetitive execution patterns lead to multiple executions of the same coverage probes
- long-running test cases over similar functionality, where the overhead associated with the multiple executions of coverage probes is compounded by the number of tests and their duration
- expensive build processes that cannot be repeated to enable the extra compilation-linking cycles to remove the coverage probes.

3.1 Algorithm and Implementation

Coverage probes normally consist of an invocation to a *Collector* method. The invoked *Collector* utilizes the covered entity's *id* to update some type of coverage vector. In general, the probes are meant to be simple to reduce overhead and any additional functionality (e.g., analyze coverage patterns or save coverage vector) is pushed to the *Collector*. The *Collector* could also be implemented as a class if additional functionality is required. At the end of program execution the coverage vector is stored in a file. Figure 1 illustrates this process through a snippet of Java code, which includes instrumentation probes and a *Collector* method to capture block-level coverage.

As mentioned before, LD is meant to remove coverage probes as soon as they are executed. It was conjectured that, even if probe removal is a relative expensive operation, long-running programs with repetitive usage patterns that would execute the same probes repeatedly, could benefit from this approach. Assuming the common scenario described in the previous paragraph, the probe removal is performed in two steps. First, the occurrence of the invocation to the *Collector* method (execution of the coverage probe) must be detected. Second, after the *Collector* has finished its execution, the invocation to *Collector* must be removed from the program.

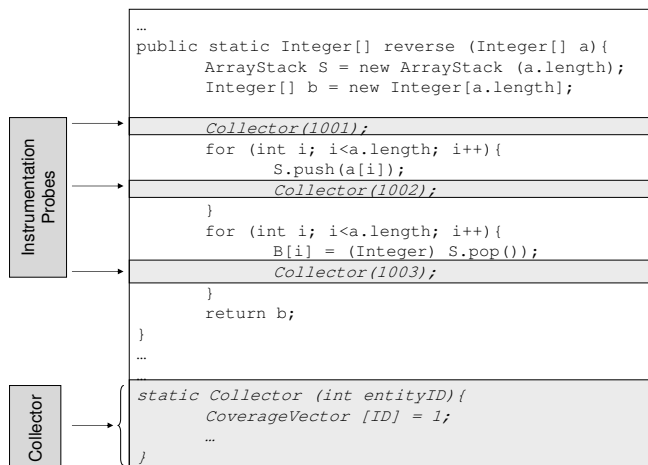


Figure 1: Sample instrumented code to collect block coverage data.

The invocation removal can happen at two levels: application level and JVM level. At the application level, the removal of an invocation can be implemented through wrapper classes that redirect the execution (e.g., hot-swapping [20]). At the JVM level, the removal can be implemented by run-time code modification. There are several tradeoffs between these approaches. The instrumentation removal at the application level offers a lot of flexibility but it can impose tremendous overhead due to the creation of and the interactions with the additional wrapper classes, and the repeated class loading. Modifying bytecodes within the JVM can be more efficient but this requires an intimate knowledge of the particular implementation and is JVM-specific. Ideally, one would implement LD by accessing an API that allows code modification at run-time, avoiding the problem. However, current JVMs do not provide yet such capability.

Given that overhead reduction is the main objective, the Kaffe JVM implementation was modified to enable LD ¹. More specifically, the execution engine within the JVM was modified. As mentioned in Section 2.3, JVMs can offer several run-time configuration modes. In this section, the algorithm and implementation of LD for the interpreter mode (the simplest but also the one supported across all JVMs) is presented. In Section 6, the algorithm, implementation, and preliminary results for LD within JIT, a more complex but efficient execution mode, will be presented.

To implement LD with a JVM in interpreter mode, a simple bytecode rewrite operation to nullify the calls to the `Collector` method was used. Since overhead is the primary concern, the expensive object-creation operation was avoided by keeping the `Collector` as a static method [25]. At execution time, for each statically invoked method, the rewrite process analyzes the method's signature at the end of its execution to determine whether the method is the `Collector` method. If it is not the `Collector` method, normal execution continues. If the `Collector` method is at the top of the stack, then the process nullifies the bytecode corresponding to the invoke instruction to `Collector` (`invokestatic` bytecode followed by its 16-bit argument) on the caller by replacing it with the bytecode 0 corresponding to the `nop` operation instruction. This nullification is

¹Although these modifications are tied to the particular JVM implementation, initial explorations indicate that the migration to IBM's JVM (jikes) and Sun's JVM can be performed with minor effort. In any event, some form of restricted run-time bytecode modification is expected to be incorporated in future Java APIs (as in JFluid [8]) which mitigates this challenge.

Algorithm 1 LD within JVM with interpreter execution engine

- 1: Caller invokes callee
 - 2: Build callee frame
 - 3: Push arguments onto frame
 - 4: Pass execution control from the caller to callee
 - 5: Execute callee’s bytecodes
 - 6: **if** *callee.methodName* = *Collector* **then**
 - 7: *Overwrite Collector invocation in caller with nop*
 - 8: **end if**
 - 9: Return control to the caller
-

semantically equivalent to the removal of the coverage probe. Using nullification does not truly dispose of the instruction space, but it is advantageous in that it does not require bytecode rearrangement. As a result of this process, the coverage probe is “removed” after its first execution.

The corresponding algorithm is presented in Algorithm 1. Note that only two new instructions (6 and 7) were added to the execution engine of the Kaffe JVM. These instructions are conceptually simple and only require the addition of approximately a dozen LOC (lines of code) to one file in the JVM. The major challenges in the LD implementation was to achieve the necessary level of understanding of the JVM to make the changes without impacting many of its sensitive activities such as the bytecode integrity verification or the handling of multithreading.

Note that instruction 6 in Algorithm 1 implies an additional comparison within the JVM for each static method invocation (calls to methods defined as static). In the presence of static methods this would signify additional overhead. Still, static methods are not prevalent enough across object-oriented applications for this to become significant (e.g., 3% of the methods in Specjvm98 benchmarks are defined as static). Furthermore, it is conjectured that, as the required granularity of coverage information decreases, the gains of instrumentation removal from the application will be more noticeable in spite of this additional comparison cost within the JVM. Last, although arguments have been provided for the potential of the LD technique to reduce the overhead associated with obtaining coverage data, empirical evidence is needed to quantify that potential. Such empirical evidence is provided in the following section.

3.2 Empirical Study

Research Question. What is the potential benefit of LD? This section investigates the level of reduction in coverage collection overhead (in terms of execution time) resulting when applying LD on top of existing instrumentation techniques.

Variables. The independent variables are the coverage instrumentation granularity and the instrumentation technique. Two coverage granularity levels were targeted: block and method, and the following techniques were considered:

1. No instrumentation (No-I). This technique defines a lower bound for overhead.
2. Instrumentation of all target entities (Basic-I). This technique is applicable at the block level and method level.

Table 1: Specjvm98 benchmark programs

Name	Description	Blocks	Blocks-Dom	Methods
jess	Expert system to solve puzzles	2672	1603	673
raytrace	Ray tracer rendering	663	394	173
db	Database builder and transaction processor	313	184	34
javac	Java compiler with sample programs	6133	3701	1179
jack	Java parser generator	2220	1350	302
compress	Stream compressor	258	161	181

3. Disposable instrumentation capabilities on Basic-I (LD-Basic-I). This technique is also applicable at the block level and method level.
4. Refined instrumentation of blocks based on their domination relationship (Dom-I). Similar to Tikir et al. [28], the technique uses dominator tree relationships to curtail the initial number of instrumentation probes in the target program at the block level.
5. Disposable instrumentation capabilities on Dom-I (LD-Dom-I). This technique is applicable only at the block level.

The dependent variable is the overhead associated by each technique at certain granularity for a specific program. To quantify the overhead, the execution time was measured through the benchmark’s built-in timer (in seconds).

Object. The Specjvm98 [26] benchmark suite (with its default configuration settings) was used to evaluate the techniques. Table 1 gives a brief description of the programs in the benchmark as designed by the non-profit organization Standard Performance Evaluation Corporation (SPEC) to measure the performance of Java virtual machines. The table also reports the number of blocks, dominator-blocks, and methods instrumented by the techniques.

Design. For the study, each granularity level was combined with each potential technique, and with each program in the benchmark. Each combination (42 in total) was then ran, and their associated overhead measured. To reduce the variation due to uncontrolled sources of variation, this study was performed five times, and the means and deviations computed across the five runs. The runs were executed on a machine with a 1.6GHz Pentium processor and 512 megabytes of memory, running Linux 9.0.

3.3 Results and Analysis

Table 2 presents the results for block instrumentation. It can be seen that the overhead caused by Basic-I ranges from 31% to 112% (fourth column), while LD-Basic-I causes overhead ranging from 6% to 12% (sixth column) when compared with No-I. The average LD-Basic-I gain over Basic-I ranges from 25% to 99%. In all cases, the standard deviations across runs are fairly consistent between techniques.

Table 3 presents the results when benchmarks were instrumented utilizing the dominator algorithm. Observe that the overhead caused by Dom-I ranges from 12% to 98% (fourth column), while LD-Dom-I

Table 2: LD: Execution time for basic-block instrumentation

Program	No-I		Basic-I		%Overhead w.r.t No-I	LD-Basic-I		%Overhead w.r.t No-I	%Gain w.r.t Basic-I
	Mean	StdDev	Mean	StdDev		Mean	StdDev		
jess	521	12	1060	36	104	562	18	8	96
raytrace	494	7	1048	27	112	555	25	12	99
db	645	42	857	8	33	712	25	10	22
javac	533	10	886	35	66	588	11	10	55
jack	322	15	422	20	31	341	15	6	25
compress	1670	24	3426	130	105	1820	52	9	96

Table 3: LD: Execution time in seconds for dominator-tree instrumentation

Program	No-I		Dom-I		%Overhead w.r.t No-I	LD-Dom-I		%Overhead w.r.t No-I	%Gain w.r.t Dom-I
	Mean	StdDev	Mean	StdDev		Mean	StdDev		
jess	521	12	964	11	85	542	21	4	81
raytrace	494	7	980	28	98	530	7	7	91
db	645	42	721	30	12	666	56	3	9
javac	533	10	790	13	48	542	23	2	46
jack	322	15	396	10	23	336	19	4	19
compress	1670	24	3202	118	92	1760	46	5	87

overhead ranges from 2% to 7% (sixth column) when compared against No-I. Overall, even when employing the dominator algorithm to reduce the number of instrumentation points, utilizing LD provided additional overhead reductions.

It may be conjectured that as the granularity of the coverage data is increased, and the number of required probes is reduced, the benefits of LD would diminish. This conjecture was confirmed by the results at the method-level. Table 4 shows that LD-Basic-I performs on average better than Basic-I at the method level even though the margin of improvement is less than at the block level (average gains for all programs over Basic is 34%). Note, however, that for one program, db, LD-Basic-I incurs an overhead of 4% more than just using Basic-I mainly because the overhead of utilizing Basic-I at the method-level is less than one percent, which leaves limited chances of improvement for LD-Basic-I. However, for some programs like raytrace, the gains achieved through LD over Basic-I at the method-level are still considerable (69%).

As stated previously, it is expected for LD to be most effective in the presence of programs that are likely to have repetitive execution patterns. This expectation was explored by computing the Redundant Probe Execution, which is defined by $RPE = \sum ProbesExecuted - \sum UniqueProbesExecuted$. To enable the comparison between programs, the RPE for each program was divided by the execution time of compress (the program with the longest execution time). This adjustment was meant to account for the fact that programs in the benchmark had different execution times which may artificially bias their RPE (e.g., long running programs may increase their chances of executing the same patterns leading to smaller RPE s). Figure 2 depicts the normalized RPE values for all programs against the percentage gain obtained on each program for Dom-I. There is a positive and linear relationship (as evidenced by the linear fit in the figure) between the gains obtained through the application of LD and the normalized RPE values, confirming the conjecture that programs with higher RPE values are likely to benefit the most from LD.

Table 4: LD: Execution time in seconds for method-level instrumentation

Program	No-I		Basic-I		%Overhead w.r.t No-I	LD-Basic-I		%Overhead w.r.t No-I	%Gain w.r.t Basic-I
	Mean	StdDev	Mean	StdDev		Mean	StdDev		
jess	521	12	756	13	45	532	7	2	43
raytrace	494	7	861	68	74	520	16	5	69
db	645	42	651	34	1	679	21	5	-4
javac	533	10	770	21	44	545	13	2	42
jack	322	15	360	20	12	330	9	2	10
compress	1670	24	2511	59	50	1721	32	3	47

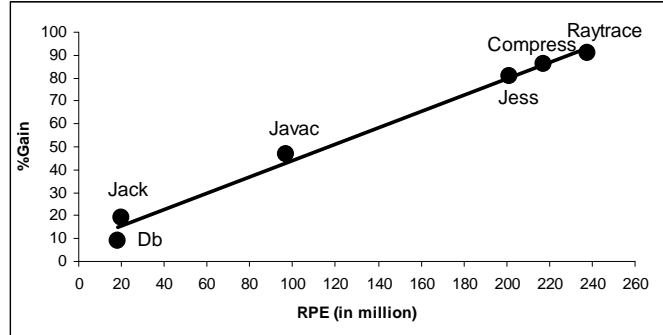


Figure 2: Gain vs. Normalized RPE.

3.4 On the limits of LD to assist coverage collection

The results have provided evidence about the potential of LD to improve the efficiency of the coverage collection process at the method and block level, which are the most popular metrics obtained through coverage assessment tools (see Section 2.1). Such benefits may extend to the collection of other types of coverage data like branch, predicate, exception as long as: 1) they include an instrumentation mechanism similar to the one described at the beginning of Section 3.1 (coverage probes that consist of an invocation to a *Collector* method), 2) the conditions to determine whether to remove or not to remove a probe are cost-effective. This second aspect is now further elaborated.

When collecting method and block coverage data, the decision process to remove a probe is trivial. In such cases, the probe invoking the *Collector* method has already achieved its data collection purpose, it is not required in any future executions, and can be removed. The same could apply to the collection of branch and condition coverage if the predicates are equipped with the prescribed instrumentation mechanism for each potential branch outcome.

However, if the target coverage data requires the analysis of data from multiple probes, then the decision process for probe removal is no longer trivial. For example, multiple probes are required to collect d-u path coverage, and the simple execution of a probe does not imply its immediate removal (e.g., a definition might relate to multiple uses). Under these circumstances, after each probe is executed, there must be a process to analyze whether all the paths relevant to a probe have been covered before the probe is removed. The development and evaluation of such analysis processes will be the subject of future work.

4 Collective Disposal

Collective Disposal (CD) enhances LD by incorporating coverage data from multiple running instances of the software to avoid executing probes that have already been covered elsewhere. The objective is to leverage many running instances of the software, avoiding the collection of the same coverage data in repeated instances. This technique would be especially valuable when:

- software has many potential configurations (configuration explosion problem [18]) that are tested in parallel
- software is deployed with coverage probes to the field to corroborate or enhance the in-house validation activities (e.g., beta testing or continuous testing)[9].

4.1 Algorithm and Implementation

CD activities are supported by a communication channel between the deployed instances. Each deployed instance is equipped with a coverage vector indicating whether an entity has been covered or not by any of the deployed instances. If the vector indicates that an entity has not yet been covered by any of the deployed instances, then the procedure is similar to applying LD. If the coverage vector indicates that an entity has been executed by another instance, then the probe is disposed without invoking the *Collector* method.

Algorithm 2 provides more details on this process. Three instructions are noticeably different from Algorithm 1. First, in the second instruction, the algorithm preventively checks whether the invocation to the *Collector* method has already been exercised by any of the deployed instances. If the particular invocation has not been performed by any other instance, then this algorithm operates similarly to the LD algorithm, executing the *Collector* code and overwriting the invocation instruction. The second difference is the 9th instruction where it proceeds to disseminate the knowledge of the newly covered statement to the other instances, and obtain an updated coverage vector with the knowledge gathered by the other instances (14th instruction). The third difference arises when a *Collector* invocation has already been executed by other instances; in this case the algorithm proceeds to overwrite the invocations without even building the corresponding stack frame (12th statement). Note that this situation implies less overhead than the LD solution since the *Collector* method is not executed.

There are several ways to provide a communication channel between the deployed instances (e.g., peer to peer, by clusters, client-server). Following a common scenario in which a company has a centralized server for deployment management, the prototyped implementation utilizes a centralized server as the communication channel. The server maintains a coverage vector indicating the probes executed by the deployed instances. There are also several potential policies to disseminate and update the coverage vector. Although frequent disseminations and updates can improve the efficiency of the probe disposal, these processes carry an associated communication overhead. One simple policy could trigger updates and disseminations at a regular interval. However, determining the right interval is not simple. Intervals that are too small could generate unnecessary communication overhead, while intervals that are too large may miss chances to reduce overhead. Furthermore, the size of the intervals should be adjusted over time as the number of uncovered

Algorithm 2 CD within and across JVMs.

```
1: if Invocation callee = Collector then
2:   if CoverageVector [invocation] = FirstTime then
3:     Caller invokes callee
4:     Build callee frame
5:     Push arguments onto frame
6:     Pass execution control from the caller to callee
7:     Execute callee's bytecodes
8:     Overwrite Collector invocation with nop
9:     Disseminate/Update Coverage Vector
10:    Return control to the caller
11:   else
12:     Overwrite Collector invocation without executing Collector
13:   end if
14: end if
```

entities is reduced. An alternative policy could use temporary buffers at each instance containing newly covered probes, and trigger the updates when the buffers are full. This policy may imply more resources for the buffers and some additional processing, but it has the advantage of triggering the transfers as a function of changes in the coverage vector. Note that a buffer of infinite size would result in a performance equivalent to LD. The policy set for the following studies is the simplest of the latter type, with a buffer of size one.

Two types of update triggers were included. First, a deployment instance gets an updated vector at the beginning of the execution when the application's classes are being loaded. The relative long and slow initial class loading process hides, to some degree, the coverage vector retrieval from the server. This update provides an updated vector to start an instance execution and leverages the knowledge collected up to that point. Second, each running instance gets an update every time it performs a disseminate operation to leverage the connection already established with the server.

As previously mentioned, it was decided that the dissemination of the information (through the connection to the server) is performed when a coverage probe is executed for the first time. This policy will generate more overhead at the beginning of deployment when probes are not yet covered, but it has the advantage of keeping an updated vector at the server at all times from which later deployed instances can benefit. By forcing an update after each dissemination, this policy also reduces the chances of an instance executing more than one coverage probe that was already covered by other instances.

In the current implementation of CD, the Kaffe JVM was modified to hold the probe coverage vector, include the Algorithm 2, and communicate with the server through a simple socket (the communication takes place at the virtual machine level). A server, which is implemented in C to match the Kaffe JVM implementation, creates multiple threads to handle the connections with the deployed instances, and maintains the centralized coverage information.

4.2 Empirical Study

Research Question. What is the potential benefit of applying CD? In particular, the study attempts to assess the reduction in coverage collection overhead (in terms of execution time) caused by CD under various

simulated scenarios.

Variables. Two independent variables have been identified: instrumentation technique and deployment scenarios. Four instrumentation techniques at the block-level are considered: 1) No-I, 2) Dom-I, 3) LD-Dom-I, and 4) CD (which by default utilizes the Dom algorithm for initial probe allocation). The second independent variable is the deployment scenario for CD. The effectiveness of CD may be affected by whether the instances are deployed concurrently, sequentially, or something closer to a diffusion adoption model [24]. First, a scenario in which all clients start executing concurrently is considered. This is the worst case scenario for CD since there are no “late starters” that can leverage the knowledge from previously deployed instances. Second, the study considers a scenario with pure sequential deployment where an instance would start executing only after another one ended, presenting a somewhat ideal scenario for CD. Third, a deployment scenario where the system is first adopted by a few users, then it starts to rapidly propagate as it gets visibility until its market is saturated, at which point the number of deployments starts to decrease. This scenario is called distributed, and it uses a normal curve to simulate it, with a peak of 15 concurrent deployments.

The dependent variables for assessing the improvement in coverage data collection are: 1) the average execution time across all deployed instances, and 2) the average throughput, which is the de-facto measure to evaluate the performance of the chosen benchmark.

Object. The Specjbb2000 [27] benchmark was utilized to evaluate the CD technique. This benchmark provides a Java server application that has a 3-tier structure. The first tier of the architecture represents warehouses that generate business transactions such as placing orders or checking orders’ status from a central server. The second tier is the business logic at the server that processes the transactions and generates requests to the third tier made up of a simulated database. The size of Specjbb2000 is nearly 26 KLoc with 732 methods and 4369 blocks. Specjbb2000 behavior can be exercised in somewhat different ways by adjusting several parameters in a configuration file such as the number of the warehouses supported by the server.

Design. To evaluate CD, an infrastructure to deploy the instances of the benchmark was required. A cluster of 38 identical computing machines powered by dual AMD 1.6 GHz athlon processors with 1GB memory served as the infrastructure. All the computing nodes were interconnected by an internal ethernet network without external interference. One node was used as the CD server and central repository of coverage information, the other 37 nodes contained the instrumented Specjbb2000 application with its satellite clients.

The Specjbb2000 was instrumented at the block level using the dominator tree algorithm (Dom-I), which resulted in 2219 probes. This instrumented version was then deployed to each node, together with a pool of configuration files. The pool consisted of 15 files with distinct property settings that were meant to exercise the application in various ways. All properties that were configurable in the benchmark were modified in the study: the number of warehouses that each Specjbb2000 had to deal with (range of 8 to 16 in either incremental or sequential form), the garbage collection strategy (trigger when the heap is full or right after

Table 5: CD: Average execution times in seconds across all deployed instances.

Technique & Scenario	Time		% Overhead w.r.t. No-I
	Mean	StdDev	
No-I	1897	93	-
Dom-I	2575	172	35
LD-Dom-I	2086	150	10
CD - concurrent	2275	208	20
CD - distributed	2003	118	6
CD - sequential	1923	131	2

each warehouse request), and the type of report generated (summarized or per warehouse).

Each combination of technique and scenario was then evaluated. For each technique and scenario, each node started Specjbb2000 (at appropriate intervals in the case of CD) with a randomly chosen property file, which made the benchmark exhibit a somewhat distinct behavior by randomly selecting a property file during its initiation. Throughout the execution of each Specjbb2000 in each node, the process kept track of each time a probe was covered, and how long it took for the application to finish execution. After all the Specjbb2000 instances finished executing, their execution times was computed and averaged. It is important to note that although the load may have varied significantly across nodes (depending on what property files were chosen by each instance of Specjbb2000), there were enough nodes in the infrastructure that the distribution of files across nodes would be similar as new techniques-scenarios were studied. Still, this process was performed twice to obtain a larger number of observations that account for any additional potential variability. Once the study setup was completed, the execution of the study required exclusive utilization of the available infrastructure for slightly over 4 days.

4.3 Results and Analysis

Table 5 summarizes the findings for each of the techniques on each of the scenarios. Dom-I has an average collection overhead that is 35% (678 seconds) higher than No-I across all deployed benchmark instances. Adding LD reduced that overhead to 10% (189 seconds). When deployment instances were started concurrently, the average execution time per instance for CD was 20% (278 seconds) higher than when No-I was performed, and overhead reduction of 15% (300 seconds) took place over Dom-I.

It is interesting to note that employing CD in a purely concurrent scenario signifies a penalty of 10% over just using LD-Dom-I. This was caused by a number of factors. First, the large number of interactions with the server from simultaneously deployed benchmarks, which caused a server collection bottleneck (server response time varied from 0.3 to 0.5 seconds). Second, the extra overhead in disseminating the new coverage information to the server. Finally, the limited opportunities to leverage the knowledge across instances since all of them execute at the same time (instances deployed concurrently executed an average of 60% of the probes, and the CD server was contacted every 0.28 seconds on average).

CD on sequential and distributed scenarios present average overhead reductions of 33% (652 seconds) and 29% (572 seconds) over Dom-I, and 8% (163 seconds) and 4% (83 seconds) over LD-Dom-I. Under these scenarios, deployed instances can better leverage the collective coverage pool knowledge by removing

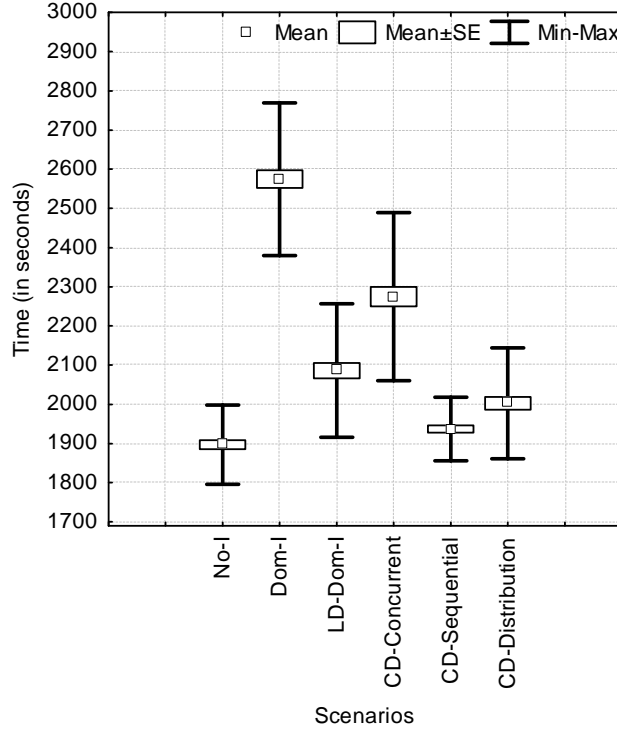


Figure 3: Comparison of execution time in seconds.

probes from some instances before they are even invoked. In the sequential scenario, there were fewer update requests to the server, which help to improve response time (ranged from 0.1 to 0.3 seconds), and each deployed instance utilized the information about the previously executed probes (deployed instances executed on average 16% of their probes, and the CD server was contacted every 0.46 seconds on average, more rarely than the in the concurrent scenario). For the distributed scenario, the deployed instances executed on average 37% of the probes in the application and the CD server was contacted every 0.30 seconds on average.

To get a better understanding of the distribution of execution time per technique and scenario, a box plot graph was generated, Figure 3, where each box depicts the 74 observations (2 simulations utilizing the 37 deployed instances) corresponding to the execution times collected for each technique and scenario combination. The line embedded in each box marks the mean value, the edges of the box are bounded by the standard error, and the whiskers extend to the minimum and maximum. Interestingly enough, the utilization of CD does not seem to add variation to the performance of Dom-I. The average performance of LD-Dom-I, CD-Concurrent, CD-Sequential, and CD-Distributed was better than the best Dom-I performance on any deployed instance. Furthermore, the average performance across deployed benchmarks when utilizing CD-sequential or CD-distributed was less than the worst case for No-I.

The average server throughput across the three scenarios for each of the techniques is shown in Figure 4. Similar to execution time, the LD-Dom-I, CD-sequential and CD-distribution techniques were beneficial, increasing the server throughput as compared to Dom-I. It is interesting to note again how CD outperforms

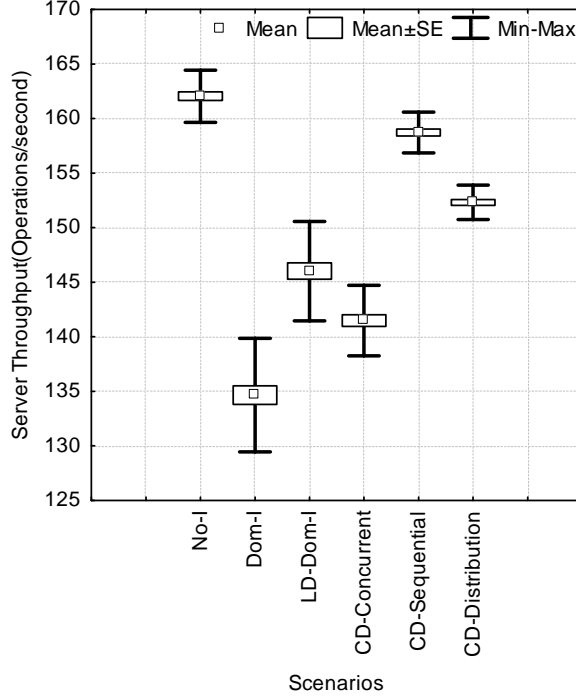


Figure 4: Comparison of throughput in operations/second.

LD-Dom-I in the non-concurrent scenarios, but it is not cost-effective in the concurrent scenario. In the concurrent scenario the cost of utilizing CD (interactions with server) overcomes the limited benefits to be gained when all deployed instances must initially include all the probes. In the non-concurrent scenarios, instances benefit by avoiding inserting probes already covered in other instances, which reduces CD communication costs since there will be less probes to trigger an interaction with the CD server (e.g., under the sequential scenario, almost half of the clients contacted the server just once).

4.4 Coverage gains from multiple deployed instances

Researchers working on leveraging field data to improve the testing activities [9, 18, 19, 22] have conjectured that monitoring a large number of deployed instances in the field will increase the chances of quickly discovering new aspects of the software behavior worth testing. This conjecture was explored under the concurrent scenario presented in Section 4, imitating a beta testing process where a set of instances is concurrently deployed for usage.

In Figure 5 a graph plotting the blocks covered over time was used to depict the rate at which deployed instances cover the code. The thick-line denotes the cumulative coverage over all deployed clients, while each dotted line represents the coverage from individual deployed instances (note that some of the 37 clients have similar behavior so their plots overlap). As expected, the aggregation of coverage information leads to a collection of higher coverage data, and at faster coverage rates. This is evident by the higher slope and reach of the thick-line representing the accumulated coverage. For example, the cumulative coverage curve reaches the coverage value of 860 blocks in 45 seconds, while the first instance to reach that level of coverage

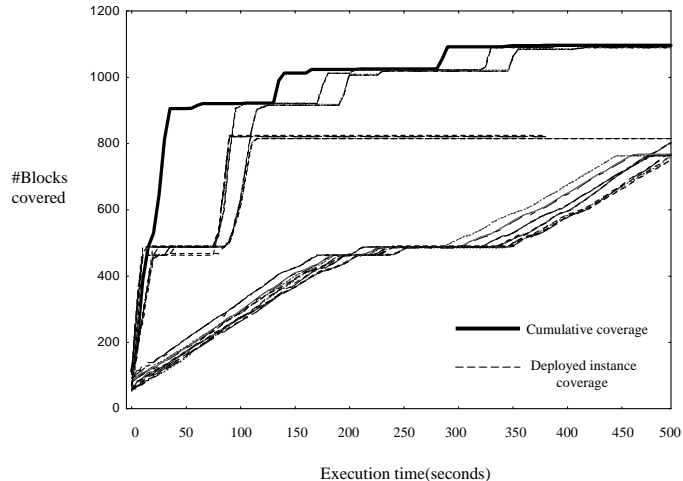


Figure 5: Coverage across multiple instances

took almost 90 seconds. Similarly, the cumulative coverage curve reaches the coverage value of 1024 blocks in 166 seconds, while the first instance to reach that level of coverage took almost 320 seconds. Note also how some instances were able to achieve almost the same overall coverage as the whole aggregated group, but in more time. Overall, this graph confirms the potential opportunity of obtaining coverage information from many deployed instances.

5 Threats to validity

The presented study, like any of this nature, has limitations which may impact the validity of the findings. Some of these limitations, their impact, and how they were address are now discussed.

First, the study is subject to threats of internal validity which could have affected the variables measured without the authors' knowledge. For example, some of the Java features may have been altered by the particular implementations. More specifically, when modifying a JVM for instrumentation manipulation, multithreading and bytecode verification must be carefully handled. To validate the implementations, the JVM modifications were carefully checked to determine that they did not impair these features, first on small programs and then on the benchmarks by checking their behavior with and without coverage probes. To avoid disturbing the verifier, the BCEL library was utilized to ensure that the stack size for each method is adjusted after the insertion of coverage probes [7]. Another threat to internal validity is the simulation setting. For example, some deployed instances may have chosen property files that configured a very large application load and state, while others configured a minimal application. This source of variation was controlled by deploying 37 instances so that the variation is likely well distributed across the study, and the simulation was performed twice.

Second, there are threats of external validity which limit the generalization of the presented techniques. For example, the promising results with LD obtained on the Specjvm98 programs may not generalize to programs that are not like the ones in this benchmark. Along the same lines, the utilization of Specjbb2000 to evaluate CD and the simulated environment for deploying it through multiple instances faces similar

problems. The simulation involved 37 instances deployed in a controlled environment, and these instances had a similar operational profile since only a limited number of program properties could be configured in the benchmark, and these were primarily workload related. In spite of these limitations, these benchmarks are recognized as de-facto standards for performance evaluation, so their generality is justified at least to some extent. Still, further evaluations with more and more diverse deployed instances are needed in future evaluations of CD.

The authors are also conscious that the techniques' implementation require JVM modification, which may not be easily transferable to practice at this time. Although the techniques may be implemented through alternative mechanisms that offer more flexibility, the intention was to reduce overhead, and the JVM offers the best chance to accomplish that goal. Furthermore, the adoption of the current approach was encouraged by the recent incorporation of some primitives into the APIs of some popular virtual machines that could enable future versions of disposable instrumentation [8].

6 LD in the presence of a JIT Engine

As noted previously in Section 2.3, a JIT execution engine differs from an interpreter engine in that it translates bytecode to native code only once. Retaining native code for subsequent executions usually leads to performance improvements. This section explores the concept of disposable instrumentation within the context of JIT.

Given the efficiency improvements introduced by JIT, it is tempting to conjecture that LD may improve the coverage collection efficiency but perhaps not in the same proportion as with an interpreter engine. However, such a conjecture does not account for two factors. First, if translation to native code is performed after instrumentation disposal, then the instrumentation is truly removed (instead of nullified like with the interpreter) from the native code making the resulting code more efficient. Second, the translation process may avoid the need to repeatedly identify the *Collector* through a targeted predicate (one of the sources of overhead associated with the implementation of LD in interpreter mode).

To quantify LD performance within JIT, the Kaffe Virtual Machine required further modification to perform LD at the method-level, the level of native code retention supported by most JVMs. The JVM using JIT maintains a dispatch table for each loaded class. Each entry in the dispatch table corresponds to a method in the class and includes a pointer to the method's address. When a method is invoked for the first time, its pointer aims to a trampoline structure (in the JIT context a trampoline is a temporary structure to direct execution towards a method's bytecode before translation). Invocation of this structure serves as a trigger for the bytecode translation, for the update of the dispatch table so that subsequent executions directly execute native code (skipping the trampoline), and for redirecting execution to the native code. Note that the three activities happen only once. Once the dispatch table is updated, subsequent calls to the method will be directed to its native code.

The JIT execution engine was adapted to include LD at the method level by performing three steps: executing the translated instrumented code, removing the instrumentation, and re-translating the code. The first time a method is called, the algorithm proceeds through a simplified version of the original trampoline

Algorithm 3 LD within JIT

```
1: Caller invokes Callee
2: Callee Trampoline is invoked
3: Translate Callee bytecodes into native code
4: if Callee invocation = first then
5:   Collector invocation bytecodes in Callee are replaced with nop
6: else
7:   if Callee invocation = second then
8:     Update Dispatch Table with native code address
9:     Update Translation flags
10:    Deallocate Trampoline for Callee {Next time Callee is invoked, execution will go to its native code}
11:   end if
12: end if
13: Jumps to Callee native code address for execution
14: Return to caller
```

Table 6: LD-JIT: Execution time for method level

Program	No-I sec.	Basic-I sec.	%Overhead w.r.t No-I	LD-Basic-I sec.	%Overhead w.r.t No-I	%Gain w.r.t to Basic-I
jess	32	49	53	33	3	50
raytrace	32	70	119	33	3	116
db	38	40	5	39	3	2
javac	34	48	41	37	9	32
jack	20	25	25	21	5	20
compress	24	58	141	27	12	129

that translates its bytecodes into native code and nullifies the call to *Collector*, but it does not update the dispatch table. When a method is called for the second time, it goes through the trampoline again, it translates the modified bytecode (without the invocation to *Collector*) to native code, it now updates the dispatch table, and proceeds to execute the native code. The next time the method is invoked, it will go directly to the native code without instrumentation. A more formal description is presented in Algorithm 3.

The LD prototype was utilized for the JIT execution engine on the Specjvm98. Table 6 presents the preliminary findings. The overhead associated with Basic-I over No-I in JIT mode ranges from 5% to 141%, whereas the overhead caused by LD-Basic-I ranges from 3% to 12%. The average LD-Basic-I overhead is nearly 6% as compared to 64% due to Basic-I. Interestingly, although the raw numbers and gains are smaller for LD-JIT than for LD, the relative efficiency gains obtained through LD while utilizing JIT is greater than when utilizing the JVM in interpreter mode.

7 Conclusions and Future Work

A new approach for reducing coverage collection overhead has been presented and quantified. This new approach differs from existing approaches in that it disposes of coverage probes at run-time, without stopping program execution, in a program's virtual execution environment. Two implementations of disposable instrumentation were constructed: LD, suitable for in-house coverage collection, and CD, suitable for reducing the overhead of gathering coverage from multiple instances that may be deployed.

The evaluations indicated that LD reduced block level coverage collection overhead by an average of 66% over a technique that instrumented all blocks. Even when the target program structures were analyzed to reduce the number of initially inserted probes, LD provided considerable gains averaging 56% across all programs and a best case of 91% reduction. The trends were consistent but less obvious at higher level coverage granularities.

In addition, the initial design and implementation of LD within JIT resulted in an order of magnitude reduction in overhead compared with an instrument-all strategy. Although this illustrates the potential of LD even in the presence of more efficient execution engines, it is currently constrained to method-level instrumentation. Given that the JVMs code retention operates exclusively at the method-level, considering smaller coverage granularities may introduce additional challenges. One possible approach could attempt to repeat the translation of bytecodes to native code (Re-JIT). Different Re-JIT policies could then be implemented to force re-translation at certain intervals. In the future, further exploration of such approaches will be necessary. Furthermore, it will be beneficial to carefully explore alternative implementations that are less dependent on the JVMs internals, which constitutes a limitation of the current prototypes.

The results on CD were also positive but highly dependent on the number of deployed instances, how those instances were exercised, and the deployment scenarios. For non-concurrent scenarios and just 37 deployed instances with fairly similar operational profiles, CD provided gains of up to 8% over LD-Dom-I and 33% over Dom-I. Further evaluation in a larger setting, with more deployed instances employed by real users, is still necessary to explore if the findings generalize. Several refinements to current implementations must also be made to investigate various update and distribution policies.

Last, the concept of disposable instrumentation could be interpreted in a broader context. Currently, instrumentation probes are discarded after just one execution because the target collection data is coverage. This interpretation could be extended by associating predicates with each probe to trigger instrumentation disposal. The goal is to allow, for example, the run-time disposal of assertions or trace statements when certain conditions have been met, enabling efficiency improvements in a larger family of program analysis and testing techniques.

Acknowledgments

This work was supported in part by a NSF-ITR program under award 0080898 and a Career Award 0347518 to University of Nebraska, Lincoln. The authors are thankful to Witawas Srisa-an for facilitating the evaluation benchmarks and for his insightful suggestions on the JVM implementation, and to the three anonymous reviewers of this paper for their suggestions and comments. The authors are also thankful to the RCF group for providing the Sandhills infrastructure as well as the Kaffe group for making the kaffe virtual machine open source.

References

- [1] The Java HotSpot Performance Engine Architecture. <http://java.sun.com/products/hotspot/>, 2005.

- [2] H. Agrawal. Efficient Coverage Testing Using Global Dominator Graphs. In *Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 11–20, 1999.
- [3] B. Pappin, H.M. Miller, N. Mehner and P. Zabelin. Hansel. <http://hansel.sourceforge.net>, 2002.
- [4] T. Ball and J. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, 1994.
- [5] T. Ball and J. Larus. Efficient path profiling. In *IEEE/ACM International Symposium on Microarchitecture*, pages 46–57, Dec. 1996.
- [6] Cenqua Organization. Clover. <http://www.cenqua.com/clover>, 2002.
- [7] M. Dahm and J. Van Zyl. Byte code engineering library. <http://jakarta.apache.org/bcel/>, June 2002.
- [8] M. Dmitriev. Profiling Java Applications Using Code Hotswapping and Dynamic Call Graph Revelation. In *Fourth International Workshop on Software and Performance*, pages 139–150, 2004.
- [9] S. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Transactions on Software Engineering*, 31(4):312–327, 2005.
- [10] EMMA. A coverage tool for java developers. <http://emma.sourceforge.net>, 2004.
- [11] P. Frankl and E. Weyuker. An applicable family of data flow testing criteria. *IEEE Transaction on Software Engineering*, 14(10):1483–1498, Oct. 1988.
- [12] J. Crisp, L. Leung and D. Holroyd. Java jvmdi coverage tool. <http://jvmdicover.sourceforge.net>, 2002.
- [13] Kaffe-Organization. Kaffe virtual machine. <http://kaffe.org>, 2005.
- [14] P. Lam, F. Qian, and O. Lhotak. Soot: a java optimization framework. <http://www.sable.mcgill.ca/soot/>, 2002.
- [15] B. Liblit, A. Aiken, Z. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *Conference on Programming Language Design and Implementation*, pages 141–154. ACM, June 2003.
- [16] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2 edition, 1999.
- [17] M. Albrecht. Grobo code coverage. <http://groboutils.sourceforge.net/codecoverage/index.html>, 2002.
- [18] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed Continuous Quality Assurance. In *International Conference on Software Engineering*, pages 459–468, 2004.
- [19] A. Orso, D. Liang, M. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *International Symposium on Software Testing and Analysis*, pages 65–69, 2002.

- [20] A. Orso, A. Rao, and M. Harrold. A technique for dynamic updating of Java software. In *IEEE International Conference on Software Maintenance*, page 649, Oct. 2002.
- [21] P. Morgan, M. Sparks and C. Lewis. Jcoverage. <http://jcoverage.com/>, 2003.
- [22] C. Pavlopoulou and M. Young. Residual Test Coverage Monitoring. In *International Conference of Software Engineering*, pages 277–284, May 1999.
- [23] Rational Software Corporation. Purecoverage. <http://www.rational.com/products/purecoverage/>, 2002.
- [24] E. Rogers. *Diffusion of Innovations*. Free Press, 5 edition, 2003.
- [25] D. Sosnoski. Java performance programming. http://www.javaworld.com/javaworld/jw-11-1999/jw-11-performance_p.html, 1998.
- [26] Standard Performance Evaluation Corporation. specjvm98 benchmarks. <http://www.spec.org/osg/jvm98>, 1998.
- [27] Standard Performance Evaluation Corporation. specjbb2000 benchmark. <http://www.spec.org/osg/jbb2000>, 2000.
- [28] M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *International Symposium on Software Testing and Analysis*, pages 86–96, July 2002.