

# Reducing Coverage Collection Overhead With Disposable Instrumentation

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. We present two techniques to mitigate this problem based on the concept of “disposable coverage instrumentation”: coverage instrumentation that is removed after its usage. The idea is to reduce coverage collection overhead by removing instrumentation probes after they have been executed. We have extended a java virtual machine to support these techniques, and show their potential through empirical studies with the Specjvm98 and Specjbb2000 benchmarks. The results indicate that the techniques can reduce coverage collection overhead between 18% and 97% over existing techniques.

## 1. Introduction

Coverage measures are important for engineers to quantify the thorough-fullness 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).

Independent of the coverage entity type, obtaining coverage data for a program  $P$  often follows a common procedure: 1) identify target entities  $E$  in  $P$ , 2) insert coverage probes  $H$  into  $P$  to capture the execution of  $E$  and build  $P'$ , 3) execute  $P'$  and obtain  $T$  trace data, and 4) process  $T$  trace data to obtain  $E$  coverage.

In spite of its deceptive simplicity, this process can be challenging when applied to a complex product and testing environment. First, as more entities are profiled, the additional coverage probes in  $P'$  can generate unacceptable execution overhead (reported to range between 10% to 390% [2, 12]). This overhead becomes even more noticeable as

more test cases are considered. Second, the build process to generate  $P'$  can be quite time consuming so changes in the instrumentation strategy to collect coverage cannot be refined to reduce such overhead.

One approach to address these problems is to remove the coverage probes after they have been executed. We call this *disposable coverage instrumentation*. Since coverage measures only require to know whether an entity is executed, coverage probes can be removed after the first time their corresponding entity is exercised without loss of information. Pavlopoulou introduced an instance of this approach called *residual testing*, where the probes in a program are removed after each test case is executed [17]. Residual testing on four small applications showed that instrumentation execution overhead becomes negligible after a few test iterations.

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

- It requires for the target program to be stopped, rebuilt, and re-executed<sup>1</sup>. 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-building processes that must be repeated.
- It cannot take advantage of multiple instances of the program that may run in parallel. Although this is important when testing is performed in parallel platforms, 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 [7, 15].

This paper introduces two disposable instrumentation techniques that address those challenges. The first technique, **local disposal (LD)**, disposes of each coverage probe  $H$  in  $P$  as soon as  $H$  is executed, without requiring for  $P$  to stop. The second technique, **collective disposal (CD)**,

<sup>1</sup>Some approaches that support run-time instrumentation removal such as Dynainst and JFluid are discussed in Section 2.

adds a cooperative flavor to LD by disposing of probes executed by any deployed instance of  $P$ . We have implemented both techniques for the java language by altering the Java Virtual Machine (Section 2.3 explains why we have targeted this environment). We have also conducted two sets of 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 includes the necessary background information and related work. Section 3 introduces the LD algorithm, implementation, and it assesses its performance under the Specjvm98 benchmark suite [21]. Section 4 introduces the CD architecture and implementation, and evaluates its performance under various scenarios within the Specjbb2000 server benchmark [22]. Section 5 provides details on additional challenges that help to scope the reach of the approach and Section 6 summarizes the findings and presents future research avenues.

## 2 Background and Related Work

We begin this section by introducing a sample of coverage tools for java programs. Section 2.2 presents the related efforts in reducing profiling overhead. Section 2.3 describes some of the Java Virtual Machine characteristics that are relevant to our work.

### 2.1 Coverage Tools

There is a myriad of commercial and freely available tools that help to collect coverage data. For example, Clover [4], PureCoverage [19], and Jcoverage [18] are some of the most popular commercial tools, while Emma [8], JVMDI [10], and GroboCodeCoverage [14] are some of the open-source coverage tools we 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 a neat interface for the tester to visualize the coverage information.

Another differentiating factor between these tools is how they obtain coverage information. The three mechanisms are: through the java debugging interface, instrumenting the java class files with the assistance of bytecode manipulation libraries, and modifying the bytecodes at loading time (when they are being setup in the java virtual machine). As we will see, the main distinction between the techniques employed by these tools and the technique we are proposing is that ours includes run-time probe re-

moval 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. For example, Agrawal introduced various techniques based on "domination relationships" to reduce the number probes in the target program without loss of coverage information [1], Ball et. al introduced a technique to reduce the probes necessary to profile acyclic intra-procedural paths [2], and Bowring et. al targeted probes within the same subtasks [9]. Although our approach uses some of these techniques to reduce the required initial number of instrumentation probes, its core capability of instrumentation removal is aimed at continuously reducing the number of probes during execution.

A second relevant approach to our work to reduce profiling overhead consists of removing instrumentation probes after 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 [17]. The prototype tool for residual test coverage [3] re-instrumented programs at the beginning of each execution. Our approach 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 for probe removal at run-time which LD resembles, but in a different target domain (technique uses trampolines for de-allocation of instrumentation in binaries executables written in C [23]). Another difference with our approach is that we can perform *distributed* instrumentation disposal, which takes into account the coverage obtained from multiple program instances.

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 through the JVM reloading mechanism [16]. The same approach could be used to dynamically manipulate instrumentation at the class level. However, the addition of wrapper classes and the repeated class loading would not be effective in reducing the coverage collection overhead, which is our main goal. Similarly, the Jfluid toolkit takes a step further by providing method level reloading [6] through a specialized API introduced into the JVM. However, this still implies reloading (at a smaller granularity) and it has restrictions on what methods can be reloaded.

### 2.3 Java Virtual Machine

The Java Virtual Machine (JVM) is a platform to run java programs. Java programs consists of classes implemented in the java programming language. These programs are compiled to produce class files containing instructions that are understandable by a JVM. A JVM implementation can perform the actions specified in the class files in various ways as long as it respects the JVM specifications [13]. JVM implementations vary in memory organization, interpretation or compilation methods, and communication to each platform. To demonstrate the feasibility of our techniques, we have modified the interpreter engine of the open source implementation of JVM by the Kaffe group (version 1.1.3) [11], which offers build configurations to run it as an interpreter or as a just-in-time compiler.

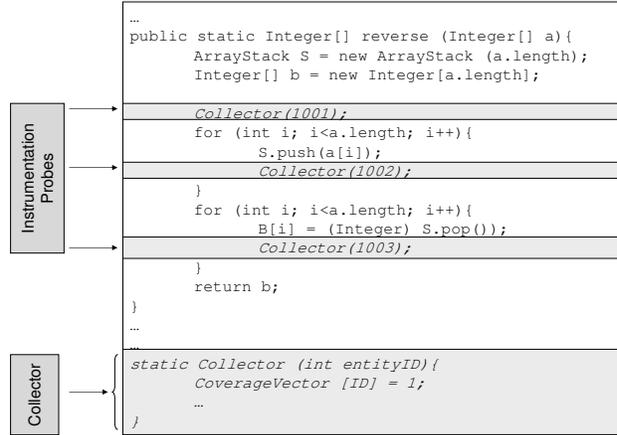
The instructions in a class file are called bytecodes, which resemble traditional machine code. The JVM instruction set has 256 instructions and the JVM’s job is to perform the functionality expected out of each bytecode it encounters during execution. The simplicity of this instruction set, their potential for direct mapping to source code, and the availability of tools for bytecode manipulation, makes the bytecode level an ideal candidate to insert coverage probes. In our studies, we have utilized the Byte Code Engineering Library (BCEL) to analyze and manipulate the class files containing bytecodes [5]. The BCEL offers the facility to place a probe at a particular chosen point in the program, while transparently handling the necessary adjustments for redirections and stack size adjustments.

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) and our relationship to companies utilizing such products also heavily influenced the direction of our implementation which exclusively targets the java programming language.

### 3 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 specially appealing for programs with:

- non-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 effect of multiple executions caused by coverage probes are compounded



**Figure 1. Instrumented Code to Collect Block Coverage.**

- expensive build process, where the mechanism to remove the coverage probes cannot afford the extra compilation-linking cycles

### 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<sup>2</sup>. At the end of program execution the coverage vector is dumped to a file. Figure 1 exemplifies this process through a short 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. We conjecture 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 removal can be performed in two steps. First, we must detect the occurrence of the invocation to the Collector method (execution of the coverage probe). Second, we must remove the invocation after the Collector has finished its execution.

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 wrap-

<sup>2</sup>Since overhead is our concern, we have avoided the expensive object creation operation and kept the Collector as a static method [20]. The Collector could also be implemented as a class if additional functionality is required.

per classes that redirect the execution (e.g., hot-swapping [16]). At the JVM level, the removal can be implemented by run-time bytecode modification. There are several tradeoffs between these approaches. The instrumentation removal at the application level offers a lot of flexibility but it can impose a 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 it requires an intimate knowledge of the particular implementation and is JVM specific. Ideally, one would implement LD by accessing an API that allows bytecodes modification at run-time avoiding the problem. However, current JVMs do not provide such capability.

Given that overhead reduction is our main objective, we decided to modify the Kaffe JVM implementation to enable LD<sup>3</sup>. More specifically, we modify the execution engine within the JVM. Our process analyzes a method’s signature at the end of each execution to determine whether it 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 we nullify the bytecode corresponding to the invoke operation (invokestatic bytecode followed by its 16-bit argument) on the caller by replacing it with the bytecode 0 corresponding to the nop operation. This nullification is equivalent to the removal of the coverage probe, but without the bytecodes rearrangement required by a true removal. As a result, the coverage probe is “removed” after its first execution.

The corresponding algorithm is presented in Algorithm 1. Note that only two new operations (6 and 7) were added to the execution engine of the Kaffe JVM. These operations are conceptually simple and only require the addition of 7 LOC 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 operation 6 in Algorithm 1 implies an additional comparison within the JVM for each static method invocation. Although this implies a slightly slower JVM, we conjecture that, as the required granularity of coverage information decreases, the gains of instrumentation removal from the application will be more noticeable in spite of the additional comparison cost within the JVM.

Although we have provided arguments for the potential of the LD technique to reduce the overhead associated with obtaining coverage data, empirical evidence is needed

<sup>3</sup>Although these modifications are tied to that implementation, our initial exploration indicates that we should be able to migrate them to IBM’s JVM (jikes) and Sun’s JVM with minor effort. In any event, we expect that some form of restricted bytecode modification would be offered as part of a future Java API (as in JFluid [6]).

---

#### Algorithm 1 LD within JVM

---

- 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
- 

to quantify that potential. We provide such empirical evidence in the following section.

### 3.2 Experimental Setting and Design

**Variables.** The independent variables are the coverage instrumentation granularity and the instrumentation technique. We investigated two coverage granularities levels: block and method, and consider three instrumentation techniques:

1. No-Instrumentation which defines a lower bound for overhead. We refer to this technique as No-I.
2. Static instrumentation of all entities with refinement based on the domination relationship between the entities. Similar to Tikir et. al [23], we use dominator tree relationships to curtail the number of instrumentation points in the target program. We refer to this technique as Basic-I.
3. Disposable instrumentation capabilities on top of Basic-I. We refer to this technique as LD.

Our dependent variable is the overhead associated by each technique at certain granularity for a specific program. To quantify the overhead we measure execution time through the benchmark’s default timer (in seconds).

**Object.** We used the Specjvm98 [21] benchmark suite to evaluate the techniques with its default configuration settings. Table 3 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.

**Design.** Our experimental design combined each technique (No-I, Basic-I, LD) with each granularity level (block and method) and with each program in the benchmark (jess, raytrace, db, javac, jack). We ran No-I, Basic-I, and LD on each one of the programs from the benchmark under both granularity levels to measure their associated overhead. To reduce the variation due to uncontrolled

**Table 1. LD: Execution time for block level**

Program	No-I sec.	Basic-I sec.	%Overhead w.r.t No-I	LD sec.	%Overhead w.r.t No-I	%Gain w.r.t to Basic-I	#Blocks Instrumented
jess	364	772	112	418	15	97	2608
raytrace	464	995	114	564	22	92	596
db	602	762	27	656	9	18	246
javac	384	630	64	428	11	53	6066
jack	357	466	31	382	7	24	2153

**Table 2. LD: Execution time for method level**

Program	No-I sec.	Basic-I sec.	%Overhead w.r.t No-I	LD sec.	%Overhead w.r.t No-I	%Gain w.r.t to Basic-I	#Methods Instrumented
jess	364	501	38	395	9	29	673
raytrace	464	871	88	552	19	69	173
db	602	606	1	640	6	-5	34
javac	384	470	22	411	7	15	1179
jack	357	382	7	376	5	2	302

**Table 3. Specjvm98 benchmark programs**

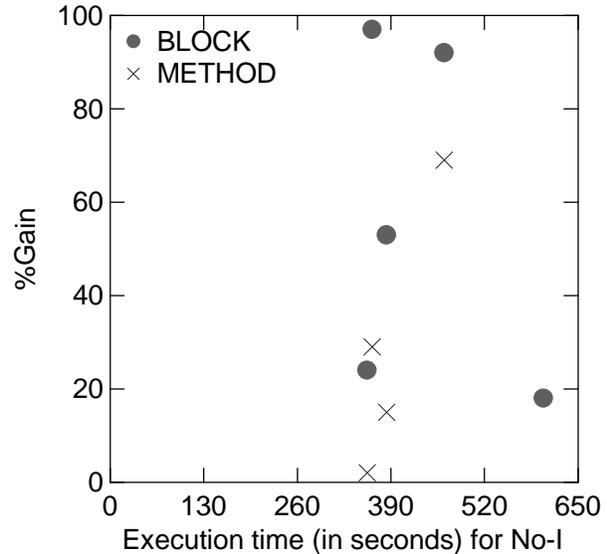
Name	Description
jess	Expert system to solve puzzles
raytrace	Ray tracer rendering
db	Database builder and transaction processor
javac	Java compiler with sample programs
jack	Java parser generator

sources, we performed this experiment 5 times and computed the means and deviations across the runs. We conducted our experiments on a 2.4GHz Pentium processor, with 512 megabytes of memory running Linux 9.0.

### 3.3 Results and Analysis

Table 1 presents our results for block level instrumentation. We see that the overhead caused by the Basic-I technique ranges from 31% to 112%, while LD causes overhead ranging from 7% to 22%. The average LD overhead is 13% (as compared to 70% for Basic-I). When comparing Basic-I with LD, we observe that LD provides up to 97% reduction of execution overhead (jess program). On average, the gains from applying LD instead of Basic-I were impressive: 57%.

We conjectured that as we increase the granularity of the coverage data, the benefits of LD diminish. This conjecture was confirmed by the results at the method coverage level. Table 2 shows that for 4 out of 5 programs LD performs better than Basic-I. However, the average gains for all programs is just 22%. More interesting is the variation of results across programs. For db, utilizing LD cost 5%

**Figure 2. Gain vs. test process duration.**

more than just using Basic-I, mainly because the overhead of Basic-I is less than one percent which leaves limited chances of improvement for LD. Still for some programs like raytrace, the gains achieved through LD are considerable (69%).

To get a better understanding of the circumstances that may have affected LD gains over Basic-I, we explored three potential factors. First, we looked at the length of the testing process. Figure 2 presents a scatter plot of the gains versus total testing time based on the No-I techniques. For the programs in the benchmark, the total test time did not

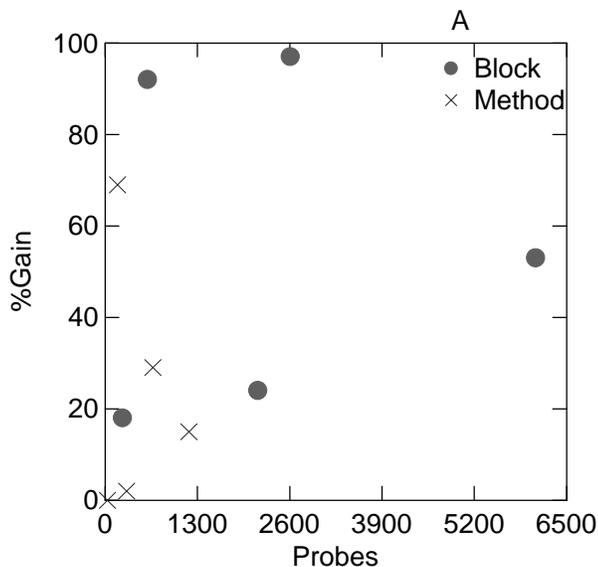


Figure 3. Gain vs. number of probes.

appear to be always associated with the potential gains of LD. The extreme case is db, which has the longest test suite execution time but also provides the lowest gains.

Second, we explored the relationship between the number of probes and the potential gains. Figure 3 presents another scatter plot but with the number of probes in the x-axis. Each observation represents one program and one level of granularity. In this figure we again see a lot of variation that is not explainable by just the number of probes. When applied to programs with less than 200 probes (e.g., raytrace with method level instrumentation), LD can provide gains of almost 70%. On the other hand, the three other instances with more than 200 probes did not provide more than 30% gains and in one case LD actually cause a detriment in performance. However, there seem to a positive trend for the LD gains as the number of probes increases.

Last, we explored the potential relationship between coverage rate and gains through Figure 4. We were interested in determining whether the similarity between test cases coverage patterns affected the gains in LD. Raytrace and jess, the programs for which LD generated more gains, present a steep coverage rate at the beginning and then level off, indicating repeated execution patterns which are beneficial to LD. Javac presents a similar scenario, although the ascend is not as fast. Jack has a shorter flat period corresponding to the repeated coverage patterns, limiting the gains from removing probes. Last, db achieves the flat state fast, but its test suite only covers 163 blocks which provides limited opportunities for LD to be effective.

Overall, our findings confirm the conjecture that objects with a large number of probes covered at a fast rate, and test

cases that generate repetitive execution patterns, are likely to benefit the most from LD.

## 4 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 deployed instances of the software, avoiding the collection of the same coverage data in repeated instances. This technique would be specially valuable when:

- 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)
- software has so many potential configurations (configuration explosion problem [24]) that the coverage adequacy goals are not achievable during the in-house testing process

### 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 equivalent 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. Four steps are noticeable different from the Algorithm 1. First, in the second statement, we preventively check 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, we operate similarly to the LD algorithm, executing the Collector code and overwriting the invocation instruction. The second difference is the 9th statement, where we proceed to disseminate the knowledge of the covered statement to the other instances.

The third difference rises when the Collector invocation has already been executed by other instances; in this case we proceed to overwrite the invocations without even building the corresponding stack frame (statement 12th). Note that this situation implies less overhead than the LD solution since the Collector method is not executed. The last distinct step is to update the instance coverage vector with the knowledge from the other instances (statement 14th).

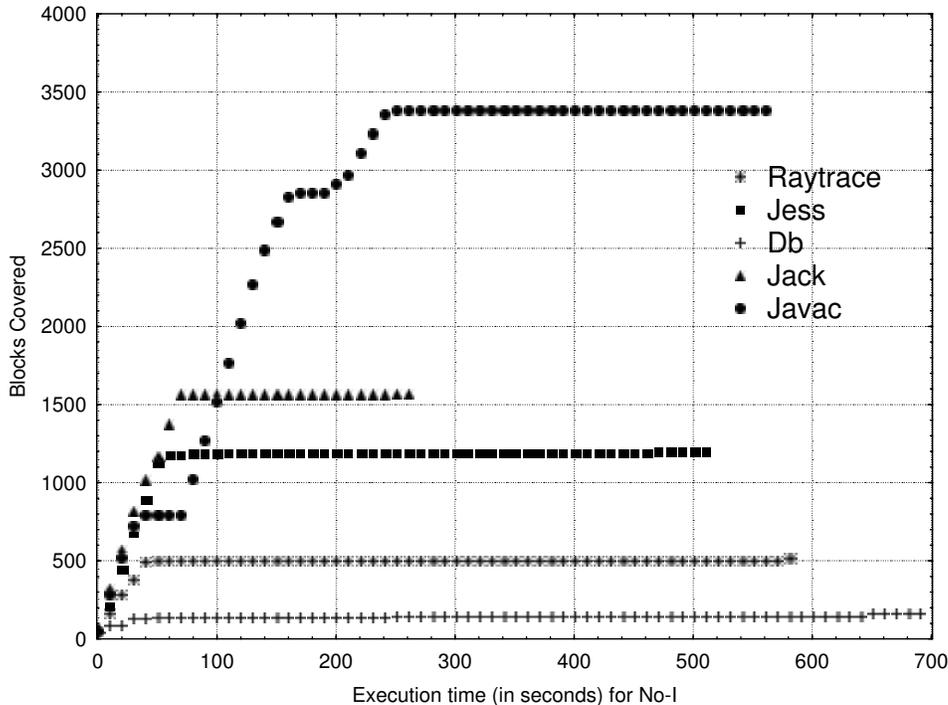


Figure 4. Coverage rate.

---

**Algorithm 2** CD within and across JVMs.

---

```

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

```

---

There are several potential 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, we implemented the communication channel with a centralized server. 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 regular intervals. However, determining the right interval's sizes 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 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. The policy in our studies is the simplest of the latter type, with a buffer of size one.

We decided to include two types of update triggers. First, a deployment instance gets an updated vector at the beginning of the execution when the application’s classes are being loaded. This slow loading process hides to some degree the coverage vector retrieval from the server. This update will provide an updated vector to start an instance execution and leverage the knowledge collected up to that point. Second, each running instance will get an update every time it performs a disseminate operation to leverage the connection already established with the server.

As we mentioned, for our experiments we decided to disseminate the information (connect to the server) every occasion 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 our implementation of CD, we modified the Kaffe JVM 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). The server, which is implemented in C to match the Kaffe JVM implementation, creates multiple threads to handle the connection with the instances.

## 4.2 Experimental Setting and Design

**Variables.** We have identified two independent variables. First, the four instrumentation techniques at the block level: 1) No-I, 2) Basic-I, 3) LD, and 4) CD.

The second independent variable is the deployment scenario. The effectiveness of CD may be affected by whether the instances are deployed concurrently, sequentially, or something in between. First, we consider a scenario in which all clients start executing concurrently. This is the worse case scenario for CD since there are no “late starters” that can leverage the knowledge from previously deployed instances. Second, we consider a scenario with pure sequential deployment where an instance would start executing only after another one ended, presenting a somewhat ideal scenario for CD. For the third scenario we estimated a deployment gap between instances. To estimate this gap, we searched for a server program similar to our object of study in an open source repository (sourceforge-jboss) and learned that it was downloaded every 30 seconds approximately (period of one week from 02/13/2004 to 02/20/2004). We then simulated a deployment scenario with a normal distribution that has a mean of 30 seconds.

The dependent variables we used for analyzing the improvement in coverage data collection are: 1) the average execution time across all deployed instances, and 2) the average server throughput which is the de-facto measure to evaluate the performance of this type of application.

**Object.** We used the Specjbb2000 [22] server benchmark to evaluate the CD technique. This benchmark provides us with a java server application that has a 3-tier structure where we can simulate multiple deployments. The first tier of the architecture represents warehouses (clients) that generate business transactions such as placing orders or checking orders’ status. The second tier is the business logic at the server that processes the transactions and generates requests to the third tier made of a database. The size of Specjbb2000 is nearly 26 KLOC. We instrumented the Specjbb2000 server at the block level using the dominator tree algorithm (Basic-I).

**Design.** To evaluate CD, we required a set of deployed instances. We simulated a deployment environment with up to 30 clients in Sandhills, a cluster of 15 computing nodes powered by dual AMD 1.2 GHz athlon processors. Each simulated instance was made to exhibit a somewhat distinct behavior by randomly selecting a property file during its initiation. This property file adjusted each simulated deployed instance by varying the changeable parameters(eg., number of warehouses, sequence of warehouses, increments of warehouses, turning on/off garbage collection). The number of warehouses that generated requests to the Specjbb2000 server varied from 8 to 16 with heap size of 512 megabytes.

We executed the server benchmark with 30 potential instances on each possible combination of technique and deployment scenario. This process was repeated five times to avoid random noise due to network load or nodes assignment. The outcome of the process was the average execution time across deployed instances and their average throughput.

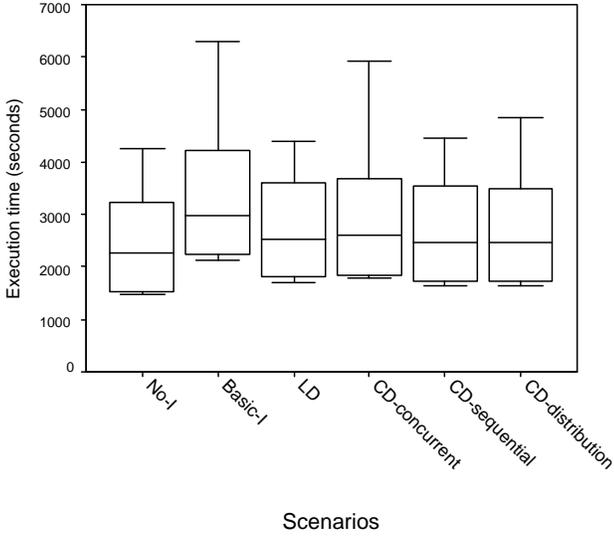
## 4.3 Results and Analysis

Table 4 summarizes the findings for each of the techniques on each one of the scenarios averaged over five runs. Each cell contains the execution time in seconds for each combination of technique and scenario, or the percentage of overhead compared with No-I. For the Specjbb2000 benchmark in our experimental setting, Basic-I has a 36% collection overhead over No-I.

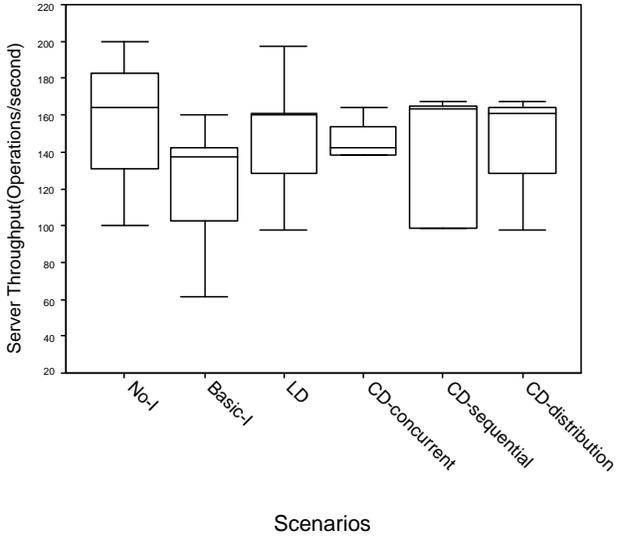
When deployment instances started concurrently, the average execution time per instance for CD was 22% higher than when no block coverage collection was performed, and overhead reduction of 14% took place over

**Table 4. CD: Execution times (in seconds).**

Deployment Scenario	No-I sec.	Basic-I sec.	%Overhead w.r.t No-I	CD sec.	%Overhead w.r.t No-I	LD	%Overhead w.r.t No-I
Concurrent	2518	3430	36	3081	22	2866	14
Sequential	2518	3430	36	2742	9	2866	14
Distributed	2518	3430	36	2744	9	2866	14



**Figure 5.** Comparison of execution time in seconds.



**Figure 6.** Comparison of throughput in operations/second.

Basic-I. It is also interesting to note that the overhead reduction seems to be driven by LD (14% reduction), not by CD. Employing CD in a purely concurrent scenario signifies a penalty of 8% over just using LD. This may have been caused by server collection bottlenecks (response time between the dissemination and update operations ranged from 0.3 to 0.5 seconds) or just the limited opportunities to leverage the knowledge across instances when all of them execute at the same time (deployed instances executed on average 77% of their probes).

CD on sequential and distributed scenarios present positive results, with overhead reductions of 27% over Basic-I and even 5% over LD. Under these scenarios, deployed instances can better leverage the coverage pool knowledge by removing probes before they are even invoked. In the sequential scenario, there were fewer update requests to the server which help to improve its response time (ranged between 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).

To get a better understanding of the distribution of the execution time per technique and scenario, we generated a box plot graph, Figure 5, where each box averages five observations corresponding to the execution times of five runs. 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 observed values. Figure 5 corroborates that there is some variation in the execution time for all the techniques, which may have been caused by the simulated node environment and the different performance between nodes. More important, the figure shows that Basic-I, LD, and CD under the concurrent scenario can, in the worst case, duplicate the execution overhead of the average No-I case. CD execution overhead variation was, however, comparable to No-I under the sequential and distributed scenario.

The average server throughput across the three scenarios for each one of the techniques is shown in Figure 6. Similar to execution time, the LD, CD-sequential and CD-distribution techniques were beneficial, increasing the

server throughput as compared to Basic-I. It is interesting to note again how CD outperforms LD in the non-concurrent scenarios but it is not cost-effective in the concurrent scenario.

## 5 Discussion

This section begins by providing the context for our findings in the way of threats to validity and explains how we addressed those threats. Sections 5.2 and 5.3 identify further limitations and potential of the techniques we developed as we try to extend their application.

### 5.1 Threats to validity

Our study, like any of this nature, has limitations which may impact the validity of our findings. We now proceed to identify some of these limitations, their impact, and how we attempted to address them.

First, we have threats of internal validity which could have affected the variables we measured without our knowledge. For example, some of the java features may have been altered by our approach. More specifically, when modifying a JVM for instrumentation manipulation, multi-threading and bytecode verification must be carefully handled. To validate our procedures we checked that our modifications 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, we utilized the BCEL library to ensure that the stack size for each method is adjusted after we insert coverage probes [5].

Second, we have threats of external validity which limit the generalization of our approach. For example, we utilized Specjvm98 to evaluate LD. The promising results of LD may not generalize to programs that are not like the ones in this benchmark. Along the same lines, we utilized Specjbb2000 to evaluate CD and a simulated environment for deploying it through multiple instances. Our simulation only involved up to 30 instances and they had a similar operation profile which may have constrained the potential benefits from CD (conservative scenarios). In spite of these limitations, these benchmarks are recognized as de-facto standards for performance evaluation, which seems quite appropriate to evaluate coverage collection overhead. Still, the next evaluation step will require the application of these techniques to other programs and scenarios.

We are also conscious that our 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, our intention was to re-

duce overhead and the JVM offers the best chance to accomplish that goal. Furthermore, we have seen repeatedly how successful techniques or mechanisms (e.g., reflection) have become a part of JVMs as they evolve or are incorporated into their API. We envision for our disposable approach to undergo a similar transition.

### 5.2 On the limits of LD to assist coverage collection

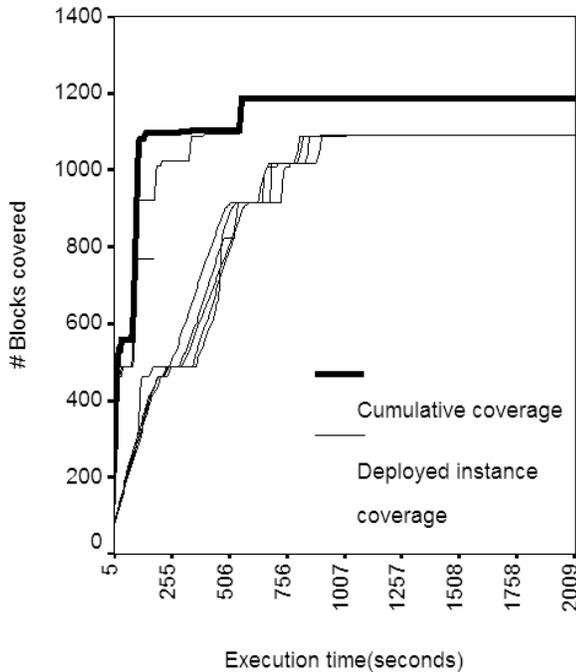
Our results have provided evidence about the potential of LD to improve the efficiency of the coverage collection process at the function/method and block/statement level. We believe that such benefits would extend to the collection of other types of coverage data as long as: 1) they include an instrumentation 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 remove a probe are cost-effective. We now elaborate further on this second aspect.

When collecting method and statement coverage data, the decision process to remove a probe is trivial. In such cases, the probe invoking the Collector method 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 not 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. We leave the development and evaluation of such analysis processes for future work.

### 5.3 Coverage gains from multiple instances

Researchers working on leveraging field data to improve the testing activities [7, 15, 17, 24] have conjectured that as the number of deployed instances increases, the chances of having users performing distinct activities in the field is likely to increase, which often results in new entities being covered. We explore this conjecture under the concurrent scenario presented in Section 4, imitating a beta testing stage where a set of instances is concurrently deployed for usage.



**Figure 7.** Cumulative coverage graph

The thick line in the Figure 7 denotes the cumulative coverage over all deployed instances, while each thin line represents the coverage from each deployed instance. As expected, we can observe how the integration of data captured in each deployed site can be aggregated to obtain a higher overall coverage. Note also how this aggregation leads to higher coverage rates which could be important when coverage criteria are involved in decision making. Some instances were able to achieve almost the same overall coverage as the whole group, but over longer periods of time. For example, the cumulative coverage curve reaches the coverage value of 1096 blocks in 145 seconds, while the first instance to reach that level of coverage took 400 seconds.

## 6 Conclusions and Future Work

We have presented and quantified a new approach for reducing coverage collection overhead. This new approach differs from existing approaches in that it disposes of coverage probes at run-time, without stopping program execution. Furthermore, the results indicate that this approach can reduce block coverage collection overhead up to 97%.

We constructed two implementations of disposable instrumentation: LD, suitable for in-house coverage collection, and CD, suitable for reducing the overhead of gathering coverage from multiple deployed instances. The evaluations indicated that LD reduced block level coverage col-

lection overhead by an average of 56% over the state of art technique. 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. We found that, for non-concurrent scenarios and just 30 deployed instances with fairly similar operational profiles, CD provided gains of about 5% over LD and 27% over Basic-I.

These results suggest several directions for future work. First, we plan to continue our evaluation in a larger setting, with more deployed instances employed by real users to explore if our findings still hold for this type of setting, specially for CD.

Second, the applicability of the implementation will certainly be affected by the efficiency of the probe removal process and the communication overhead in the case of CD. We expect to make several refinements to these implementations. For example, we will experiment with various update and distribution policies for CD and we plan to incorporate disposable instrumentation within the Just-In-Time(JIT) version of the execution engine which translates the bytecodes to native code and retains the native code to gain speed during subsequent executions.

Last, the concept of disposable instrumentation could be interpreted in a broader context. Currently, we dispose of instrumentation after just one execution because we are aiming to collect coverage. We plan to extend this interpretation by associating predicates with each probe to trigger instrumentation disposals. The goal is to allow, for example, the run-time disposal of assertions or trace statements when certain conditions have been met.

## 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. We thank Witawas Srisa-an for facilitating the evaluation benchmarks and for his insightful suggestions on the JVM implementation. We also thank the three anonymous reviewers of this paper for their suggestions and comments.

## References

- [1] H. Agrawal. Dominators, super blocks and program coverage. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 25–34, 1994.
- [2] T. Ball and J. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, 1994.
- [3] Brill Pappin, Harald M. Miller, Niklas Mehner, Paul Zabelin. Hansel. <http://hansel.sourceforge.net>, 2002.

- [4] Cenqua Organization. Clover. <http://www.cenqua.com/clover>, 2002.
- [5] M. Dahm. Byte Code Engineeing Library. In *Proceedings of JIT*, pages 267–277, 1999.
- [6] 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.
- [7] S. Elbaum and M. Hardojo. An empirical study of profiling strategies for released software and their impact on testing activities. In *International Symposium of Software Testing and Analysis*, pages 65–75, July 2004.
- [8] EMMA. A coverage tool for java developers. <http://emma.sourceforge.net>, 2004.
- [9] J.Bowring, A. Orso, and M. Harrold. Monitoring deployed software using software tomography. In *Workshop on Program analysis for software tools and engineering*, pages 2–9, 2002.
- [10] Joel Crisp, Lawrence Leung, David Holroyd. Java jvmdi coverage tool. <http://jvmdicover.sourceforge.net>, 2002.
- [11] Kaffe-Organization. Kaffe virtual machine. <http://kaffe.org>.
- [12] 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.
- [13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2 edition, April 1999.
- [14] Matt Albrecht. Grobo code coverage. <http://groboutils.sourceforge.net/codecoverage/index.html>, 2002.
- [15] 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.
- [16] A. Orso, A. Rao, and M. Harrold. A technique for dynamic updating of Java software. In *Proceedings of the IEEE International Conference on Software Maintenance*, page 649, Oct. 2002.
- [17] C. Pavlopoulou and M. Young. Residual Test Coverage Monitoring. In *International Conference of Software Engineering*, pages 277–284, May 1999.
- [18] Peter Morgan, Malcolm Sparks, Colin Lewis. jcoverage/gpl. <http://jcoverage.com/>, 2003.
- [19] Rational Software Corporation. Purecoverage. <http://www.rational.com/products/purecoverage/>, 2002.
- [20] D. Sosnoski. Java performance programming. [http://www.javaworld.com/javaworld/jw-11-1999/jw-11-performance\\_p.html](http://www.javaworld.com/javaworld/jw-11-1999/jw-11-performance_p.html), 1998.
- [21] Standard Performance Evaluation Corporation. specjvm98 benchmarks. <http://www.spec.org/osg/jvm98>, 1998.
- [22] Standard Performance Evaluation Corporation. specjbb2000 benchmark. <http://www.spec.org/osg/jbb2000>, 2000.
- [23] M. Tikir and Hollingsworth.J.K. Efficient instrumentation for code coverage testing. In *International Symposium on Software Testing and Analysis*, pages 86–96, July 2002.
- [24] C. Yelmaz, A. Porter, and A. Schimdt. Distributed continuous quality assurance:The Skoll Project. In *Workshop on Remote Analysis and Monitoring Software Systems*, pages 16–19, 2003.