Unit Test Virtualization with VMVM

Jonathan Bell
Columbia University
500 West 120th St, MC 0401
New York, NY USA
jbell@cs.columbia.edu

Gail Kaiser
Columbia University
500 West 120th St, MC 0401
New York, NY USA
kaiser@cs.columbia.edu

ABSTRACT

Testing large software packages can become very time intensive. To address this problem, researchers have investigated techniques such as Test Suite Minimization. Test Suite Minimization reduces the number of tests in a suite by removing tests that appear redundant, at the risk of a reduction in fault-finding ability since it can be difficult to identify which tests are truly redundant. We take a completely different approach to solving the same problem of long running test suites by instead reducing the time needed to execute each test, an approach that we call Unit Test Virtualization. With Unit Test Virtualization, we reduce the overhead of isolating each unit test with a lightweight virtualization container. We describe the empirical analysis that grounds our approach and provide an implementation of Unit Test Virtualization targeting Java applications. We evaluated our implementation, VMVM, using 20 real-world Java applications and found that it reduces test suite execution time by up to 97% (on average, 62%) when compared to traditional unit test execution. We also compared VMVM to a well known Test Suite Minimization technique, finding the reduction provided by VMVM to be four times greater, while still executing every test with no loss of fault-finding ability.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools*

General Terms

Reliability, Performance

Keywords

Testing, test suite minimization, unit test virtualization

1. INTRODUCTION

As developers fix bugs, they often create regression tests to ensure that should those bugs recur, they will be detected by the test suite. These tests are added to existing unit test suites and in an ideal continuous integration environment, executed regularly (e.g., upon code check-ins, or nightly). Because developers are often creating new tests, as software grows in size and complexity, its test suite frequently grows similarly. Software can reach a point where its test suite has gotten so large that it takes too long to regularly execute — previous work has reported test suites in industry taking several weeks to execute fully [37].

To cope with long running test suites, testers might turn to Test Suite Minimization or Test Suite Prioritization [44]. Test Suite Minimization techniques such as [15, 16, 23, 24, 28, 29, 39, 42] seek to reduce the total number of tests to execute by approximating redundant tests. However, identifying which tests are truly redundant is hard, and Test Suite Minimization approaches typically rely on coverage measures to identify redundancy, which may not be completely accurate, leading to a potential loss in fault-finding ability. Furthermore, Test Suite Minimization is an NP-complete problem [24], and therefore existing algorithms rely on heuristics. Test Suite Prioritization techniques such as [19, 20, 37, 38, 41] re-order test cases, for example so that given the set of changes to the application since the last test execution, the most relevant tests are executed first. This technique is useful for prioritizing test cases to identify faults earlier in the testing cycle, but does not actually reduce the total time necessary to execute the entire suite.

Rather than focus our approach on reducing the number of tests executed in a suite, we have set our goal broadly on minimizing the total amount of time necessary to execute the test suite as a whole, while still executing all tests and without risking loss in fault-finding ability. We conducted a study on approximately 1,200 large and open source Java applications to identify bottlenecks in the unit testing process. We found that for most large applications each test executes in its own process, rather than executing multiple tests in the same process. We discovered that this is done to isolate the state-based side effects of each test from skewing the results for future tests. The upper half of Figure 1 shows an example of a typical test suite execution loop: before each test is executed, the application is initialized and after each test, the application terminates. In our study we found that these initialization steps add an overhead to testing time of up to 4,153% of the total testing time (on average, 618%).

At first, it may seem that the time spent running tests could be trivially reduced by removing the initialization step from the loop, performing initialization only at the beginning of the test suite. In this way, that initialized application

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/05...$15.00
http://dx.doi.org/10.1145/2568225.2568248

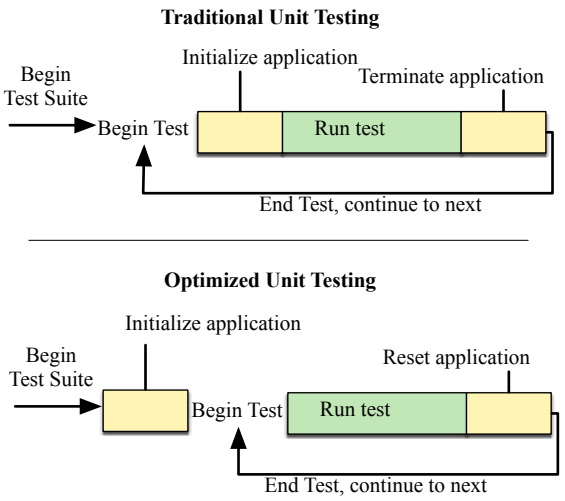

Figure 1: The test execution loop: In traditional unit testing, the application under test is restarted for each test. In optimized unit testing, the application is started only once, then each test runs within the same process, which risks in-memory side effects from each test case.

could be reused for all tests (illustrated in the bottom half of Figure 1), cutting out this high overhead. In some cases this is exactly what testers do, writing pre-test methods to bring the system under test into the correct state and post-test methods to return the system to the starting state.

In practice, these setup and teardown methods can be difficult to implement correctly: developers may make explicit assumptions about how their code will run, such as permissible in-memory side-effects. As we found in our study of 1,200 real-world Java applications (described further in §2), developers often sacrifice performance for correctness by isolating each test in its own process, rather than risk that these side-effects result in false positives or false negatives.

Our key insight is that in the case of memory-managed languages (such as Java), it is not actually necessary to reinitialize the entire application being tested between each test in order to maintain this isolation. Instead, it is feasible to analyze the software to find all potential side-effect causing code and automatically reinitialize only the parts necessary, when needed, in a "just-in-time" manner.

In this paper we introduce *Unit Test Virtualization*, a technique whereby the side-effects of each unit test are efficiently isolated from other tests, eliminating the need to restart the system under test with every new test. With a hybrid static-dynamic analysis, Unit Test Virtualization automatically identifies the code segments that may create side-effects and isolates them in a container similar to a lightweight virtual machine. Each unit test (in a suite) executes in its own container that isolates all in-memory side-effects to contain them to affect only that suite, exactly mimicking the isolation effect of executing each test in its own process, but with much lower overhead. This approach is relevant to any situation where a suite of tests is executed and must be isolated such as regression testing, continuous integration, or test-driven development.

We implemented Unit Test Virtualization for Java, creating our tool VmVm (pronounced "vroom-vroom"), which transforms application byte code directly without requiring modification to the JVM or access to application source code. We have integrated it directly with popular Java testing and build automation tools JUnit [2], ant [5] and maven [6], and it is available for download via GitHub [7].

We evaluated VmVm to determine the performance benefits that it can provide and show that it does not affect fault finding ability. In our study of 1,200 applications, we found that the test suites for most large applications isolate each unit test into its own process, and that in a sample of these applications VmVm provides up to a 97% performance gain when executing tests. We compared VmVm with a well known Test Suite Minimization process and found that the performance benefits of VmVm exceed those of the minimization technique without sacrificing fault-finding ability.

The primary contributions of this paper are:

1. A categorical study of the test suites of 1,200 open source projects showing that developers isolate tests
2. A presentation of Unit Test Virtualization, a technique to efficiently isolate test cases that is language agnostic among memory managed languages
3. An implementation of our technique for Java, VmVm (released freely via GitHub [7]), evaluated to show its efficacy in reducing test suite runtime and maintaining fault-finding properties

2. MOTIVATION

This work would be unnecessary if we could safely execute all of an application's tests in the same process. Were that the case, then the performance overhead of isolating test cases to individual processes could be trivially removed by running each test in the same process. We have discovered, however, that developers rely on process separation to ensure that their tests are isolated and execute correctly.

In this section, we answer the following three motivation questions to underscore the need for this work.

MQ1: Do developers isolate their unit tests?
MQ2: Why do developers isolate their unit tests?
MQ3: What is the overhead of the isolation technique that developers use?

2.1 MQ1: Do Developers Isolate Their Tests?

To answer **MQ1** we analyzed the 1,200 largest open source Java projects listed by Ohloh, a website that indexes open source software [3]. At time of writing, Ohloh indexed over 5,000 individual sources such as GitHub, SourceForge and Google Code, comprising over 550,000 projects and over 10 billion lines of code [10]. We restricted ourselves to Java projects in this study due to the widespread adoption of test automation tools for Java, allowing us to easily parse configuration files to determine if the project isolates its test cases (a process described further below).

Using the Ohloh API, we identified the largest open source Java projects, ranked by number of active committers in the preceding 12 months.

Table 1: Statistics for subjects retrieved from Ohloh

	Min	Max	Avg	Std dev
LOC	268	20,280.14k	519.40k	1,515.48k
Active Devs	3.00	350.00	15.88	28.49
Age (Years)	0.17	16.76	5.33	3.24

Table 2: Projects creating a process per test, grouped by tests per project and by lines of code per project

# of Tests in Project	# of Projects Creating New Processes Per Test	
0-10	24/71	(34%)
10-100	81/235	(34%)
100-1000	97/238	(41%)
>1000	38/47	(81%)
All Projects	240/591	(41%)

Lines of Code in Project	# of Projects Creating New Processes Per Test	
0-10k	7/42	(17%)
10k-100k	60/200	(30%)
100k-1m	115/267	(43%)
>1m	58/82	(71%)
All Projects	240/591	(41%)

From the 1,200 projects, we downloaded the source code for 2,272 repositories (each project may have several repositories to track different versions or to track dependencies). We captured this data between August 15 and August 20, 2013. Basic statistics (as calculated by Ohloh) for these projects appear in Table 1, showing the aggregate minimum, maximum, average and standard deviation for lines of code, active developers, and age in years. A complete description of the entire dataset is available in the technical report that accompanies this paper [8].

The two most popular build automation systems for Java are ant [5] and maven [6]. These systems allow developers to write build scripts in XML, with the build system managing dependencies and automatically executing pre-deployment tasks such as running tests. Both systems can be configured to either run all tests in the same process or to create a new process for each test to execute in. From our 1,200 projects, we parsed these XML files to identify those that use JUnit as part of their build process and of those, how many direct JUnit to isolate each test in its own process. Then, we parsed the source files for each of the projects that use JUnit to determine the number of tests in each of these projects.

Next, we broke down the projects both by the number of tests per project and by the number of lines of code per project. Table 2 shows the result of this study. We found that 81% of those projects with over 1,000 tests create a new process for each test when executing it — only 19% do not isolate their tests in separate processes. When grouping by lines of code, 71% of projects with over one million lines of code create new processes for each test case. Overall, 41% of those projects in our sample that use JUnit create separate processes for each test. With these findings, we are confident in our claim that it is common practice, particularly among large applications (which may have the longest running test suites), to isolate each test case into its own process.

2.2 MQ2: Why Isolate Tests?

Understanding now that it is common practice for developers to isolate unit tests into separate processes, we next sought to answer **MQ2** — why developers isolate tests.

Perhaps in the ideal unit testing environment each unit test could be executed in the same application process, with pre-test and post-test methods ensuring that the application under test is in a "clean" state for the next test. However, handwritten pre-test and post-test teardown methods can place a burden on developers to write and may not always be correct. When these pre-test and post-test methods are not correct tests may produce false negatives, missing bugs that should be caught or false positives, incorrectly raising an exception when the failure is in the test case, not in the application begin tested.

For example, Muşlu et al. [32] discuss a bug in the Apache Commons CLI library that took approximately four years from initial report to reach a confirmed fix. This bug could be detected by running the application's existing tests independently of each other, but when running on the same instance of the application (using only the developer-provided pre and post-test methods to reset the application), it did not present because it was masked by a hidden dependency between tests that was not automatically reset.

There can be many confounding factors that create such hidden dependencies between tests. For instance, methods may have side effects that are undocumented. In a complex codebase with hundreds of thousands of lines of code, it may be very difficult to identify all potential side effects of an action. When a tester writes the test case for a method, they will be unable to properly reset the system state if they are unaware of that method's implicit side effects. To avoid this sort of confusion, testers may decide to simply execute each test in a separate process — introducing significant runtime overhead to their test suite.

In the remainder of this subsection, we describe these dependencies as they appear in the Java programming language and show a real-world example of one such dependency. Although some terminology is specific to Java, these concepts apply similarly to other languages.

Consider the following real Java code snippet from the Apache Tomcat project shown in Listing 1. This single line of code is taken from the "CookieSupport" class, which defines a series of configuration constants. In this example, the field "`ALLOW_EQUALS_IN_VALUE`" is defined with the modifiers `static final`. `static` signifies that it can be referenced by any object, regardless of position in the object graph. The `final` modifier indicates that its value is constant — once it is set, it can never be changed. The value that it is assigned on the right hand side of the expression is derived from a "System Property" (a Java feature that mirrors environmental variables).

This initializer is executed only once in the application: when the class containing it is initialized. If a test case depends on the value of this field then it must set the appropriate system property *before* the class containing the field is initialized. Imagine the following test execution: first, a test executes and sets the system property to false. Then the

```
public static final boolean
    ALLOW_EQUALS_IN_VALUE = Boolean.valueOf(
    System.getProperty(``org.apache.tomcat.
    util.http.ServerCookie.
    ALLOW_EQUALS_IN_VALUE'', ``false'')).
    booleanValue();
```

Listing 1: CookieSupport.java: An example of Java code that breaks test independence

initializer runs, setting the field `ALLOW_EQUALS_IN_VALUE` to false. Then the next test executes, setting the system property to true, expecting that `ALLOW_EQUALS_IN_VALUE` will be set to true when the field is initialized. However, because the value has already been set it will remain as it is: false, causing the second test to fail unexpectedly. This scenario is exactly what occurs in the Tomcat test suite and in fact, in the source code for several tests that rely on this property the following comment appears: "Note because of the use of static final constants in Cookies, each of these tests must be executed in a new JVM instance" [1].

Although the above example was from a Java application, the sort of leakage that occurred could happen in practically any language, provided that the developers follow a similar pattern. In any situation where a program reads in some configuration from a file and stores it in memory, there is the potential for such leakage.

There are certainly other potential sources of leakage between test executions. For instance in Java, the system property interface mentioned above allows developers to set properties that are persisted for the entire execution of that process. There are also various forms of registries provided by the Java API to allow developers to register services and lookup environments — these too, provide avenues through which data could be leaked between executions.

While in some cases it is possible (although perhaps complicated and time consuming) to write post-test methods to efficiently reset system state, take note that our example, the `static final` field can not be manually reset. The only option left to developers is to re-architect their codebase to make testing easier, for example by removing such fields (at the cost of the time to re-architect it and potential defects introduced by the new implementation) or to isolate each test to a separate process.

2.3 MQ3: The Overhead of Test Isolation

To gauge the overhead of test isolation we compared the execution time of several application test suites running in isolation with the execution time running without isolation. From the set of approximately 50 projects that include build scripts with JUnit tests that executed without modification or configuration on our test machine, we selected 20 projects for this study with the aim of including a mix of both widely used and recognizable projects (e.g., the Apache Tomcat project, a popular JSP server with 8537 commits and 15 recent 47 contributors overall), and smaller projects as well (e.g., JTor, an alpha-quality Tor implementation in Java with only 445 commits and 6 contributors overall). Details about each project including a direct link to download the application used can be found on Ohloh and are archived in our accompanying technical report [8].

Modifying each project's build scripts, we ran the test suite for each project twice: once with all tests executing in the same process, and once with one process per test. Then we calculated the overhead of executing each test in a separate process as $100 \times \frac{T_n - T_o}{T_o}$, where T_n is the absolute time to execute all of the tests in their own process, and T_o is the absolute time to execute all of the tests in the same process. We performed this study on our commodity server running Ubuntu 12.04.1 LTS and Java 1.7.0_25 with a 4-core 2.66Ghz Xeon processor and 8GB of RAM.

Table 3 shows the results of this study. For each project studied, we have included the total lines of code in the project (as counted by Ohloh), the overhead of isolating each test in its own process, and an indicator as to whether that project executes each test in its own process by default. On average, the overhead of executing each test in its own process is stunningly high: 618% on average. We investigated further the subjects "Bristlecone" and "upm," the subjects with the lowest and highest overhead respectively. We observed that Bristlecone had a low number of tests total (only four test classes in total), with each test taking on average approximately 20 seconds. Meanwhile, in the upm subject, there were 10 test classes total, and each test took on average approximately 0.15 seconds. In general, in test suites that have very fast tests (such as upm), the testing time can be easily dominated by setup and teardown time to create new processes. On the other hand, for test suites with longer running tests (such as Bristlecone), the setup and teardown time is masked by the long duration of the tests themselves.

Table 3: Overhead of isolating tests in new processes. Bolded applications normally isolate each test case. Additional descriptions of each subject appear in Table 5.

Project	LOC (in k)	Test Classes	Overhead
Apache Ivy	305.99	119	342%
Apache Nutch	100.91	27	18%
Apache River	365.72	22	102%
Apache Tomcat	5692.45	292	42%
betterFORM	1114.14	127	377%
Bristlecone	16.52	4	3%
btrace	14.15	3	123%
Closure Compiler	467.57	223	888%
Commons Codec	17.99	46	407%
Commons IO	29.16	84	89%
Commons Validator	17.46	21	914%
FreeRapid Downloader	257.70	7	631%
gedcom4j	18.22	57	464%
JAXX	91.13	6	832%
Jetty	621.53	6	50%
JTor	15.07	7	1,133%
mkgmap	58.54	43	231%
Openfire	250.79	12	762%
Trove for Java	45.31	12	801%
upm	5.62	10	4,153%
Average	475.30k	56.4	618%

3. APPROACH

Our key insight that enables Unit Test Virtualization is that it is often unnecessary to completely reinitialize an application in order to isolate its test cases. As shown in Figure 2, Unit Test Virtualization fits into a traditional unit testing process. During each test execution, Unit Test Virtualization determines what parts of the application will need to be reset during future executions. Then, during future executions, the affected memory is reset just before it is accessed. This section describes how we determine which parts of the application need to be reset and how we reset just those components.

Unit Test Virtualization relies on both static and dynamic analyses to detect what memory areas need to be reset after each test execution. This approach leverages the runtime

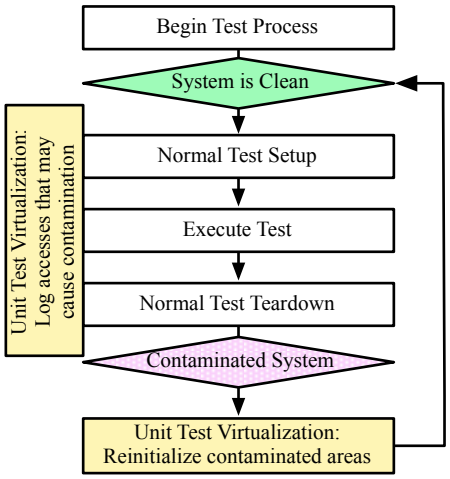

Figure 2: Unit Test Virtualization at the high level

performance benefits of static analysis (i.e., that the analysis is precomputed) with the precision of dynamic analysis.

Before test execution, a static analysis pass occurs, placing each addressed memory region into one of two categories: M_s ("safe") and M_u ("unknown"). Memory areas that are in M_s can be guaranteed to never be shared between test executions, and therefore do not need to be reset. An area might be in M_s because we can determine statically that it is never accessed, or that it is always reset to its starting condition at the conclusion of a test. This static analysis can be cached at the module-level, only needing to be recomputed when the module code changes. All stack memory can be placed in M_s because we assume that the test suite runner (which calls each individual test) does not pass a pointer to the same stack memory to more than one test (we also assume that code can only access the current stack frame, and no others). We find this reasonable, as it only places a burden on developers of test suite runners (not developers of actual tests), which are reusable and often standardized.

Memory areas that are placed in M_u are left to a runtime checker to identify those which are written to and not cleared. As each test case executes, memory allocations and accesses are tracked, specifically tracking each allocation that occurs in M_u. During future executions we ensure that accesses to that same location in M_u are treated as if the location hadn't been accessed before.

This is a general approach and indeed is left somewhat vague, as the details of exactly how M_s is built and how M_u is checked at runtime will vary from language to language. Further detail for the implementation of Unit Test Virtualization as applied to Java programs is provided in the Implementation section that follows.

4. IMPLEMENTATION

To evaluate the performance of Unit Test Virtualization we created a fully-functioning implementation of it for Java. We call our implementation VmVm, named after its technique of building a Virtual Machine-like container within the Java Virtual Machine. VmVm (pronounced "vroom-vroom") is released under an MIT license and is available on GitHub [7]. VmVm is compatible with any Java bytecode, but the runtime depends on newer language features, requiring a JRE version 5 or newer. We integrated VmVm with the

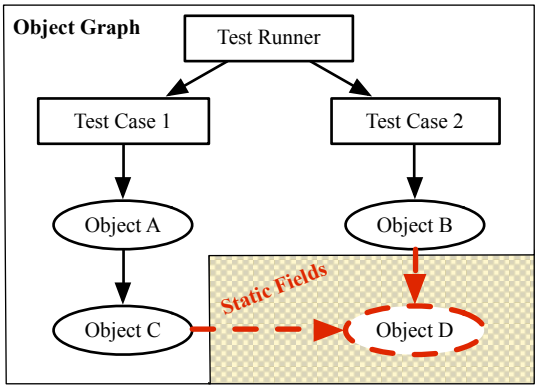

Figure 3: A leaked reference between two tests. Notice that the only link between both test cases is through a static field reference.

popular test utility JUnit and two common build systems: ant and maven, to reset the test environment between automated test executions with no intervention. VmVm requires no modification to the host machine or JVM, running in a completely unmodified environment. This detail is key in that VmVm is portable to different JVMs running on different platforms. VmVm requires no access to source code, an important feature when testing applications that use third party libraries (for which the source may not be available).

Architecturally, VmVm consists of a static bytecode instrumenter (implemented with the ASM instrumentation library [12]) and a dynamic runtime. The static analyzer and instrumenter identify locations that may require reinitializing and insert code to reinitialize if necessary at runtime. The dynamic runtime tracks what actually needs to be reset and performs this reinitialization between each JUnit test. These components are shown at a high level in Figure 4.

4.1 Java Background

Before describing the implementation details for VmVm, we first briefly provide some short background on memory management in Java. In a managed memory model, such as in Java, machine instructions can not build pointers to arbitrary locations in memory. Without pointer manipulation, the set of accessible memory S to a code region R in Java is constrained to all regions to which R has a pointer, plus all pointers that may be contained in that region. In an object oriented language, this is referred to as an *object graph*: each object is a node, and if there is a reference from object A to object B, then we say that there exists an edge from A to B. An object can only access other objects which are children in its object graph, with the exception of objects that are referred to by fields declared with the `static` modifier. The `static` keyword indicates that rather than a field belonging to the instances of some object of some class, there is only one instance of that field for the class, and therefore can be referenced directly, without already having a reference to an object of that class. It is easy to see how to systematically avoid leaking data between two tests through non-static references:

Consider the simple reference graph shown in Figure 3. Test Case 1 references Object A which in turn references Object C. For Test Case 2 to also reference Object A, it would be necessary for the Test Runner (which can reference Object A) to explicitly pass a reference to Object A to Test

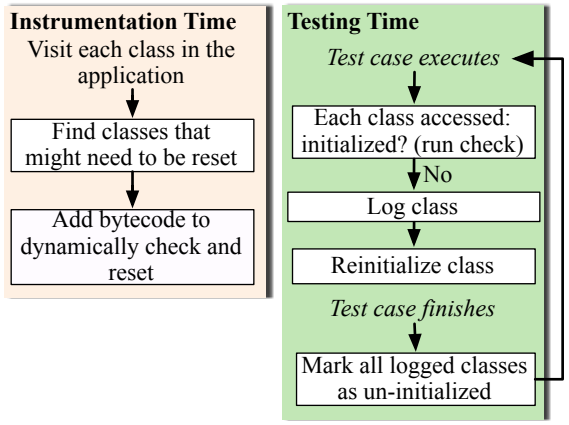

Figure 4: Implementation of VMVM

Case 2. As long as the test runner never holds a reference to a prior test case when it creates a new one, then this situation can be avoided easily. That is, the application being tested or the tests being executed could not result in such a leak: only the testing framework itself could do so, therefore, this sort of leakage is not of our concern as it can easily be controlled by the testing framework. Therefore, all memory accesses to non-`static` fields are automatically placed in M_s by VmVm, as we are certain that those memory regions will be "reset" between executions.

The leakage problem that we are concerned with comes from `static` fields: in the same figure, we mark "Object D" as an object that is statically referenced. Because it can be referenced by any object, it is possible for Test Case 1 and Test Case 2 to both indirectly access it - potentially leaking data between the tests. It is then only `static` fields that VmVm must analyze to place in M_u or M_s.

4.2 Offline Analysis

VmVm must determine which `static` fields are safe (i.e., can be placed in M_s). For a `static` field to be in M_s, it must not only hold a constant value throughout execution, but its value must not be dependent on any non-constant values. This distinction is important as it prevents propagating possibly leaked data into M_s. Listing 2 shows an example of a class with three fields that meet these requirements: the first two fields are set with constant values, and the third is set with a value that is non-constant, but dependent only on another constant value. We determine that a field holds a constant value if it is a `final` field (a Java keyword indicating that it is of constant value) referencing an immutable type (note that this is imprecise, but accurate).

In normal operation, when the JVM initializes a class, all `static` fields of that class are initialized. To emulate the behavior of stopping the target application and restarting it (in a fresh JVM), VmVm does not reinitialize individual `static` fields, instead reinitializing entire classes at a time.

```
public class StaticExample {
  public static final String s = "abcd";
  public static final int x = 5;
  public static final int y = x * 3;
}
```

Listing 2: Example of static fields

Therefore, to reinitialize a field, we must completely reinitialize the class that owns that field, executing all of the initialization code of that class (it could be possible to only reinitialize particular fields, but for simplicity of implementation, we did not investigate this approach). As a performance optimization, VmVm detects which classes need never be reinitialized. In addition to having no `static` fields in M_u, the initialization code for these classes must create no side-effects for other classes. If these conditions are met then the entire class is marked as safe and VmVm never attempts to reinitialize it.

This entire analysis process can be cached per-class file, and as the software is modified, only the analysis for classes affected need be recomputed. Even if it is necessary to execute the analysis on the entire codebase, the analysis is fairly fast. We measured the time necessary to analyze the entire Java API (version 1.7.0_25, using the rt.jar archive) and found that it took approximately 20 seconds to analyze all 19,097 classes. Varying the number of classes analyzed, we found that the duration of the analysis ranged from 0.16 seconds for 10 classes to 2.74 seconds for 1,000 classes, 12.07 seconds for 10,000 classes, and finally capping out at 21.21 seconds for all 19,097 classes analyzed.

4.3 Bytecode Instrumentation

Using the results of the analysis performed in the previous step, VmVm instruments the application code (including any external libraries, but excluding the Java runtime libraries to ensure portability) to log the initialization of each class that may need to be reinitialized. Simultaneously, VmVm instruments the application code to preface each access that could result in a class being initialized with a check, to see if it should be reinitialized by force. Note that because we initialize all static fields of a class at the same time, if a class has at least one non-safe static field, then we must check every access to that class, including to safe fields of the class. The following actions cause the JVM to initialize a class (if it hasn't yet been initialized):

1. Creation of a new instance of a class
2. Access to a static method of a class
3. Access to a static field of a class
4. Explicitly requesting initialization via reflection

VmVm uses the same actions to trigger re-initialization. Actions 1-3 can occur in "normal" code (i.e., by the developer writing code such as `x.someStaticMethod()` to call a method), or dynamically through the Java reflection interface, which allows developers to reference classes dynamically by name at runtime. Regardless of how the class is accessed, VmVm prefaces each such instruction with a check to determine if the class needs to be reinitialized. This check is synchronized, locking on the JVM object that represents the class being checked. This is identical to the synchronization technique specified by the JVM [31], ensuring that VmVm is fully-functional in multithreaded environments. Note that programmers can also write C code using the JNI bridge that can access classes — in these cases, VmVm can not automatically reinitialize the class if it is first referenced from JNI code (instead, it would not be reinitialized until it is first referenced from Java code). In these cases, it would require modification of the native code (at the source level) to function with VmVm, by providing a hint to VmVm the

first time that native code accesses a class. None of the applications evaluated in §5 required such changes.

4.4 Logging Class Initializations

Each class in Java has a special method called `<clinit>` which is called upon its initialization. For classes that may need to be reinitialized, we insert our logging code directly at the start of this initializer, recording the name of the class being initialized.

We store this logged information in two places for efficient lookup. First, we store the initialization state of the class in a `static` field that we add to the class itself. This allows for fast lookups when accessing a class to determine if it's been initialized or not. Second, we store an index that contains all initialized classes so that we can quickly invalidate those initializations when we want to reinitialize them.

4.5 Dynamically Reinitializing Classes

To reinitialize a class, VmVm clears the flag indicating that the class has been initialized. The next time that the class is accessed (as described in §4.3), the initializer is called. However, since we only instrument the application code (and not the Java core library set), the above process is not quite complete: there are still locations within the Java library where data could be leaked between test executions.

For instance, Java provides a "System Property" interface that allows applications to set process-wide configuration properties. We scanned the Java API to identify public-facing methods that set `static` fields which are internal to the Java API, first using a script to identify possible candidates, then verifying each by hand to identify false positives. In total, we found 48 classes with methods that set the value of some `static` field within the Java API. For each of these methods, VmVm provides copy-on-write functionality, logging the value of each internal field before changing it, and then restoring that value when reinitializing the application. To provide such support, VmVm prefaces each such method with a wrapper to record the value in a log, and then scans the log at reinitialization time to restore the values.

4.6 Test Automation Integration

VmVm plugs directly into the popular unit testing tool JUnit [2] and build automation systems ant [5] and maven [6]. This integration is important as it makes the transition from isolating tests by process separation to isolating tests by VmVm as painless as possible for developers.

Both ant and maven rely upon well-formed XML configuration files to specify the steps of the build (and test) process. VmVm changes approximately 4 lines of these files, modifying them to include VmVm in the classpath, to execute all tests in the same process, and to notify VmVm after each test completion so that shared memory can be reset automatically. As each test completes VmVm marks each class that was used (and not on its list of "safe" classes) as being in need of reinitialization.

Although we integrated VmVm directly into these popular tools, it can also be used directly in any other testing environment. Both the ant and maven hooks that we wrote consist of only a single line of code: `VirtualRuntime.reset()`, which triggers the reinitialization process.

Further details regarding the implementation of VmVm, including a more detailed and technical discussion of the instrumentation passes performed, are available in our accompanying technical report [8] or directly on GitHub [7].

5. EXPERIMENTAL RESULTS

To evaluate the performance of VmVm we pose and answer the following three research questions (RQ):

RQ1: How does VmVm compare to test suite minimization in terms of performance and fault-finding ability?

RQ2: In general, what performance gains are possible when using VmVm compared to creating a new process for each test?

RQ3: How does VmVm impact fault-finding ability compared to using traditional isolation?

We performed two studies to address these research questions. Both studies were performed in the same environment as our study from §2 — on our commodity server running Ubuntu 12.04.1 LTS and Java 1.7.0_25 with a 4-core 2.66Ghz Xeon processor and 8GB of RAM.

5.1 Study 1: Comparison to Minimization

We address **RQ1**, comparing VmVm to Test Suite Minimization (TSM), by turning to a study performed by Zhang et al. [45]. Zhang et al. applied TSM to Java programs in the largest study that we could find comparing TSM algorithms using Java subjects. In particular, they implemented four minimization techniques (each implemented four different ways, for a total of 16 implementations): a greedy technique [16], Harrold et al's heuristic [24], the GRE heuristic [15,16], and an ILP model [9]. Zhang et al. studied the reduction of test suite size and reduction of fault-finding ability of these TSM implementations using four real-world Java programs as subjects, comparing across several versions of each. The programs were selected from the Software-artifact Infrastructure Repository (SIR) [18]. The SIR is widely used for measuring the performance of TSM techniques, and includes test suites written by the original developers as well as seeded faults for each program.

We downloaded the same 19 versions of the same four applications evaluated in [45] from the SIR and instrumented them with VmVm. We executed each test suite twice: once with each test case running in its own process, and once with all test cases running in the same process but with VmVm providing isolation. The test scripts included by SIR with each application isolate each test case in its own process, so to execute them with VmVm we replaced the SIR-provided scripts with our own, running each in the same process and calling VmVm to reset the environment between each test. For each version of each application, we calculated the reduction in execution time (RT) for both VmVm and TSM as $RT = 100 \times \frac{|T_n| - |T_{new}|}{|T_n|}$ where T_n is the absolute time to execute each test in its own process, and T_{new} is the absolute time to execute all of the tests in the same process using VmVm, or the absolute time to execute the minimized test suite. For each version of the application with seeded tests we calculated the reduction in fault-finding ability (RF) as $RF = 100 \times \frac{|F_n| - |F_{vmvm}|}{|F_n|}$ where F_n is the number of faults detected by executing each test in its own process and F_{vmvm} is the number of faults detected by executing all tests in the same process using VmVm. Zhang

Table 4: Test suite optimization with VmVm and with Harrold et al's Test Suite Minimization (TSM) technique [24]. We show reduction in test suite size (*RS*, calculated by [45]) for TSM as well as reduction in test execution time (*RT*) for TSM, VmVm, and the combination of VmVm with TSM.

Application	LOC (in k)	Test Classes	TSM *RS*	TSM *RT*	VmVm *RT*	Combined *RT*
Ant v1	25.83k	34	3%	4%	39%	40%
Ant v2	39.72k	52	0%	0%	36%	37%
Ant v3	39.80k	52	0%	1%	36%	37%
Ant v4	61.85k	101	7%	4%	34%	37%
Ant v5	63.48k	104	6%	11%	25%	26%
Ant v6	63.55k	105	6%	11%	26%	27%
Ant v7	80.36k	150	11%	21%	28%	38%
Ant v8	80.42k	150	10%	18%	27%	37%
JMeter v1	35.54k	23	8%	2%	42%	42%
JMeter v2	35.17k	25	4%	1%	41%	42%
JMeter v3	39.29k	28	11%	5%	44%	48%
JMeter v4	40.38k	28	11%	5%	42%	47%
JMeter v5	43.12k	32	16%	8%	50%	52%
jtopas v1	1.90k	10	13%	34%	75%	77%
jtopas v2	2.03k	11	11%	31%	70%	76%
jtopas v3	5.36k	18	17%	27%	48%	68%
xml-sec v1	18.30k	15	33%	22%	69%	73%
xml-sec v2	18.96k	15	33%	26%	79%	80%
xml-sec v3	16.86k	13	38%	19%	54%	55%
Average	37.47k	51	12%	13%	46%	49%

et al. similarly calculated *RS* as the reduction in total suite size (number of tests) and *RF*.

Table 4 shows the results of this study (*RF* is not shown in the table, as it is 0 in all cases). Note that for each subject, Zhang et al. compared 16 minimization approaches, yet we display here only one value per subject. Specifically, Zhang et al. concluded that using Harrold et al's heuristic [24] applied at the test case level using statement level coverage (one of the 16 approaches evaluated in their work) yielded the best overall reduction in test suite size with the minimal cost to fault-finding ability. Therefore, in this experiment, we compared VmVm to this recommended technique.

To answer **RQ1**, we found that in almost all cases the reduction in testing time was greater from VmVm than from the TSM technique. On average, VmVm performed quite favorably, reducing the testing time by 46%, while the TSM technique reduced the testing time by only 13%. We also investigated the combination of the two approaches: using VmVm to isolate a minimized test suite, with results shown in the last column of Table 4. We found that in some cases, combining the two approaches yielded greater reductions in testing time than either approach alone. However, the speedup is not purely additive, since for every test case removed by TSM, the ability for VmVm to provide a net improvement is lowered (as it reduces the time between tests).

The *RF* values observed for VmVm are constant at zero, and every test case is still executed in the VmVm configuration. Although the TSM technique also had $RF = 0$ on all seeded faults, such a technique always risks a potential loss of fault finding ability. In fact, studies using the same algorithm on other subjects have found *RF* values up to 100% [36] (i.e., finding no faults). In general, our expectation is that VmVm results in no loss of fault-finding ability because it still executes all tests in a suite (unlike TSM). Our concerns for the impact of VmVm on fault-finding ability are instead related to its correctness of isolation: does VmVm properly isolate applications? We evaluate the correctness of VmVm further from this perspective in the following study of 20, large, real-world applications.

5.2 Study 2: More Applications

To further study the overhead and fault-finding implications of VmVm we applied it to the same 20 open source Java applications used for our motivating study. Most of the applications are well-established, averaging approximately 452,660 lines of code and having an average lifetime of 7 years. These applications are significantly larger than the SIR applications used in Study 1, for which the average application had only 25,830 lines of code. Additional information about each project is available on that project's information page on the Ohloh website [3] and is permanently archived in our accompanying technical report [8].

For each subject in this study we executed the test suite three times, each time recording the duration of the execution and the number of failed tests. First, we executed the test suite isolating each test case in its own process (what we will refer to as "traditional isolation"). Second, we executed the test suite with no isolation, with all test cases executed in the same process (which we will refer to as "not isolated"). Finally, we instrumented the subject with VmVm and executed all tests cases in the same process but with VmVm providing isolation. We then calculated the reduction in execution time *RT* as in Study 1 to address **RQ2**. Half of these subjects isolate test cases by default (i.e., half do not normally isolate their tests), yet we include these subjects in this study to show the potential speedup available if the subject did indeed isolate its test cases.

To answer **RQ3** (beyond the evidence found in the first study) we wanted to exercise VmVm in scenarios where we knew that the test cases being executed had side-effects. When tests have side-effects on each other they can lead to false positives (e.g., a test case that fails despite the code begin tested being correct) and false negatives (e.g., a test case that passes despite the code being tested being faulty). In practice, we were unable to identify known false negatives, and therefore studied the effect of VmVm on false positives, identifiable easily as instances where a test case passes in isolation but fails without isolation. We evaluated the effectiveness of VmVm's isolation by observing the false positives that occur for each subject when executed without isolation, comparing this to the false positives that occur for each subject when executed with VmVm isolation. We use the test failures for each subject in traditional isolation as a baseline. In all cases, the same tests passed (or failed) when using VmVm and when using traditional isolation.

The results of this study are shown in Table 5. Note that for each application we executed our study on the most recent (at time of writing) development version, identified by its revision number shown in Table 5.

On average, the reduction in execution time was slightly higher than in Study 1: 62% (56% when considering only the subjects that isolate their tests by default), providing strong support for **RQ2**: VmVm yields significant reductions in test suite execution time. We identified the "Bristlecone" subject

Table 5: Reduction in testing time (RT) and number of false positives for VMVM over 20 subjects. Here, false positives refer to tests that failed but should have passed. There were no cases of tests passing when expected to fail. We also include the overhead of isolation from both VMVM and forking each test. Bolded projects isolated their tests by default. The average is segregated into projects that isolate their tests by default, and those that did not isolate their tests.

		LOC	Age	# of Tests		Overhead			False Positives	
Project	Revisions	(in k)	(Years)	Classes	Methods	VMVM	Forking	RT	VMVM	No Isolation
Apache Ivy	1233	305.99	5.77	119	988	48%	342%	67%	0	52
Apache Nutch	1481	100.91	11.02	27	73	1%	18%	14%	0	0
Apache River	264	365.72	6.36	22	83	1%	102%	50%	0	0
Apache Tomcat	8537	5,692.45	12.36	292	1,734	2%	42%	28%	0	16
betterFORM	1940	1,114.14	3.68	127	680	40%	377%	71%	0	0
Bristlecone	149	16.52	5.94	4	39	6%	3%	-3%	0	0
btrace	326	14.15	5.52	3	16	3%	123%	54%	0	0
Closure Compiler	2296	467.57	3.85	223	7,949	174%	888%	72%	0	0
Commons Codec	1260	17.99	10.44	46	613	34%	407%	74%	0	0
Commons IO	961	29.16	6.19	84	1,022	1%	89%	47%	0	0
Commons Validator	269	17.46	6.19	21	202	81%	914%	82%	0	0
FreeRapid Downloader	1388	257.70	5.10	7	30	8%	631%	85%	0	0
gedcom4j	279	18.22	4.44	57	286	141%	464%	57%	0	0
JAXX	44	91.13	7.44	6	36	42%	832%	85%	0	0
Jetty	2349	621.53	15.11	6	24	3%	50%	31%	0	0
JTor	445	15.07	3.94	7	26	18%	1,133%	90%	0	0
mkgmap	1663	58.54	6.85	43	293	26%	231%	62%	0	0
Openfire	1726	250.79	6.44	12	33	14%	762%	87%	0	0
Trove for Java	193	45.31	11.86	12	179	27%	801%	86%	0	0
upm	323	5.62	7.94	10	34	16%	4,153%	97%	0	0
Average	1356.3	475.30	7.32	56.4	717	34%	618%	62%	0	3.4
Average (Isolated)	1739.3	743.16	8.86	58.7	419	12%	648%	56%	0	6.8
Average (Not Isolated)	973.3	207.43	5.79	54.1	1,015	57%	588%	68%	0	0

as a worst case style scenario that occurred in our study. In our original motivating study (described previously in Table 3), we found that there was almost no overhead (3%) to isolating the tests in this subject, due to the relatively long amount of time spent executing each individual test, and the very few number of tests. Therefore, we were unsurprised to see VmVm provide no reduction in testing time for this subject (and in fact, a slight overhead). On the other hand, we identified the "upm" subject as a near best case: with fast tests, the overhead of creating a new process for each test was very high (4,153%), providing much room for VmVm to provide improvement.

In no cases did we observe any false positives when isolating tests with VmVm, despite observing false positives in several instances when using no isolation at all. That is, no test cases failed when isolated with VmVm that did not fail when executed with traditional isolation. This finding further supports our previous finding for **RQ3** from Study 1, that VmVm does not decrease fault finding ability.

5.3 Limitations and Threats to Validity

The first key potential threat to the validity of our studies is the selection of subjects used. However, we believe that by using the standard SIR artifact repository (which is used by other authors as well, e.g., [23, 26, 39] and more) we can partially address this concern. The applications that we selected for Study 2 were larger on average, a deliberate attempt to broaden the scope of the study beyond the SIR subjects. It is possible that they are not representative of some class of applications, but we believe that they show both the worst and best case performance of VmVm: when there are very few, long running tests and when there are very many, fast running tests.

Our initial claim that these subjects represent the largest Java projects is based on two assumptions: first that number of contributing developers is an indicator of project size, and second that the projects in the Ohloh repository are a representative sample of all Java projects. We believe that we have captured all of the "largest" Java projects in our dataset regardless of the metric, given the very large number of projects retrieved. Additionally, given the overall size of Ohloh's data (which includes all repositories from, among other sources, GitHub and SourceForge) we believe that our study is at least as broad as previous work by other authors that utilized primarily test subjects from the SIR.

Unit Test Virtualization is primarily useful in cases where the time between tests is a large factor in the overall test suite execution time. Consider an extreme example: if some tests require human interaction, and others are fully automated, then the reduction in total cost of execution by removing the interaction-based tests from the suite may be significantly higher than what VmVm can provide by speeding up the automated component. If such a scenario arises, then it may be efficient to combine VmVm with Test Suite Minimization in order to realize the benefits of both approaches. However, in the programs studied, this is not the case: no test cases require tester input, and the setup time for each test was significant enough for VmVm to provide a (sometimes quite sizable) speedup.

Although we provide a high level approach to Unit Test Virtualization that is language agnostic (particularly among memory managed languages), we implemented it in Java.

The performance benefits that we revealed could be biased to the language features of Java. For instance, it may be that Java programmers more frequently isolate their unit tests in separate processes than other developers, in which case this approach may not provide such large performance benefits to test suites in other languages.

The final limitation that we discuss is the level of isolation provided by VmVm. VmVm is designed to be a drop-in replacement for "traditional" isolation where only in-memory state is isolated between test cases. It would be interesting to extend VmVm beyond this "traditional" isolation to also isolate state on disk or in databases. Such isolation would need to be integrated with current developer best practices, and we consider it to be outside of the scope of this paper.

6. RELATED WORK

Unit Test Virtualization can be seen as complementary to Test Suite Minimization (TSM), an approach where test cases that do not increase coverage metrics for the overall suite are removed, as redundant [24]. This optimization problem is NP-complete, and there have been many heuristics developed to approximate the minimization [15, 16, 23, 24,28,29,39,42]. TSM can be limited not only by imprecision of minimization approximations but also by the strength of optimization criteria (e.g., statement or branch coverage), a problem potentially abated by optimizing over multiple criteria simultaneously (e.g., [26]). We have shown that it is feasible to combine TSM with Unit Test Virtualization, minimizing both the number of tests executed and the amount of time spent executing those tests.

The effect of TSM on fault finding ability can vary greatly with the structure of the application being optimized and the structure of its test suite. Wong et al. found an average reduction of fault finding ability of less than 7.28% in two separate studies [40, 42]. On larger applications, Rothermel et al. reported a reduction in fault finding ability of over 50% for more than half of the suites considered [36]. Rothermel et al. suggested that this dramatic difference in results could be best attributed to the difference in the size of test suites studied, suggesting that Wong et al's [42] selection of small test suites (on average, less than 7 test cases) reduced the opportunities for loss of fault finding effectiveness [36]. The test suites studied in our first study averaged 51 test classes, and the suites in the second study averaged 56 test classes and over 700 individual test methods.

Similar to TSM is Test Suite Prioritization, where test cases are ordered to maximize the speed at which faults are detected, particularly in regression testing [19, 20, 37, 38, 41]. In this way, large test suites can still run in their entirety, with the hopes that faults are detected earlier in the process. We see Test Suite Prioritization and Unit Test Virtualization as complementary (and perhaps, able to be used simultaneously): Unit Test Virtualization increases the rate at which test suites execute, while prioritization increases the rate at which faults are detected by a test suite.

Muşlu et al. studied the effect of isolating unit tests on several software packages, finding isolation to be helpful in finding faults, but computationally expensive [32]. DTDetector detects (but can not repair) dependencies between tests by perturbing test execution ordering and observing test outcomes [46], and can only report dependencies that manifest by causing tests to pass or fail. VmVm both detects and prevents test dependencies by analyzing test code, which enables it to prevent dependencies even if they do not directly impact test outcomes. Holmes and Notkin created an approach to identify program dependencies using a hybrid static-dynamic analysis [25], which could be used to detect hidden dependencies between tests. Pinto et al. studied the evolution of test suites throughout several versions of seven real-world Java programs, measuring the sort of changes made to the test suites [35]. It would be interesting to study specifically the kinds of modifications made to test suites in order to support isolation of unit tests.

Unit Test Virtualization can be seen as similar in overall goal to sandboxing systems [4, 27, 30, 34]. However, while sandbox systems restrict all access from an application (or a subcomponent thereof) to a limited partition of memory, our goal is to allow that application normal access to resources, while recording such accesses so that they can be reverted, more similar to checkpoint-restart systems (e.g., [11, 14, 17, 21, 22]). Most relevant are several checkpointing systems that directly target Java. Nikolov et al. presented recoverable class loaders, allowing for more efficient reinitialization of classes, but requiring a customized JVM [33], whereas VmVm functions on any commodity JVM. Xu et al. created a generic language-level technique for snapshotting Java programs [43], however our approach eliminates the need for explicit checkpoints, instead always reinitializing the system to its starting state.

Unit Test Virtualization may be more similar to microrebooting, a system-level approach to reinitializing small components of applications [13], although microrebooting requires developers to specifically decouple components to enable microrebooting, while Unit Test Virtualization requires no changes to the application under test.

7. CONCLUSIONS AND FUTURE WORK

Unit Test Virtualization is a powerful new approach to reduce the time necessary to execute long test suites by reducing the overhead of isolating individual tests. We have shown the applicability of such an approach by studying 1,200 of the largest Java applications, showing that of the largest, over 80% isolate their test cases, and in general, 40% do. We implemented Unit Test Virtualization for Java, creating our tool VmVm (pronounced "vroom-vroom"), and showed that in our sample of applications, it reduced testing time by up to 97% (on average, 62%), while still executing all test cases and without any loss of fault finding ability. We are interested in exploring further the research challenges challenges of implementing Unit Test Virtualization for non-memory managed languages such as C, as well as the technical challenges in extending VmVm to other languages that target Java byte code (such as Scala).There is also further room for research in the implementation of VmVm: for instance, it may be possible to use program slicing to identify initializers for individual fields, hence relieving the need to reinitialize entire classes at a time.

8. ACKNOWLEDGMENTS

The authors thank Sidharth Shanker for his implementation of a Test Suite Minimizer, and Lingming Zhang for sharing additional details about his study [45]. The authors are members of the Programming Systems Laboratory, funded in part by NSF CCF-1161079, NSF CNS-0905246, and NIH U54 CA121852.

9. REFERENCES

[1] Cookiesbasetest.java. `http://svn.apache.org/repos/asf/tomcat/trunk/test/org/apache/tomcat/util/http/CookiesBaseTest.java`.

[2] Junit: A programmer-oriented testing framework for java. `http://junit.org/`.

[3] Ohloh, inc. http://www.ohloh.net.

[4] J. Ansel, P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 355–366, New York, NY, USA, 2011. ACM.

[5] Apache Software Foundation. The apache ant project. `http://ant.apache.org/`.

[6] Apache Software Foundation. The apache maven project. `http://maven.apache.org/`.

[7] J. Bell and G. Kaiser. Vmvm: Unit test virtualization in java. https://github.com/Programming-Systems-Lab/vmvm.

[8] J. Bell and G. Kaiser. Unit test virtualization with vmvm. Technical Report CUCS-021-13, Columbia University Dept of Computer Science, `http://mice.cs.columbia.edu/getTechreport.php?techreportID=1549&format=pdf`, September 2013.

[9] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 106–115, Washington, DC, USA, 2004. IEEE Computer Society.

[10] Black Duck Software. Black duck unveils ohloh open data initiative, launches beta code search capability. `http://www.blackducksoftware.com/news/releases/2012-07-18`.

[11] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under unix. *ACM Trans. Comput. Syst.*, 7(1):1–24, Jan. 1989.

[12] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[13] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot: A technique for cheap recovery. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[14] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[15] T. Chen and M. Lau. A new heuristic for test suite reduction. *Information and Software Technology*, 40(5–6):347 – 354, 1998.

[16] T. Chen and M. Lau. A simulation study on some heuristics for test suite reduction. *Information and Software Technology*, 40(13):777 – 787, 1998.

[17] G.-M. Chiu and C.-R. Young. Efficient rollback-recovery technique in distributed computing systems. *IEEE Trans. Parallel Distrib. Syst.*, 7(6):565–577, June 1996.

[18] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[19] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a junit testing environment. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 113–124, 2004.

[20] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 329–338, Washington, DC, USA, 2001. IEEE Computer Society.

[21] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Comput.*, 41(5):526–531, May 1992.

[22] E. Gelenbe. A model of roll-back recovery with multiple checkpoints. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 251–255, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[23] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel. On-demand test suite reduction. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 738–748, Piscataway, NJ, USA, 2012. IEEE Press.

[24] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993.

[25] R. Holmes and D. Notkin. Identifying program, test, and environmental changes that affect behaviour. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 371–380, New York, NY, USA, 2011. ACM.

[26] H.-Y. Hsu and A. Orso. Mints: A general framework and tool for supporting test-suite minimization. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 419–429, Washington, DC, USA, 2009. IEEE Computer Society.

[27] S. Jain, F. Shafique, V. Djeric, and A. Goel. Application-level isolation and recovery with solitude. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 95–107, New York, NY, USA, 2008. ACM.

[28] D. Jeffrey and N. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Trans. Softw. Eng.*, 33(2):108–123, Feb. 2007.

[29] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Trans. Softw. Eng.*, 29(3):195–209, Mar. 2003.

[30] Z. Liang, W. Sun, V. N. Venkatakrishnan, and R. Sekar. Alcatraz: An isolated environment for experimenting with untrusted software. *Transactions*

on Information and System Security (TISSEC), 12(3):14:1–14:37, Jan. 2009.

[31] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification*, Java SE 7 edition, Feb 2013.

[32] K. Muşlu, B. Soran, and J. Wuttke. Finding bugs by isolating unit tests. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 496–499, New York, NY, USA, 2011. ACM.

[33] V. Nikolov, R. Kapitza, and F. J. Hauck. Recoverable class loaders for a fast restart of java applications. *Mobile Networks and Applications*, 14(1):53–64, Feb. 2009.

[34] M. Payer and T. R. Gross. Fine-grained user-space security through virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, pages 157–168, New York, NY, USA, 2011. ACM.

[35] L. S. Pinto, S. Sinha, and A. Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 33:1–33:11, New York, NY, USA, 2012. ACM.

[36] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *In Proceedings of the International Conference on Software Maintenance*, pages 34–43.

[37] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Test case prioritization: an empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99)*, pages 179–188, 1999.

[38] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 97–106, New York, NY, USA, 2002. ACM.

[39] S. Tallam and N. Gupta. A concept analysis inspired greedy algorithm for test suite minimization. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '05, pages 35–42, New York, NY, USA, 2005. ACM.

[40] W. Wong, J. Horgan, A. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: a case study in a space application. In *Computer Software and Applications Conference, 1997. COMPSAC '97. Proceedings., The Twenty-First Annual International*, pages 522–528, 1997.

[41] W. E. Wong, J. R. Horgan, S. London, and H. A. Bellcore. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, ISSRE '97, Washington, DC, USA, 1997. IEEE Computer Society.

[42] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 41–50, New York, NY, USA, 1995. ACM.

[43] G. Xu, A. Rountev, Y. Tang, and F. Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 85–94, New York, NY, USA, 2007. ACM.

[44] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, Mar. 2012.

[45] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. An empirical study of junit test-suite reduction. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 170–179, 2011.

[46] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. Technical Report 2014-01-01, University of Washington, `ftp://ftp.cs.washington.edu/tr/2014/01/UW-CSE-14-01-01.PDF`, 2014.