VMVM: Unit Test Virtualization for Java

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

As software evolves and grows, its regression test suites tend to grow as well. When these test suites become too large, they can eventually reach a point where they become too length to regularly execute. Previous work in Test Suite Minimization has reduced the number of tests in such suites by attempting to identify those that are redundant (e.g. by a coverage metric). Our approach to ameliorating the runtime of these large test suites is complementary, instead focusing on reducing the overhead of running each test, an approach that we call Unit Test Virtualization. This Tool Demonstration presents our implementation of Unit Test Virtualization, VmVm (pronounced "vroom-vroom") and summarizes an evaluation of our implementation on 20 real-world Java applications, showing that it reduces test suite execution time by up to 97% (on average, 62%). A companion video to this demonstration is available online, at `https://www.youtube.com/watch?v=sRpqF3rJERI`.

Categories and Subject Descriptors

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

General Terms

Reliability, Performance

Keywords

Testing, test optimization, unit test virtualization

1. MOTIVATION AND OVERVIEW

In the process of maintaining software, developers often create regression tests to ensure that in the event that bugs recur, they will be detected by the test suite. While these test suites can be very useful for ensuring that previously-repaired faults are not reintroduced, they can become cumbersome to run: because developers are often creating new

To appear at the 36th International Conference on Software Engineering, Hyderabad India, June 2014 (ICSE '14).

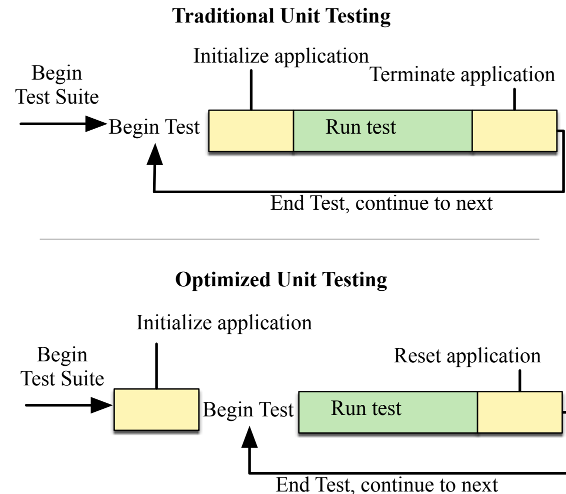

Figure 1: The test execution loop

tests, the test suite can grow very large. In fact, previous work has reported cases in industry where test suites can take up to several weeks to execute fully [14]. Long running test suites are contradictory to modern continuous integration environments, where tests are expected to be run frequently.

In order to reduce the cost of running these test suites, researchers have widely investigated the field of Test Suite Minimization [6,8–11,15,16]. Test Suite Minimization (TSM) is a general approach to reducing the size of test suites by identifying tests that are redundant — for example, those that do not add additional statement or branch coverage to the test suite. However, identifying exact duplicates is a hard problem, as such coverage metrics are not necessarily accurate.

We present a complementary approach to reducing the cost of running long test suites, instead focusing on reducing the overhead of the test suite as a whole[1]. Figure 1 shows a high level overview of the implementation of normal unit testing. For each test in the suite, the system under test is initialized, tested, then shut down. We observe that these initialization steps can be time consuming (relative to the tests themselves), and that perhaps test suite execution can be sped up by removing the initialization phase from the loop. In an ideal unit testing world, we would instead initialize the system under test only once, and then after

[1] Our ICSE 2014 research track publication [3] describes this approach in much greater detail, although without focusing on the usage of the tool as much as this demonstration does

Table 1: Number of projects that restart the system under test for each test, grouped by tests per project

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

each test execution, reset the system to its starting state, a process suggested by the bottom part of Figure 1.

At first, it may seem like this should be a non-problem: do developers actually create test suites where each test executes on a freshly started instance of the system under test? Why wouldn't developers simply write a post-test method to ensure that the system under test is in the correct state for the next test, without this costly application restart? We performed a study of the 591 largest open-source Java projects (that contained unit tests) in the Ohloh repository [1] to determine the extent to which developers create test suites where each test occurs in its own process. This study is described in much greater detail (including the selection of the projects) in Section 2.1 of our accompanying technical paper [3], and the results of the study are partially reproduced in Table 1 in this document. Overall 41% of all projects surveyed executed each test against a new process; in projects with many tests (over 1,000 tests), 81% of the projects did this. We find this to be an indicator that particularly in large, complex applications (which have many tests), it is necessary to execute each test in its own process.

To understand the overhead of isolating test cases, we selected 20 projects from this same corpus of large Java projects which include JUnit tests and executed their test suites several times: once with each test case running in its own process (isolated) and once with all test cases running in the same process (no isolation). We found the average runtime overhead of isolating test cases in this way to be 627% (this study is described in much greater detail in Section 2.3 of our accompanying technical paper [3]) — clearly presenting room for improvement.

For further insight into *why* this is necessary, we turned to studying the test suite for the Apache Tomcat J2EE server, version 7.0.42 (a project which executes each test case in a new process). We modified the test runner to execute each test in the same process (without fully restarting the application in-between tests) and observed 16 test cases fail (which didn't fail when executing the test suite normally). Upon inspection of one of the failing tests, we found the following comment: "Note because of the use of static final constants in Cookies, each of these tests must be executed in a new JVM instance" [2]. In this case, developers required that each test be executed in its own process because tests are dependent on certain application constants being set differently upon initialization, and the only way to re-execute that initialization code is to restart the application.

In the case of complex applications with hundreds of thousands of lines of code and many dependencies it may be difficult or impossible to identify all potential side effects of running a test. As an example of another such dependency, Muşlu et al. [12] discuss a bug in the Apache Commons CLI library that took approximately four years from initial report to reach a confirmed fix. The Apache Commons CLI project is an example of a project that does *not* isolate its test cases to execute on clean instances of the application (presumably to avoid the performance penalty). Part of the confusion in this case was that there had been a test case designed specifically to detect the failure reported, and that that test case passed, yet the failure remained. In fact, the reason that the test case passed was that the error was dependent on classes being initialized in a certain order: if the test case in question was executed before any others, it failed, and if it executed in the order that it was in by default, it passed. This error could have been detected immediately if each test were isolated by executing them in separate processes, on freshly started instances of the application under test. To avoid the potential for unexpected errors in tests, we believe that many developers (81% of the large applications surveyed) choose to execute each test case in its own process, hence isolating its side effects.

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

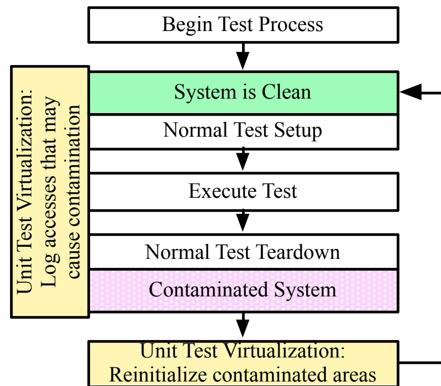

Figure 2: Unit Test Virtualization Overview

Unit Test Virtualization is our approach to compromise between the opposing tradeoffs of test isolation and test performance: a lightweight technique to achieve the same test case isolation that is otherwise achieved by restarting the application for every test execution. With Unit Test Virtualization, we execute each unit test in its own virtual container, effectively isolating all possible side effects of each test. This formal demonstration describes the usage of our implementation of Unit Test Virtualization in Java, VmVm, which is released under an MIT license and available for download on GitHub [2].

2. UNIT TEST VIRTUALIZATION

Our key insight that enables Unit Test Virtualization is that it's often unnecessary to completely reinitialize the system under test in-between every test case execution in order to still isolate application side-effects. Figure 2 presents a high-level overview of our approach: during each test execution, we log any memory accesses that may cause side-effects. Then, after each test completes, we reinitialize only those areas that are contaminated. Pronounced "vroom-vroom," VmVm supports such reinitialization by creating a Virtual Machine-like runtime within the Java Virtual Machine, and is our implementation of Unit Test Virtualization for Java.

In a static analysis pass, VmVm determines what Java Classes may possibly become contaminated with side-effects.

This analysis is coarse grained, and considers any Class that has mutable, static fields as targets for contamination. We leverage the managed memory model of Java here, in that it is straightforward to detect what memory could be shared between two test case executions. During execution, VmVm detects when such Classes are loaded, and after a test case completes, VmVm efficiently causes all such classes to be re-initialized upon their next access, hence resetting all of these mutable static fields which could cause information leakage between executions. We do not consider state shared between test cases in external files or databases, but consider this acceptable as such state leakage is also not addressed by running each test case in its own process. This demonstration will focus primarily on the application of VmVm (rather than on the details of its implementation). For additional details regarding its implementation, please refer to our accompanying technical paper [3].

2.1 Usage

VmVm is available for download via github [2], and is designed to be easy for developers to use, in a two step process. First, developers use VmVm to instrument their applications (including dependent libraries) with instructions to support efficient re-initialization. This process uses the ASM [4] byte code manipulation library to automate the instrumentation. VmVm provides a simple interface for instrumenting applications, taking as input a folder containing an application (and all of its dependent libraries) and outputting another folder containing a replica of the input, but with VmVm instrumentation added. The specific usage syntax is `java -cp lib/asm-all-4.1.jar:vmvm.jar edu.columbia.cs.psl.vmvm.Instrumenter <folder-to-instrument> <dest>`.

The second step, after instrumenting their application, is for developers to modify their application test scripts to execute test cases in the same process, and to notify VmVm when a test case completes so that it can reinitialize the effected portions of the application. This notification is made by simply calling our API method, `VirtualRuntime.reset()`. This entire process is simplified, as there are two common build automation systems used in Java, `ant` and `maven`, for which we provide the necessary code and detailed instructions to include VmVm in the testing process.

2.1.1 *Using* VmVm *with ant projects*

For ant projects, developers must only modify their ant build.xml file to add to the classpath two jars (the VmVm jar and the VMVM-AntMvn-Listener jar file, which contains classes specific for interacting with ant), to add our ant JUnit test listener to the configuration, and to execute all test cases in the same process. Specifically, the following lines are added:

```
<classpath>
 <pathelement path="ant-mvn-formatter.jar" />
 <pathelement location="vmvm.jar"/>
</classpath>
<jvmarg value="-Xbootclasspath/a:vmvm.jar:asm-all-4.1.jar"/>
<formatter classname="edu.columbia.cs.psl.vmvm.AntJUnitTestListener" extension=".xml"/>
```

To modify ant's configuration to execute all of the test cases in the same process, a developer would add the option `forkMode="once"` to the `junit` tag of the build.xml file.

2.1.2 *Using* VmVm *with maven projects*

Developers can modify their maven projects to use the VmVm isolation mechanism by the same two jar files in their test configuration, similar to the process for ant-based systems. In the case of maven, we provide test execution listener that hooks into the surefire testing plugin — developers simply modify their testing configuration to include our jars in the test classpath, and to register our listener:

```
<configuration>
 <additionalClasspathElements>
  <additionalClasspathElement>vmvm.jar</additionalClasspathElement>
  <additionalClasspathElement>ant-mvn-formatter.jar</additionalClasspathElement>
 </additionalClasspathElements>
 <properties>
  <property>
   <name>listener</name>
   <value>edu.columbia.cs.psl.vmvm.MvnVMVMListener</value>
  </property>
 </properties>
</configuration>
```

3. EVALUATION

To evaluate the performance of VmVm, we first compared it to several Test Suite Minimization (TSM) techniques. To compare VmVm to TSM, we turned to a study performed by Zhang et al. [18], which evaluated the performance of four TSM techniques. Zhang et al. studied the performance of each technique in terms of fault finding reduction and test suite size reduction across several version of four real-world Java applications downloaded from the Software Infrastructure Repository [7]. We downloaded the same applications, replicating their experiments using VmVm instead of the TSM mechanisms from their study, comparing the results. VmVm showed a reduction in test suite execution time of 42% with no loss of fault finding ability, while the TSM technique we compared to (there were several studied by [18]; we selected the most stable in terms of fault finding ability) showed a reduction in test suite size of 12%. From these results (described further in [3]), we conclude that VmVm can provide a significant reduction in test suite execution time, compared to existing TSM mechanisms.

We evaluated VmVm further by selecting 20 open source Java applications that contain JUnit test suites. Projects were selected 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 47 total contributors), and smaller projects (e.g. JTor, an alpha-quality Tor implementation with only 445 commits and 6 contributors overall). On average, each application had 475,000 lines of code, 56 test classes, and was 7.32 years old (significantly larger than the applications used to compare to TSM and used by [18]). For all twenty applications tested, VmVm was able to successfully instrument the application and isolate the test cases effectively, providing an average speedup of 62% with a maximum speedup of 97% (compared to isolating each test case in its own process).

More detailed results from this study are available in our accompanying technical paper [3].

4. 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 [9]. This optimization problem is NP-complete, and there have been many heuristics developed to approximate the minimization [6, 9, 11] (and others, not specified due to space constraints). We believe that it may be 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 implementation of Unit Test Virtualization can be seen as similar in overall goal to sandboxing systems. 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, but to record such accesses so that they can be reverted, more similar to checkpoint-restart systems. Most relevant are several checkpoint style systems that directly target Java. Nikolov et al. presented recoverable class loaders, allowing for more efficient reinitialization of classes, but requiring a customized JVM [13], whereas VmVm functions on any commodity JVM. Xu et al. created a generic language-level technique for snapshotting Java programs [17], 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 [5], although microrebooting requires developers to specifically decouple components to enable microrebooting, while Unit Test Virtualization requires no changes to the application under test. A much more thorough discussion of the relevant literature can be found in our technical paper [3].

5. CONCLUSIONS

Unit Test Virtualization is a powerful new approach to reduce the execution time of long test suites. VmVm (pronounced "vroom-vroom") is an easy to use implementation of Unit Test Virtualization, implemented for Java, and available for download via github [2]. VmVm provides a significant reduction in testing time in applications that isolate their test cases. We are very interested in developer feedback regarding the usability and applicability of VmVm, and are actively promoting it to relevant open source projects.

6. ACKNOWLEDGMENTS

The authors are members of the Programming Systems Laboratory, funded in part by NSF CCF-1161079, NSF CNS-0905246, and NIH U54 CA121852.

7. REFERENCES

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

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

[3] J. Bell and G. Kaiser. Unit Test Virtualization with VMVM. In *Proceedings of the 2014 International Conference on Software Engineering*, ICSE 2014, Piscataway, NJ, USA, Jun 2014. IEEE Press.

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

[5] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot: A technique for cheap recovery. OSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

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

[7] 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.

[8] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel. On-demand test suite reduction. ICSE 2012, pages 738–748, Piscataway, NJ, USA, 2012. IEEE Press.

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

[10] 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.

[11] 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.

[12] K. Muşlu, B. Soran, and J. Wuttke. Finding bugs by isolating unit tests. ESEC/FSE '11, pages 496–499, New York, NY, USA, 2011. ACM.

[13] 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.

[14] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Test case prioritization: an empirical study. volume ICSM '99, pages 179–188, 1999.

[15] 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.

[16] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. ICSE '95, pages 41–50, New York, NY, USA, 1995. ACM.

[17] G. Xu, A. Rountev, Y. Tang, and F. Qin. Efficient checkpointing of java software using context-sensitive capture and replay. ESEC-FSE '07, pages 85–94, New York, NY, USA, 2007. ACM.

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