Vroom: Faster Build Processes for Java

Jonathan Bell, Columbia University

Eric Melski and Mohan Dattatreya, Electric Cloud

Gail E. Kaiser, Columbia University

// To speed up testing, researchers combined two complementary approaches. Unit test virtualization isolates in-memory dependencies among test cases. Virtualized unit test virtualization isolates external dependencies such as files and network ports while long-running tests execute in parallel. //

SLOW SOFTWARE BUILD CYCLES substantially hinder continuous integration during development. They can be an even more significant nuisance for continuous delivery and other release processes. As a complex software system evolves and its compilation and packaging process becomes more complicated, building changes from a process that developers perform frequently on their desktop machines after every small code edit, to one performed nightly on a dedicated build machine, to one that can't even be performed in its entirety overnight. We aim to significantly reduce build time, with a sufficiently general solution applicable to both full ("clean") and incremental builds.

We decided to reduce building time by reducing testing time. To do this, we developed a system that combines two approaches. The first approach, *unit test virtualization*, isolates in-memory dependencies among test cases, which otherwise are isolated inefficiently by restarting the Java Virtual Machine (JVM) before every test. We call our implementation of this approach VMVM (Virtual Machine in the Virtual Machine, pronounced "vroom vroom"). The second approach, *virtualized unit test virtualization*, isolates external dependencies such as files and network ports while long-running tests execute in parallel. We call our implementation of this approach VMVMVM (Virtual Machine in a Virtual Machine on a Virtual Machine—"vroom vroom vroom").

The Dominance of Testing Time

We've found that the testing phase for real-world Java-based build processes often dominates compilation, packaging, and other traditional contributors to the build time. So, we focus on reducing the clock time needed to run test suites. Some of our industry partners report that they've been forced to remove testing from their regular build process as a stopgap solution. For instance, one partner reported that its Java-based build process took about eight hours—long enough to be problematic even for nightly build cycles.

To obtain concrete data on this problem, we measured the compilation, test, and other build phases for 20 popular open-source Java projects. We found the situation could be even worse than in our partner's anecdote: the testing phase took more than four times as long, on average, as the rest of the build (see Figure 1).

Unacceptably long test cycles aren't new. Previous research tried to reduce the time to run test suites.[1] It focused on

- selecting the smallest subset of relevant tests deemed most likely

0740-7459/15/$31.00 © 2015 IEEE

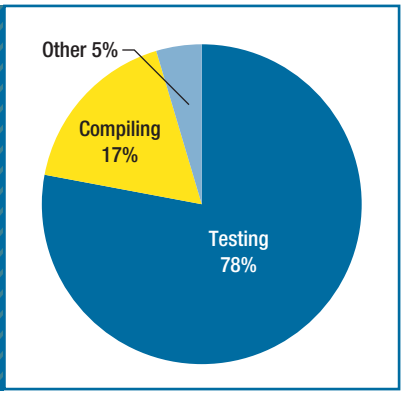

FIGURE 1. How build time is spent. By reducing testing time, we aim to significantly reduce build time, with a sufficiently general solution applicable to both full and incremental builds.

to find the faults for a given change set or
- reordering tests to execute those more likely to fail sooner.

The former approach can find only the defects affected by that change set. The latter approach might find defects sooner but only reduces the total time needed if testing halts after a time-out.

Furthermore, the approach that reduces the number of tests in a suite isn't sound—there's always a risk that some tests are deemed irrelevant when they aren't. (In the general case, it's undecidable whether one test suite is equivalent to another.) So, we seek to reduce testing time while still executing the entire test suite, with no loss of fault-finding ability.

Isolation Inefficiency

We studied the testing process that many Java projects employ, using the popular JUnit framework. JUnit can be used for integration and full system end-to-end tests as well as unit testing. We observed a common yet inefficient practice: each test executed in a fresh process—that is, in its own Java Virtual Machine (JVM).

An important implicit assumption in testing is that the result of test T shouldn't depend on the execution of some previous test T_p. This assumption of independent test cases, a part of the *controlled regression testing assumption* (applicable to other kinds of testing besides regression), is difficult to achieve efficiently.[2] Ideally, it's enforced with pretest setup methods and post-test tear-down methods.

However, for complex software (for example, software that uses black-box third-party APIs), ensuring these methods' correctness can be particularly difficult. The testing code might be buggy or incomplete, just like the application code being tested. Moreover, testers might miss resetting the state of some part of the system under test (and hence cause an unexpected dependency between one test and another). In this case, the results can range from false positives (where tests incorrectly raise an alarm when the code is correct) to, what's worse, false negatives (where tests fail to raise an alarm despite errors in the code).

So in practice, each test often executes in a separate process, which ensures that the tests are isolated (and don't have hidden dependencies), greatly simplifying writing the pretest and post-test methods. This isolation comes at a significant cost. We studied the overhead of executing each test in its own process, relative to the time needed to simply execute each test in the same process, for the 20 Java projects we mentioned earlier. We found it to be astonishingly high: on average, 618 percent (and up to 4,153 percent!).

Completely removing this isolation from the testing process in these applications would reduce application build time a net 56 percent. However, removing test isolation can have disastrous consequences on test suite correctness. In our study, we found 70 test cases that passed in isolation yet failed unexpectedly without it. Even worse, there are reports of test cases that erroneously pass when not isolated (despite a defect in the application under test) and fail only under isolation.

VMVM and VMVMVM

To combat this overhead while maintaining test case isolation, we developed unit test virtualization,[3] which automatically and efficiently isolates the side-effects of unit tests and other tests. The system reinitializes only that part of memory written by some previous test that could be read by the next test (determined by static and dynamic analysis), rather than restarting the entire process to reinitialize the entire in-memory state. This provides the same level of isolation that running each test class in its own JVM would provide. (Multiple test methods in the same test class still can have dependencies, which current versions of JUnit allow.)

We implemented our approach for Java in VMVM. As we show later, when we applied VMVM to the test suites of the 20 projects, it achieved an impressive average net speedup of the entire build time (compared to running each in a separate process). Because all tests executed, no loss of fault-finding ability occurred.

VMVM works well for speeding up test suites when the overhead of restarting the JVM between tests (usually a constant 1 to 2 seconds) constitutes a significant portion of the testing time. However, in cases with only a few test classes that are

very long (for example, 10 seconds each), removing the overhead of restarting the JVM and adding the overhead of our dynamic analysis can slow down testing.

In those latter cases, we take a complementary approach to reducing clock time: we leverage modern multicore hardware to execute multiple test cases in parallel. A simple approach to parallelizing test cases (offered by the most recent versions of Ant and Maven) uses a controller thread to distribute test cases in round-robin manner to several workers executing in parallel on the same machine. (This simply spawns extra processes running in the same directory.) However, of our 20 projects, five had tests fail erroneously with this parallelization, with test cases racing for access to shared resources (for example, files or sockets). To safely execute multiple tests simultaneously, each must have its own virtual file system and network interface.

So, we developed virtualized unit test virtualization, which leverages a distributed architecture. Each worker process executes in a distinct conventional VM (as in VMware or VirtualBox), with its own virtual file system and other system resources. This architecture is effective at simultaneously executing multiple tests that use local files and network resources (for example, binding to a socket and connecting back to that socket). However, it doesn't address test cases that interact with remote servers and databases; such interactions are usually avoided during testing and didn't occur in the test suites we examined. As we mentioned before, we call our implementation for Java VMVMVM.

The result is an integrated two-tier system that reduces the build time for Java projects by reducing testing time in two ways. First, VMVM reduces the time between short test cases that's taken to isolate the test cases. Second, VMVMVM reduces the total time for long test cases by letting them run in parallel (and using VMVM internally so that the same JVM can be used for the sequence of test jobs run in the same worker VM).

> We leverage modern multicore hardware to execute multiple test cases in parallel.

The Problem Scope

To empirically ground our efforts to reduce build time, we asked three main questions:

- Does testing take a significant portion of build time?
- Do developers isolate their test cases?
- If they do, is this isolation sufficient to let tests run in parallel?

To answer these questions, we downloaded the 1,200 largest free and open-source Java projects from the indexing website Ohloh (now Open Hub; www.openhub.net). From those, we selected the projects that executed JUnit tests during their Ant- or Maven-based builds. We tried to build all the 591 projects that use JUnit but found that only about 50 worked out of the box without significant configuration (for example, worked by running a single command such as `ant test` or `mvn test`). From those, we selected the 20 projects we've been talking about. This was a manageable set of projects that ensured a diversity of widely used, recognizable projects (for example, the Apache Tomcat JavaServer Pages server) and smaller projects (for example, JTor, an alpha-quality Tor implementation with a very small contributor base).

Measuring Testing Time

To answer the first question, we executed the entire build process for each application (using its Ant or Maven build script), recorded the time each build step took, and aggregated the time for all the testing (junit) steps and all the compilation (javac) steps. We executed this process 10 times, averaging the results.

Table 1 shows the results, which roughly matched our expectations based on our anecdotal industry evidence. For this study, we ensured that all tests were isolated in their own process; those projects that already performed this isolation are bold in Table 1. Testing took on average 78 percent of the build time.

Isolating Test Cases

To answer the second question, we statically analyzed the test scripts for the 591 projects to determine the percentage of them that executed each test case ("test class" in JUnit terminology) in its own process. Of those projects with the most test cases (over 1,000 test cases; 47 total), 81 percent executed each test in its own process. Overall, 41 percent of all the projects executed each test in its own process.

TABLE 1

The build speedup for 20 popular open source Java projects.*

Project	No. of classes	Test LOC × 1,000	Build time spent testing (%)	Build speedup (%)	
				VMVM†	VMVMVM‡
Apache Commons Codec	46	17.99	91	83	85
Apache Commons Validator	21	17.46	93	31	34
Apache Ivy	119	305.99	95	70	86
Apache Nutch	27	100.91	92	13	16
Apache River	22	365.72	74	41	43
Apache Tomcat	292	5,692.45	99	28	68
betterFORM	127	1,114.14	98	73	90
Bristlecone Performance Test Tools	4	16.52	94	–2	12
btrace	3	14.15	49	23	–20
Closure Compiler	223	467.57	93	63	75
Commons IO	84	29.16	96	52	85
FreeRapid Downloader	7	257.70	43	43	41
gedcom4j	57	18.22	98	57	75
JAXX	6	91.13	48	45	34
Jetty—Java HTTP Servlet Server	6	621.53	64	18	16
JTor	7	15.07	61	64	63
mkgmap	43	58.54	88	59	68
Openfire	12	250.79	32	32	31
Trove for Java	12	45.31	56	60	59
Universal Password Manager	10	5.62	97	95	70
Average	56	475.30	78	47	52

** Bold indicates that, in the default configuration, the build isolated each test by executing it in its own process and ran all the test processes sequentially in the same OS on the same machine (no virtual machines). Otherwise, the default configuration didn't isolate tests but ran them all in the same process.*

† Virtual Machine in the Virtual Machine

‡ Virtual Machine in a Virtual Machine on a Virtual Machine

Test isolation is necessary for many complex software systems. Otherwise, the testers would need to write additional test cases for the pretest setup and post-test tear-down methods, to test the tests. Kıvanç Muşlu and his colleagues pointed out a perfect example of what can happen when tests aren't isolated.[4] They found that a fault that took four years to resolve (Apache Commons CLI-26, 186 and 187) could have been detected immediately (even before users reported it) if the project's test cases had been isolated.

The problem was that several test cases checked the Apache library's behavior under varying configurations, and their setup stored these configurations in a `static` field. However, other tests assumed that the system under test would be clean, in a default configuration. But some of the configuration-modifying tests

happened to be earlier in the test suite and didn't restore the static field when finishing. So, the later tests in the test suite that should have caught the defect passed (because the defect occurred in only the default configuration), and the defect went undetected.

In Sai Zhang and his colleagues' sample of popular open-source Java software, 96 tests displayed similar dependencies.[5] Of those dependencies, 61 percent arose from side effects from accesses to static fields.

Isolation and Parallelism

Our final motivating study addressed the need to enforce further isolation than process separation provides, when tests run in parallel. Although executing each test in its own process eliminates in-memory dependencies between test cases, other persistent state could cause dependencies among tests. For example, when multiple tests read and write from the same file on disk, the test run's results might depend on their execution order. Even if each test properly cleans up after itself (for example, deleting the file), these tests still can't execute concurrently on the same machine because they would compete for simultaneous access to the same file.

We executed the test suites for each of the 20 Java projects several more times. We isolated each test case in a separate process but ran up to eight tests from each test suite concurrently on the same machine (using the parallelization option available in the most recent versions of Ant and Maven). As we mentioned before, five projects (Apache Ivy, Apache Nutch, Apache Tomcat, mkgmap, and Jetty) had tests fail erroneously when executed concurrently. Moreover, even more failures might have occurred; we didn't explore all possible scheduling combinations in which tests might execute concurrently.

Examining these five projects' source code, we found two sources of dependencies: files and network ports. For example, some tests created temporary directories, wrote files to them, and deleted the directories when the test ended. This caused conflicts when two test cases executed simultaneously. Both tests used the same temporary directory; the test that finished first deleted the directory, causing the second test to unexpectedly fail.

We also saw several cases of conflicting bindings to network ports. In these cases, part of the test setup started a mock server listening to some predefined port and then connected the code under test to that port. The first test to bind to the port succeeded; subsequent tests that executed while that first test was running unexpectedly failed, unable to bind to the port. None of the observed dependencies occurred when the test suites executed serially; all the tests correctly cleaned up the environment state at their conclusion. None of these projects had conflicts on resources external to the machine (for example, remote servers).

> We found two sources of dependencies: files and network ports.

Reducing Testing Time

Our key insight is that we can provide the same level of test case isolation as process separation without paying the high overhead cost of restarting the JVM for each test case. The typical reason engineers ask the build process to restart the JVM before every test case is to eliminate in-memory dependencies between test cases. Our insight relies on the observation that these dependencies are easy to track and manage within a single JVM (without restarting) with much lower overhead and thus shorter testing time. Then, we found that we can speed up testing further by parallelizing—running multiple tests simultaneously—if we can also remove system-level dependencies.

Isolating Object Graphs

The JVM provides a managed memory environment in which code can't construct pointers to arbitrary memory locations. Instead, the memory M accessible to some executing function F is constrained to only that memory reachable from F's object graph, plus any static fields. The object graph encompasses the traditional object-oriented view of memory: F can receive several pointers to objects as parameters, those objects can in turn have pointers to other objects, and so on.

It's easy to imagine how to isolate this object graph between test executions. Assume that the test runner (which is instantiating each test) constructs new arguments to pass to each test case and doesn't pass a reference to any of the same objects to multiple tests, as would normally

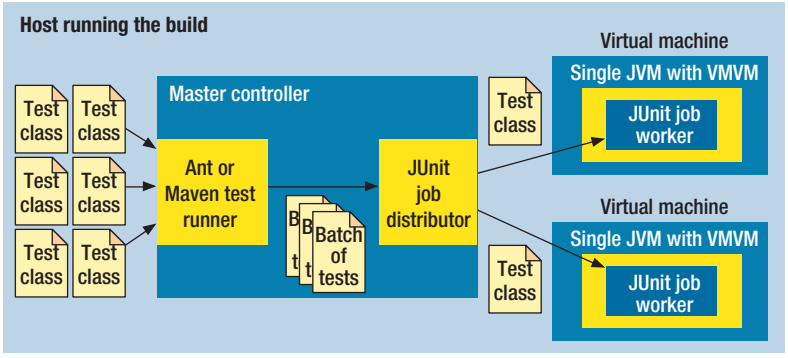

FIGURE 2. The high-level architecture of VMVMVM (Virtual Machine in a Virtual Machine on a Virtual Machine). Our system integrates directly with test execution initiated by Ant or Maven, or via JUnit directly. Each test class execution is intercepted and sent to a master controller that delegates test cases to workers.

be the case. Then (at their creation) no two tests' object graphs will have overlapping nodes. Because the test runner is standardized to the testing framework, ensuring that tests at this level are isolated is easy.

Isolating static Fields

In contrast, Java **static** fields are like global variables: they're directly referenced by their field name and class name (no additional pointers needed). So, we must isolate them.

Our approach to isolating **static** fields is simple and emulates exactly what happens when the JVM restarts. Between each pair of test cases, we reexecute the initializer for every **static** field, effectively eliminating **static** fields as a source of dependencies between tests. VMVM optimizes this basic approach to further reduce the overhead of isolating test cases. It reinitializes only the mutable **static** fields of classes used during prior test executions when they're needed and always ignores fields that are immutable (guaranteed to be unchanged).

VMVM performs offline static analysis and bytecode instrumentation before test execution. This analysis and instrumentation occurs each time the application code or tests change. In this phase, VMVM determines which classes contain no mutable **static** fields and thus won't ever need reinitialization.

VMVM then emulates exactly the process JVM uses internally for initializing a class. It inserts guards (in the bytecode) around every access to the class to check whether the class must be reinitialized and, if so, to reinitialize it. It inserts these guards before every instruction that might create a new instance of a class (the **new** bytecode instruction), access a class's **static** method (the **INVOKESTATIC** bytecode instruction), or access a class's **static** field (the **GETSTATIC** and **PUTSTATIC** instructions). VMVM also intercepts calls to Java's reflection library that would dynamically perform the same operations, adding guards on the fly. In addition, it modifies each class initializer to insert instructions to log its execution. This lets VMVM efficiently determine exactly which classes were used by previous test cases and thus will need to be reinitialized in the next test case that references them.

VMVM performs all these instrumentations on only the application bytecode (not code in the Java core library set). To reinitialize **static** fields belonging to classes in the Java core libraries, we wrote a tool that scans the Java API to identify public-facing methods that set **static** fields. We then verified each result by hand (this process would have to be repeated only for new versions of Java). We found 48 classes with methods that set the value of some **static** field in the Java API. For each of these methods, VMVM provides copy-on-write functionality, logging each internal field's value before changing it and then restoring that value when reinitializing. To provide this support, VMVM prefaces each such method call with a wrapper to record the value in a log and then scans the log at reinitialization (between each pair of test cases) to restore the value.

Running Tests in Parallel

As we mentioned before, our VMVMVM prototype lets us execute test cases in parallel without interference by employing a conventional VM to ensure that each simultaneously executing test has its own file system and virtual network interface (see Figure 2). VMVMVM still relies on a test's manually written pretest and post-test methods to clean up system resources between test classes, so that no two tests are dependent as a result of some shared file. In our study of the 20 projects, we found no test classes that were dependent (when executed sequentially) because of shared system resources. Other researchers have confirmed that such dependencies are uncommon.[5]

To run *N* tests in parallel, we create *N* VMs, with a single daemon running in each one. Each daemon listens for requests from our master controller process, executes the tests submitted by the controller, and returns the results. The controller collects the results, reorders them to appear as if they executed serially, and returns them to the original invoker of the test suite as if they had executed sequentially on the same machine. The daemons use VMVM to provide in-memory isolation between test cases, so they don't start a new JVM for each test case.

For easy integration, we provide a drop-in replacement for the Ant JUnit task, the Maven JUnit target, and a custom JUnit runner. Engineers need only change their build configuration to use our JUnit target (which accepts the exact same arguments as the normal target); test cases are automatically parallelized.

For instance, when using our Ant task, VMVMVM will automatically start a local socket server, spin up worker processes, distribute the test requests, and return the results (in serial order) to the Ant task. Existing test listeners and custom test runners continue to work normally.

Evaluation

We evaluated how our approaches reduced the 20 projects' build time. For each application, we first ran the entire test suite with each test case isolated in its own process (the baseline configuration). Then, we ran the suite with all tests executing in the same process, but using VMVM to provide isolation. Finally, we ran the suite distributed across three workers, each one running all its tests in the same process, again with VMVM providing the isolation. We performed this entire process 10 times, averaging the results.

We performed this study on our commodity server running Ubuntu 12.04.1 LTS (Long Term Support) and Java 1.7.025 with a four-core 2.66-GHz Xeon processor and 32 Gbytes of RAM. Each worker ran in its own VMWare Workstation 10 VM, running Ubuntu 12.04.1 LTS and allocated 2 Gbytes of RAM and two cores.

Table 1 shows the results. All speedups are relative to the length of a build that isolated each test by executing it in its own process and then ran all the test processes sequentially in the same OS on the same machine (no VMs). If this was a project's default configuration, the table shows it in bold; otherwise, the default configuration didn't isolate tests but ran them all in the same process.

The average speedups provided by both solutions (VMVM alone and VMVMVM parallelized in multiple VMs) were comparable. Build time decreased by 47 percent when we used VMVM to isolate test cases and by 52 percent when we added VMVMVM.

We were interested most in the cases in which one approach significantly eclipsed the other. For example, for Apache Tomcat, VMVM sped up the overall build by only 28 percent, whereas VMVMVM sped it up by 68 percent. For btrace, VMVM sped up the overall build by 23 percent, whereas VMVMVM—with VMVM included—slowed it down by 20 percent. Tomcat had almost 300 test classes, with a fairly even distribution of test lengths, so parallelization was quite effective. On the other hand, btrace had only three test classes, taking 1,410 ms, 36 ms, and 23 ms, respectively.

For btrace, parallelization provided no significant benefit because a single test class dominated the testing time. The communication overhead of distributing the tests to the workers showed through, causing VMVMVM to provide a slowdown compared to VMVM alone. In the other applications in which VMVMVM didn't perform as well as VMVM, the overall number of test classes was nearly the same as the number of workers (three), and one or two of the tests dominated the others in execution time. In such cases, parallelizing test classes wasn't effective; using only VMVM increased speedup.

> In projects with a diverse range of test classes, VMVMVM greatly reduced the time to run a complete build.

Our study shows that in projects with a diverse range of test classes, VMVMVM greatly reduced the time to run a complete build. On popular open source software, such as Apache Tomcat, this reduction was huge. We've released a stand-alone version of VMVM under an MIT license via GitHub (https://github.com/Programming-Systems-Lab/vmvm). We're working with our

industrial partners to release a full version of VMVMVM. We hope our efforts to reduce Java build times can help relieve release engineers from long-running builds.

Acknowledgments

Jonathan Bell and Gail Kaiser are members of Columbia University's Programming Systems Laboratory, which is funded partly by US National Science Foundation awards CCF-1302269, CCF-1161079, and CNS-0905246 and US National Institutes of Health grant U54 CA121852.

References

1. S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, 2012, pp. 67–120.
2. G. Rothermel and M.J. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE Trans. Software Eng.*, vol. 22, no. 8, 1996, pp. 529–441.
3. J. Bell and G. Kaiser, "Unit Test Virtualization with VMVM," *Proc. 36th Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 550–561.
4. K. Muşlu, B. Soran, and J. Wuttke, "Finding Bugs by Isolating Unit Tests," *Proc. 19th ACM SIGSOFT Symp. and 13th European Conf. Foundations of Software Eng.* (ESEC/FSE 11), 2011, pp. 496–499.
5. S. Zhang et al., "Empirically Revisiting the Test Independence Assumption," *Proc. 2014 Int'l Symp. Software Testing and Analysis* (ISSTA 14), 2014, pp. 384–396.

ABOUT THE AUTHORS

JONATHAN BELL is a PhD student in software engineering at Columbia University. His research interests include software testing, program analysis, and fault reproduction. Bell received an M Phil in computer science from Columbia University. He's a member of the IEEE Computer Society. Contact him at jbell@cs.columbia.edu.

ERIC MELSKI is the chief architect at Electric Cloud and has been developing build optimization software there for more than 12 years. His research interests include distributed systems, high-performance computing, parallel programming, and kernel development. Melski received a BS in computer science from the University of Wisconsin. Contact him at eric@electric-cloud.com.

MOHAN DATTATREYA is the senior director of engineering at Electric Cloud. His research interests include software-defined networks, application acceleration, and distributed-systems performance engineering. Dattatreya received an MS in computer science from Stanford University. Contact him at mohan@electric-cloud.com.

GAIL E. KAISER is a professor of computer science at Columbia University. Her research interests include software reliability and robustness, information management, social software engineering, and software development environments and tools. Kaiser received a PhD in computer science from Carnegie Mellon University. She was a founding associate editor of *ACM Transactions on Software Engineering and Methodology* and has been an editorial board member of *IEEE Internet Computing*. She's a senior member of IEEE. Contact her at kaiser@cs.columbia.edu.