Distributed In Vivo Testing of Software Applications

Matt Chu, Christian Murphy, Gail Kaiser
Department of Computer Science, Columbia University, New York NY 10027
{*mwc2110, cmurphy, kaiser*}*@cs.columbia.edu*

Abstract

The in vivo software testing approach focuses on testing live applications by executing unit tests throughout the lifecycle, including after deployment. The motivation is that the "known state" approach of traditional unit testing is unrealistic; deployed applications rarely operate under such conditions, and it may be more informative to perform the testing in live environments. One of the limitations of this approach is the high performance cost it incurs, as the unit tests are executed in parallel with the application. Here we present distributed in vivo testing*, which focuses on easing the burden by sharing the load across multiple instances of the application of interest. That is, we elevate the scope of in vivo testing from a single instance to a community of instances, all participating in the testing process. Our approach is different from prior work in that we are actively testing during execution, as opposed to passively monitoring the application or conducting tests in the user environment prior to execution. We discuss new extensions to the existing in vivo testing framework (called Invite) and present empirical results that show the performance overhead improves linearly with the number of clients.*

1. Introduction

Thorough testing of a commercial software product is unquestionably a crucial part of the development process, but the ability to faithfully detect all defects ("bugs") in an application is severely hampered by numerous factors. For large, complex software systems, it is typically impossible in terms of time and cost to reliably test all configuration options or to anticipate all the different system states before releasing the product into the field.

The in vivo testing approach [9] seeks to address this by extending the testing phase into deployed environments, which are subject to real world workloads and changes in state. The claim is that many (if not all) deployed software products still have latent defects and these may reveal themselves in application states that were unanticipated and/or untested in the development and testing environments. In addition, bugs may also exist in the unit tests themselves, so a unit test that passes during development may not necessarily pass after deployment, and a unit test that passes does not ensure it is without flaw.

In vivo testing approaches this problem by executing unit tests in the context of the running application within the deployment environment, as opposed to a controlled or "clean" state. Tests are run continuously through the lifetime of the application at arbitrary points. Crucial to the approach is the notion that the test itself must not alter the state of the application; this is clearly undesirable. To ensure this, the test is executed in a separate process which is an exact replica of the original.

This important guarantee leads to a major limitation of in vivo testing: the high cost of replicating a process. One possibility is to limit the number of tests that are run, but this also limits the effectiveness of the approach.

This paper introduces the idea of *distributed in vivo testing*: applying the in vivo testing approach to a community of applications, where the size of the community can be leveraged to detect bugs and reduce overhead.

Applying a distributed approach to in vivo testing is motivated by two reasons. First, amortizing the workload over many instances tackles a major limitation, high performance impact, without sacrificing the quantity of tests being conducted. Second, in vivo testing relies upon testing as many unexpected permutations of state as possible, in the hopes that one will be encountered that is not correctly handled by the code. Having a community of applications collaboratively working together increases the possibility that an instance will find these erroneous permutations of state.

2. Related work

Our previous research into in vivo testing [9] was principally inspired by the notion of "perpetual testing" [12] [14] [13] [18], which suggests that analysis and testing of software should continue into the deployment phase and throughout the entire lifetime of the application. Perpetual testing advocates that analysis and testing should be ongoing activities that improve quality through several generations of the product, in the development environment (the

lab) as well as the deployment environment (the field). The in vivo testing approach is a type of perpetual testing in which the same unit tests can be used in both environments with only minor modifications, and the tests do not alter the state of the application under test.

The Skoll project [6] [8] has extended the notion of continuous testing [15] [16] into the deployment environment by carefully managed facilitation of the execution of tests at distributed installation sites, and then gathering the results back at a central server. The principal idea is that there are simply too many possible configurations and options to test in the development environment, so tests can be run on-site to ensure proper quality assurance. Whereas the Skoll work to date has mostly focused on acceptance testing of compilation and installation on different target platforms and performance testing, distributed in vivo testing is different in that it seeks to execute unit tests within the application while it is running under normal operation.

Other approaches to perpetual testing include the monitoring and profiling of deployed software, as surveyed in [5], though many of these do not use a distributed approach [17] [4]. One that does, however, is the GAMMA system [11] [10], which uses software tomography for dividing monitoring tasks and reassembling gathered information; this can then be used for onsite modification of the code (for instance, by distributing a patch) to fix defects. The principle difference is that GAMMA is a monitoring tool that passively measures path or data access coverage, or memory access, and expects users to report bugs. However, with distributed in vivo testing, we actively test the applications of interest by executing the unit tests and automatically report any failures found.

Lastly, distributed in vivo testing utilizes the notion of "application communities" [7], in which application instances in a software monoculture share information. This allows in vivo testing to distribute the testing load in space as well as in time. Our work is different in that we are not concerned with repair policy or security violations, but rather with conducting functional testing while the application is running in the field.

3. The distributed Invite framework

The original implementation of the in vivo approach was the Invite (IN VIvo TEsting) framework [9], which works as follows: the developer specifies a list of target classes in the application under test to instrument, and a set of unit tests to execute. Invite then instruments each method in the list of classes so that with a predetermined probability one of the unit tests are executed. If a unit test is to run, the process is forked, the child process executes the test, the results are recorded, and the child process terminates. Thus (by design) a unit test can only be run if an instrumented method is called; if the application under test is idle, then no tests will ever be run. There are a number of requirements the code must satisfy (the tests must be in their own class, test methods must follow a common convention, *etc.*), which are fully detailed in [9].

In this work we have extended the original framework with a distributed component which follows a simple server-client model, where each application under test includes the Invite client. The Invite server is a separate standalone component that runs on a separate machine.

The new distributed Invite framework seeks to reduce the overhead of each Invite client by reducing the number of tests each instance has to run while maintaining the same global quantity. This distributed effort is coordinated by the central Invite server. The basic protocol is:

1. The Invite server is initialized and ready to receive Invite client requests.

2. When an instance of the instrumented application begins, it logs into the Invite server for the first time and registers itself as an Invite client as part of its initialization process. The Invite client receives a unique client id, a list of tests to run, a probability p to run the tests, and a time t to reconnect to the Invite server.

3. The application under test executes normally, except that with probability p the Invite client (randomly) executes one of the assigned tests using the in vivo testing approach. When the test is complete the results are sent back to the Invite server.

4. At time t the Invite client connects to the Invite server with its id as identification, and receives a new time t' to reconnect, as well as possibly a new list of tests and/or a new probability p'. In this way the Invite server can dynamically adjust to changes in community size by modifying the values it assigns.

The Invite server handles all the bookkeeping of maintaining client ids, distribution and assignment of the test suite to the community, the results of each test and the Invite client that executed it, and the reconnect times. The full test suite is intelligently partitioned by the Invite server based on the size of the community, and these partitions are rotated periodically (as in [11] and [7]). There is also a simple console client that allows a developer to query the Invite server for reports. Currently, the server can report the number of clients in its community, the number of tests run by each client, and the results of each individual test.

4. Experiments

We will show in the following experiments that distributing the testing load as described improves performance by

decreasing the number of tests each individual member in the community has to run.

4.1. Setup

We experimented on the open-source application Jetty, a webserver/JSP container written purely in Java [3]. Instrumentation was performed using AspectJ [1] and stress testing was performed using The Grinder [2]. The experiments were conducted on Sun Sparc machines with dual 300 MHz CPUs and 512 MB of RAM running SunOS 5.8, with each instance of Jetty running on its own machine. The entire community was on the same internal 100 MBit LAN.

The class within Jetty that we instrumented was `BufferCache`, a class that deals with the internal page cache. We chose this class because of its high frequency of getting called. The list of tests came from the `BufferCacheTest` class. The Grinder was configured to use one thread. We made 5,000 requests for specific static and dynamic pages from each Jetty instance. Each Invite client was assigned the full test suite instead of (disjoint) subsets for the purpose of experimentation.

We conducted two sets of experiments. First, to establish the minimal amount of overhead, we ran a single instance of Jetty with the Invite framework disabled via a compile-time flag. Second, to measure the effect community size has on performance, we enabled the Invite framework and ran the experiment on community sizes of one, two, three, and four clients. We were limited to such small community sizes because of a lack of machines with the same specifications (to ensure that differences in measurements were not the result of differences in hardware); however there is nothing inherent in the in vivo testing approach that precludes a heterogeneous application community. Each member of the community always received the same load (i.e. 5,000 requests), independent of community size, while the rate of test execution decreased (approximately, due to randomization) linearly with the number of clients, ensuring that the global number of tests remained constant. The base value of p was arbitrarily set at 1.0%.

4.2. Analysis & discussion

In the context of these experiments we define performance as the mean time that The Grinder receives the first byte of the response from the instrumented Jetty server (hereon denoted $\bar{T}_1$); this definition is appropriate because the Invite client is executed *before* Jetty begins to send its response to The Grinder. In contrast to this definition is the mean time for The Grinder to completely receive the whole response (hereon denoted $\bar{T}_*$). While $\bar{T}_*$might appear to be a better definition of performance than $\bar{T}_1$, this definition is misleading for two reasons: 1) the Invite framework forks the application, so the test (child) process becomes a process just like any other process, subject to the whims of the operating system scheduler; 2) the size of the webpage being requested affects $\bar{T}_*$but not $\bar{T}_1$. Table 1 shows the raw data we collected.

# Clients	p	Avg # Test Per Client	Total # Tests	Mean Time to 1^{st} Byte	% Overhead
N/A	N/A	N/A	N/A	3.05	0.0%
0	0.00%	0	0	3.13	2.4%
1	1.00%	449	449	17.1	460%
2	0.50%	227	454	10.0	227%
3	0.33%	151.67	454	7.34	141%
4	0.25%	112.75	451	6.19	103%

Table 1. Jetty reponses times (in milliseconds) with varying community sizes

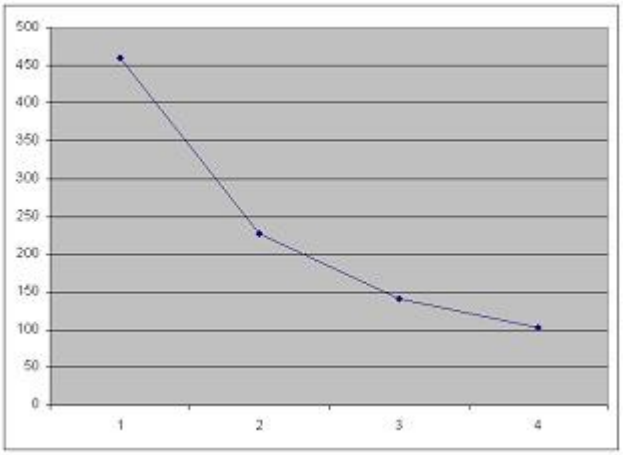

Figure 1. Community Size (X Axis) vs Percent Overhead (Y Axis).

We begin with a comparison of the performance of Jetty without the Invite framework and the performance of Jetty with the Invite framework enabled but with no tests run, i.e. with an empty list of tests. This comparison approximates the overhead of Invite itself at 0.08 ms, a 2.4% increase in overhead. This overhead stems from the Invite framework itself, as no tests were actually executed. Thus, we expect to see this overhead regardless of the size of the community, and we seek to only approach this optimal minimum.

When we increased the community size from zero to one (i.e. having a single client perform all the tests) the overhead of $\bar{T}_1$ jumped to 460%, while a community size of two (i.e. each Invite client performing half as many tests) resulted in an overhead of 227%. We continued this up to four clients, and plotted Figure 1. What is of importance is not the absolute amount of overhead but the fact that it decreases linearly as the size of the community grows. The graph clearly shows this; as we increase the size of the com-

munity, we are able to reduce the p we assign to each Invite client, which results in a lower overhead per Invite client while maintaining the same number of tests being run globally. That is, this lower performance overhead is achieved without sacrificing the original goal of in vivo testing.

If we extrapolate from our data, we predict that a community size of around 18 would reduce overhead to 25%, a size of around 45 to 10%, and a size of around 90 to 5%.

5. Future work & conclusion

Distributed in vivo testing shows great promise in alleviating one of the biggest problems of the in vivo testing approach (extremely high performance penalties) by distributing the number of tests that need to be run onto the entire community of applications. A central server coordinates this effort by monitoring the size of the community and collecting results for later analysis. This distribution not only does not sacrifice testing to reduce the overhead as the quantity of tests being conducted remains the same, but it also has the additional benefit of running the tests in more diverse environments, thereby increasing the chances of finding an anomalous state that causes a test to fail.

Having a community of applications with a central server enables interesting possibilities for more sophisticated testing techniques. One such technique that we hope to explore in a future work is what to do when a test fails. We envision a kind of "differential diagnosis" wherein the Invite server then assigns this test to all the members of the community. Comparing which Invite clients passed with which failed would be of great assistance to the development team in pinpointing the source of the defect.

We see a number of ways to further improve performance: one way would be to modify the Invite server so that it could more intelligently distribute the test suite, either by offloading busy instances onto less busy instances, or by ensuring equality across the community. Similarly, a single instance could adjust its own p value based on its current load. Another way which considers the community at large is to determine the best strategies to dynamically adjust the global workload as the size of the community changes, as in [11]. Specifically, future work would involve experimentation with the frequency of Invite client reconnections to the Invite server.

Another direction involves deciding what tests actually get assigned. The current implementation requires that the developers specify which tests get run, and then the system arbitrarily assigns partitions of the test suite to members of the community. However, the server could more intelligently choose which tests to run and which not to. For example, if a specific test has consistently passed for many cycles on a given Invite client, the Invite server can then remove this test from that client so that it focuses more attention on its other tests. This should result in the discovery of more bugs, enhancing the usefulness of the in vivo testing approach.

6. Acknowledgments

Murphy and Kaiser are members of the Programming Systems Lab, funded in part by NSF CNS-0717544, CNS-0627473, CNS-0426623 and EIA-0202063, NIH 1 U54 CA121852-01A1.

References

[1] Aspectj. http://www.eclipse.org/aspectj/.
[2] The grinder, a java load testing framework. http://grinder.sourceforge.net/.
[3] Jetty, a java web server. http://www.mortbay.org/.
[4] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. *29th ICSE*, pages 261–270, 2007.
[5] S. Elbaum and M. Hardojo. An empirical study of profiling strategies for released software and their impact on testing activities. *ISSTA 2004*, pages 65–75, 2004.
[6] A. Krishna et al. A distributed continuous quality assurance process to manage variability in performance-intensive software. *19th ACM OOPSLA Workshop on Component and Middleware Performance*, 2004.
[7] M. Locasto, S. Sidiroglou, and A. Keromytis. Software self-healing using collaborative application communities. *NDSS 2006*, pages 95–106, Feb 2006.
[8] A. Memon and A. Porter et al. Skoll: distributed continuous quality assurance. *26th ICSE*, pages 459–468, 2004.
[9] C. Murphy, G. Kaiser, and M. Chu. Towards in vivo testing of software applications. Technical Report cucs-037-07, Columbia University, Dept. of Computer Science, 2007.
[10] A. Orso, T. Apiwattanapong, and M. Harrold. Leveraging field data for impact analysis and regression testing. *9th ESEC*, pages 128–137, 2003.
[11] A. Orso, D. Liang, and M. Harrold. Gamma system: Continuous evolution of software after deployment. *ISSTA 2002*, pages 65–69, 2002.
[12] L. Osterweil. Perpetually testing software. *9th ISQW*, 1996.
[13] D. Richardson, L. Clarke, L. Osterweil, and M. Young. Perpetual testing project. http://www.ics.uci.edu/ djr/edcs/PerpTest.html.
[14] D. Rubenstein, L. Osterweil, and S. Zilberstein. An anytime approach to analyzing software systems. *10th FLAIRS*, pages 386–91, May 1997.
[15] D. Saff. Automated continuous testing to speed software development. Master's thesis, MIT, Feb 2004.
[16] D. Saff and M. Ernst. An experimental evaluation of continuous testing during development. *ISSTA 2004*, pages 76–85, 2004.
[17] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing production run failures at the users site. *21st SOSP*, pages 131–144, Oct 2007.
[18] M. Young. Perpetual testing. Technical Report AFRL-IF-RS-TR-2003-32, Feb 2003.