Optimizing unit test execution in large software programs using dependency analysis

Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich
MIT CSAIL

Abstract

TAO is a system that optimizes the execution of unit tests in large software programs and reduces the programmer wait time from minutes to seconds. TAO is based on two key ideas: First, TAO focuses on efficiency, unlike past work that focused on avoiding false negatives. TAO implements simple and fast function-level dependency tracking that identifies tests to run on a code change; any false negatives missed by this dependency tracking are caught by running the entire test suite on a test server once the code change is committed. Second, to make it easy for programmers to adopt TAO, it incorporates the dependency information into the source code repository. This paper describes an early prototype of TAO and demonstrates that TAO can reduce unit test execution time in two large Python software projects by over 96% while incurring few false negatives.

1 Introduction

After fixing a bug or implementing a new feature in a software project, programmers typically run unit tests to check that the individual components of the software continue to function correctly, before committing their code changes to the software's source code repository. Many open source projects, such as the Twisted networking engine [3] and the Django web application framework [1], mandate that all unit tests pass before accepting a programmer's commits into the source code repository [2].

Because programmers run unit tests interactively, a unit test suite is generally expected to run within a few seconds. However, large software projects like Django have unit test suites with several thousand tests and take tens of minutes to complete. This significantly increases the wait time of programmers, making them less productive. It also raises the likelihood that a programmer does not run the unit test suite before committing code changes, increasing the chances of a bug in the committed code.

Therefore, in practice, software projects use several approaches to reduce running time of unit tests. One approach is to offload running of unit tests to a test execution server *after* the programmer commits the changed code to the repository; however, unlike in interactive test execution, it takes longer for the programmer to learn of a test failure and fix the bug, thereby reducing programmer productivity. Another approach is to have the programmer manually select the tests to run; for example, the programmer could run the full test suite once and if a test fails, only rerun that test while fixing the bug that caused it, and finally rerun the full suite before committing to the source repository. This approach, however, burdens the programmer in avoiding unintentional changes tested by other unit tests, thus making them less productive in developing large software projects.

Past work has proposed regression test selection (RTS) techniques [5, 8, 12] to address the limitations of the above approaches. At a high level, RTS's goal is to identify and run the tests whose results may change due to a source code modification. To identify those tests, RTS analyzes a software's source code as well as the past executions of the software's test suite, and computes dependencies of each test on the software's code elements such as functions, classes, or basic blocks. When a programmer modifies the source code, RTS identifies the code elements affected by the modification, and uses its dependency information to flag the tests whose results may change and thus need to be rerun.

Though RTS has the potential to reduce unit test execution time, it is not used in practice, and we are not aware of a real-world unit testing framework that employs RTS. An important reason for this is that existing RTS techniques focus on avoiding false negatives, and so flag *any* test that may be affected by a code change; in practice, this results in too many false positives. For example, if a commit adds a new static variable to a Python class, existing RTS techniques rerun all tests that loaded the class. However, in practice many of these are false positives (see §4 and [7]); typically it is sufficient to run a much smaller set of tests that only executed functions which were modified to access the new variable.

We used this observation to design TAO,[1] a practical system to optimize unit test execution. TAO's key contribution is in making RTS practical by addressing two challenges. The first challenge is that TAO should be efficient—the total time to identify and execute the affected tests with TAO should be a fraction of time needed to run the unit tests without TAO. TAO does this by tracking only function-level dependencies, which is fast and in practice has low false positives. During test execution, TAO tracks which functions were accessed by each test; when the programmer modifies the program's source, TAO analyzes the diff to determine the functions that were modified and executes only the tests that accessed those functions. In return for this efficiency, TAO gives up on false negatives; as function-level dependency tracking is not complete, TAO can miss some tests whose results may change due to the code change (see §2 for examples). In practice, such false negatives are uncommon (§4), and to catch them TAO runs the complete test suite on a test server after every commit; only for these few false negatives does the programmer learn of any failures at a later time.

The second challenge faced by TAO is bootstrapping TAO's dependency analysis and keeping it up-to-date as the source code evolves: TAO loses its usefulness if a programmer has to run all unit tests to generate TAO's dependency information every time she updates from the source repository. To solve this problem, TAO augments a source tree's repository with a *dependency server* that stores the dependency information for each version of the source tree, much like symbol servers that store debug symbols for different versions of software packages. TAO uses dependency information from the dependency server to determine unit tests to run for a code change. Before code is committed to the source code repository, TAO runs the unit tests affected by the code changed by the commit, generates incremental dependency information for the affected unit tests, and updates the dependency server.

We have implemented a prototype of TAO for Python, and integrated it with PyUnit, the standard unit-testing framework for Python. This makes it easy for Python programmers who already use PyUnit to take advantage of TAO. We evaluated TAO with two large Python software projects, Django and Twisted, and demonstrate that for recent source code commits to these projects, TAO runs unit tests in an average of 0.3% and 3.1% of the time required to run the unit tests without TAO, suggesting the TAO's unit test optimization works well in practice.

In the rest of this paper, §2 describes TAO's design, §3 describes TAO's implementation, §4 presents the results of our preliminary evaluation, §5 summarizes related work, and §6 concludes.

[1] TAO stands for Test Analyzer and Optimizer.

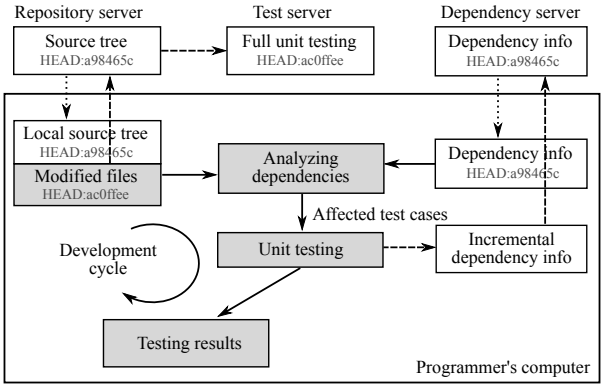


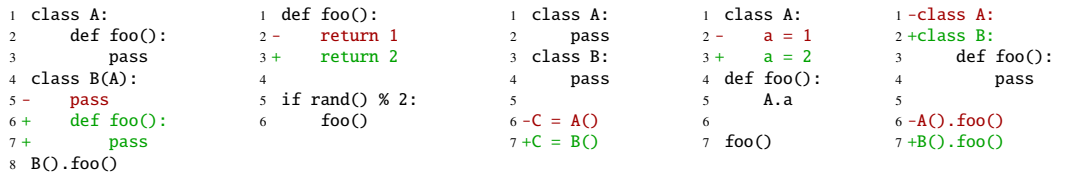
Figure 1: Programmer workflow while using TAO for unit testing. The dotted lines show the update of the local source tree from the repository server, and local dependency information from the dependency server. The shaded boxes illustrate the programmer's development cycle and TAO's optimized execution of unit tests. The dashed lines indicate committing changes in code and dependency information to the servers, and the running of the full unit test suite on the test server after every commit.

2 Design

Figure 1 illustrates the programmer workflow with TAO. When a programmer clones a local source tree from a repository server, the dependency information for unit tests in that version of the source is downloaded by TAO from the dependency server. After modifying the source, when the programmer runs unit tests, TAO analyzes the source modifications to determine the modified functions, and uses the dependency information to select tests to run. Finally, when the programmer commits changed source to the repository server, TAO generates the dependency information for the new version of the source and uploads it to the dependency server. The rest of this section describes TAO's dependency analysis and management in more detail.

Dependency tracking. Dependency tracking in TAO consists of runtime interception and static analysis components. When unit tests are run, TAO intercepts every function executed by each unit test, and constructs a *dependency map* of functions to unit tests that executed the functions. Functions are named by a tuple consisting of the function name, class name, and source file name.

TAO statically analyzes the changed source code to identify modified functions, as follows. TAO first parses the source-level diffs of the changed source to determine the source files and the line numbers in the files that changed. Then, TAO parses the original and modified versions of each source file to construct abstract syntax trees. Finally, it determines the functions in the original and modified versions that contain the changed line numbers—these are the modified functions. These functions include those that were newly added to the code as well those that were removed. TAO looks up the modified functions in



(a) Inter-class dependency (b) Non-determinism (c) Global scope (d) Class variables (e) Lexical dependency

Figure 2: Examples of dependencies that TAO's function-level tracking misses. TAO misses (a) because its runtime interception records that `A`'s `foo` was called; whereas its patch analysis reports that `B.foo` was added, as the patch analysis does not understand that `B` is a subclass of `A`. TAO misses (b) if the result of `rand` was odd during runtime interception. TAO misses (c), (d), and (e) because it tracks only function execution and not statements executed in global or class scope.

the dependency map to determine the affected tests and runs those tests. All new unit tests are run as well.

Dependency server. TAO requires the dependency map to determine unit tests to run on a code change. One option is to generate the dependency map by running all unit tests whenever the programmer updates the local source tree from the repository server; however, this defeats the purpose of TAO's unit test optimization.

To solve this issue, TAO stores the dependency map on a dependency server, with a separate version of the dependency map for each version of the source tree. When the local source tree is updated to a new version, TAO downloads the dependency map for that version and uses it for analysis. Two challenges arise while implementing dependency servers. First, managing a dependency server and its permissions places extra burden on the repository maintainers. To avoid this issue, TAO stores the dependency map as a file in the source code repository itself; thus, programmers who have write access to the repository can also update the dependency map. Second, updating the entire dependency map on every commit wastes network bandwidth, as it needs to be downloaded on an update as well. Instead, TAO generates an incremental dependency map for each commit, which contains only the map for the tests that were affected by the committed code change and were rerun. Periodically, TAO generates a full dependency map on a commit to avoid having to collapse long chains of incremental dependency maps.

Missing dependencies. TAO's function-level dependency tracking is not complete and can miss some dependencies of tests on changed code. Figure 2 shows examples of missing dependencies, which can sometimes result in TAO not rerunning tests whose outputs may have changed due to the changed code (i.e., false negatives); §4 evaluates the frequency of TAO's false negatives.

3 Implementation

We have implemented a prototype of TAO for PyUnit, the standard unit testing framework for Python, and the Git version control system. The implementation is 720 lines of Python code in total, as shown in Figure 3.

Component	Lines of code
Execution tracing	210 lines of Python
Dependency analysis	350 lines of Python
Django test runner	50 lines of Python
Twisted test runner	110 lines of Python

Figure 3: Lines of code of each component of the TAO prototype, including the incremental test runners for Django and Twisted.

TAO intercepts function calls with `settrace`, Python's system tracing function used to implement Python debuggers. The dependency map is stored as a log of functions executed for each unit test case. Incremental maps are stored in the same format as well; merging them is simply a concatenation of the logs, with later entries for a test overriding the previous entries. The dependency maps are compressed with Gzip to reduce storage overhead.

TAO's patch analysis gets a Git diff of the modified source, identifies the modified Python source files from the diff, and then uses Python's standard `ast` module to parse those source files into abstract syntax trees and identify the functions that were modified.

4 Evaluation

Our preliminary evaluation of TAO answers the following questions:

- For real software, how many functions are modified in each commit?
- What fraction of unit tests can be skipped for each commit, and how much execution time is saved as a result?
- How many missed dependencies and false negatives does TAO incur?
- What is the overhead of TAO's dependency tracking?

In our evaluation we used the source code repositories of Django [1] and Twisted [3], and the patches for the most recent 100 commits in those projects. By design, TAO reruns tests affected by a commit to find potential

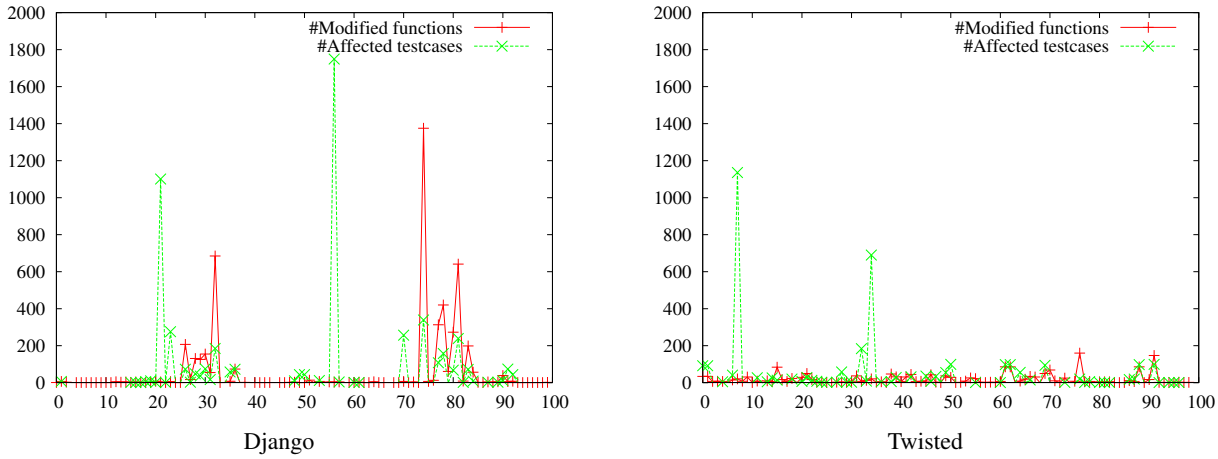


Figure 4: The number of modified functions and affected tests cases (Y-axis) for each of the last 100 commits (X-axis) of Django (left) and Twisted (right), respectively. The total number of functions in Django and Twisted were 13,190 and 23,096, and the total number of test cases were 5,166 and 7,150.

Project	Number of Tests			Runtime	
	All	RTS	TAO	All	TAO
Django	5,166	759.7	50.8	520.3s	1.7s
Twisted	7,150	1,668.3	28.7	72.1s	2.2s

Figure 5: TAO unit test execution performance. The number of tests columns show the total number of tests, and the average numbers of tests executed by an existing RTS technique and TAO for the latest 100 commits in Django and Twisted. The runtime columns show the time to execute all tests and the average time taken by TAO to execute unit tests for the latest 100 commits.

bugs; however, it did not catch any bugs because all unit tests pass for all these commits, likely because developers diligently followed the checkin policies of these projects and ensured that all unit tests passed before committing their code modifications to the repositories. The experiments were run on a computer with a 2.80 GHz Intel Core i7-860 processor and 8 GB of RAM running Ubuntu 12.10.

4.1 Functions modified in each commit

We analyzed the patches for the last 100 commits in Django and Twisted to understand the typical number of functions modified per commit in real-world software. The results are shown in the top graphs of Figure 4. Out of a total number of 13,190 and 23,096 functions in Django and Twisted, respectively, only 50.8 and 18.2 functions (or 0.3% and 0.07%) were modified on average by each commit, and 90% of the commits modified less than 10 functions. As unit tests generally test isolated portions of code, this indicates that the number of affected test cases would be low as well.

4.2 Reduction in test execution time

For the same 100 commits as in the previous experiment, we measured the number of test cases executed by TAO for each commit. The aggregate results are shown in Figure 5. Django and Twisted have 5,166 and 7,150 unit test cases and running all the tests takes 520 seconds (8.7 min) and 72 seconds, respectively. On an average, TAO ran only 50.4 and 28.7 tests per commit (1% and 0.4% of all tests), and took 1.7 seconds and 2.2 seconds to run them (0.3% and 3.1% of the total time). This shows that TAO is very effective in reducing execution time of unit tests. To compare with existing RTS techniques, we also implemented an RTS technique that avoids false negatives by tracking class, module, and global scope, and measured its performance; on an average it executes 759.7 and 1668.3 tests (14.7% and 23.3% of all tests), which is significantly more than TAO.

The top plots of Figure 4 show the distribution of number of tests executed by TAO for each of the 100 commits. The plots show that the number of tests executed is mostly correlated with the number of modified functions (i.e., greater number of modified functions results in more number of tests executed). However, the graphs illustrate exceptions that fall into two interesting categories. First, some commits change only a few functions, but affect many tests; this happens because those commits modify common functions that are invoked in many tests. For example, the `WSGIRequest` class represents a web request in Django and is instantiated in 1,094 out of 5,166 unit tests; a commit that changes the constructor of that class would therefore require rerunning all those tests. Similarly, in Twisted, the `Platform.getType` function that checks the environment affects 1,054 out of 7,150 test cases.

Second, some commits change a large number of functions but affect only a few tests. This happens when commits contain new features, perform code refactoring, or code maintenance. For example, in Django, a commit to support `unittest2` modifies 1,375 functions but affects

Project	FN/I	FN/N	FN/G	FN/C	FN/L
Django	0/0	0/0	2/ 8	1/ 3	1/23
Twisted	1/2	0/0	1/20	1/17	0/11

Figure 6: False negatives and missed dependencies due to TAO for the latest 100 commits of Django and Twisted. Each FN/X (where X is one of I, N, G, C, L, corresponding to the missed dependency types in Figure 2,) represents the number of commits with false negatives (FN) and the number of commits with a missed dependency of type X. A commit can have more than one type of missed dependency and more than one type of false negative. The total number of false negatives was 3 for both projects.

only 341 out of 5,166 test cases. In Twisted, implementing a SSH command end point modifies 160 functions but affects only 20 out of 7,150 test cases.

4.3 False negatives

We manually inspected each of the latest 100 commits for Django and Twisted, and counted the number of commits with code changes that TAO's function-level dependency tracking misses. These missed dependencies fell into the categories in Figure 2. For each of the missed dependencies, we also manually identified the tests that needed to be executed to test that dependency. For some commits, TAO's function-level dependency tracking may have already flagged all the tests corresponding to the missed dependencies because the commit also included function-level changes. In this case TAO does not have false negatives for that commit; Figure 7 illustrates an example scenario for this case. However, if TAO's function-level algorithm missed flagging any of those tests, we counted that commit as having a false negative.

Figure 6 shows the results of this experiment. Although the total number of commits with missed dependencies is about 25%, only 3 commits out of 100 in either project had any false negatives. For the remaining 97 commits, the result of execution of all tests is the same as that of TAO's optimized test execution on the programmers' computers; so, TAO's test execution suffices to catch bugs in 97% of the commits. Combined with the more than 96% reduction in unit test execution time, this shows that TAO in practice strikes a good balance between optimizing test execution time and the number of commits with false negatives.

4.4 Performance overhead

To measure the overhead of TAO's dependency tracking, we ran the unit tests of Django and Twisted, with and without TAO, and measured the running time and the size of the compressed dependency maps. The results are shown in Figure 8. TAO's dependency tracking adds an overhead of 60% and 117% for Twisted and Django, respectively; this high overhead is due to Python's `settrace` debug function being slow. Implementing TAO's dependency tracking in the Python runtime should significantly reduce

```
class DecimalField(IntegerField):
    default_error_messages = {
        ...
-       'max_digits': _(msg)
+       'max_digits': ungettext_lazy(msg)
    ...

    def __init__(...):
        ...
-           raise ValidationError(oldmsg)
+           raise ValidationError(newmsg)
```

Figure 7: Simplified snippet of the Django commit be9ae693 that illustrates why missed dependencies typically do not result in false negatives for TAO. In addition to a missed class variable dependency, the commit also includes a change to the `__init__` function; the tests flagged by this function-level dependency on the `__init__` function are a superset of the tests affected by the change to the class variable.

Project	Runtime		Storage	
	no TAO	TAO	Full	Incremental
Django	520.3s	1,129.1s	9.9MB	270KB
Twisted	72.1s	115.6s	1.3MB	280KB

Figure 8: TAO overhead during execution of Django and Twisted unit tests. The runtime columns show total times taken to run all the unit tests without TAO and with TAO. The storage columns show the size of TAO's compressed full dependency maps and the average size of TAO's compressed incremental dependency maps for each commit.

this overhead. More importantly, the significant reduction in the number of executed tests dwarfs this high overhead, with the bottom line result that the overall execution time of unit tests is still a small fraction of the time taken to execute all the tests.

TAO's dependency information occupies 10MB and 1.3MB for Django and Twisted unit tests, respectively. This corresponds to 2KB and 0.18 KB per test. For the tests that TAO reruns for the recent 100 commits, the average size of incremental dependency maps is less than 300KB per commit, for both Django and Twisted.

5 Related work

The existing work closest to TAO is research on regression test selection [4, 5, 8, 12]. TAO's contributions are in making RTS practical by exploiting the insight that simple and fast function-level dependency tracking works well in practice; and in making it easy for programmers to adopt TAO as part of their workflow by integrating with PyUnit and including dependency information as part of each commit.

Other methods have been proposed to reduce unit test execution time: for example, Khalek et al. [9] show that eliminating common setup phases of unit tests in Java can speed up test execution. However, Khalek et al. require the programmer to define undo methods for all operations in a unit test, which places a significant burden on the programmer that TAO avoids with its automatic dependency tracking.

Past work has also used dependency tracking for applications other than test optimization: Poirot [10] and TaintCheck [11] have used dependency analysis and taint tracking for intrusion recovery, and TaintDroid [6] has used taint tracking for privacy monitoring. TAO shows that dependency tracking can be applied to optimize unit test execution.

6 Summary

TAO helps programmers run unit tests faster by running only the tests that may be affected by the programmers' source code modifications. TAO does this by tracking during test execution the functions executed by each test; analyzing changed code to determine the modified functions; and combining this information to determine the affected tests. To bootstrap dependency information without needing the programmer to run all the unit tests, TAO stores this information along with the source code repository. An initial prototype of TAO reruns an average of less than 1% of the unit tests for each commit in two large Python software projects.

Acknowledgments

Thanks to the anonymous reviewers for their feedback. This work was supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143.

References

[1] Django: The Web framework for perfectionists with deadlines. `http://www.djangoproject.com`, 2013.

[2] Unit tests: Django documentation. `https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/unit-tests`, 2013.

[3] Twisted. `http://twistedmatrix.com`, 2013.

[4] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London. Incremental regression testing. In *Proceedings of the IEEE Conference on Software Maintenance*, September 1993.

[5] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran. Regression test selection techniques: A survey. *Informatica*, 35(3):289–321, October 2011.

[6] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, October 2010.

[7] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, April 2001.

[8] M. Harrold and M. Souffa. An incremental approach to unit testing during maintenance. In *Proceedings of the Conference on Software Maintenance*, pages 362–367, 1988.

[9] S. A. Khalek and S. Khurshid. Efficiently running test suites using abstract undo operations. *IEEE International Symposium on Software Reliability Engineering*, pages 110–119, 2011.

[10] T. Kim, R. Chandra, and N. Zeldovich. Efficient patch-based auditing for web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, October 2012.

[11] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005.

[12] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing Verification and Reliability*, 22(2), March 2012.