Optimizing unit test execution in large software programs using dependency analysis

Taesoo Kim,
Ramesh Chandra and Nickolai Zeldovich

MIT CSAIL
Running unit tests takes too long

It’s our policy to make sure all tests pass at all times.

- Large software programs often require running full unit tests for each commit
- But, unit tests take about 10 min in Django
- With our work, it can be done within 2 sec!
Current approaches for shortening testing time

- **Modular unit tests (e.g., testsuite)**
  - Run a certain set of unit tests that might be affected

- **Test bot (e.g., gtest, autotest)**
  - Run unit tests remotely and get the results back
Problem: current approaches are very limited

- Manual efforts involved
  - Maintaining multiple test suites

- Overall testing still takes too long
  - Waiting for Test bot to complete full unit testing
Research: regression test selection (RTS)

- **Goal:** run **only necessary** tests instead of full tests
  - identify test cases whose results might change due to the current code modification
  - **Step 1:** analyze test cases (e.g., execution traces)
  - **Step 2:** syntactically analyze code changes
  - **Step 3:** output the affected test cases
Problem: RTS techniques are never adopted in practice

- “Soundness” of RTS techniques kills adoption
  - Soundness means no false negatives
  - Impose non-negligible perf. overheads (analysis/runtime)
  - Select lots of test cases (particularly in dynamic languages)
  - e.g., changes in a global variable → run all test cases
Goal: make RTS practical

• Idea 1: trade off soundness for performance
  – Keep track of function-level dependency / changes
  – Fewer tests selected, may have false negatives

• Idea 2: integrate test optimization into dev. cycle
  – Maintain dependency information in code repository
Current development cycle

Repository server

Source tree

<HEAD>

① Check out code

<HEAD>

Local repo.

Programmer's computer
Current development cycle

Repository server

Source tree

<HEAD>

① Check out code

<HEAD>

② Changes

Local repo.

Programmer's computer
Current development cycle

Repository server

Source tree

<HEAD>

① Check out code

<HEAD>

② Changes

Local repo.

Development cycle

③ Unit testing

④ Test results
New development cycle

1. Check out code

2. Changes

Repository server

Source tree <HEAD>

Local repo.

Diff

3. Analyzing dependencies

Development cycle

4. Unit testing

Affected test cases

5. Test results

Programmer's computer

Test case information
New development cycle

Repository server

Source tree
<HEAD>

① Check out code

<HEAD>

② Changes

Local repo.

Diff

③ Analyzing dependencies

Affected test cases

Test case information

Programmer's computer

Development cycle

④ Unit testing

⑤ Test results
Identifying affected test cases by the code modification

• **Plan:** track which tests execute which functions
  
  – **Step 1:** generate function-level dependency info.
    
    • **Map:** invoked functions ↔ test case
    • Construct map by running all unit tests
  
  – **Step 2:** identify modified func., given code changes
  
  – **Step 3:** identify tests that ran the modified func.
Identifying affected test cases by the code modification

- **Plan:** track which tests execute which functions
  - **Step 1:** generate function-level dependency info.
    - **Map:** invoked functions ↔ test case
    - Construct map by running all unit tests
  - **Step 2:** identify modified func., given code changes
  - **Step 3:** identify tests that ran the modified func.
Bootstrapping dependency info.

Repository server

Source tree

<HEAD>

Check out code

<HEAD>

Changes

Diff

Local repo.

Generated by running full unit tests

Analyzing dependencies

Unit testing

Testing results

Development cycle

Programmer's computer

Dep. info
Bootstrapping dependency info.

Repository server

Source tree
<HEAD>

Check out code

<HEAD>

Changes

Local repo.

Diff

Dep. info
<HEAD>

Dependency server

Check out dep. info

Dep. info
<HEAD>

Development cycle

Analyzing dependencies

Testing results

Unit testing

Programmer's computer
Update dependency information

Repository server
- Source tree <HEAD>
- Changes

Dependency server
- Dependency info <HEAD>

Local repo.
- Diff
- Analyzing dependencies
- Unit testing
- Incremental dep. info

Development cycle
- Testing results
- <0xac0ffee>

Programmer's computer
Update dependency information

Repository server
- Source tree `<HEAD>`
- `<HEAD>`
- Changes
  - Local repo.
  - Diff

Dependency server
- Dependency info `<HEAD>`
- `<HEAD>`
- Dependency info
  - Incremental dep. info

Development cycle
- Unit testing
- Testing results

Programmer's computer
Problem: false negatives

• Function-level tracking can **miss some dependencies** and cause **false negatives**
  - Failed to identify some test cases that are actually affected

• Identified **five types** of missing dependencies
  - Inter-class dependency
  - Non-determinism
  - Class variable
  - Global-scope
  - Lexical dependency
Problem: false negatives

• Function-level tracking can miss some dependencies and cause false negatives
  – Failed to identify some test cases that are actually affected

• Identified five types of missing dependencies
  – Inter-class dependency
  – Non-determinism
  – Class variable
  – Global-scope
  – Lexical dependency
Example: inter-class dep. in Python

class A:
    def foo():
        return 1

class B(A):
    pass

def testcase():
    assertEquals(B().foo(), 1)
Example: inter-class dep. in Python

class A:
    def foo():
        return 1

class B(A):
    pass

def testcase():
    assertEquals(B().foo(), 1)

Dependency info:
testcase() →
   B.__init__()
   A.foo()
Example: inter-class dep. in Python

class A:
    def foo():
        return 1
class B(A):
    pass

def testcase():
    assertEqual(B().foo(), 1)

Example: inter-class dep. in Python

Dependency info:

testcase() →
    B.__init__()  
    A.foo()

Modified functions:

B.foo()
Example: missing dep. because of non-determinism in Python

```python
def foo():
    return 1

def testcase():
    if rand()%2:
        assertEqual(foo(), 1)
```

Dependency info:

Modified functions:

- `testcase()`
- `rand()`
- `foo()`

or

- `testcase()`
- `rand()`
Example: missing dep. because of non-determinism in Python

```
def foo():
    return 1
+   return 2

def testcase():
    if rand()%2:
        assertEqual(
            foo(), 1)
```

Dependency info:

```
testcase() →
  rand()
  foo()
```

Modified functions:

```
foo()
```
Example: class-var. dep. in Python

class C:
    - a = 1
    + a = 2

def foo():
    return C.a

def testcase():
    assertEqual(foo(), 1)

Dependency info:

testcase() → foo()

Modified functions:

N/A
Solution: test server runs all tests async.
Test server also verifies dep. info

Repository server
- Source tree
- <HEAD>
- Changes

Test server
- Full unit testing
- <HEAD>
- Analyzing dependencies
- Unit testing
- Testing results
- Incremental dep. info

Dependency server
- Dependency info
- <HEAD>

Development cycle
- Local repo.
- Diff
- Programmer's computer

Verify
TAO: a prototype for PyUnit
Implementation

- **TAO**: a prototype for PyUnit
  - Extending standard `python-unittest` library
  - Patch analysis: using `ast/diff` python module
  - Dependency tracking: using `settrace()` interface
  - 800 Lines of code in Python
Evaluation

• How many functions are modified in each commit in large software programs?
• How much testing time can be saved as result?
• How many false negatives does TAO incur?
• What is the overall runtime overhead of TAO?
Experiment setup

- Two popular projects: Django and Twisted
  - **Django**: a web application framework
  - **Twisted**: a network protocol engine
  - Use existing unit tests of both projects
  - Integrate TAO into both projects
  - Analyze the latest **100 commits** of each project
Small number of functions are modified in each commit

- **Django**: 50.8 / 13k functions (0.3%)
- **Twisted**: 18.2 / 23k functions (0.07%)
Small number of functions are modified in each commit

- Django: 50.8 / 13k functions (0.3%)
- Twisted: 18.2 / 23k functions (0.07%)
Small number of functions are modified in each commit

- **Django**: 50.8 / 13k functions (0.3%)
- **Twisted**: 18.2 / 23k functions (0.07%)
Small number of functions are modified in each commit

- **Django**: 50.8 / 13k functions (0.3%)
- **Twisted**: 18.2 / 23k functions (0.07%)
Small number of test cases need to be rerun

- **Django**: 50.4 / 5k test cases (1.0%)
- **Twisted**: 28.7 / 7k test cases (0.4%)
Small number of test cases need to be rerun

- **Django**: 50.4 / 5k test cases (1.0%)
- **Twisted**: 28.7 / 7k test cases (0.4%)
Trend 1: 

The number of affected test cases is correlated with the number of modified functions.

![Graph showing correlation between affected test cases and modified functions for Django.](image-url)
Trend 2: many modified functions, few affected test cases
Trend 2: many modified functions, few affected test cases

Refactoring (maintenance): e.g., unittest2()
Trend 3: few modified functions, many affected test cases

![Graph showing commit IDs vs modified functions and affected testcases for Django](image-url)
Trend 3: few modified functions, many affected test cases

Changes in “hot” funcs: e.g., WSGIRequest()
TAO can improve the overall execution time for unit testing

<table>
<thead>
<tr>
<th>Project</th>
<th>#Test cases</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All</td>
<td>TAO</td>
</tr>
<tr>
<td>Django</td>
<td>5,166</td>
<td>50.8</td>
</tr>
<tr>
<td>Twisted</td>
<td>7,150</td>
<td>28.7</td>
</tr>
</tbody>
</table>

- Django: 520.3s → 1.7s (5k → 50.8 test cases)
- Twisted: 72.1s → 2.2s (7k → 29.7 test cases)
TAO has few false negatives (FN)

<table>
<thead>
<tr>
<th>Project</th>
<th>FN/I (inter-class)</th>
<th>FN/N (non-det.)</th>
<th>FN/G (global scope)</th>
<th>FN/C (class var.)</th>
<th>FN/L (lexical dep.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Django</td>
<td>0/0</td>
<td>0/0</td>
<td>2/8</td>
<td>1/3</td>
<td>1/23</td>
</tr>
<tr>
<td>Twisted</td>
<td>1/2</td>
<td>0/0</td>
<td>1/20</td>
<td>1/17</td>
<td>0/11</td>
</tr>
</tbody>
</table>

- We **manually identified** types of missing dependencies and false negatives on each commit
- Django: 3 false negatives (one commit is counted in both G/L)
- Twisted: 3 false negatives
TAO has few false negatives (FN)

Among class variable deps we identified, how many false negatives end up getting at?

<table>
<thead>
<tr>
<th>Project</th>
<th>FN/I (inter-class)</th>
<th>FN/N (non-det.)</th>
<th>FN/G (global scope)</th>
<th>FN/C (class var.)</th>
<th>FN/L (lexical dep.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Django</td>
<td>0/0</td>
<td>0/0</td>
<td>2/8</td>
<td>1/3</td>
<td>1/23</td>
</tr>
<tr>
<td>Twisted</td>
<td>1/2</td>
<td>0/0</td>
<td>1/20</td>
<td>1/17</td>
<td>0/11</td>
</tr>
</tbody>
</table>

- We **manually identified** types of missing dependencies and false negatives on each commit
- Django: 3 false negatives (one commit is counted in both G/L)
- Twisted: 3 false negatives
Example: not all missing deps cause false negatives

```python
class DecimalField(IntegerField):
    default_error_messages = {
        'max_digits': ungettext_lazy(msg)
    }

def __init__(...):
    ...
    raise ValidationError(newmsg)
```

Function-level dependency
Dependency tracking imposes performance overheads

<table>
<thead>
<tr>
<th>Project</th>
<th>Runtime</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no TAO</td>
<td>TAO</td>
</tr>
<tr>
<td>Django</td>
<td>520.3s</td>
<td>1,129.1s</td>
</tr>
<tr>
<td>Twisted</td>
<td>72.1s</td>
<td>115.6s</td>
</tr>
</tbody>
</table>

- Django: 10 min (117%) to generate dep. info (9.9MB)
- Twisted: <1 min (60%) to generate dep. info (1.3MB)
- Performance can be improved if we implement function-level tracing natively, instead of using settrace() library.
Incremental dependency information is small

<table>
<thead>
<tr>
<th>Project</th>
<th>Runtime</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no TAO</td>
<td>TAO</td>
</tr>
<tr>
<td>Django</td>
<td>520.3s</td>
<td>1,129.1s</td>
</tr>
<tr>
<td>Twisted</td>
<td>72.1s</td>
<td>115.6s</td>
</tr>
</tbody>
</table>

- Django: 270KB incremental dep. info (per commit)
- Twisted: 280KB incremental dep. info (per commit)
Related work

• Regression test selection:
  – RTS [Biswas '11]: survey of available RTS techniques
    → Simple function-level dependency is effective in practice
    → TAO can be integrated into the programmer's workflow

• Dependency tracking:
  – Poirot [Kim '12]: intrusion recovery
  – TaintDroid [Enck '12]: privacy monitoring
    → Dependency tracking can optimize unit test execution
Summary

TAO: a system that optimizes unit test execution using dependency analysis

- Tracks function-level dependency of each unit test
- Analyzes code changes to find the affected test cases
- Runs only affected test cases (but few false negative)
- Integrated into programmer's development cycle