Unit Test Virtualization with VMVM

Jonathan Bell and Gail Kaiser
Columbia University
Good news: We have tests!

No judgement on whether they are complete or not, but we sure have a lot of them

<table>
<thead>
<tr>
<th></th>
<th>Number of tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Tomcat</td>
<td>1,734</td>
</tr>
<tr>
<td>Closure Compiler</td>
<td>7,949</td>
</tr>
<tr>
<td>Commons I/O</td>
<td>1,022</td>
</tr>
</tbody>
</table>
Bad news: We have to run a lot of tests!
Bad news: We have to run a lot of tests!

• Much work has focused on improving the situation:

• Test Suite Prioritization

  • E.g. Wong [ISSRE ’97], Rothermel [ICSM ’99]; Elbaum [ICSE ’01]; Srivastava [ISSTA ’02] and more
Bad news: We have to run a lot of tests!

- Much work has focused on improving the situation:
  - Test Suite Prioritization
    - E.g. Wong [ISSRE ’97], Rothermel [ICSM ’99]; Elbaum [ICSE ’01]; Srivastava [ISSTA ’02] and more
  - Test Suite Minimization
    - E.g. Harrold [TOSEM ’93]; Wong [ICSE ’95]; Chen [IST ’98]; Jones [TOSEM ’03]; Tallam [PASTE ’05]; Jeffrey [TSE ’07]; Orso [ICSE ’09] Hao [ICSE ’12] and more
Testing still takes too long.
Our Approach:
Unit Test Virtualization
Our Approach:
Unit Test Virtualization

Reduces test execution time by up to 97%, on average 62%
Our Approach:
Unit Test Virtualization

Reduces test execution time by up to 97%, on average 62%

Apache Tomcat: From 26 minutes to 18 minutes
Our Approach:
Unit Test Virtualization

Reduces test execution time by up to 97%, on average 62%

Apache Tomcat: From 26 minutes to 18 minutes

Integrates with JUnit, ant, and mvn on unmodified JVMs.
Our Approach:
Unit Test Virtualization

Reduces test execution time by up to 97%, on average 62%

Apache Tomcat: From 26 minutes to 18 minutes

Integrates with JUnit, ant, and mvn on unmodified JVMs.

Available on GitHub
JUnit Test Execution

Start Test Suite → Begin Test → Execute Test → Start JVM

Terminate App
JUnit Test Execution

Start Test Suite → Begin Test → Start JVM → Execute Test → Terminate App
JUnit Test Execution

Start Test Suite → Begin Test → Start JVM → Execute Test → Terminate App

1.4 sec (combined) For EVERY test!
JUnit Test Execution

Overhead of restarting the JVM?

Start Test Suite → Begin Test → Execute Test → Terminate App

1.4 sec (combined) For EVERY test!
JUnit Test Execution

Overhead of restarting the JVM?

Unit tests as fast as 3-5 ms

1.4 sec (combined) For EVERY test!
JUnit Test Execution

Overhead of restarting the JVM?

Unit tests as fast as 3-5 ms

JVM startup time is fairly constant (1.4 sec)

1.4 sec (combined)

For EVERY test!
JUnit Test Execution

Overhead of restarting the JVM?

Unit tests as fast as 3-5 ms

JVM startup time is fairly constant (1.4 sec)

Up to 4,153%, avg 618%

*From our study of 20 popular FOSS apps*
Do applications really use a new JVM for each test?

- Checked out the 1,000 largest Java projects from Ohloh
- 81% of projects with more than 1,000 tests do it
- 71% of projects with more than 1 million LOC do it
- Overall: 41% of all of the projects do
Test Independence

• We typically assume that tests are order-independent
Test Independence

• We typically assume that tests are order-independent

• Might rely on developers to completely reset the system under test between tests

  • Who tests the tests?
Test Independence

• We typically assume that tests are *order-independent*

• Might rely on developers to completely reset the system under test between tests

  • Who tests the tests?

• Dangerous: If wrong, can have false positives or false negatives (Muşlu [FSE ’11], Zhang [ISSTA ’14])
Test Independence

/** If true, cookie values are allowed to contain an equals character without being quoted. */
public static boolean ALLOW_EQUALS_IN_VALUE =
    Boolean.valueOf(System.getProperty("org.apache.tomcat.util.http.ServerCookie.ALLOW_EQUALS_IN_VALUE","false"))
    .booleanValue();
Test Independence

This field is set once, when the class that owns it is initialized:

```java
/** If true, cookie values are allowed to contain an equals character without being quoted. */
public static boolean ALLOW_EQUALS_IN_VALUE = Boolean.valueOf(System.getProperty("org.apache.tomcat.util.http.ServerCookie.ALLOW_EQUALS_IN_VALUE","false")) .booleanValue();
```
Test Independence

This field is set once, when the class that owns it is initialized.

```java
/** If true, cookie values are allowed to contain an equals character without being quoted. */
public static boolean ALLOW_EQUALS_IN_VALUE =
    Boolean.valueOf(System.getProperty(
        "org.apache.tomcat.util.http.ServerCookie.ALLOW_EQUALS_IN_VALUE",
        "false"))
    .booleanValue();
```

This field’s value is dependent on an external property.
A Tale of Two Tests

```java
public static boolean ALLOW_EQUALS_IN_VALUE = Boolean.valueOf(
    ALLOW_EQUALS_IN_VALUE","false"));
```

TestAllowEqualsInValue  TestDontAllowEqualsInValue
A Tale of Two Tests

public static boolean ALLOW_EQUALS_IN_VALUE = Boolean.valueOf(
    ALLOW_EQUALS_IN_VALUE","false"))booleanValue();

TestAllowEqualsInValue
TestDontAllowEqualsInValue

Sets environmental variable to true
Start Tomcat, run test
A Tale of Two Tests

```java
public static boolean ALLOW_EQUALS_IN_VALUE = Boolean.valueOf(
    ALLOW_EQUALS_IN_VALUE","false")) booleanValue();
```

**TestAllowEqualsInValue**
Sets environmental variable to **true**
Start Tomcat, run test

**TestDontAllowEqualsInValue**
Sets environmental variable to **false**
Start Tomcat, run test
A Tale of Two Tests

```
public static boolean ALLOW_EQUALS_IN_VALUE = Boolean.valueOf(
    ALLOW_EQUALS_IN_VALUE","false"));
```

**TestAllowEqualsInValue**
Sets environmental variable to `true`
Start Tomcat, run test

**TestDontAllowEqualsInValue**
Sets environmental variable to `false`
Start Tomcat, run test

But our static field is stuck!
A Tale of Two Tests

```java
public static boolean ALLOW_EQUALS_IN_VALUE = Boolean.valueOf(System.getProperty("org.apache.tomcat.util.http.ServerCookie.ALLOW_EQUALS_IN_VALUE","false")).booleanValue();
```

**TestAllowEqualsInValue**
Sets environmental variable to **true**
Start Tomcat, run test

**TestDontAllowEqualsInValue**
Sets environmental variable to **false**
Start Tomcat, run test
Our Approach

Unit Test Virtualization: Allow tests to leave side-effects. *But* efficiently contain them.
How do Tests Leak Data?

Java is **memory-managed**, and **object oriented**

We think in terms of **object graphs**
How do Tests Leak Data?

Java is **memory-managed**, and **object oriented**

We think in terms of **object graphs**

Test Runner
Instance
How do Tests Leak Data?

Java is **memory-managed**, and **object oriented**

We think in terms of **object graphs**

Diagram:

```
Test Case 1
  references

Accessible Objects
```

```
Test Runner Instance
```

```
references
```
How do Tests Leak Data?

Java is **memory-managed**, and **object oriented**

We think in terms of **object graphs**

![Diagram](image-url)
How do Tests Leak Data?

Java is **memory-managed**, and **object oriented**

We think in terms of **object graphs**

```
<table>
<thead>
<tr>
<th>Test Runner Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>references</td>
</tr>
<tr>
<td>references</td>
</tr>
<tr>
<td>references</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Case 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>references</td>
</tr>
<tr>
<td>Accessible Objects</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Case 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>references</td>
</tr>
<tr>
<td>Accessible Objects</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Case n</th>
</tr>
</thead>
<tbody>
<tr>
<td>references</td>
</tr>
<tr>
<td>Accessible Objects</td>
</tr>
</tbody>
</table>
```
How do Tests Leak Data?

Java is **memory-managed**, and **object oriented**

We think in terms of **object graphs**

- Test Runner Instance
  - Test Case 1
    - Accessible Objects
  - Test Case 2
    - Accessible Objects
  - Test Case n
    - Accessible Objects

No cross-talk
How do Tests Leak Data?

Java is **memory-managed**, and **object oriented**

We think in terms of **object graphs**

Class A
- Static Fields

Class B
- Static Fields

Static fields: owned by a class, NOT by an instance
How do Tests Leak Data?

Java is **memory-managed**, and **object oriented**

We think in terms of **object graphs**

Static fields: owned by a class, NOT by an instance

These are leakage points
Isolating Side Effects

Class A
Static Fields

Test 1

Class B
Static Fields

Test 2

Class C
Static Fields
Isolating Side Effects

Class A
- Static Fields
  - Writes
  - Test 1

Class B
- Static Fields
  - Test 2

Class C
- Static Fields
Isolating Side Effects

Class A
Static Fields

Test 1

Class B
Static Fields

Test 2

Class C
Static Fields
Isolating Side Effects

Test 1

Class A
Static Fields

Write
Reads

Test 2

Class B
Static Fields

Write

Class C
Static Fields

Write
Isolating Side Effects

Class A

Static Fields

Test 1

Writes

Reads

Class B

Static Fields

Test 2

Reads

Writes

Class C

Static Fields

Test 1

Class A

Static Fields

Writes

Test 2

Reads

Writes

Class C

Static Fields
Isolating Side Effects

Class A
- Static Fields

Test 1
- Writes
- Reads

Class B
- Static Fields

Test 2
- *Interception*
- Reads
- Writes

Class C
- Static Fields
Isolating Side Effects

Class A
Static Fields
Reads
Writes
Test 1

Class B
Static Fields
Reads
Writes
Test 2

Class C
Static Fields

*Interception*
Isolating Side Effects

Class A
Static Fields

Class B
Static Fields

Class C
Static Fields

Test 1
*Interception*
Reads
Writes

Test 2
Reads
Writes
Isolating Side Effects

Class A
Static Fields

Class B
Static Fields

Class C
Static Fields

Test 1
Writes
Reads

Test 2
Reads
Writes

*Interception*

These classes had no possible conflicts
So, don’t touch them!
Isolating Side Effects

Key Insight:
No need to re-initialize the entire application in order to isolate tests

So, don’t touch them!

These classes had no possible conflicts
VMVM: Unit Test Virtualization

• Isolates in-memory side effects, just like restarting JVM
VMVM: Unit Test Virtualization

• Isolates in-memory side effects, just like restarting JVM

• Integrates easily with ant, maven, junit
VMVM: Unit Test Virtualization

- Isolates in-memory side effects, just like restarting JVM
- Integrates easily with ant, maven, junit
- Implemented completely with application byte code instrumentation
VMVM: Unit Test Virtualization

- Isolates in-memory side effects, just like restarting JVM
- Integrates easily with ant, maven, junit
- Implemented completely with application byte code instrumentation
- No changes to JVM, no access to source code required
Efficient Reinitialization

Emulate exactly what happens when a class is initialized the first time.
Efficient Reinitialization

Emulate *exactly* what happens when a class is initialized the first time

First new instance or static reference of $T$
Efficient Reinitialization

Emulate *exactly* what happens when a class is initialized the first time

First new instance or static reference of $T$

Acquire lock on $T$
Efficient Reinitialization

Emulate exactly what happens when a class is initialized the first time

First new instance or static reference of $T$

Acquire lock on $T$

Check initialization status
Efficient Reinitialization

Emulate *exactly* what happens when a class is initialized the first time

- First new instance or static reference of $T$
- Acquire lock on $T$
- Check initialization status
- Release lock on $T$
Efficient Reinitialization

Emulate exactly what happens when a class is initialized the first time

First new instance or static reference of $T$

Acquire lock on $T$

Check initialization status

Not initialized

Release lock on $T$

Initialize $T$'s super classes
Efficient Reinitialization

Emulate *exactly* what happens when a class is initialized the first time

1. First new instance or static reference of $T$
2. Acquire lock on $T$
3. Check initialization status

- Not initialized

4. Release lock on $T$
5. Initialize $T$'s super classes
6. Run initializer for $T$
Efficient Reinitialization

Emulate *exactly* what happens when a class is initialized the first time

1. First new instance or static reference of $T$
2. Acquire lock on $T$
3. Check initialization status
4. Not initialized
5. Acquire lock on $T$
6. Release lock on $T$
7. Initialize $T$'s super classes
8. Run initializer for $T$
Efficient Reinitialization

Emulate *exactly* what happens when a class is initialized the first time

- First new instance or static reference of $T$
  - Acquire lock on $T$
  - Check initialization status

- Not initialized

  - Release lock on $T$
  - Initialize $T$'s super classes
  - Run initializer for $T$

- Mark init done

- Acquire lock on $T$
Efficient Reinitialization

Emulate *exactly* what happens when a class is initialized the first time

- First new instance or static reference of $T$
  - Acquire lock on $T$
  - Check initialization status

- Release lock on $T$
  - Initialize $T$'s super classes
  - Run initializer for $T$

- Acquire lock on $T$
  - Mark init done
  - Release lock on $T$
Efficient Reinitialization

Emulate *exactly* what happens when a class is initialized the first time

First new instance or static reference of $T$ *per test*

Acquire lock on $T$

Check initialization status

Not initialized

Release lock on $T$

Initialize $T$'s super classes

Run initializer for $T$

Acquire lock on $T$

Mark init done

Release lock on $T$
Efficient Reinitialization

Emulate *exactly* what happens when a class is initialized the first time

1. Acquire lock on $T$
2. Check initialization status
3. Re-initialize $T$'s super classes
4. Run initializer for $T$
5. Mark init done
6. Release lock on $T$
7. Release lock on $T$
Efficient Reinitialization

Emulate exactly what happens when a class is initialized the first time

First new instance or static reference of $T$ per test

- Acquire lock on $T$
- Check initialization status

If not initialized:

- Re-initialize $T$'s super classes
- Re-initialize $T$

Release lock on $T$

Acquire lock on $T$

Mark init done

Release lock on $T$
Efficient Reinitialization

• Does not require any modifications to the JVM and runs on commodity JVMs
Efficient Reinitialization

• Does not require any modifications to the JVM and runs on commodity JVMs

• The JVM calls a special method, <clinit> to initialize a class
Efficient Reinitialization

• Does not require any modifications to the JVM and runs on commodity JVMs

• The JVM calls a special method, <clinit> to initialize a class

• We do the same, entirely in Java
Efficient Reinitialization

• Does not require any modifications to the JVM and runs on commodity JVMs

• The JVM calls a special method, <clinit> to initialize a class

• We do the same, entirely in Java

• Add guards to trigger this process
Efficient Reinitialization

- Does not require any modifications to the JVM and runs on commodity JVMs

- The JVM calls a special method, `<clinit>` to initialize a class

- We do the same, entirely in Java

- Add guards to trigger this process

- Register a hook with test runner to tell us when a new test starts
Experiments

• RQ1: How does VMVM compare to Test Suite Minimization?
Experiments

- RQ1: How does VMVM compare to Test Suite Minimization?
- RQ2: What are the performance gains of VMVM?
Experiments

- RQ1: How does VMVM compare to Test Suite Minimization?
- RQ2: What are the performance gains of VMVM?
- RQ3: Does VMVM impact fault finding ability?
RQ1: VMVM vs Test Minimization

- Study design follows Zhang [ISSRE ‘11]’s evaluation of four minimization approaches
RQ1: VMVM vs Test Minimization

- Study design follows Zhang [ISSRE ‘11]’s evaluation of four minimization approaches
- Compare to the minimization technique with least impact on fault finding ability, Harrold [TOSEM ‘93]'s technique
RQ1: VMVM vs Test Minimization

• Study design follows Zhang [ISSRE ‘11]'s evaluation of four minimization approaches

• Compare to the minimization technique with least impact on fault finding ability, Harrold [TOSEM ‘93]'s technique

• Study performed on the popular Software Infrastructure Repository dataset
RQ1: VMVM vs Test Minimization

Reduction in Testing Time

Test Suite Minimization  VMVM  Combined

Application

ICSE 2014  @_jon_bell_  June 5, 2014
RQ1: VMVM vs Test Minimization

Reduction in Testing Time

- Test Suite Minimization
- VMVM
- Combined

Larger is better

13%

Application

Ant v1, Ant v2, Ant v3, Ant v4, Ant v5, Ant v6, Ant v7, Ant v8, JMeter v1, JMeter v2, JMeter v3, JMeter v4, jtopas v1, jtopas v2, jtopas v3, xml-sec v1, xml-sec v2, xml-sec v3

ICSE 2014

@_jon_bell_

June 5, 2014
RQ1: VMVM vs Test Minimization

Reduction in Testing Time

Application

Ant v1  Ant v2  Ant v3  Ant v4  Ant v5  Ant v6  Ant v7  JMeter v1  JMeter v2  JMeter v3  JMeter v4  Jtopas v1  Jtopas v2  Jtopas v3  xml-sec v1  xml-sec v2  xml-sec v3

VMVM  Combined

Test Suite Minimization

Larger is better

ICSE 2014

@_jon_bell_

June 5, 2014
RQ1: VMVM vs Test Minimization

Reduction in Testing Time

<table>
<thead>
<tr>
<th>Application</th>
<th>Test Suite Minimization</th>
<th>VMVM</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant v1</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>Ant v2</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>Ant v3</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>Ant v4</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>Ant v5</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>Ant v6</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>Ant v7</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>Ant v8</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>JMeter v1</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>JMeter v2</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>JMeter v3</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>JMeter v4</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>JMeter v5</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>Jtopas v1</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>Jtopas v2</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>Jtopas v3</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>xml-sec v1</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>xml-sec v2</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
<tr>
<td>xml-sec v3</td>
<td>13%</td>
<td>46%</td>
<td>49%</td>
</tr>
</tbody>
</table>

Larger is better
RQ2: Broader Evaluation

- Previous study: well-studied suite of 4 projects, which average 37,000 LoC and 51 test classes
RQ2: Broader Evaluation

• Previous study: well-studied suite of 4 projects, which average 37,000 LoC and 51 test classes

• This study: manually collected repository of 20 projects, average 475,000 LoC and 56 test classes
RQ2: Broader Evaluation

- Previous study: well-studied suite of 4 projects, which average 37,000 LoC and 51 test classes

- This study: manually collected repository of 20 projects, average 475,000 LoC and 56 test classes

  - Range from 5,000 LoC - 5,692,450 LoC; 3 - 292 test classes; 3.5-15 years in age
RQ2: Broader Evaluation

[Bar chart showing relative speedup for various projects, larger is better]

- Bristlecone
- Apache Nutch
- Apache Tomcat
- Jetty
- Commons IO
- Apache River
- btrace
- gedcom4j
- mkgmap
- Apache Ivy
- betterFORM
- Closure Compiler
- Commons Codec
- Commons Validator
- JAXX
- FreeRapid Downloader
- Trove for Java
- Openfire
- JTor
- upm

Relative Speedup
Larger is better
RQ2: Broader Evaluation

![Bar chart showing relative speedup with larger being better. The average speedup is 62%.](image)

**Average: 62%**

- Bristlecone
- Apache Nutch
- Apache Tomcat
- Jetty
- Commons IO
- Apache River
- btrace
- gedcom4j
- mkgmap
- Apache Ivy
- betterFORM
- Closure Compiler
- Commons Codec
- Commons Validator
- JAXX
- FreeRapid Downloader
- Trove for Java
- Openfire
- JTor
- upm

ICSE 2014  @ _jon_bell_  June 5, 2014
RQ2: Broader Evaluation

![Bar chart showing relative speedup of various projects. The x-axis represents percentage speedup from 0% to 100%, and the y-axis lists project names. The chart indicates that the average speedup is 62%, with a maximum of 97%. Larger is better indicates that a higher percentage indicates better performance.](chart.png)

**Projects**:
- Bristlecone
- Apache Nutch
- Apache Tomcat
- Jetty
- Commons IO
- Apache River
- btrace
- gedcom4j
- mkgmap
- Apache Ivy
- betterFORM
- Closure Compiler
- Commons Codec
- Commons Validator
- JAXX
- FreeRapid Downloader
- Trove for Java
- Openfire
- JTor
- upm

**Performance Metrics**:
- **Average**: 62%
- **Max**: 97%
Factors that impact reduction

- Looked for relationships between number of tests, lines of code, age of project, total testing time, time per test, and VMVM’s speedup
Factors that impact reduction

• Looked for relationships between number of tests, lines of code, age of project, total testing time, time per test, and VMVM’s speedup

• Result: Only average time per test is correlated with VMVM’s speedup (in fact, quite strongly; $p < 0.0001$)
RQ3: Impact on Fault Finding

- No impact on fault finding from seeded faults (SIR)
RQ3: Impact on Fault Finding

- No impact on fault finding from seeded faults (SIR)
- Does VMVM correctly isolate tests though?
RQ3: Impact on Fault Finding

- No impact on fault finding from seeded faults (SIR)
- Does VMVM correctly isolate tests though?
- Compared false positives and negatives between unisolated execution, traditionally isolated execution, and VMVM-isolated execution for these 20 complex applications
RQ3: Impact on Fault Finding

- **No impact on fault finding** from seeded faults (SIR)
- Does VMVM correctly isolate tests though?
- Compared false positives and negatives between un-isolated execution, traditionally isolated execution, and VMVM-isolated execution for these 20 complex applications
- **Result:** False positives occur when not isolated. VMVM shows no false positives or false negatives.
Conclusions

- Most large applications isolate their test cases
Conclusions

• Most large applications isolate their test cases

• VMVM provides up to a 97% reduction in testing time through more efficient isolation (average 62%)
Conclusions

• Most large applications isolate their test cases

• VMVM provides up to a 97% reduction in testing time through more efficient isolation (average 62%)

• VMVM does not risk a reduction in fault finding
Unit Test Virtualization with VMVM

Jonathan Bell and Gail Kaiser
Columbia University

https://github.com/Programming-Systems-Lab/vmvm

See a demo of VMVM at 2:30 today! Room MR G1-3