Unit Test Virtualization: Optimizing Testing Time

Jonathan Bell and Gail Kaiser
Computer Science Department, Columbia University

The Problem: Isolating Test Cases

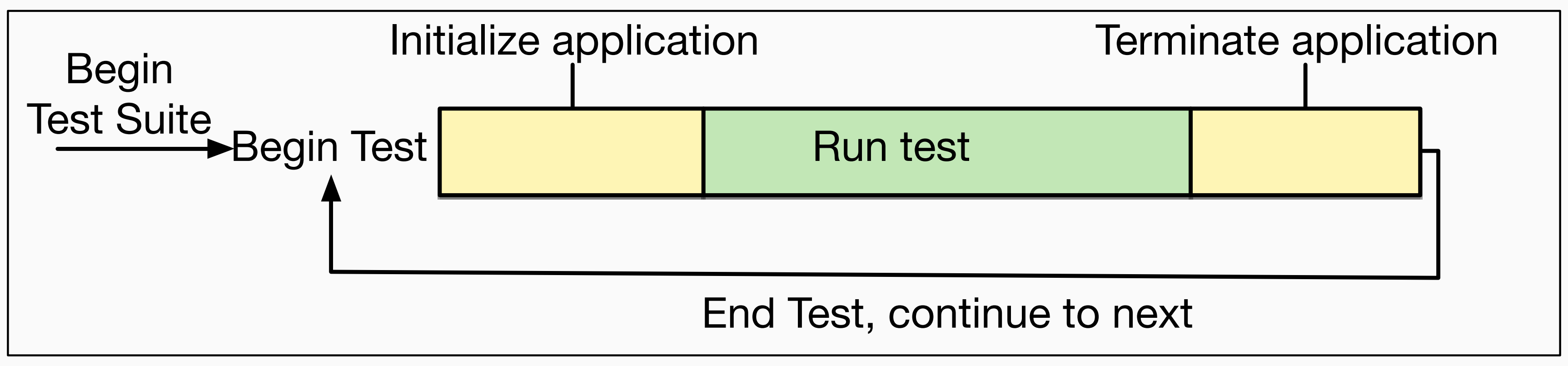
Developers can accidentally create code that makes testing difficult

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

Code sample from Apache Tomcat that demonstrates the sort of code that can create unexpected test case dependencies: ALLOW_EQUALS_IN_VALUE can be set only once: on subsequence executions within the same process, its value will not change, even if the system property does. This sort of dependency is non-trivial to detect (in fact, NP-complete).

Standard fix: Execute every test case in its own process

By executing every test in its own process, such side-effects can be ignored, as they are only persisted as part of the in-memory application state. Restarting the application clears this state.



This fix is very commonly used in large Java projects, and is very slow

We mined the top 1,000 Java projects on ohloh. We looked at those using ant or maven to run automated tests to see how many isolate test cases in separate processes. For 20 of these, we calculated the overhead of isolating each test (shown in bottom table).

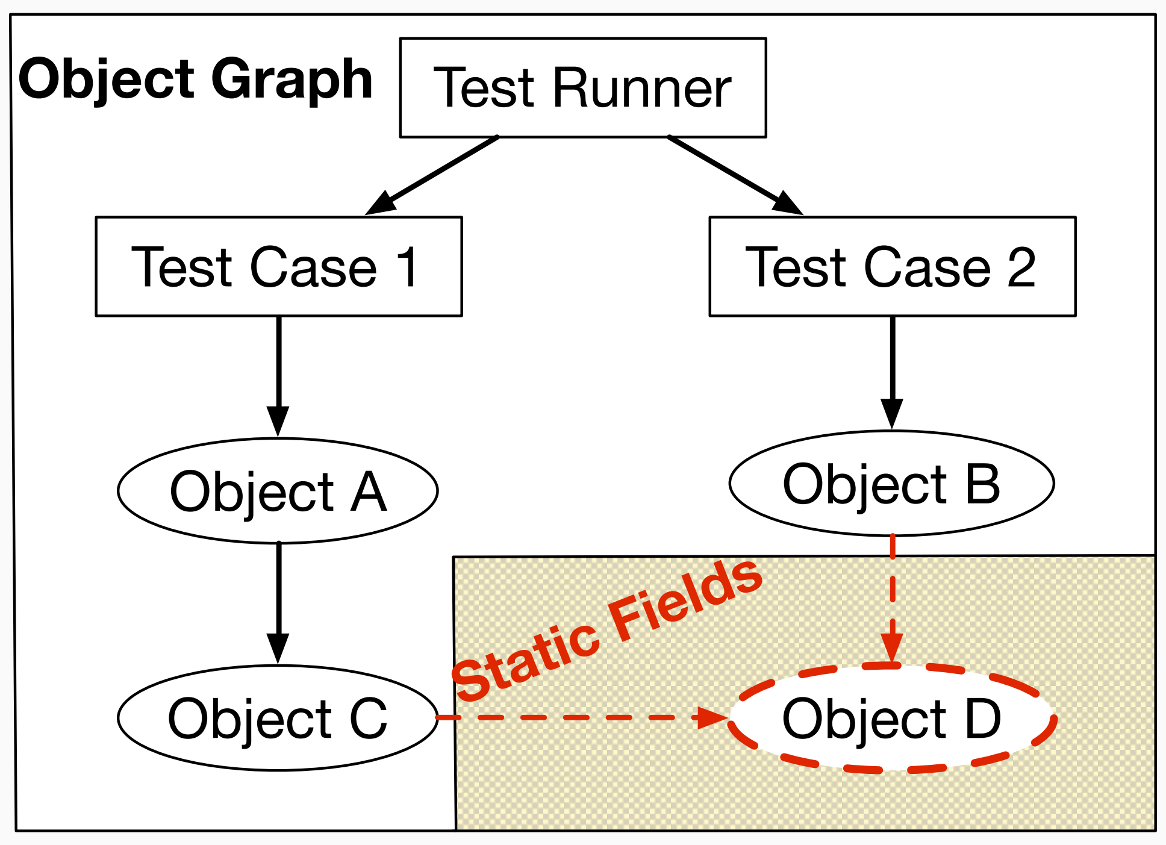
Number of tests in project	Number of projects creating a new process for each test	
0-10	24/71	(34%)
10-100	81/235	(34%)
100-1000	97/238	(41%)
>1000	38/47	(81%)
(Overall)	240/591	(41%)

Number of lines of code in project	Number of projects creating a new process for each test	
0-10k	7/42	(17%)
10k-100k	60/200	(30%)
100k-1m	114/267	(43%)
> 1m	58/82	(71%)
(Overall)	240/591	(41%)

Our solution: VMVM's Unit Test Virtualization

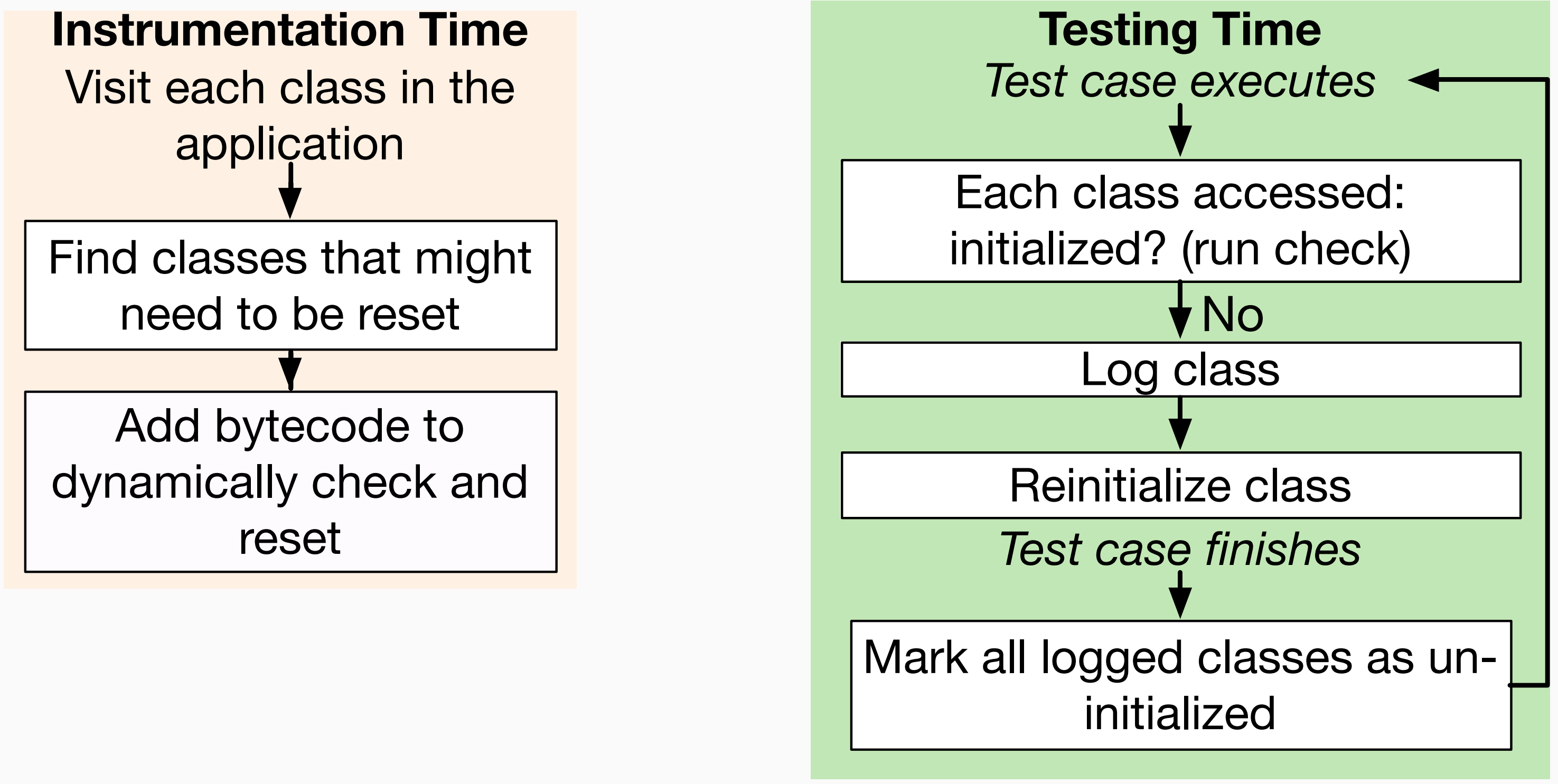
Efficiently reset Java applications to their starting state

Assuming that classes are not reused between test executions (by the test runner), only possible leakage is through static fields. The graph below shows how such a leakage could occur.



VMVM uses a hybrid static-dynamic analysis

VMVM efficiently resets these static fields on-demand using a two-phase static/dynamic byte code analysis. Statically, VMVM identifies classes that may possibly need to be reset and inserts guards. At runtime, these guards are checked.


Instrumentation Time
Visit each class in the application
Find classes that might need to be reset
Add bytecode to dynamically check and reset
Testing Time
Test case executes
Each class accessed: initialized? (run check)
No
Log class
Reinitialize class
Test case finishes
Mark all logged classes as un-initialized

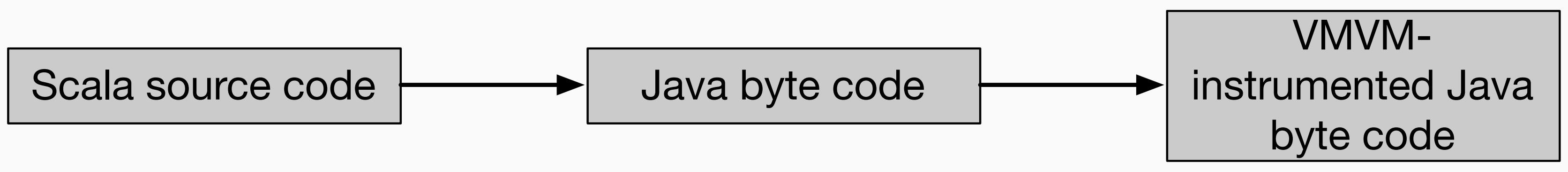
VMVM is much faster than running each test in its own process

We compared VMVM's overhead to that of traditional, process-based isolation, finding it significantly reduced test execution time.

Project	LOC	Test Classes	Isolation Overhead	VMVM Overhead
Apache Ivy	**305,991**	**119**	**342.47%**	**48.06%**
Apache Nutch	**100,911**	**27**	**17.50%**	**1.01%**
Apache River	**365,722**	**22**	**101.98%**	**1.42%**
Apache Tomcat	**5,692,447**	**292**	**42.01%**	**2.46%**
betterFORM	1,114,142	127	377.20%	40.19%
Bristlecone	16,516	4	3.02%	5.59%
btrace	**14,145**	**3**	**123.11%**	**2.76%**
Closure Compiler	467,571	223	887.61%	174.36%
Commons Codec	17,992	46	407.40%	34.19%
Commons IO	**29,163**	**84**	**88.78%**	**0.91%**
Commons Validator	17,456	21	914.32%	81.35%
FreeRapid Downloader	257,697	7	630.62%	8.36%
gedcom4j	18,224	57	463.93%	140.84%
JAXX	91,127	6	832.05%	42.31%
Jetty	**621,532**	**6**	**49.57%**	**3.19%**
JTor	15,073	7	1,133.18%	17.52%
mkgmap	58,541	43	231.18%	26.09%
Openfire	**250,792**	**12**	**762.08%**	**14.42%**
Trove for Java	**45,305**	**12**	**800.67%**	**27.00%**
upm	**5,617**	**10**	**4,152.66%**	**15.63%**
Average	475,298	56.4	618.07%	34.38%

Applications to non-Java languages

VMVM targets JVM, but is tightly integrated with JUnit. We are currently integrating VMVM with the Scala compiler's *partest.* The Scala compiler test suite contains over 3,500 test cases, each executed in their own process.



Additional challenges:
- Dependence on custom system class loaders
- Dependence on custom JVM launch options

VMVM is on GitHub: http://github.com/Programming-Systems-Lab/vmvm