Testing in Resource Constrained Execution Environments

Gregory M. Kapfhammer
Department of Computer Science
Allegheny College

Mary Lou Soffa
Department of Computer Science
University of Virginia

Daniel Mossé
Department of Computer Science
University of Pittsburgh
Contributions

- Use of native code unloading during test suite execution in a resource constrained environment
- Identification of the testing techniques that yield the greatest reduction in execution time and native code size
- Characterization of how software applications and test suites restrict and/or support resource constrained testing
- Cost-benefit analysis for the use of sample-based and exhaustive profiles of program testing behavior
- **Executes test suites faster when memory resources are limited!**
During testing the JVM must manage limited resources.
Resource Constrained Testing

Memory Resident Native Code Bodies

Program $P$

$m_s$ \ldots \hspace{1cm} m_t$

- inv ct: 8
- exec time: 1%
- size: 100 KB
- inv ct: 1200
- exec time: 15%
- size: 64 KB

Test Executor

$TE_u$ \ldots \hspace{1cm} TE_v$

- inv ct: 50
- exec time: 22%
- size: 75 KB
- inv ct: 15
- exec time: 2%
- size: 50 KB

Test Suite $T$

$T_1$ \ldots \hspace{1cm} T_n$

- inv ct: 2
- exec time: 2%
- size: 64 KB
- inv ct: 1
- exec time: 1%
- size: 50 KB

All Tests size: 128 KB

- JIT compiler produces native code that exhausts limited memory resources
- Frequent invocation of GC increases testing time

Test Suite Execution Strategies

- **Omit tests?** - Could reduce overall confidence in the correctness of $P$
- **Use non-constrained environment?** - Defects related to $P$’s interaction with environment might not be isolated
- **Execute tests individually?** - Might increase overall testing time and violate test order dependencies
- **Unload with offline profile?** - Not useful if $P$ and $T$ change frequently during regression testing
- **Our Approach**: Use online behavior profiles to guide the unloading of native code
Experiment Goals and Design

Research Question: Can an adaptive code unloading JVM reduce time and space overheads associated with memory constrained testing?

Experiment Metrics: percent reduction in time, $T_R^\% (P, T)$ and space, $S_R^\% (P, T)$

- Jikes RVM 2.2.1, JUnit 3.8.1, GNU/Linux 2.4.18
- Sample-based ($S$) and exhaustive ($X$) program profiles
- Timer ($TM$), garbage collection ($GC$), and code cache size ($CS$) triggers the unloading technique
### Case Study Applications

<table>
<thead>
<tr>
<th>Name</th>
<th>\textit{Min} Size (MB)</th>
<th># Tests</th>
<th>NCSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>UniqueBoundedStack (UBS)</td>
<td>8</td>
<td>24</td>
<td>362</td>
</tr>
<tr>
<td>Library (L)</td>
<td>8</td>
<td>53</td>
<td>551</td>
</tr>
<tr>
<td>ShoppingCart (SC)</td>
<td>8</td>
<td>20</td>
<td>229</td>
</tr>
<tr>
<td>Stack (S)</td>
<td>8</td>
<td>58</td>
<td>624</td>
</tr>
<tr>
<td>JDepend (JD)</td>
<td>10</td>
<td>53</td>
<td>2124</td>
</tr>
<tr>
<td>IDTable (ID)</td>
<td>11</td>
<td>24</td>
<td>315</td>
</tr>
</tbody>
</table>

- Empirically determined the \textit{Min} Jikes RVM heap size

---

When memory is not constrained, testing time is acceptable.

Testing time increases significantly when memory is Min.
### Summary of Reductions for Library

<table>
<thead>
<tr>
<th>Name</th>
<th>$T_R(P, T)$</th>
<th>$S_R(P, T)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-GC</td>
<td>32.7</td>
<td>78.8 ✓</td>
</tr>
<tr>
<td>X-GC</td>
<td>32.1</td>
<td>65.0</td>
</tr>
<tr>
<td>S-TM</td>
<td>32.0</td>
<td>72.8</td>
</tr>
<tr>
<td>X-TM</td>
<td>31.5</td>
<td>62.3</td>
</tr>
<tr>
<td>S-CS</td>
<td>34.3 ✓</td>
<td>61.4</td>
</tr>
<tr>
<td>X-CS</td>
<td>33.4</td>
<td>59.8</td>
</tr>
</tbody>
</table>

→ Significant reductions in time and space required for testing
Testing Time Overhead: Library

$S$ vs. $X$: Similar reductions for all code unloading techniques
Testing Space Overhead: Library

Code size reduction does not always yield best testing time
Code Size Fluctuation: Library

$S$-GC causes code size fluctuation that increases testing time
## Summary of Reductions for ID

<table>
<thead>
<tr>
<th>Name</th>
<th>$T_R^% (P, T)$</th>
<th>$S_R^% (P, T)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-GC</td>
<td>-1.1</td>
<td>42.5</td>
</tr>
<tr>
<td>X-GC</td>
<td>-1.1</td>
<td>26.7</td>
</tr>
<tr>
<td>S-TM</td>
<td>-1.2</td>
<td>44.5</td>
</tr>
<tr>
<td>X-TM</td>
<td>-.29 ✓</td>
<td>28.8</td>
</tr>
<tr>
<td>S-CS</td>
<td>-.77</td>
<td>51.4</td>
</tr>
<tr>
<td>X-CS</td>
<td>-1.4</td>
<td>61.4 ✓</td>
</tr>
</tbody>
</table>

- Unloading can decrease code size while still creating an overall increase in testing time.
Number of Code Unloads

All techniques cause ID to perform few unloading sessions
ID’s large working set forces unloading of many code bodies
Summary of Reductions

<table>
<thead>
<tr>
<th>Name</th>
<th>$\mathcal{T}_R(\mathcal{P}, \mathcal{T})$</th>
<th>$\mathcal{S}_R(\mathcal{P}, \mathcal{T})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-GC</td>
<td>16.1</td>
<td>68.4 ✓</td>
</tr>
<tr>
<td>X-GC</td>
<td>16.4</td>
<td>52.8</td>
</tr>
<tr>
<td>S-TM</td>
<td>17.1</td>
<td>62.6</td>
</tr>
<tr>
<td>X-TM</td>
<td>16.4</td>
<td>45.9</td>
</tr>
<tr>
<td>S-CS</td>
<td>17.6 ✓</td>
<td>58.8</td>
</tr>
<tr>
<td>X-CS</td>
<td>15.3</td>
<td>54.8</td>
</tr>
</tbody>
</table>

- Across all applications, adaptive code unloading techniques reduce both testing time and space overhead.
Conclusions and Future Work

- Dynamic compilation in JVMs can increase testing time if memory is constrained
- Adaptive unloading of native code often reduces memory overhead, avoids GC invocation, and reduces testing time
- Impact of unloading varies with respect to size of application’s working set and program testing behavior
- Regression test suite prioritization and reduction techniques that consider structural coverage and time and space overheads

http://cs.allegeheny.edu/~gkapfham/research/juggernaut/