Performance Regression Testing of Concurrent Classes

BY MICHAEL PRADEL, MARKUS HUGGLER, THOMAS R. GROSS
Goal

Automate for 2 versions of a concurrent class:

- Performance measurements
  - Reliable (no false positives or true negatives)
  - Meaningful (representative of real-world usage)

- Measurement evaluation
Algorithm overview: SpeedGun

1. Performance test generation
2. Performance measurements
3. Measurement evaluation
Test generation

Given a class to evaluate, generate...

- Sequential initialization (called **prefix**)
- Concurrent usage (called **suffixes**)

```java
ExpandoMetaClassInit v0 = new ExpandoMetaClassInit();
ExpandoMetaClass v1 = v0.unInitializedExpandoMetaClass();
Class v2 = v1.getJavaClass();
ExpandoMetaClass x = new ExpandoMetaClass(v2, true);
x.getExpandoMethods();

Thread 1  Thread N
```

```
String v4 = x.toString();
x.respondsTo(v4, v4, null);
x.isModified();
...
x.initialize();
x.getClassName();
x.getProperties();
...
```
Test generation (suffixes)

Which methods to test?

- Common interface of classes under test
  - Enables performance comparison
- Focus on methods with altered implementation
  - Only performance differences are interesting
- Other methods potentially required, can’t be ignored
Test generation (suffixes)

Test length?
- OS-provided timers have limited accuracy
  ⇒ No accurate measurement of short tests possible
- Too long tests have disadvantages, too
  - Expensive to generate: $O(n^2)$, $n := \text{length of call sequence}$
  - Little additional measurement value

Solutions:
⇒ Binary search for $n$ via trial and error
⇒ Call sequence of length $\sqrt{n}$ repeated $\sqrt{n}$ times
Performance measurements

Warm-up phase
(allow JIT to optimize)

Measure a fixed amount of times

Keep measuring until variance acceptable

---

**Algorithm 2** Gather execution times of a test.

**Input:** Test \( T \); Number of repetitions \( r_w \) and \( r_s \) for the warm-up phase and the steady-state phase, respectively

**Output:** Set \( \mathcal{M} \) of execution times or *inconclusive*

1. `runGarbageCollection()`
2. `repeat(T, r_w)`  \( \triangleright \) Warm-up phase
3. \( \mathcal{M} \leftarrow \emptyset \)  \( \triangleright \) Start of steady-state phase
4. `repeat`
5. \( \mathcal{M} \leftarrow \mathcal{M} \cup \text{repeatAndMeasure}(T, r_s) \)
6. `until` \( m_{\text{min}} \) measurements done
7. `while` \( \sigma(\mathcal{M}) > \overline{\mathcal{M}} \cdot \sigma_{\text{stop}} \) do
8. \( \mathcal{M} \leftarrow \mathcal{M} \cup \text{repeatAndMeasure}(T, r_s) \)
9. `if` \( |\mathcal{M}| = m_{\text{max}} \) then
10. `if` \( \sigma(\mathcal{M}) \leq \overline{\mathcal{M}} \cdot \sigma_{\text{acceptable}} \) then
11. `return` \( \mathcal{M} \)
12. `else`
13. `return` *inconclusive*
14. `end if`
15. `end if`
16. `end while`
17. `return` \( \mathcal{M} \)  \( \triangleright \) End of steady-state phase
Performance measurements

How to measure execution time of a test?

- Only **suffixes** are relevant
- May or may not measure **suffixes** individually, depending on use case

```
Algorithm 3 repeatAndMeasure(T, r)
Input: Test T; Number of repetitions r
Output: Execution time t
1:   t ← 0
2:   repeat
3:       Execute prefix of T
4:       Setup threads for suffixes of T
5:       start ← currentTime()  ▶ Start measurement
6:       for each thread do
7:           Execute a suffix of T
8:       end for
9:   t ← t + currentTime() − start  ▶ Stop measurement
10:  Clean up threads
11:  until r repetitions done
12:  return t
```
Measurement evaluation

Given execution times for a particular test...

- Compute mean and confidence interval
- Report performance difference if...
  - Confidence intervals don’t overlap
  - Difference between performances bigger than threshold
Measurement evaluation

Given evaluations of all tests concerning a class...

- Report performance difference if the majority of the tests show a performance difference in one direction
Real-world experiment setup

- Manual analysis of changes to Java code bases based on commit messages
- Comparison of SpeedGun’s results with manual analysis
Experimental results

<table>
<thead>
<tr>
<th>ID</th>
<th>Code base</th>
<th>Class</th>
<th>Revision</th>
<th>Description</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Baseline</td>
</tr>
<tr>
<td>(1)</td>
<td>Pool</td>
<td>GenericObject</td>
<td>774007</td>
<td>Finer-grained locking to avoid deadlocks described in Issue 125</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pool</td>
<td></td>
<td>Replace synchronized methods with volatile fields to address Issue 113</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>Pool</td>
<td>GenericObject</td>
<td>602773</td>
<td>Fix of a performance problem (Issue 93) by introducing more fine-grained locking</td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td>Collections</td>
<td>Static</td>
<td>BucketMap</td>
<td>1076039</td>
<td>Fix of a correctness bug (Issue 334) by adding synchronization</td>
</tr>
<tr>
<td>(5)</td>
<td>JodaTime</td>
<td>Date/Time</td>
<td>v2.1</td>
<td>Newer version is reported to decrease performance over v1.5.2 due to additional synchronization (Issue 153)</td>
<td></td>
</tr>
<tr>
<td>(6)</td>
<td>Groovy</td>
<td>ExpandoMetaCl</td>
<td>d3k3a3a44</td>
<td>Add synchronized blocks to fix correctness problem</td>
<td></td>
</tr>
<tr>
<td>(7)</td>
<td>Groovy</td>
<td>ExpandoMetaCl</td>
<td>1c947d6b</td>
<td>Replace synchronized collections with project-internal concurrent collections</td>
<td></td>
</tr>
<tr>
<td>(8)</td>
<td>Groovy</td>
<td>ExpandoMetaCl</td>
<td>2b09801c</td>
<td>Add synchronized block to fix correctness problem</td>
<td></td>
</tr>
<tr>
<td>(9)</td>
<td>Groovy</td>
<td>ExpandoMetaCl</td>
<td>feff5190</td>
<td>Synchronize methods to fix correctness bug (Issue 2166)</td>
<td></td>
</tr>
<tr>
<td>(10)</td>
<td>Groovy</td>
<td>ExpandoMetaC</td>
<td>83629dc1</td>
<td>Patch to improve (sequential) performance</td>
<td></td>
</tr>
<tr>
<td>(11)</td>
<td>Groovy</td>
<td>ExpandoMetaC</td>
<td>77822d4e</td>
<td>Replace project-internal concurrent collections with java.util.concurrent collections</td>
<td></td>
</tr>
<tr>
<td>(12)</td>
<td>Groovy</td>
<td>ExpandoMetaC</td>
<td>6e349ed9</td>
<td>Large patch without any obvious effects on performance</td>
<td></td>
</tr>
<tr>
<td>(13)</td>
<td>Groovy</td>
<td>ExpandoMetaC</td>
<td>26f21b00</td>
<td>Replace synchronized methods with volatile field to fix performance bug (Issue 3557)</td>
<td></td>
</tr>
<tr>
<td>(14)</td>
<td>Groovy</td>
<td>ExpandoMetaC</td>
<td>d92c12ab</td>
<td>Replace volatile fields with synchronized methods to fix correctness problem</td>
<td></td>
</tr>
<tr>
<td>(15)</td>
<td>Groovy</td>
<td>ExpandoMetaC</td>
<td>48269129</td>
<td>Replace synchronized method with volatile fields to address performance problem (Issue 4182)</td>
<td></td>
</tr>
<tr>
<td>(16)</td>
<td>Groovy</td>
<td>ExpandoMetaC</td>
<td>cc39843</td>
<td>Supposed performance improvement by replacing synchronized method with explicit locks</td>
<td></td>
</tr>
<tr>
<td>(17)</td>
<td>Groovy</td>
<td>ExpandoMetaC</td>
<td>d38da33c</td>
<td>Replace volatile field with synchronized method to fix correctness bug</td>
<td></td>
</tr>
</tbody>
</table>
Experimental results

- **SpeedGun** coincides with majority of manual analysis
- Identified where expected performance improvements did not happen and vice versa
- Quality versus quantity of reports controlled by threshold
Conclusion

Results:
- Goal met! No obvious issues found in real-world experiment.

Limitations:
- Running time of several hours per class
- Automatic test generation may be too artificial for real world
Questions?
Thank you!