The Role of Method Call Optimizations in the Efficiency of Java Generics

Jeffrey D. Lindblom, Seth Sorensen, Elena Machkasova

April 14, 2012
Terminology

Java HotSpot Virtual Machine (JVM)
An application developed by Oracle that interprets a compiled Java program.

Just-in-Time Compiler (JIT)
A part of the JVM that optimizes code through recompilations at run-time.

Java Generics
A type in Java that allow the contents of a container to be bounded to a single, specified type. (E.g. ArrayList<String>).
Describe the influence of Java Generics on run times of Java programs

Detect the presence of optimizations such as inlining and devirtualization

Explore tools and methodology for observing JIT optimizations of Java Generics:
  - Profilers such as XProf
  - Internal logging of JIT
Challenges: JVM Complexity

- The HotSpot JVM documentation is not detailed and often not up to date
- Which JIT optimizations matter and why is difficult to assess
- The HotSpot JVM is multi-threaded
- JIT optimizations may be scheduled differently among multiple runs of the same program
Challenges: JVM Diagnostics

- **Observer Effect:**
  - Profilers can influence JIT optimizations as well as program run times.

- **Absence of Relevant Data:**
  - Differences among run times for multiple runs of the same program may not be explainable by using the tools at our disposal.

- **Presence of Irrelevant Data:**
  - Tools can provide overwhelming amounts of information that may or may not be useful in describing observations.
Java Execution Model

Java code is executed through a two-phase compilation process:

- Initial compilation into bytecode
- Additional recompilation by the JIT

Three internal representations exist:

- Bytecode
- Native code produced by the JIT
- The Sea of Nodes within the JIT
During JIT compilation, optimizations are made to increase efficiency and decrease run time of the program.

**Devirtualization**
- The JVM uses *Virtual Method Lookup* to locate the correct method.
- JIT replaces these calls with jumps after repeated look-up.

**Inlining**
- The method call is replaced by the code it represents.
- A method call threshold must be reached before optimizations take place.
public class ArrayList<T>

ArrayList<String> strArrayList = new ArrayList<String>();

public class ArrayListInteger extends ArrayList<Integer>

The last example is referred to as *bound narrowing*, where the element type of a class is more specific than that of its superclass.
Bound Narrowing

- public class Generic<K, V> extends HashMap<K, V>
- public class Narrowed extends HashMap<Integer, String>
- hashMap = new Generic<Integer, String>();
- hashMap = new Narrowed();
public boolean containsValue(String value) {
    // some unimportant code removed
    Entry[] tab = table;
    for (int i = 0; i < tab.length; i++)
        for (Entry e = tab[i]; e != null; e = e.next)
            if (value.equals(e.value))
                return true;
    return false;
}
Running times for Narrowed and Generic runs.

- 100,000,000 method calls for `containsValue`
- 10 test runs for each of Narrowed and Generic.
Running the same code multiple times may result in differing run times: instability

<table>
<thead>
<tr>
<th></th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Run 6</th>
<th>Run 7</th>
<th>Run 8</th>
<th>Run 9</th>
<th>Run 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Narrowed</td>
<td>6.87</td>
<td>8.09</td>
<td>8.24</td>
<td>8.81</td>
<td>6.87</td>
<td>8.58</td>
<td>6.87</td>
<td>8.03</td>
<td>6.87</td>
<td>6.61</td>
</tr>
<tr>
<td>Generic</td>
<td>8.55</td>
<td>8.53</td>
<td>7.84</td>
<td>7.83</td>
<td>8.59</td>
<td>8.82</td>
<td>7.82</td>
<td>7.83</td>
<td>7.83</td>
<td>8.60</td>
</tr>
</tbody>
</table>

- In some cases two runs may produce identical logs (Generic)
- In other cases there are differences in logs (Narrowed)
- We can use the differences in logs to explain the second type of instability
LogCompilation of Two Narrowed Test Runs: Compilations

- Slow Run: 8.87 s

```
Task: compile_id = 3, method = TestNarrowed innerLoop (ILMap;[Ljava/lang/String;)Z, bytes = 34, count = 10000, backedge_count = 5386, icount = 5, stamp = 0.131

Method: id = 604, name = innerLoop, bytes = 34, icount = 5,
Method: id = 609, name = containsValue, bytes = 0, icount = 1,
Call: method = 609, count = 43394, prof_factor = 1, virtual = 1, inlining = 1, receiver = 607, receiver_count = 43394,
Method: id = 610, name = containsValue, bytes = 9, compile_id = 2, compiler = C2, level = 2, icount = 10000,
Call: method = 610, count = 43394, prof_factor = 1, inlining = 1, inlining_reason = already compiled into a big method

Task done: success = 1, nmsize = 316, count = 10000, backedge_count = 5386, stamp = 0.134
```

- Fast Run: 6.97 s

```
Task: compile_id = 3, method = TestNarrowed innerLoop (ILMap;[Ljava/lang/String;)Z, bytes = 34, count = 2, backedge_count = 5000, icount = 2, stamp = 0.121

Method: id = 604, name = innerLoop, bytes = 34, icount = 2,
Method: id = 609, name = containsValue, bytes = 0, icount = 1,
Call: method = 609, count = 6701, prof_factor = 1, virtual = 1, inlining = 1, receiver = 607, receiver_count = 6701,
Method: id = 610, name = containsValue, bytes = 9, icount = 10000,
Call: method = 610, count = 6701, prof_factor = 1, inlining = 1,
Method: id = 612, name = containsValue, bytes = 64, compile_id = 1, compiler = C2, level = 2, icount = 2501,
Call: method = 612, count = 6701, prof_factor = 0.6701, inlining = 1,
Method: id = 621, name = equals, bytes = 88, icount = 6612,
Call: method = 621, count = 4189, prof_factor = 1, inlining = 1,

Task done: success = 1, nmsize = 1456, count = 10000, backedge_count = 5342, inlined_bytes = 152, stamp = 0.159
```
LogCompilation of Two Narrowed Test Runs: Nodes

- **Slow Run: 8.87 s**
  
  In Thread 1
  - Task: compile_id = 2, method = Narrowed containsValue (Ljava/lang/Object;)Z, bytes = 9, count = 5000, icount = 10000, stamp = 0.112,

  Phase: name = parse, nodes = 3, stamp = 0.012,
  Phase: name = optimizer, nodes = 403, stamp = 0.113,
  Phase: name = matcher, nodes = 815, stamp = 0.116,
  Phase: name = regalloc, nodes = 446, stamp = 0.117,
  Phase: name = output, noded = 775, stamp = 0.124.

  Task done: success = 1, amsize = 1040, count = 5000, inlined_bytes = 152, stamp = 0.124.
  - Task: compile_id = 3, method = TestNarrowed innerLoop (ILMap;Ljava/lang/String;Z)Z, bytes = 34, count = 10000, backedge_count = 5396, icount = 5, stamp = 0.131,

  Phase: name = parse, nodes = 3, stamp = 0.132,
  Phase: name = optimizer, nodes = 156, stamp = 0.132,
  Phase: name = matcher, nodes = 177, stamp = 0.133,
  Phase: name = regalloc, nodes = 120, stamp = 0.133,
  Phase: name = output, noded = 170, stamp = 0.134.

  Task done: success = 1, amsize = 316, count = 10000, backedge_count = 5386, stamp = 0.134.

- **Fast Run: 6.97 s**
  
  In Thread 1
  - Task: compile_id = 2, method = Narrowed containsValue (Ljava/lang/Object;)Z, bytes = 9, count = 5000, icount = 10000, stamp = 0.112,

  Phase: name = parse, nodes = 3, stamp = 0.012,
  Phase: name = optimizer, nodes = 403, stamp = 0.113,
  Phase: name = matcher, nodes = 815, stamp = 0.116,
  Phase: name = regalloc, nodes = 446, stamp = 0.117,
  Phase: name = output, noded = 775, stamp = 0.134.

  Task done: success = 1, amsize = 1040, count = 9901, inlined_bytes = 152, stamp = 0.134.
  
  In Thread 2
  - Task: compile_id = 3, method = TestNarrowed innerLoop (ILMap;Ljava/lang/String;Z)Z, bytes = 34, count = 2, backedge_count = 5000, icount = 2, stamp = 0.121,

  Phase: name = parse, nodes = 3, stamp = 0.121,
  Phase: name = optimizer, nodes = 496, stamp = 0.123,
  Phase: name = matcher, nodes = 949, stamp = 0.126,
  Phase: name = regalloc, nodes = 524, stamp = 0.136,
  Phase: name = output, noded = 1042, stamp = 0.158.

  Task done: success = 1, amsize = 1456, count = 10000, backedge_count = 5342, inlined_bytes = 152, stamp = 0.159.
## XProf

**Slow Run: 8.87 s**

<table>
<thead>
<tr>
<th>Method</th>
<th>Compiled + native</th>
<th>Native Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Narrowed.containsValue</td>
<td>77.0%  676 + 0</td>
<td></td>
</tr>
<tr>
<td>TestNarrowed.innerLoop</td>
<td>21.8%  191 + 0</td>
<td></td>
</tr>
<tr>
<td>Total compiled</td>
<td>98.7%  867 + 0</td>
<td></td>
</tr>
</tbody>
</table>

**Fast Run: 6.97 s**

<table>
<thead>
<tr>
<th>Method</th>
<th>Compiled + native</th>
<th>Native Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestNarrowed.innerLoop</td>
<td>99.1%  672 + 0</td>
<td></td>
</tr>
<tr>
<td>Narrowed.containsValue</td>
<td>0.1%   1 + 0</td>
<td></td>
</tr>
<tr>
<td>Total compiled</td>
<td>99.3%  673 + 0</td>
<td></td>
</tr>
</tbody>
</table>
Conclusions

- Able to classify and distinguish instability through:
  - Differences in LogCompilation
  - Differences in XProf output
- Observed evidence of specific methods being inlined
- Developed strategies for describing specific behaviors of JIT
Use these strategies to explain other behaviors associated with Java generics

- More recent versions of Java SE 6
- Extend to Java SE 7