Updating Gin’s Profiler for Current Java
Myles Watkinson, Sandy Brownlee
Presentation Structure

1. Explaining what Gin is
2. What role does a profiler play in a GI framework
3. Selecting a profiler for Gin
4. Comparing the new profiler to the old
5. Conclusion
Gin: A Toolbox for Genetic Improvement

- Created to stimulate genetic improvement research
- Designed to be simple and understandable
- Implementations of common edits, build pipeline, testing, speed and memory measurement, and profiling
How GI (in Gin) Works

Code

Delete
Swap
Replace

Mutate
Fails tests
Slower

Search
Gin is stuck in Java 8

HPROF profiler
Only compatible in Java 8 and below

Reflection security
Only able to reflect easily in Java 8 and below

Gin
Java 8

Gin
Java 9 onwards
Why so much effort on the profiler?

- GI to improve runtime of Java code while retaining functionality
  - edits are targeted at “hot methods”
  - these are where the CPU spends the most time

- Profiler:
  - selects the hot methods
  - determines the order the hot methods are ranked

- Need to consider how the profiler plugs into Gin and is used by it
Choosing a new Profiler: Criteria

- The profiler should plug straight into Gin as HPROF did

- The profiler should produce a similar output to that below that can be read and utilised by Gin
Choosing a new Profiler: Criteria

- Input and output needs to be handled automatically by Gin, no visual interfaces.

- Low overhead is needed as running all unit tests may take time. It is preferable that a profiler adds a little time as possible.

- Gin is a research tool. External profilers used should be free and simple to use.

- The profiler needs to accurately count running functions from the specific codebase.
Candidates

**Visual interface:**
VisualVM, Java Mission Control, NetBeans profiler

**Cost Associated:**
JProfiler

**Large Overhead:**
JConsole

**Potential Profiler:**
Java Flight Recorder (JFR)
# Integrating JFR into Gin

<table>
<thead>
<tr>
<th>HPROF</th>
<th>JFR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outputs to <code>.txt</code> file</td>
<td>Outputs to <code>.jfr</code> file</td>
</tr>
<tr>
<td>Gives most commonly seen methods</td>
<td>Gives call stack</td>
</tr>
<tr>
<td>Skips Java language functions</td>
<td>Call stacks contain all Java language functions</td>
</tr>
<tr>
<td>Profiles all threads</td>
<td>Doesn’t profile sleeping or waiting threads</td>
</tr>
</tbody>
</table>
Processing JFR call stacks

HPROF pre-processes call stacks and only outputs functions from the program being run most commonly seen in the call stack.

JFR outputs a raw call stack which often contains Java language functions.

- `Java.vector.indexOf()`
- `Java.vector.copyOf()`
- `MainProgram.begin()`

On top of call stack before processing.

Function added to profiled method count.
Experiments

Two experiments were run to compare HPROF and JFR

1. Profiling a set of simple functions that calculate primes

2. Profiling a more realistic program
Profiling prime number calculations

Only 1 function running at a time with an understanding of how the calculation of primes scales with time

Calculating 5,000 through to 25,000 primes, taking raw time and method count
# Prime calculation profiling results

<table>
<thead>
<tr>
<th>Primes found</th>
<th>Time taken (ms)</th>
<th>Call stack samples found with HPROF</th>
<th>Call stack samples found with JFR</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>24</td>
<td>1.25</td>
<td>1.75</td>
</tr>
<tr>
<td>10,000</td>
<td>75</td>
<td>5.75</td>
<td>4</td>
</tr>
<tr>
<td>15,000</td>
<td>153</td>
<td>12.5</td>
<td>8.5</td>
</tr>
<tr>
<td>20,000</td>
<td>266</td>
<td>23</td>
<td>14.75</td>
</tr>
<tr>
<td>25,000</td>
<td>412</td>
<td>34</td>
<td>22.75</td>
</tr>
<tr>
<td>30,000</td>
<td>588</td>
<td>54.5</td>
<td>32.25</td>
</tr>
</tbody>
</table>
Reasons for different profiling results

Different Java versions?
- HPROF and JFR had to be run in Java 8 and 9 respectively, although, there was almost no difference in the runtime between each version.

Thread in a state not profilable by JFR?
- JFR omits samples if the thread sampled is in a WAITING, SLEEPING or BLOCKED state. Although, the program simply adds numbers to a vector if they are primes, there is no waiting or sleeping done in the program.
Further Investigation

When profiling this code:

```java
long start = System.currentTimeMillis();
Long now = 0;
while (now < 2000)
{
    Now = System.currentTimeMillis() - start;
    //JFR doesn’t profile the above system call
}
```

HPROF consistently returns 127 samples

JFR returns between 10 and 30
Profiling a more realistic program

The program profiled was Spark, a Java web framework
https://github.com/perwendel/spark

The standard Gin interface was used, Sparks unit tests were run
and profiled to produce a hot method summary
Spark profiling results

Spearman Coefficient for top 10 JFR methods and corresponding HPROF methods: 0.29

Spearman Coefficient for top 20 JFR methods and corresponding HPROF methods: 0.8
Conclusion

→ Gave an overview of the importance of a profiler in a GI framework

→ Proposed a set of criteria for selecting a profiler

→ Ran experiments to compare two profilers

→ Integrated this profiler into Gin to boost it into current Java versions retaining its efficacy as a tool for GI research
Any Questions?

Link to Gin repository: https://github.com/gintool/gin
Email: myleswatkinson1@gmail.com