Exhaustive Testing of Safety Critical Java

Tomas Kalibera
Pavel Parizek
Michal Malohlava

Martin Schoeberl

Charles University
Technical University of Denmark
Exhaustive Testing with Java PathFinder (JPF)

- JPF is a specialized Java Virtual Machine (JVM)
  - Runs Java programs
  - Saves program state and backtracks over different scheduling sequences
  - Looks for error states (exceptions, races, ...)
- Optimizations
  - Re-scheduling only at operations that are not thread local (partial order reduction)
  - Detection of visited states (state matching)
- Designed for plain Java

(there is much more to it, see http://babelfish.arc.nasa.gov/trac/jpf/ )
Our Goal: Tool for Exhaustive Testing of SCJ Programs

- **Features sought**
  - Find races (SCJ L1 and higher)
  - Find SCJ specific errors and plain Java errors even if scheduling sequence dependent

- **Challenges**
  - Cover all possible scheduling sequences with a real-time scheduler
  - Fight state explosion so that we can check non-toy programs
Our Contribution: Tool for Exhaustive Testing of SCJ

• Prototype implementation $\text{Rs}_\text{sj}$ – JPF extension
  – Detects invalid memory assignments, potential races, regular Java errors, failed assertions
  – Supports subset of SCJ L0/L1, only periodic handlers
  – Tested with Collision Detector and PapaBench

• SCJ L0,L1 scheduling algorithm for JPF
  – Reduction of the number of states with execution time estimator for target platform
  – Tested with Java Optimized Processor (JOP)
SCJ L0, L1 Scheduling for JPF
The Notion of Time at SCJ Level 0

- Only one valid scheduling sequence
- Notion of time is only needed for
  - The application – Clock.getTime
  - Diagnostics – detect possible frame overruns
The Notion of Time at SCJ Level 0

startup

mission init

H1

H2

H3

idle

H2

idle

Time

t = 0

t = 0 + “length of frame 1”
The Notion of Time at SCJ Level 0

\[
t_{\text{min}} = 0 \\
t_{\text{max}} = 0 \\
t_{\text{min}} = t_{\text{min}} + \text{“lower bound for execution time of insn1”} \\
t_{\text{max}} = t_{\text{max}} + \text{“upper bound for execution time of insn1”}
\]
The Notion of Time at SCJ Level 1

Startup mission init

H1 preempts H2

H1 completes, H2 continues

H2

H2

H1

H3

idle

H2

Releases

Time
The Notion of Time at SCJ Level 1

- Notion of time needed for scheduling
- Imprecise notion of time results in multiple valid scheduling sequences
The Notion of Time at SCJ Level 1

Releases

\[ t = 0 \]

\[ t = \text{"release offset of H2"} + 1 \times \text{"period of H2"} \]
Non-deterministic Execution at SCJ Level 1

startup mission init H2 H1 H3 idle H2

Releases

insn1 insn2 insn3

t_R
Non-deterministic Execution at SCJ Level 1

Is $t_{\text{min}} \leq t_{R} \leq t_{\text{max}}$?
(Can the release happen now?)

If NOT, keep executing H2

If YES, choose non-deterministically whether to release or not

$t_{\text{min}} = 0$
$t_{\text{max}} = 0$

$t_{\text{min}} = t_{\text{min}} + \text{"lower bound for execution time of insn1"}$
$t_{\text{max}} = t_{\text{max}} + \text{"upper bound for execution time of insn1"}$
Evaluating RSJ

Does it scale to real programs?
What are the caveats of our scheduling algorithm?
## Testing with Application Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># of Tasks</th>
<th>SCJ</th>
<th>Checking Time</th>
<th>Memory Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDx – no simulator</td>
<td>1</td>
<td>L0</td>
<td>8s</td>
<td>490M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L1</td>
<td>12s</td>
<td>490M</td>
</tr>
<tr>
<td>CDx – with simulator</td>
<td>2</td>
<td>L0</td>
<td>34s</td>
<td>580M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L1</td>
<td>35s</td>
<td>710M</td>
</tr>
<tr>
<td>PapaBench</td>
<td>14</td>
<td>L0</td>
<td>15min</td>
<td>14G</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L1</td>
<td>31min</td>
<td>15G</td>
</tr>
</tbody>
</table>

### CDx

Collision Detector benchmark (Purdue), aircraft collision detection. We implemented the SCJ port of CDx with simulator and the L1 version.

### PapaBench

Based on Paparazzi UAV auto-pilot. We translated the C version of PapaBench to Java and extended it to be executable.
Java PapaBench: A Better RT Java Application Benchmark

- **Paparazzi Project**
  - Free auto-pilot (free sw, open-design hw)
  - Implemented in C, has flown real UAVs

- **C PapaBench**
  - A subset of an earlier version of Paparazzi, intended for testing WCET analysis tools
  - IRIT, France

- **Java PapaBench**
  - Java/RTSJ/SCJ translation of PapaBench
  - Includes environment simulation to be executable
  - Michal Malohlava, Charles University
(Java) PapaBench Components

• Autopilot
  – Produces low-level flight commands to FBW
  – Follows a pre-configured high-level flight plane
  – Reacts to input from GPS and IR

• Fly-by-wire (FBW)
  – Low-level access to aircraft hardware

• Simulator
  – GPS, IR interrupt source
  – Physical environment simulation
Checking RT Programs: Lessons Learned
• State matching needs revisiting
  – Current time is part of program state – SM has to be disabled, otherwise we fail to fully check a program
• Partial order reduction does not apply
  – Scheduler decisions in a real system are deterministic
  – Potential preemption points have to be fine grained (i.e. a single instruction in Rstu) to bound release jitter
• More work is needed to customize JPF-core
  – By default, states are saved even at deterministic thread switch
See the official RTEembed extension of JPF at
http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/rtembed for our related efforts.