Ji.Fi: Visual Test and Debug Queries for Hard Real-Time

Ethan Blanton\textsuperscript{1}, Demian Lessa\textsuperscript{2}, Lukasz Ziarek\textsuperscript{1,2}, Bharat Jayaraman\textsuperscript{2}

Fiji Systems Inc.\textsuperscript{1}, SUNY Buffalo\textsuperscript{2}

October 25, 2012
Motivation

- Debugging **real-time systems** is difficult
  - Symbolic debugger (gdb, jdb) overhead
  - Stop-and-examine difficult or **entirely precluded**
- Cost per line of certified code is very high
  - $1000 per line, versus $15-30 for non-certified code\(^1\)
- We need more tools for SCJ and RTSJ
- We want a tool for teaching undergraduates SCJ/RTSJ

JIVE is a visual debugger.
- Uses extended UML object and sequence diagrams
- Declarative queries reduce visualization complexity
- The necessary data and no more

Built on Eclipse
- Extended evaluation in the literature
JIVE Suitability

JIVE provides many nice features

- Powerful visualization
- Forward- and reverse-stepping
- Queries to reduce cognitive burden: “when” style questions

However...

- No sense of real time, only logical time
- No offline debugger
- JDI event granularity is too fine
- No monitors or synchronization events
Extending JIVE: Ji.Fi

- System split into two levels
- **Replaceable bottom half** (Use your own compiler/VM!)
  - Our implementation uses the Fiji VM
- Modified JIVE **top half**
Visualizations

- RT Object Diagram
- RT Sequence Diagram
- Thread State Diagram
Thread State Diagram

- User Thread:
  - Start
  - Sleep
  - Notify

- Main Thread:
  - Start
  - Wait
  - Synchronized

Legend:
- Ready
- Running
- Waiting
- Blocked
Dynamic Query-Based Analysis

Temporal Queries provide analysis of temporal behavior
- Natural representation of a real-time system log
- Complex behaviors reduced to SQL-like expressions
- Recursive queries are also expressible
- Questions like:
  - Which monitors are accessed by multiple threads?
  - Does this execution trigger priority inversion avoidance?
  - Which periods include deadline misses?
Contended Monitors

```sql
SELECT m1.threadId AS h, m2.threadId AS b,
       m1.monitor AS m, m2.time
FROM (event NATURAL JOIN event_monitor) m1,
     (event NATURAL JOIN event_monitor) m2,
     (event NATURAL JOIN event_monitor) m3
WHERE
  m1.monitor = m2.monitor AND
  m1.monitor = m3.monitor AND
  m1.threadId = m3.threadId AND
  m1.threadId <> m2.threadId AND
  m1.time < m2.time AND m2.time < m3.time AND
  m1.kind IN (12, 13) AND
  m2.kind = 11 AND
  m3.kind IN (15, 16) AND
NOT EXISTS (SELECT 1
FROM (event NATURAL JOIN event_monitor) mx
WHERE mx.kind IN (15,16) AND
      m1.monitor = mx.monitor AND
      m1.threadId = mx.threadId AND
      m1.time < mx.time AND mx.time < m3.time);
```
Contended Monitors: Materialized Relations

```sql
1. SELECT holder AS h, threadId AS b, monitor AS m, time
2. FROM event NATURAL JOIN event_monitor
3.   NATURAL JOIN monitor
4. WHERE
5.     -- event is LOCK BEGIN
6.     kind = 11 AND
7.     -- monitor is locked
8.     lockCount > 0 AND
9.     -- lock is held by a different thread
10.    holder <> threadId
```
Making Queries Easier

- JI.FI will include relevant materialized relations, *etc.*
- Designing queries is still complicated
- **Canned queries** can be presented to users as *Forms*
  - Users fill out *simple forms*, get *sophisticated answers*
  - *e.g.*, “Show me every period where PIP was invoked on monitor <X>”
- More work to be done on this front
Log Structure

- There are two levels of logs:
  - Low-level **VM logs**
  - High-level **XML logs**
- Neither level of log is VM-specific
  - *Although, translation between the two may be!*
- VM logs are designed for speed and predictability
- XML logs are designed for expression and extensibility
Real-Time Log Entry

<table>
<thead>
<tr>
<th>Type</th>
<th>Subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>24B</td>
<td></td>
</tr>
</tbody>
</table>

- **Compile-time constant**
- **Run-time dynamic**
- **Compile-time constant**
- **Run-time constant**
- **Run-time dynamic**

- **Type**:
  - Low-level Log

- **Subtype**:
  - Event-Specific Data
  - Timestamp
  - Thread ID

- **Dimensions**:
  - 0-32b
  - 0-24B

*Ji.Fi: Visual Test and Debug Queries for Hard Real-Time*
Log Buffers

- Each Java thread maintains a buffer of log entries
- Individual log entries are lock-free writes
- Handoff to a log-flushing thread has a tight critical window
- Log buffer management requires more work
Log Validation

Both low-level and high-level log manipulations involve validation

- Verify that VM log emissions are complete and correct
  - Monitor actions balance
  - Method calls nest
  - *etc.*
- Validate XML structure
- Check equivalence of VM and JIVE semantics
Runtime Instrumentation

Code paths with neat insertion points get runtime instrumentation

- Monitors
- Thread state changes
- Priority changes

```java
1  static void sleep(long ms, int ns) throws InterruptedException {
2      long t = Time.nanoTime() + ms * 1000 * 1000 + ns;
3      FlowLog.log(FlowLog.TYPE_THREAD, FlowLog.SUBTYPE_SLEEP, t);
4      sleepAbsolute(t);
5      FlowLog.log(FlowLog.TYPE_THREAD, FlowLog.SUBTYPE_WAKE, t);
6  }
```
Compiler Instrumentation

Logging within user code and compiler-generated code is inserted by the bytecode-to-C compiler

- Method calls
- Returns
Instrumentation Challenges

- Keeping log fields constant
  - *e.g.*, Assign **unique 32-bit integer** to every method

- Filtering uninteresting methods
  - Rule-based (user-provided code vs. VM code)
  - Annotations

- Choosing **minimal but sufficient** data for events

- Timestamps can be **very expensive!**
Evaluation Methodology

- Evaluated using CDj
- CDj simulates an air traffic control collision detector
- Many tunable parameters
  - 6 aircraft
  - 50 frames per second (20 ms release)
  - Varying number of frames
Computational Overhead

- Pure software implementation
- Architecturally optimized, but not optimized “in the small”
- **67% overhead** for extensive logging
  - Covers all user-provided code plus entry into system code
  - Overhead doubles to include getters/setters/trivial methods
- Substantial overhead reduction by turning off features
- Method tracing is the most expensive event class
Predictability

![Graph showing predictability over release number]

- **No logging**
- **Complete logging**

Duration (CPU cycles) vs. Release number
Next Steps: SCJ + RTSJ

- Visualizations: Missions, Scopes, Parameters
- Temporal Queries: Form Queries
- Logging: Scope instrumentation
- Generate scope size estimates (unlimited mode)
Next Steps: Provably Predictable Logging

- WCET Analysis of the logging framework
- Schedulability of program + logging framework
- Log impact report via Visualization
Next Steps: Multi-Run Executions

- Motivation: Cannot log everything
- Produce multiple version of the program, each instrumented to log different data
- Post execution, merge logs