Programming energy aware systems in Safety Critical Java

Bent Thomsen
CISS/Dept. Computing, AAU
ICT Energy Conference
18.8.2016
Joint work with:

- Allan Mycroft
  - Cambridge University
- Corina S. Pasareanu
  - NASA Ames, CA, USA
- Hans Søndergaard, Stephan Korsholm
  - Via University College
- Thomas Bøgholm, Kasper Søe Luckow, Anders P. Ravn, Kim G. Larsen, Rene R. Hansen and Lone Leth Thomsen
  - CISS/Department of Computer Science, Aalborg University
Embedded Control Systems

• Over 90% of all microprocessors are used for real-time and embedded systems
  – Market growing 10% year on year

• Usually programmed in C or Assembler
  – Hard, error prone, work
  – But preferred choice
    • Close to hardware
    • No real alternatives

  – Difficult to find new skilled programmers
    • Jackson Structured Development (1975) still widely used
    • EE Times calling for re-introducing C programming at US Uni

Well ... ADA – 10th on the list of most wanted skills
We need to look for other languages

• The number of embedded systems is growing
• More functionality in each system is required
• More reliable systems are needed
• Time to market is getting shorter
• Increase productivity
  – Software engineering practices (OOA&D) – 10%
  – Tools (IDEs, analyzers and verifiers) – 10%
  – New Languages -700%
    • 200%-300% in embedded systems programming (Atego)
Java

• Most popular programming language ever!
  – In 2005 Sun estimated 4.5 million Java programmers
  – In 2010 Oracle estimated 9 million Java programmers
  – 61% of all programmers are Java programmers

• Originally designed for setop-boxes
  – 3 billion devices run Java

• But propelled to popularity by the internet
  – Write once, Run everywhere

http://jaxenter.com/how-many-java-developers-are-there-10462.html
What is the problem with Java?

• Unpredictable performance
  – Memory
    • Garbage collected heap
  – Control and data flow
    • Dynamic class loading
    • Recursion
    • Unbounded loops
    • Dynamic dispatch
    • Exceptions
  – Scheduling
  – Lack high resolution time

• JVM
  – Good for portability – bad for predictability
Real-Time Specification for Java (RTSJ)

- **Java Community Standard (JSR 1, JSR 282)**
  - Started in 1998
    - January 2002 – RTSJ 1.0 Accepted by JSP
    - Spring 2005 – RTSJ 1.0.1 released
    - Summer 2006 – RTSJ 1.0.2 initiated
    - March 2009 Early draft of RTSJ version 1.1 now called JSR 282.
    - March 2015 Early draft review 2

- **New Thread model: NoHeapRealtimeThread**
  - High resolution time and timer
  - Clear definition of scheduler
    - Extends Java’s 10 priority levels to 28
    - Priority inheritance protocol
  - Scoped memory to avoid GC
    - Never interrupted by Garbage Collector
    - Threads may not access Heap Objects
  - Low-level access through raw memory
RTSJ Guiding Principles

• Backward compatibility to standard Java
• No Syntactic extension
• Reflected current real-time practice anno 1998
• Allow implementation flexibility
• Rather complex and very dynamic
• Write Once, Run Anywhere
  – But execution time is platform dependent
• Most common for real-time Java applications
  – Especially on Wall Street

• Does not address certification of Safety Critical applications
Observation

There is essentially only one way to get a more predictable language:

• namely to select a set of features which makes it controllable.

• Which implies that a set of features can be deselected as well.
Safety-Critical Java (SCJ)

• Java Specification Request 302
• Aims for DO178B, Level A (IEC 61508/ISO 26262)
• Restricted/extended subset of RTSJ
• Three Compliance Points (Levels 0, 1, 2)
  – Level 0 provides a cyclic executive (single thread), no wait/notify
  – Level 1 provides a single mission with multiple schedulable objects
  – Level 2 provides nested missions with (limited) nested scopes
• More analysis friendly
  – Especially Worst Case Execution Time Analysis
  – Write once, Run where ever possible
Listing 1: An SCJ handler for methane level [16].

Listing 2: Detecting the methane level [16].
SCJ

- Only RealtimeThreads are allowed
- Notions of missions and handlers
- No heap objects/ no GC
- Restricted use of scopes
Predicatble JVM

- **JOP**
  - Java Optimized Processor
  - JVM in Hardware (FPGA)

- **HVM**
  - Java-to-C compiler with an embedded interpreter
  - Execution on the bare metal
  - Run in 256 KB ROM and 20 KB RAM
  - Interpreted or AOT compiling
  - 1st level interrupt handlers in Java
  - Runs on ATmega2560, CR16C, ARM7, ARM9 and x86

- **JamaicaVM**
  - Industrial strength real-time JVM from Aicas
  - Enroute for Certification for use in Airplanes and Cars
The Predictable Real-time HVM

- Time predictable implementations of Interpreter loop and each bytecode

```c
1  static int32  methodInterpreter(const
2   MethodInfo*  method,  int32*  fp) {  
3     unsigned char*  method_code;
4     int32*  sp;
5     const MethodInfo*  methodInfo;
6     
7     start:  method_code = (unsigned char*)
8           pgm_read_pointer(&method->code, unsigned
9           char**);
10    sp = &fp[pgm_read_word(&method->maxLocals)
11                  + 2];
12    
13    loop:  while (1) {
14       unsigned char  code = pgm_read_byte(
15          method_code);
16       switch (code) {
17          case ICONST_0_OPCODE:
18             //ICONST_X Java Bytecodes  
19          case ICONST_5_OPCODE:
20             *sp++ = code - ICONST_0_OPCODE;
21             method_code++;
22             continue;
23          case FCONST_0_OPCODE:
24             //Remaining Java Bytecode impl...  
25          }
26    }
27  }
```
What about Time Analysis?

Utilisation-Based Analysis

- A simple sufficient but not necessary schedulability test exists

\[ U \equiv \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{1/N} - 1) \]

\[ U \leq 0.69 \text{ as } N \to \infty \]

Where C is WCET and T is period

Response Time Equation

\[ R_i = C_i + \sum_{j \in hp(i)} \left[ \frac{R_j}{T_j} \right] C_j \]

Where hp(i) is the set of tasks with priority higher than task i

Solve by forming a recurrence relationship:

\[ w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left[ \frac{w_j^n}{T_j} \right] C_j \]

The set of values \( w_i^n, w_i^1, w_i^2, \ldots, w_i^n \) is monotonically non decreasing

When \( w_i^n = w_i^{n+1} \) the solution to the equation has been found, \( w_i^n \) must not be greater than \( R_i \) (e.g. 0 or \( C_i \))

- Traditional approaches to analysis of RT systems are hard and conservative
- Very difficult to use with Java because of JVM (and Object Orientedness)
Model based Analysis

• Translate timing analysis problems into analysis of properties of timed Automatas
  – TIMES
    • Model based schedulability tool based on UPPAAL
  – WCA
    • WCET analysis for JOP
  – SARTS
    • Schedulability on JOP
  – TetaJ
    • WCET analysis for SW JVM on Commodity HW
  – TetaSARTS
    • Schedulability analysis for SW JVM on Commodity HW and JOP
  – SymRT
    • Combines Symbolic execution and model-based timing analysis
• TetaJ
  – WCET analysis tool
  – Analysis at method level
  – Can be used interactively
  – Takes VM into account
  – Takes HW into account
TetaSARTS
Minepump example

https://www.youtube.com/watch?v=DbR42p5vU2M&feature=player_detailpage
Minepump example
Write once – run whereever possible

### Table 2. Using TetaSARTS with various execution environments.

<table>
<thead>
<tr>
<th>Execution Environment</th>
<th>Water Deadline</th>
<th>Methane Deadline</th>
<th>Schedulable</th>
</tr>
</thead>
<tbody>
<tr>
<td>HVM + AVR @ 10 MHz</td>
<td>12 ms</td>
<td>12 ms</td>
<td>✓</td>
</tr>
<tr>
<td>HVM + AVR @ 5 MHz</td>
<td>12 ms</td>
<td>12 ms</td>
<td>×</td>
</tr>
<tr>
<td>HVM + AVR @ 10 MHz</td>
<td>6 ms</td>
<td>6 ms</td>
<td>×</td>
</tr>
<tr>
<td>JOP @ 100 MHz</td>
<td>6 ms</td>
<td>6 ms</td>
<td>✓</td>
</tr>
<tr>
<td>JOP @ 100 MHz</td>
<td>12 μs</td>
<td>12 μs</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Table 1. Results obtained using TetaSARTS and SARTS.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Exec. Env.</th>
<th>Optimised</th>
<th>Analysis Time</th>
<th>Mem. Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minepump</td>
<td>HVM + AVR</td>
<td>✓</td>
<td>15h 25m 16s</td>
<td>17933 MB</td>
</tr>
<tr>
<td>Minepump</td>
<td>JOP</td>
<td>✓</td>
<td>7s</td>
<td>27 MB</td>
</tr>
<tr>
<td>Minepump</td>
<td>JOP</td>
<td>×</td>
<td>6m 18s</td>
<td>62 MB</td>
</tr>
<tr>
<td>SARTS Minepump</td>
<td>JOP</td>
<td>N/A</td>
<td>21s</td>
<td>42 MB</td>
</tr>
<tr>
<td>Simple System</td>
<td>HVM + AVR</td>
<td>✓</td>
<td>49s</td>
<td>168 MB</td>
</tr>
<tr>
<td>Simple System</td>
<td>HVM + AVR</td>
<td>×</td>
<td>22m 58s</td>
<td>238 MB</td>
</tr>
<tr>
<td>Simple System</td>
<td>JOP</td>
<td>✓</td>
<td>0.05s</td>
<td>7 MB</td>
</tr>
<tr>
<td>Simple System</td>
<td>JOP</td>
<td>×</td>
<td>0.5s</td>
<td>20 MB</td>
</tr>
</tbody>
</table>
Energy Optimized Applications

<table>
<thead>
<tr>
<th>Execution Environment</th>
<th>Clock Freq.</th>
<th>Schedulable</th>
</tr>
</thead>
<tbody>
<tr>
<td>HVM + AVR</td>
<td>10 MHz</td>
<td>✓</td>
</tr>
<tr>
<td>HVM + AVR</td>
<td>5 MHz</td>
<td>×</td>
</tr>
<tr>
<td>JOP</td>
<td>2 MHz</td>
<td>✓</td>
</tr>
<tr>
<td>JOP</td>
<td>1 MHz</td>
<td>×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RTSM</td>
<td>100 MHz</td>
<td>48.5 μs</td>
<td>4.0 ms</td>
</tr>
<tr>
<td>RTSM</td>
<td>60 MHz</td>
<td>80.8 μs</td>
<td>4.0 ms</td>
</tr>
<tr>
<td>Minepump</td>
<td>100 MHz</td>
<td>25.9 μs</td>
<td>2.0 ms</td>
</tr>
<tr>
<td>Minepump</td>
<td>10 MHz</td>
<td>259 μs</td>
<td>11.8 ms</td>
</tr>
</tbody>
</table>

Future work and speculations

- SCJ (JSR 302) has the notions of MissionSequenser
- When a mission terminates the getNextMission() method is invoked
  - This method can only be called by the infrastructure
  - It invokes the Missions cleanup() method
- The application can invoke requestSequenceTermination()
- The speed of CPU could be throttled during this transition!
Implementation and Analysis

• Not so easy to implement though
  – On the AVR Atmega speed is set through CLKPR register
  – However the C compiler assumes the CPU speed to be constants and defined by #define F_CPU
  – This constants is use by the _delay() function
  – However, a set of predefined speeds and associated delays could be compiled, at the cost of some code bloat

• Not so easy to analyze
  – Each mission in a missionSequence must be analyzed for schedulability to ensure schedulability of the application.
  – This include analysis of the transitions between missions
    • Initialize() and Cleanup() methods
  – Would require change to UPPAAL model for analysis
Battery discharge

Fig. A.2: Simulation of an ideal energy source (solid line, unit Coulombs) during a periodic load (dashed line, unit milliamperes). The energy runs out just before time 720.

Fig. A.3: Simulation of the same periodic load as in Fig. A.2 powered by a battery with the same capacity. The upper solid line represents the total charge in the battery, i.e., the sum of the bound charge \( b \), the upper dashed line and the available charge \( a \), the lower solid line. The alternating dotted/dashed line represents the height \( h_b \) scaled so it can be compared to the available charge as if it were \( h_a \) (in reality, \( h_a = \frac{1}{l} = 6a \) and \( h_b = \frac{1}{l} h_a = \frac{1}{6} h_a \), so we scale both by \( \frac{1}{6} \) so \( a \) and \( h_b \) coincide). Just after time 320 the available charge runs out and the system fails even though the battery has expended only a good of half its charge. A system with more lenient requirements would be able to wait for more charge to become available and thus exploit more of the total charge.

Fig. A.1: Mental model of the kinetic battery model.

Fig. B.5: Four example SOC profiles and their scores.
Trinity of tools, platform and programming model

Real-Time Execution Platform (JVM + HW)  
Timing Analysis Tools  
Real-Time Programming Model
Thank You