Safety-Critical Java for Low-End Embedded Platforms

Stephan E. Korsholm & Hans Søndergaard
VIA University College, Horsens, DK

Anders P. Ravn
CISS, Aalborg University, DK

JTRES October 2012
The Problem

• Low-End Industrial Platforms
  - KT4585 from Polycom
  - ATMega2560 from AVR
  - NEC-V850 e.g. used by Grundfos
  - Typical memory resources
    - 16 kB RAM, 256 kB ROM

• Safety-Critical Java impl. using RTSJ
  - Based on Java RTS (SUN)
  - Recommended Requirements
    - CPU system with 512 MB
    - Real-Time OS: Linux
Plan to Solve the Problem

Reduce each layer of the architecture

<table>
<thead>
<tr>
<th>Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCJ</td>
</tr>
<tr>
<td>RTSJ</td>
</tr>
<tr>
<td>JDK</td>
</tr>
<tr>
<td>VM</td>
</tr>
<tr>
<td>OS</td>
</tr>
</tbody>
</table>
Operating System

No Operating System
Instead:

- Hardware Objects for device control
- 1\textsuperscript{st} level interrupt handling in Java space
- Minimal native layer for context switch between tasks
VM:
Hardware near Virtual Machine (HVM)

• Lean
  – Java-to-C compiler with embedded interpreter
  – Program specialization
    – Classes & methods
    – Bytecode selection

• No dependencies on external libraries
• Portable
  – Strict ANSI-C
  – All usual C compilers can be used
  – Simple build procedure
JDK

• No special JDK required
  - Uses Java 1.6 (Other JDKs supported as well)
  - Reduced through program specialization
• Dependency leaks
  - System.out.println leaks, but
  - Collection classes (e.g. ArrayList) do not
• A bare metal implementation
  – No RTSJ
  – The VM interface
Scoped Memory

Scoped Memory:

- base
- free

Backing store

size
Scoped Memory

Java:

```java
public class AllocationArea {
    protected int base;
    protected int size;
    protected int free;

    @IcecapCVar
    private static int HVMbase;
    @IcecapCVar
    private static int HVMfree;
    @IcecapCVar
    private static int HVMsize;

    @IcecapCompileMe
    public static void switchAllocationArea(AllocationArea newScope,
        AllocationArea oldScope) {
        oldScope.base = HVMbase;
        oldScope.free = HVMfree;
        oldScope.size = HVMsize;

        HVMbase = newScope.base;
        HVMfree = newScope.free;
        HVMsize = newScope.size;
    }
}
```

C:

```c
unsigned char* HVMbase;
uint32 HVMfree;
uint32 HVMsize;
```
Scheduling

- Context switch through the layers

<table>
<thead>
<tr>
<th>C</th>
<th>C - Assembler</th>
<th>Java</th>
<th>Java</th>
</tr>
</thead>
</table>

**Diagram:**

- **HVM**
  - `interrupt !!!`
  - `store context`
  - `stackPointer = sp`
  - `sp = schedulerStack`

- **Assembler Stub**
  - `interrupt !!!`
  - `store context`
  - `stackPointer = sp`
  - `sp = schedulerStack`
  - `return !!!`

- **Process Sequencer**
  - `interrupt()`
  - `process.sp = stackPointer`
  - `return`
  - `process = getNextProcess()`

- **Process Scheduler**
  - `process = getNextProcess()`
Real-Time Clock

• Platform specific
  – E.g. KT4585,
    ```java
    @IcecapCVar
    private static int systemTick;
    ```
  – ATMega2560
    – Hardware clock
      – Configured using Hardware Objects
      – Tick interrupt handled in Java
Evaluation

• SCJ Level 1:
  1 Mission, 3 Handlers, KT4585

  - ROM: 35 kB
  - RAM: 10 kB

<table>
<thead>
<tr>
<th>SCJ related</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘Main’ stack</td>
<td>1024</td>
</tr>
<tr>
<td>Mission sequencer stack</td>
<td>1024</td>
</tr>
<tr>
<td>Scheduler stack</td>
<td>1024</td>
</tr>
<tr>
<td>Idle task stack</td>
<td>256</td>
</tr>
<tr>
<td>3xHandler stack</td>
<td>3072</td>
</tr>
<tr>
<td>Immortal memory</td>
<td>757</td>
</tr>
<tr>
<td>Mission memory</td>
<td>1042</td>
</tr>
<tr>
<td>3xHandler memory</td>
<td>3x64 = 192</td>
</tr>
</tbody>
</table>

**HVM infrastructure**

| Various                            | 959   |
| Class fields                        | 557   |
| **Total**                           | 9907  |
Evaluation

- **MiniCDj, ATMega2560**
  - ROM

<table>
<thead>
<tr>
<th></th>
<th>Classes</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.*</td>
<td>46</td>
<td>171</td>
</tr>
<tr>
<td>java.util.*</td>
<td>10</td>
<td>42</td>
</tr>
<tr>
<td>javax.safetycritical.*</td>
<td>46</td>
<td>185</td>
</tr>
<tr>
<td>minicdj.*</td>
<td>49</td>
<td>216</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>151</td>
<td>614</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>miniCDj benchmark</th>
<th>ROM (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mostly interpreted</td>
<td>94682</td>
</tr>
<tr>
<td>Compilation only</td>
<td>282166</td>
</tr>
</tbody>
</table>

- RAM, more than 300 kB
Related JVMs

• JamaicaVM
  – Hard real-time execution guarantees
  – Real-time GC
  – SCJ on top of RTSJ
  – High-end embedded platforms

• FijiVM
  – Efficient Java-to-C compiler
  – Real-time GC
  – SCJ Level 0 with native function layer
  – High-end embedded platforms

• KESO VM
  – Lean VM. Efficient Java-to-C compiler
  – GC support
  – HVM SCJ ported to KESO?
  – Low-end embedded platforms
Conclusion

A SCJ Level 0 + 1 implementation for low-end platforms by means of:

- A bare metal implementation of SCJ using a VMInterface
- No special JDK required
- A lean and portable HVM, no library dependencies
- Hardware near features like Hardware Objects

Typical memory resources

16 kB RAM, 256 kB ROM
Are we happy now?

• Ensure SCJ compatibility
• Development environment
• Improve Java SCJ infrastructure
• Learn efficient compilation from Fiji