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

Well … ADA – 10th on the list of most wanted skills

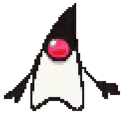
  - Difficult to find new skilled programmers
    - Jackson Structured Development (1975) still widely used
    - EE Times calling for re-introducing C programming at US Uni

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

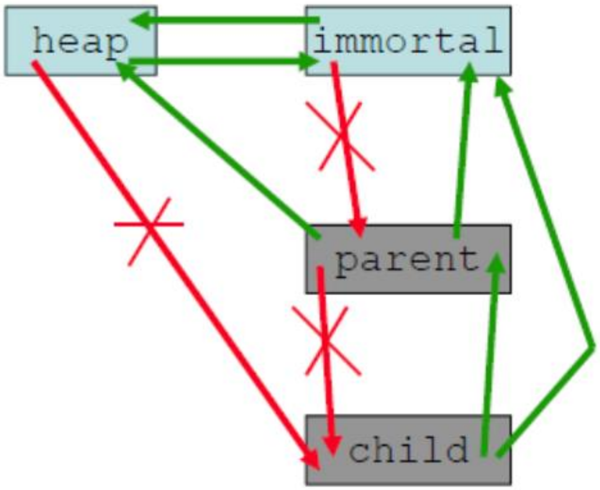
http://jaxenter.com/how-many-java-developers-are-there-10462.html

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

```java
PeriodicMethaneDetection
  methaneDetection =
    new PeriodicMethaneDetection(
      new PriorityParameters(METHANE_DETECTION_PRIORITY
           ),
      new PeriodicParameters(
          new RelativeTime(0, 0),
          new RelativeTime(PERIODIC_GAS_PERIOD, 0)),
      new StorageParameters(
          SCOPED_MEMORY_BACKING_STORE_SIZE,
          NATIVE_STACK_SIZE,
          JAVA_STACK_SIZE),
      methaneSensor,
      waterpumpActuator);

methaneDetection.register();
```

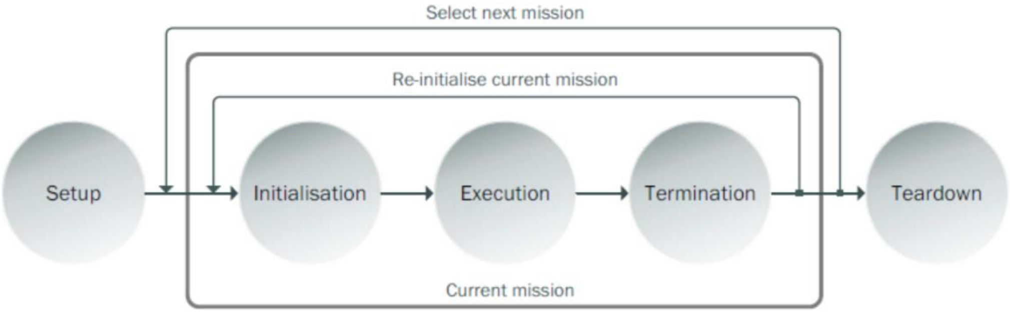
Listing 1: An SCJ handler for methane level [16].

```java
public void handleEvent() {
  waterpumpActuator.emergencyStop(
    methaneSensor.isCriticalMethaneLevelReached()
  );
}
```

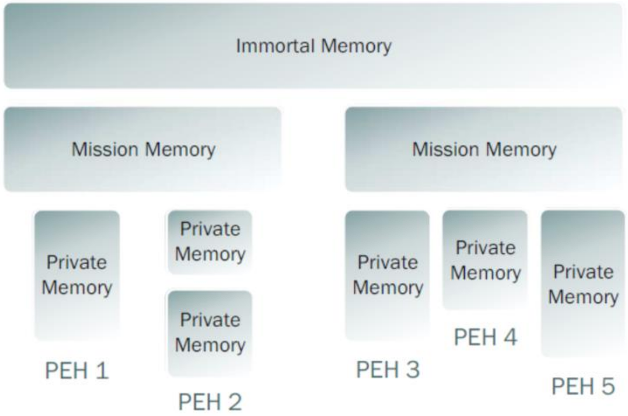
Listing 2: Detecting the methane level [16].

SCJ

- Only RealtimeThreads are allowed
- Notions of missions and handlers



- No heap objects/ no GC
- Restricted use of scopes



Predicatble JVM

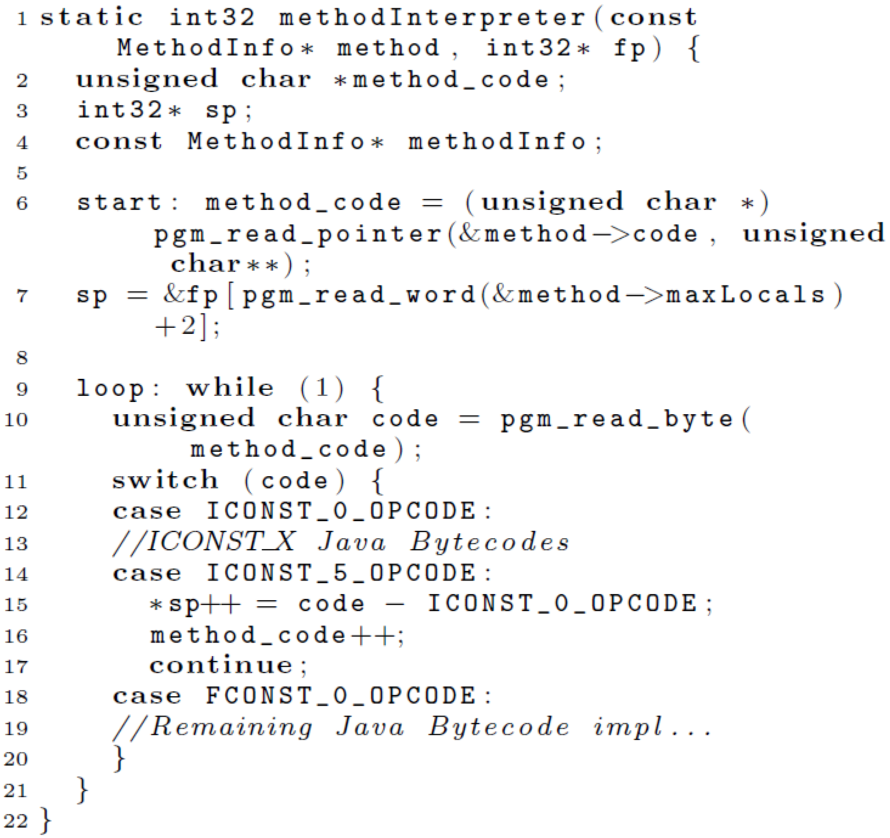
- JOP
  - Java Optimized Processor
  - JVM in Hardware (FPGA)

- 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 compilering
  - 1st level interupt 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

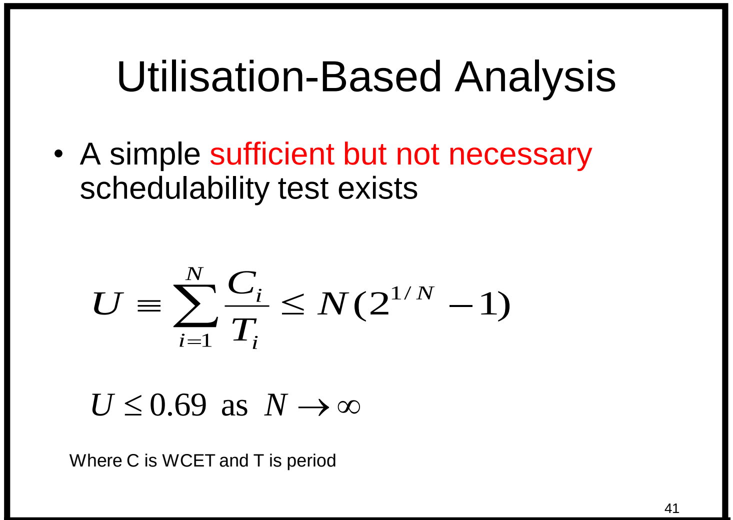


```
static int32 methodInterpreter(const
      MethodInfo* method, int32* fp) {
   unsigned char *method_code;
   int32* sp;
   const MethodInfo* methodInfo;

   start: method_code = (unsigned char *)
         pgm_read_pointer(&method->code, unsigned
          char**);
   sp = &fp[pgm_read_word(&method->maxLocals)
         +2];

   loop: while (1) {
      unsigned char code = pgm_read_byte(
            method_code);
      switch (code) {
      case ICONST_0_OPCODE:
      //ICONST_X Java Bytecodes
      case ICONST_5_OPCODE:
         *sp++ = code - ICONST_0_OPCODE;
         method_code++;
         continue;
      case FCONST_0_OPCODE:
      //Remaining Java Bytecode impl...
      }
   }
}
```

What about Time Analysis?



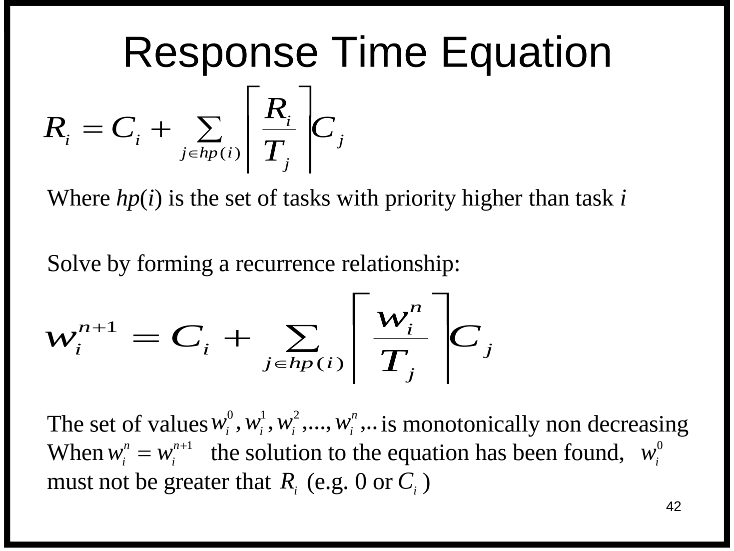
Utilisation-Based Analysis

- A simple sufficient but not necessary schedulability test exists

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

$$U \leq 0.69 \text{ as } N \rightarrow \infty$$

Where C is WCET and T is period



Response Time Equation

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

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

Solve by forming a recurrence relationship:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

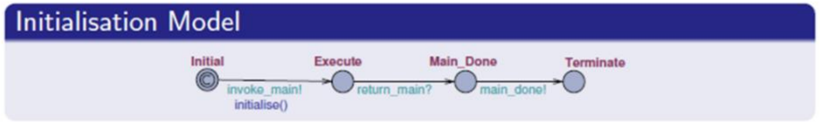
The set of values $w_i^0, w_i^1, w_i^2, \ldots, w_i^n, \ldots$ is monotonically non decreasing
When $w_i^n = w_i^{n+1}$ the solution to the equation has been found, w_i^0 must not be greater that R_i (e.g. 0 or C_i)

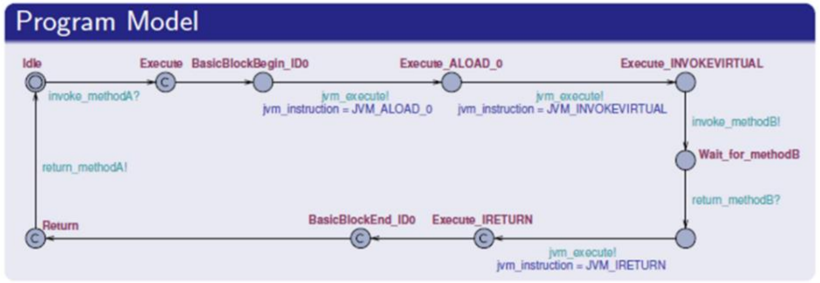
- Traditional approaches to analysis of RT systems are hard and conservative
- Very difficult to use with Java because of JVM (and Object Orientedness)

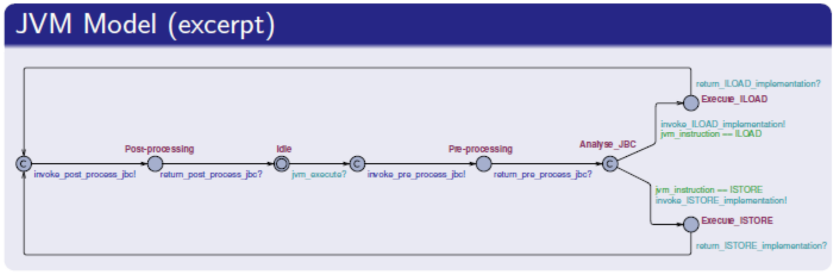
Model based Analysis

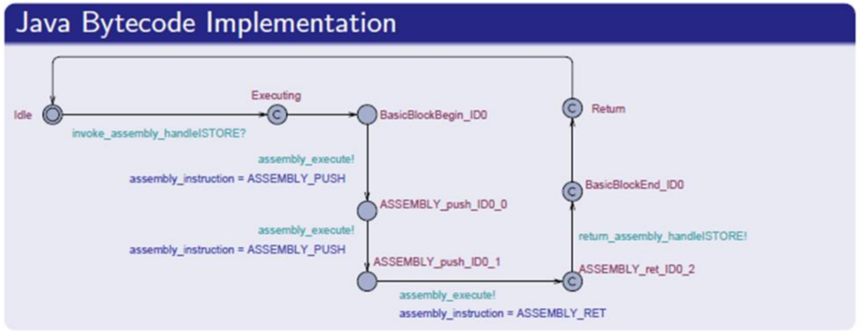
- Translate timing analysis problems into analysis of properties of timed Automatas
  - TIMES
    - Model based schedulability tool based on UPPAAL
  - WCA
    - WCET analysis for JOP
  - SARTS
    - Schedulability on JOP
  - TetaJ
    - WCET analysis for SW JVM on Commodity HW
  - TetaSARTS
    - Schedulability analysis for SW JVM on Commodity HW and JOP
  - SymRT
    - Combines Symbolic executioon and modelbased timing analysis

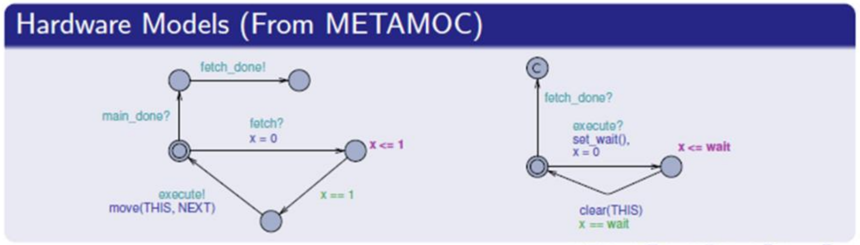
- TetaJ
  - WCET analysis tool
  - Analysis at method level
  - Can be used interactively
  - Takes VM into account
  - Takes HW into account


Initialisation Model

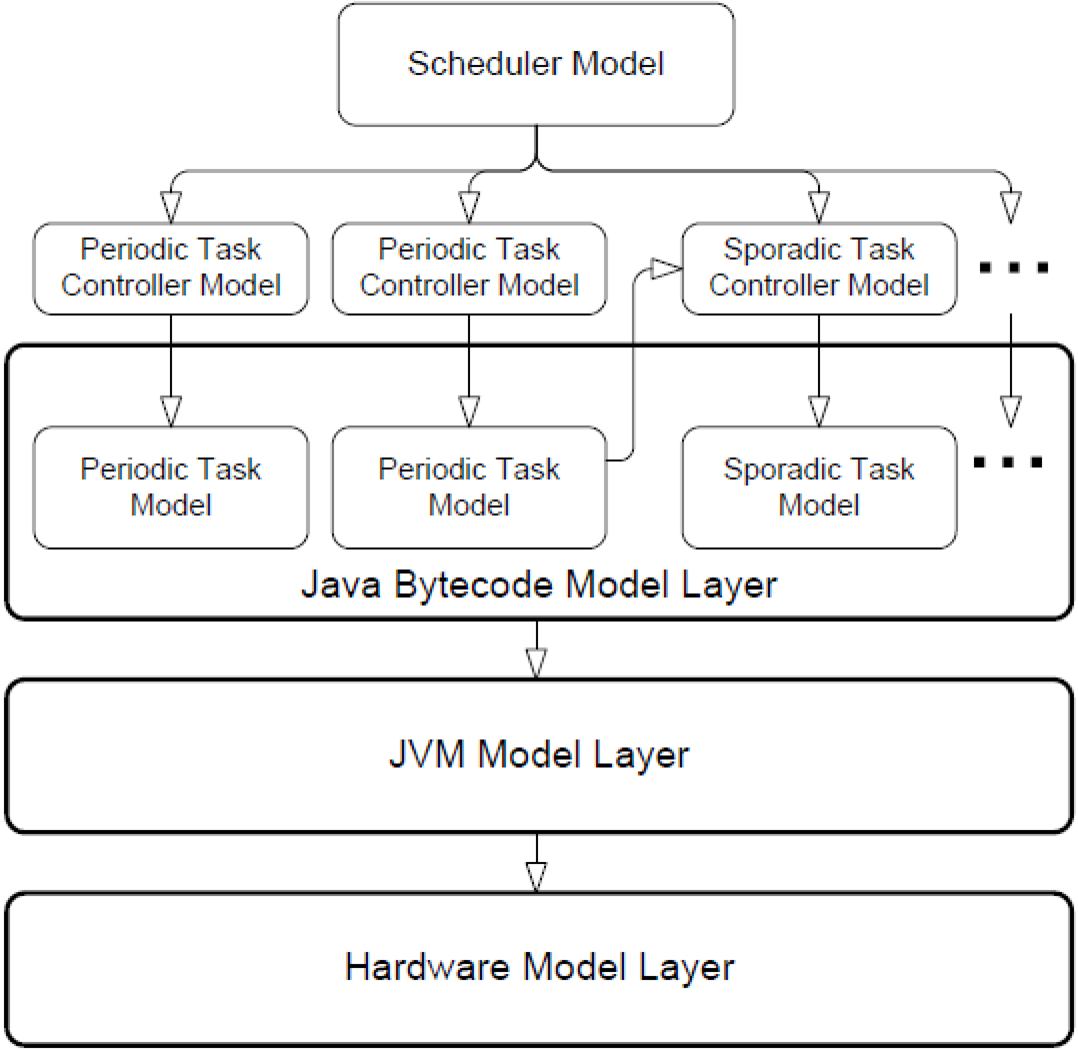

Program Model


JVM Model (excerpt)

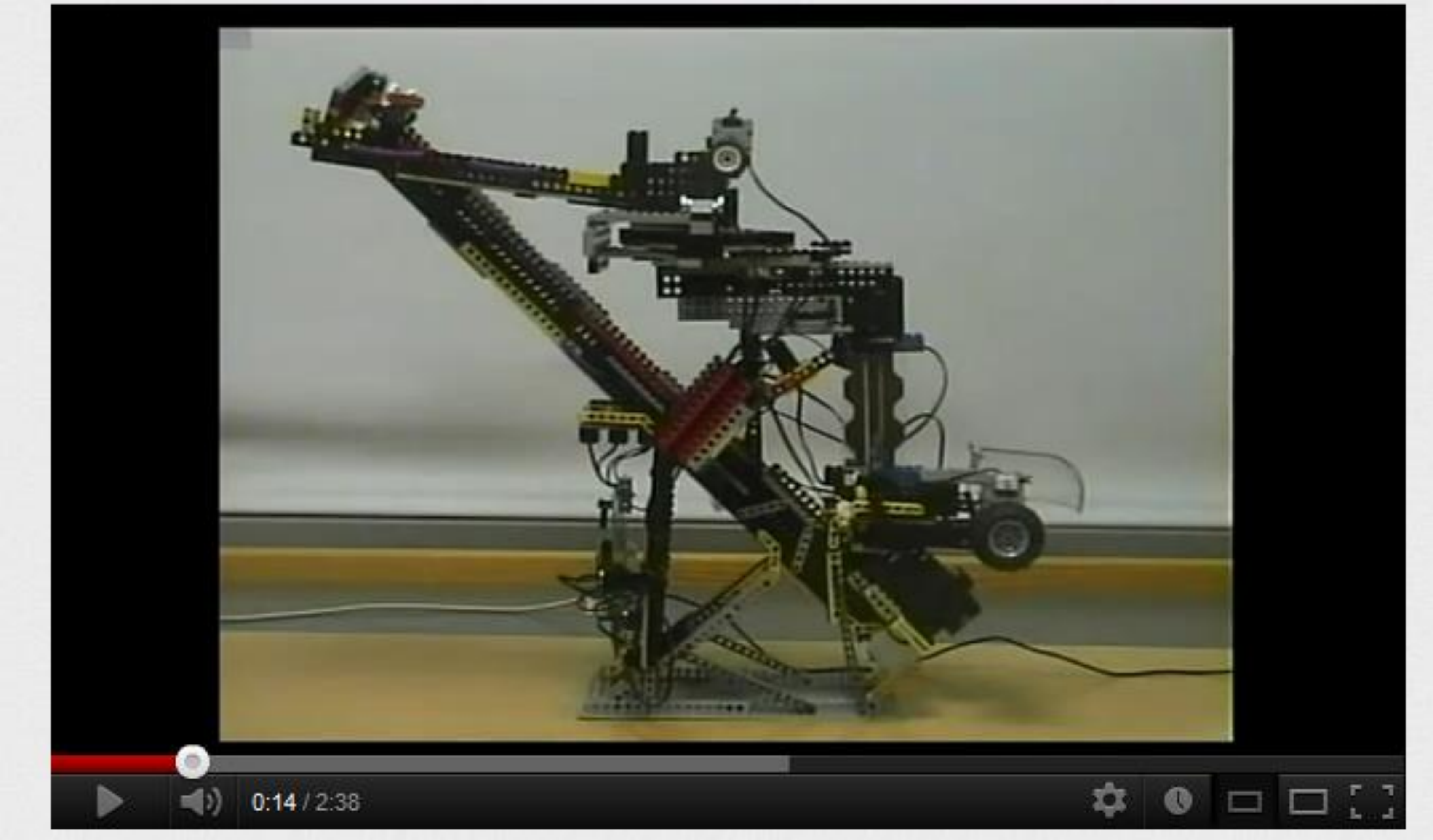

Java Bytecode Implementation


Hardware Models (From METAMOC)

TetaSARTS

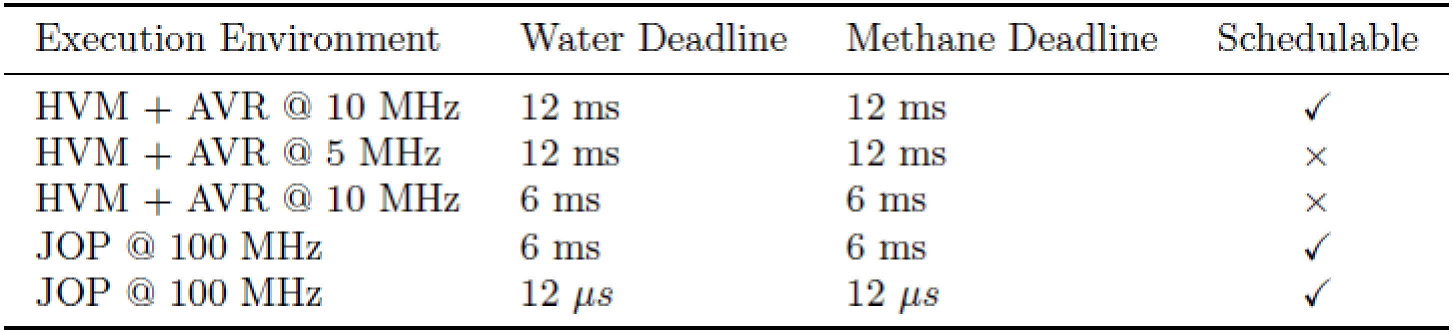

Scheduler Model
Periodic Task Controller Model
Periodic Task Controller Model
Sporadic Task Controller Model
. . .
Periodic Task Model
Periodic Task Model
Sporadic Task Model
. . .
Java Bytecode Model Layer
JVM Model Layer
Hardware Model Layer

Minepump example



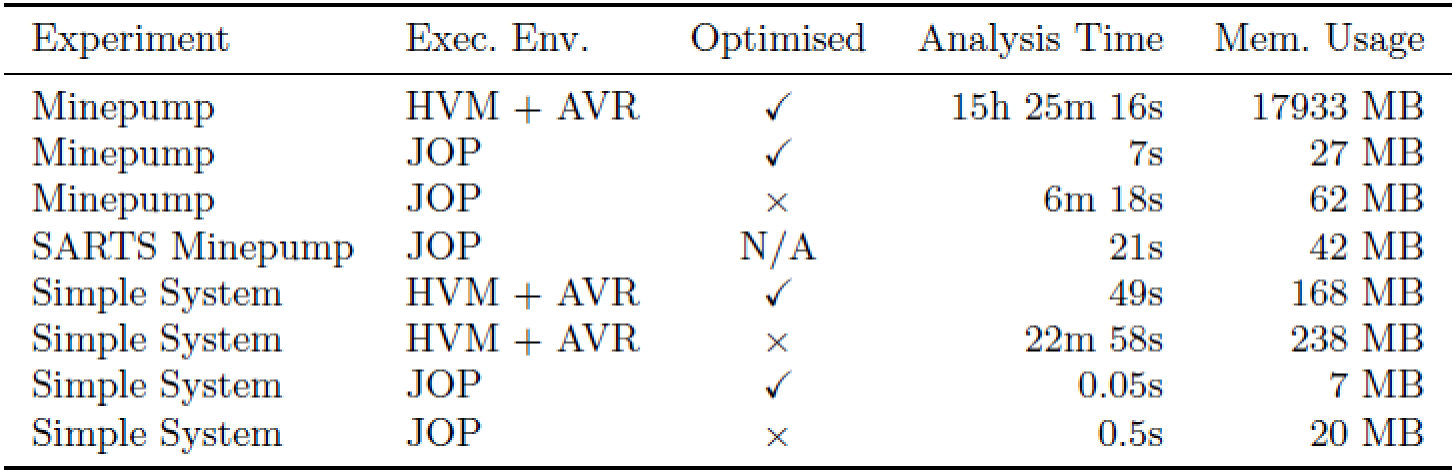
https://www.youtube.com/watch?v=DbR42p5vU2M&feature=player_detailpage

Minepump example
Write once – run whereever possible



Execution Environment	Water Deadline	Methane Deadline	Schedulable
HVM + AVR @ 10 MHz	12 ms	12 ms	✓
HVM + AVR @ 5 MHz	12 ms	12 ms	×
HVM + AVR @ 10 MHz	6 ms	6 ms	×
JOP @ 100 MHz	6 ms	6 ms	✓
JOP @ 100 MHz	12 μs	12 μs	✓

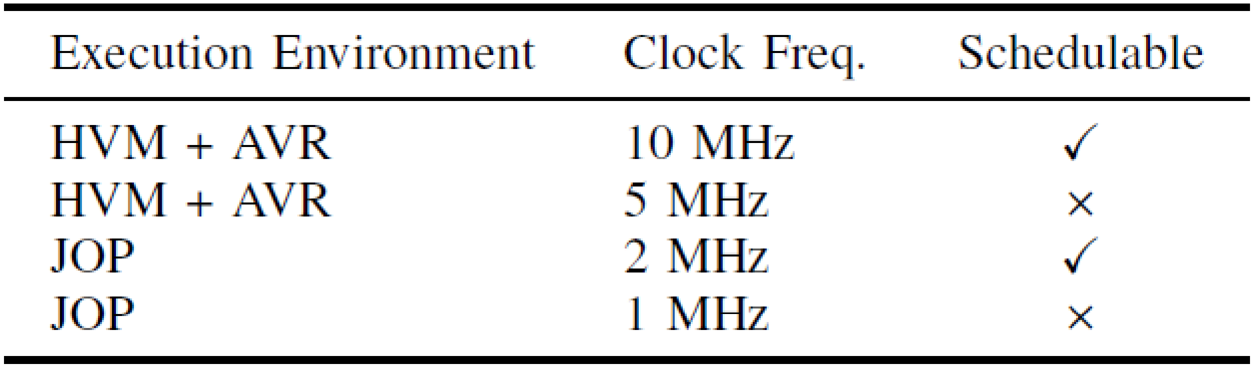
Table 2. Using TetaSARTS with various execution environments.



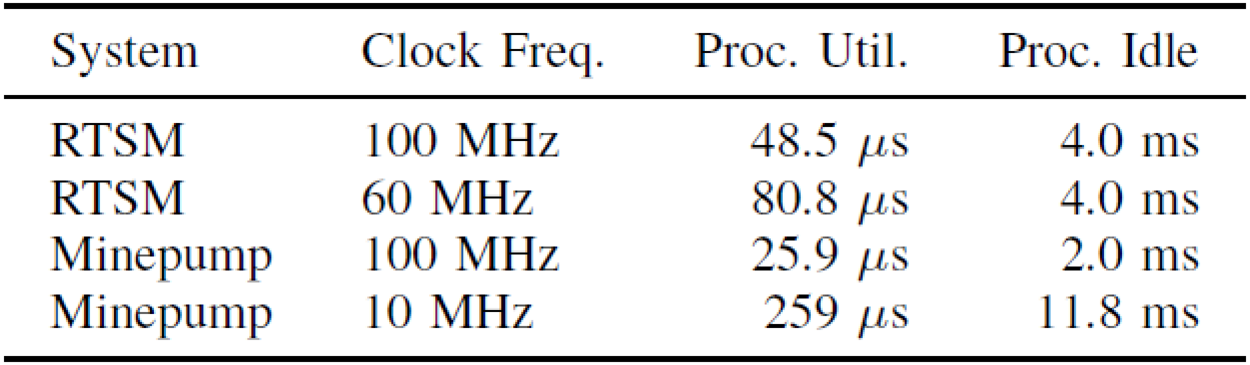
Experiment	Exec. Env.	Optimised	Analysis Time	Mem. Usage
Minepump	HVM + AVR	✓	15h 25m 16s	17933 MB
Minepump	JOP	✓	7s	27 MB
Minepump	JOP	×	6m 18s	62 MB
SARTS Minepump	JOP	N/A	21s	42 MB
Simple System	HVM + AVR	✓	49s	168 MB
Simple System	HVM + AVR	×	22m 58s	238 MB
Simple System	JOP	✓	0.05s	7 MB
Simple System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

Table 1. Results obtained using TetaSARTS and SARTS.

Energy Optimized Applications



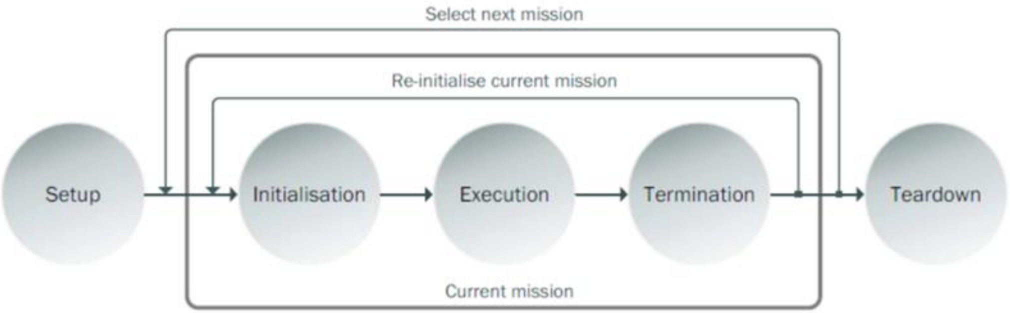
Execution Environment	Clock Freq.	Schedulable
HVM + AVR	10 MHz	✓
HVM + AVR	5 MHz	×
JOP	2 MHz	✓
JOP	1 MHz	×



System	Clock Freq.	Proc. Util.	Proc. Idle
RTSM	100 MHz	48.5 μs	4.0 ms
RTSM	60 MHz	80.8 μs	4.0 ms
Minepump	100 MHz	25.9 μs	2.0 ms
Minepump	10 MHz	259 μs	11.8 ms

Luckow, Kasper Søe, Thomas Bøgholm, and Bent Thomsen. "Supporting development of energy-optimised java real-time systems using tetasarts." Wip Proceedings of the 19th Ieee Real-time and Embedded Technology and Applications Symposium. 2013.

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

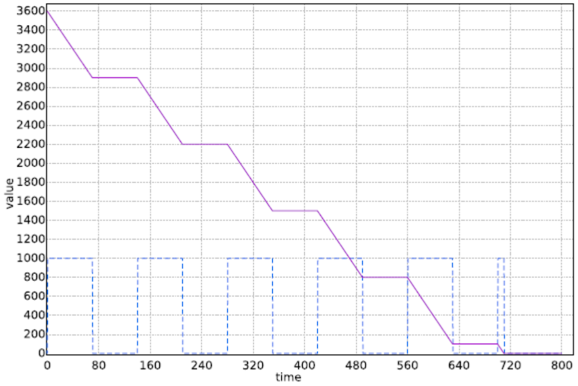


Fig. A.2: Simulation of an ideal energy source (solid line, unit Coulombs) during a periodic load (dashed line, unit milliamperes). The energy runs out just before time 720.

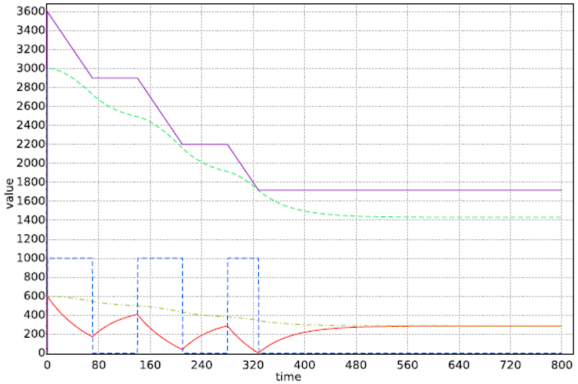


Fig. A.3: Simulation of the same periodic load as in Fig. A.2 powered by a battery with the same capacity. The upper solid line represents the total charge in the battery, i.e., the sum of the bound charge (*b*, the upper dashed line) and the available charge (*a*, the lower solid line). The alternating 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, $h_a = \frac{a}{c} = 6a$ and $h_b = \frac{b}{1-c} = \frac{6b}{5}$, so we scale both by $\frac{1}{6}$ so a and h_a 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.

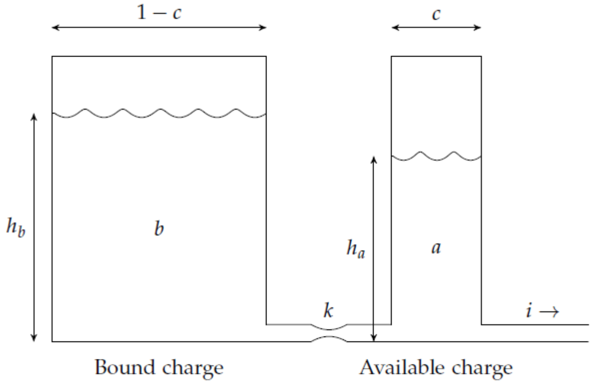


Fig. A.1: Mental model of the kinetic battery model

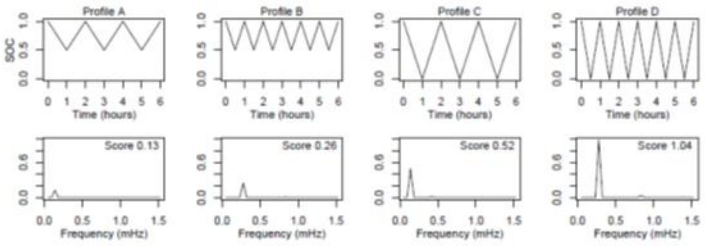
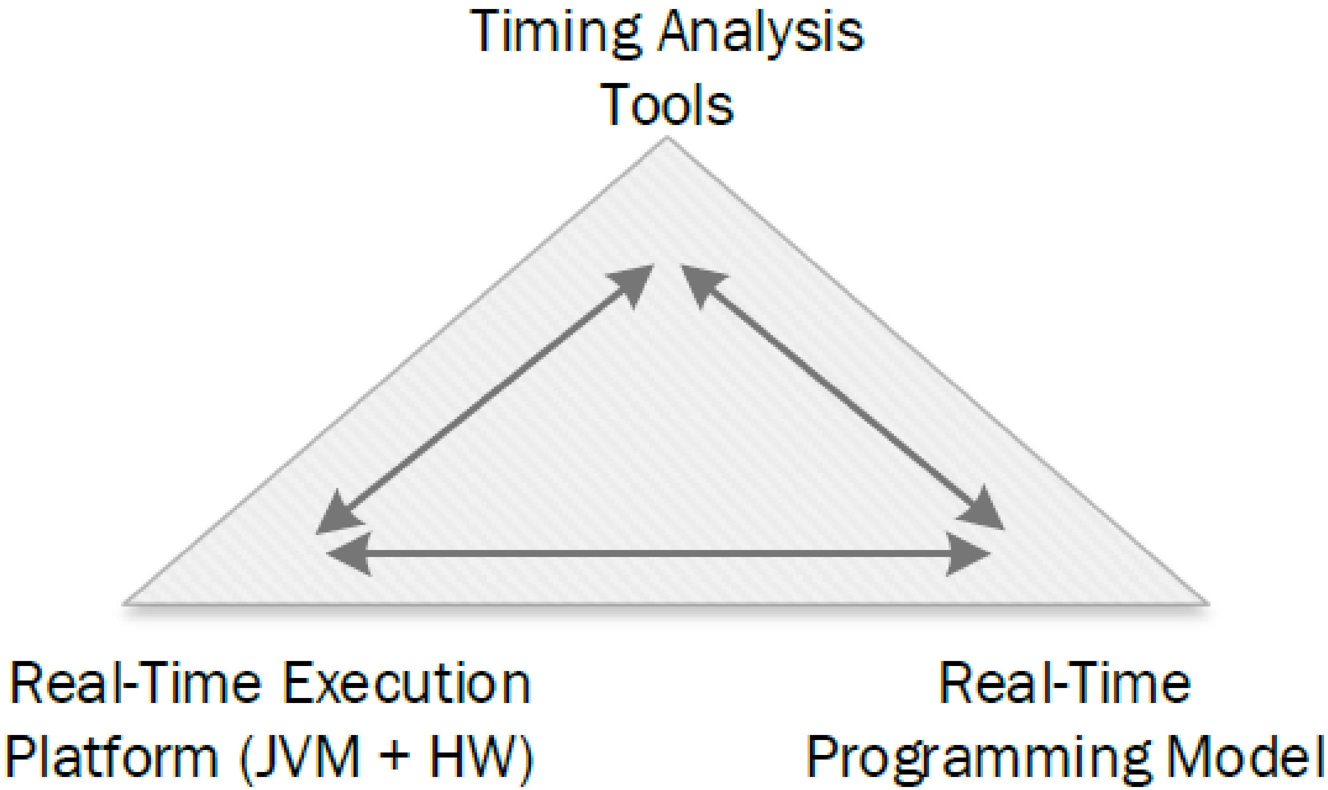


Fig. B.5: Four example SOC profiles and their scores.

Trinity of tools, platform and programming model



Timing Analysis
Tools

Real-Time Execution
Platform (JVM + HW)

Real-Time
Programming Model

Thank You