HVM_{TP}: A Time Predictable and Portable Java Virtual Machine for Hard Real-Time Embedded Systems

JTRES 2014

Kasper Søe Luckow[1] Bent Thomsen[1] Stephan Erbs Korsholm[2]

[1]Department of Computer Science
Aalborg University
Denmark

[2]VIA University College
Horsens
Denmark

Motivation

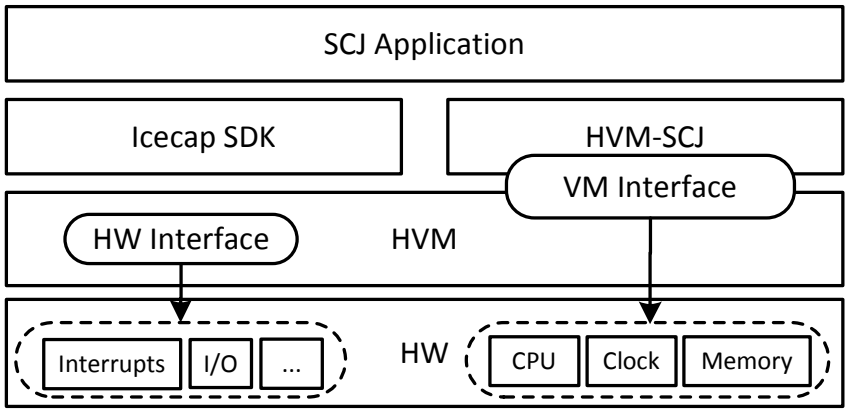
- WCET analysis necessitates that the temporal behavior of the execution environment can be analysed
- Java Optimized Processor[1]
  - Hardware Java Virtual Machine
  - Execution times of the Java Bytecodes can be predicted
- This work addresses:
  - Software Java Virtual Machine
  - (Commodity) embedded hardware

[1] http://www.jopdesign.com/

Contributions

- Time-predictable, software Java Virtual Machine
  - Temporal behavior of Java Bytecodes can be modeled and analysed
  - HVM_{TP}
- Accompanying tool support
  - $TETASARTS_{JVM}$
  - $TETASARTS_{TS}$

HVM_{TP} at a Glance



- Based on the Hardware near Virtual Machine (HVM)[2]
- Java-to-C compiler
  - ICECAP-TOOLS
  - Supports (iterative) interpretation
  - Ahead-Of-Time compilation
  - Tailors and optimises HVM for the hosted program
- Requirements: 256 kB flash and 20 kB RAM
- Self-contained (runs on bare metal), ANSI C
  - ARM, AVR, x86, cr16c, ...

[2] http://icelab.dk/

Time-Predictability of HVM_{TP}

- Time-predictability is possible by
  - Harnessing the SCJ programming model
    - HVM_{TP} implements SCJ Level 1
  - Harnessing information obtained statically (ICECAP-TOOLS)
- This work focuses on the iterative interpreter (constant time stages)
- Many Java Bytecodes from HVM are time-predictable
- Re-design comprises
  - Object allocation
  - Exceptions
  - Method invocation
  - Type checking of reference types
  - . . . and a few others

Object Allocation

- HVM performs zeroing at allocation time
  - Linear time operation
- In HVM_{TP} the heap structure is zeroed at Safelet initialisation
- Zeroing happens when scoped memory is exited
- Performed in Java space using native variables
  - Variables in the HVM accessible in Java space

Exceptions

- SCJ permits exception objects to be pre-allocated before entering a time critical phase
- ICECAP-TOOLS approximates the set of exceptions that can be thrown
  - E.g. athrow and idiv
- Exception handler is located in the call stack (linear time)
- Maximum call stack depth is estimated by ICECAP-TOOLS
  - Reconstructs call graph
  - Recursion is not allowed

Method Invocation

```
1 case INVOKEVIRTUAL_OPCODE: {
2   const MethodInfo* mInfo;
3   signed short excep;
4   mInfo = findMethodInfo(&sp[top], &
          method_code[pc]);
5   excep = methodInterpreter(mInfo, &sp[top]);
6   //...
7 }
```

Listing 1 : Original `invokevirtual`.

```
1 case INVOKEVIRTUAL_OPCODE: {
2   //...
3   unsigned short pc = method_code - (
          unsigned char *)
          pgm_read_pointer(&method->code,
           unsigned char**);
4   fp = pushStackFrame(mInfo, method, pc, fp, sp);
5   method = mInfo;
6   //...
7 }
```

Listing 2 : Using stack frames.

- The HVM employed recursion
  - Difficult to analyse and model
- HVM_{TP} implements a call stack
- HVM_{TP} attempts to devirtualise call sites (using VTA)
- Method dispatch at virtual call sites (`invokevirtual` and `invokeinterface`)
  - Treated (almost) equally for simplicity
  - Consult method tables of *objectref*'s class and superclasses
  - Bounded by maximum height of class hierarchy
  - (Obvious) future work: generate dispatch table (`invokevirtual`)

Type Checking Reference Types

- The HVM iteratively consults *objectref*'s class and superclasses
- HVM_{TP} exploits availability of the class hierarchy at HVM_{TP} construction time
- A bit matrix is constructed with entries denoting the type compatibility of (x, y)

Tool Support

- Tools for HVM_{TP}:
  - $TETASARTS_{JVM}$
  - $TETASARTS_{TS}$

TETASARTS$_{JVM}$

```
 1 case I2L_OPCODE: {
 2 #if defined(INSTRUMENT)
 3     BEGIN_JBC(I2L_OP);
 4 #endif
 5     int32 lsb = *(--sp);
 6     if(lsb < 0) {
 7         *sp++ = -1;
 8     } else {
 9         *sp++ = 0x0;
10     }
11     *sp++ = lsb;
12     method_code++;
13 #if defined(INSTRUMENT)
14     END_JBC(I2L_OP);
15 #endif
16 }
```

Listing 3 : `i2l`.

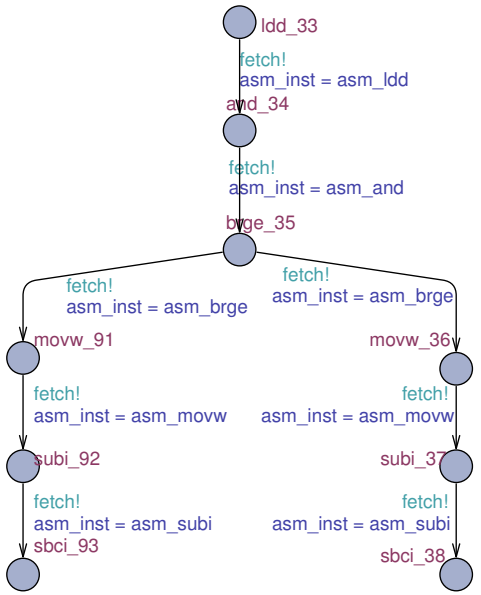


Figure : Excerpt of TA for `i2l`.

- Generates a JVM Timing Model
- Timed Automata (TA) (UPPAAL[3] model checker)
- Executable is instrumented
- Loop bounds provided comment-style
  - All bounds are provided by ICECAP-TOOLS
- Reconstructs Control-Flow Graph and translates to TA

[3] http://www.uppaal.org

TETASARTS$_{JVM}$ Cont'd

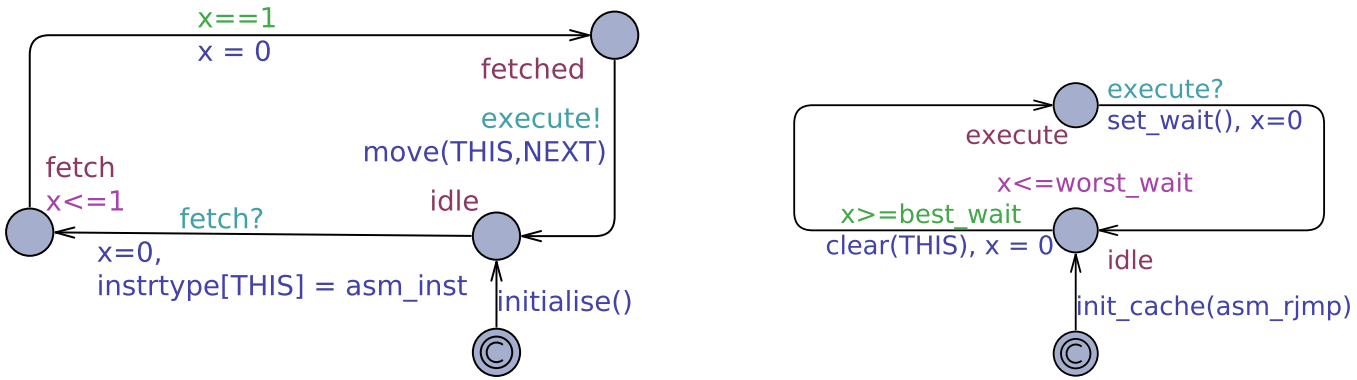


Figure : Fetch and execute TA from METAMOC[4].

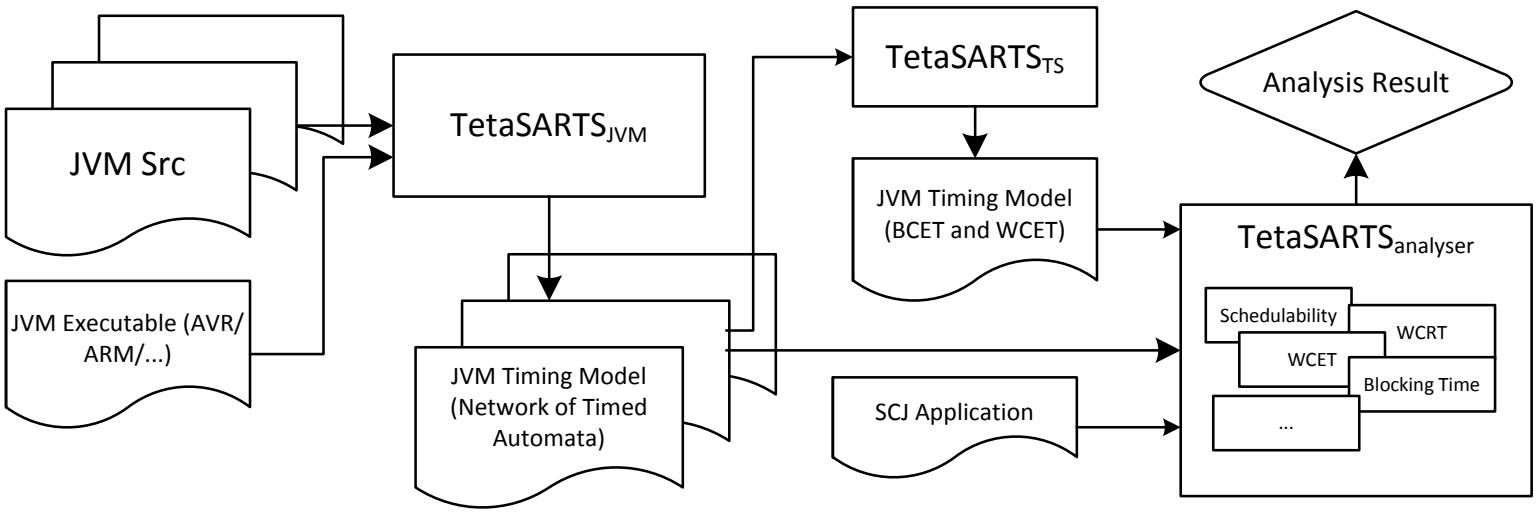
- Composition with HW TA yields the JVM Timing Model
- Verification of properties (TCTL)
  - E.g. estimate execution times of the Java Bytecodes

[4] http://metamoc.dk/

TETASARTS$_{TS}$

- TETASARTS$_{TS}$ generates a timing scheme from the JVM Timing Model
- A timing scheme captures an abstract timing model of the execution environment
  - $[BCET_i, WCET_i]$ for instruction i

The Big Picture



Results

- Constructing complete JVM Timing Model: 16*s*
- Generating a timing scheme for **all** Java Bytecodes:
  - $\sim$ 4.5*hours* (without exception handling)
  - $\sim$ 5*days* (with exception handling)
- Application-dependent Java Bytecodes:
  - Only these must be re-analysed if the program is modified
  - 13 (without exception handling)
  - 47 (with exception handling)
- In reality, only a subset of the Java Bytecodes are used
  - The Minepump uses 49 distinct Bytecodes $\rightarrow$ 5*s* (JVM Timing Model) and 6*m* (timing scheme)
  - Only two Java Bytecodes are application-dependent (`invokevirtual` and `invokeinterface`)

Results Cont'd

Bytecode	TETASARTS$_{TS}$ BCET	WCET	Measured Avg	Low	High
i2l	129	136	130	130	130
aload_*	79	79	79	79	79
new	469	1715	1568	1568	1568
ireturn	505	1080	893	865	976
invokespecial	501	977	710	639	772
iinc	191	194	192	192	192

Times are represented in clock cycles.

- Simulation on Atmel AVR
- Measurements obtained from Atmel Studio 6
- Safety: $BCET \leq Low$ and $High \leq WCET$

Future Work

- Further improve HVM_{TP}
  - E.g. `invokevirtual`
- Improve precision of JVM Timing Model
  - CFG contains both feasible and infeasible execution paths
  - Symbolic execution
- Evaluate analysis approach on other (and more complex) hardware models