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

Kasper Søe Luckow, Bent Thomsen
Department of Computer Science,
Aalborg University, Denmark
{luckow,bt}@cs.aau.dk

Stephan Erbs Korsholm
VIA University College,
Horsens, Denmark
sek@viauc.dk

ABSTRACT

We present HVM$_{TP}$, a time predictable and portable Java Virtual Machine (JVM) implementation with applications in resource-constrained, hard real-time embedded systems, which implements the Safety Critical Java (SCJ) Level 1 specification.

Time predictability is achieved by a combination of time predictable algorithms, exploiting the programming model of the SCJ profile, and harnessing static knowledge of the hosted SCJ system.

This paper presents HVM$_{TP}$ in terms of its design and capabilities, and demonstrates how a complete timing model of the JVM represented as a Network of Timed Automata can be obtained using the tool TetaSARTS$_{JVM}$. The timing model readily integrates with the rest of the TetaSARTS tool-set for temporal verification of SCJ systems. We will also show how a complete timing scheme in terms of safe Worst Case Execution Times and Best Case Execution Times of the Java Bytecodes can be derived from the model.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Object-oriented languages; D.3.4 [**Processors**]: Run-time environments; C.3 [**Special-Purpose and Application-Based**]: Real-time and embedded systems

General Terms

Verification, Languages, Experimentation

Keywords

Java virtual machine, Real-time Java, Time predictability, Model checking

JTRES'14, October 13 - 14 2014, Niagara Falls, NY, USA

http://dx.doi.org/10.1145/2661020.2661022

1. INTRODUCTION

Java is widely accepted as programming language for many application domains and is increasingly seeing popularity in educational institutions. Due to its high popularity, pushing the use of Java into application domains for which Java was not originally intended, has fostered large research communities. One of these is the embedded real-time systems domain for which timeliness of real-time tasks is an imperative. However, the original system model of Java does not accommodate the requirements and constraints dictated by embedded real-time systems for reasons such as the lack of high-resolution real-time clocks, insufficiently tight thread semantics, and, most notably, memory management traditionally handled by a garbage collector whose execution and execution time are highly unpredictable.

To accommodate this, much research has been devoted to develop appropriate programming models in Java facilitating real-time systems development such as the Real-Time Specification for Java (RTSJ) [10] and the Safety Critical Java (SCJ) [20] profile. The specification of the latter is still a draft, but it is reasonable to assume that the development towards applicable real-time programming models for Java has come a long way.

Having an appropriate programming model is a necessary, but not the only, step towards use in hard real-time embedded systems. Equally important is that the underlying execution environment exhibits temporally predictable behavior to allow reasoning about timeliness. Many embedded hardware platforms are relatively simple and do not include the same amount of technologies for improving average case execution time as is the case on most desktop hardware platforms. As a result, the temporal behavior of the machine instructions of embedded microcontrollers can be modelled and can be considered time predictable. Hence, the Java Virtual Machine (JVM) is the remaining component in enabling temporal verification of hard real-time systems written in Java.

Evidently, the JVM adds to the complexity in performing static analyses of Java compared to C, which traditionally can be executed on bare metal. To mitigate this complexity and direct attention towards the programming model, efforts have been made in establishing a similar execution environment as that for C, that is, removing the (traditionally) software implemented JVM from the equation. Particularly, aJile Systems [2] and the Java Optimized Proces-

sor (JOP) project [27] have both implemented a JVM in hardware thereby effectively achieving native execution of the resulting Java bytecode. The JOP has documented and predictable execution times of the Java bytecodes. A problem with hardware implemented JVMs is that they necessitate special-purpose hardware such as FPGAs which may be a costly solution especially considering that embedded systems are sometimes produced in large quantities. A more general solution is to allow real-time Java systems to be executed on common embedded hardware used in industry, such as ARM and AVR while remaining amenable to static analyses. This necessarily demands that the JVM is amenable to static analyses as well, which is difficult, since the JVM specification [19] allows for high flexibility; it emphasises on *what* a JVM implementation must do i.e. the semantics of the bytecode instructions, but it does not specify *how*.

The contribution of this paper is threefold: Our first contribution is to demonstrate how a JVM implementation, the Hardware near Virtual Machine (HVM), originally intended for embedded systems can be redesigned to exhibit complete time predictable behavior. The redesign is possible by combining statically inferable knowledge about the hosting application, the SCJ programming model, and time predictable algorithms with bounded execution times. We denote this time predictable JVM implementation HVM_{TP}, and based on this, our second contribution is the derivation of a complete timing model that captures the temporal behavior of the JVM and allows reasoning on Timed Computation Tree Logic (TCTL) properties. The timing model is derived using the complementary tool, $\text{TetaSARTS}_{\text{JVM}}$, a model-based and highly flexible tool for safe timing analysis of JVM executables part of the TetaSARTS collection of timing analysis tools [21, 22, 23]. Our third contribution, is the $\text{TetaSARTS}_{\text{TS}}$ tool, which processes the timing model to derive safe Worst Case Execution Times (WCET) and Best Case Execution Times (BCET) of all supported instructions collectively forming a *timing scheme*. The timing model (or timing scheme) of the JVM can subsequently be used for verifying temporal properties of the hosted SCJ hard real-time system.

The paper is structured as follows; Section 2 reviews related work and is followed by Section 3, which introduces our real-time Java framework in terms of timing analysis tools, programming model, and execution platform. This puts HVM_{TP} in perspective and is used for the reasoning in Section 4, which contains the core of the paper; the HVM_{TP} in terms of the time-predictable revision the HVM has underwent. Section 5 outlines the $\text{TetaSARTS}_{\text{JVM}}$ and $\text{TetaSARTS}_{\text{TS}}$ tools and presents the results followed by Section 6, which contains conclusive remarks.

2. RELATED WORK

Oracle Java Real Time System (Java RTS) [24], OVM [3], FijiVM [25], KESO VM [13], and JamaicaVM [1] are all examples of JVM implementations for embedded real-time systems. Java RTS is a Just-In-Time (JIT) compiling JVM implementation supporting the Real-Time Specification for Java (RTSJ) [10]; a set of extensions to the JVM and class libraries facilitating real-time systems development in Java. Java RTS is not supported by timing analysis tools and as such cannot be used for reasoning about hard real-time constraints. OVM is an Ahead-Of-Time (AOT) compiling RTSJ-compliant JVM implementation also lacking tool support for timing analysis. Based on OVM, the FijiVM was fostered that is also based on AOT-compilation with additional support for the SCJ [20] specification based on the open-source SCJ implementation, oSCJ [26]. FijiVM, as opposed to OVM, has sufficiently low memory demands and is applicable for embedded systems. It does however require a POSIX-like OS. The KESO VM is in many respects similar to the HVM; it is an AOT compiling JVM that exploits static configuration knowledge for tailoring an optimised execution environment for the hosting Java system. It uses an OSEK/VDX system model for OS functionality. Similar to FijiVM, JamaicaVM is also targeting embedded systems and uses various optimisation techniques for mitigating application size and resource demands on run-time.

In contrast to the aforementioned, a direction of research has focused on implementing the JVM in hardware [27, 2]. The purpose is to provide a complete environment for Java real-time systems, that is, a time-predictable JVM, and accompanying tool support for conducting WCET analysis using the WCET Analyzer tool (WCA) [29], and schedulability analysis using either SARTS [9] or TetaSARTS [22, 21, 23]. The JOP has been realised using a Field-Programmable Gate Array (FPGA).

Providing timing analysis of execution environments containing a software implementation of the JVM running on embedded hardware, has been attempted by XRTJ [16], TetaJ [14] and TetaSARTS. XRTJ presents a framework for portable timing analysis based on the concept of Virtual Machine Timing Models (VMTMs) [15], which allow for expressing the cost of individual Java Bytecode instructions for a particular execution environment. [15, 16] put forward two strategies for deriving VMTMs: one based on profiling and one based on benchmarks, but both approaches are measurement-based as detailed analysis of the Java program, JVM implementation, and hardware platform is judged too complex. Due to potentially under-approximating the temporal behavior, such models are not applicable for making hard real-time guarantees. TetaJ is a model-based WCET analysis tool, which translates the JVM and hardware state information, e.g. cache and pipeline behavior, into a Network of Timed Automata (NTA) model which is amenable to model checking using Uppaal [4]. TetaSARTS uses a similar approach, but incorporates a number of analyses and optimisations to the model, and thus scales to analysis of bigger systems. It also facilitates other analyses to be conducted including Worst Case Response Time (WCRT) analysis, schedulability analysis, and more.

3. REAL-TIME JAVA FRAMEWORK

The contributions of this paper are within the scope of our ongoing work on a real-time Java framework whose components form the trinity shown in Figure 1. While the components can be used in other contexts, their trinity provides a synergy effect; e.g. the precision and capabilities of the Timing Analysis Tools are enhanced if the JVM exhibits a certain structure (and if certain hardware is used) which is due to a model-based timing analysis approach. In this section, we review the framework which subsequently will be used for reasoning on the JVM design.

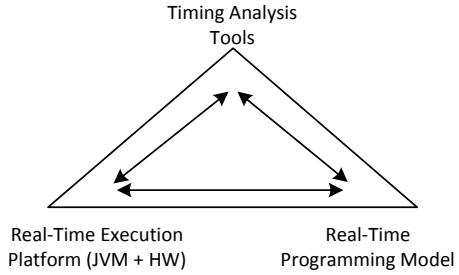


Figure 1: The real-time Java framework.

3.1 Real-Time Programming Model

Our programming model is based on the SCJ profile level 1. Most importantly, the profile addresses inherent issues related to memory management and concurrency semantics, but it also imposes restrictions on the use of certain classes from the Java class library as well as adding functionality required to support real-time concepts e.g. high-resolution timers and hardware device interactions. Here, we review some important areas and concepts of the SCJ programming model that are particularly relevant to our work.

3.1.1 Missions

A mission encapsulates a specific functionality or phase in the lifetime of the real-time system as a set of schedulable entities. For instance, a flight-control system may be composed of take-off, cruising, and landing each of which can be assigned a dedicated mission. A schedulable entity handles a specific functionality and has release parameters describing the release pattern and temporal scope e.g. release time and deadline. The release pattern is either periodic or aperiodic.

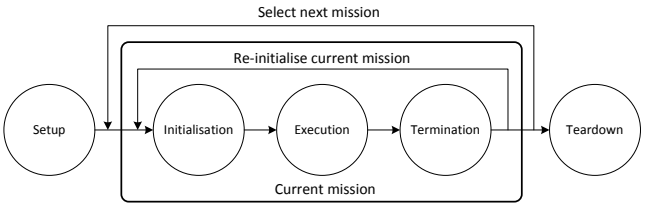


Figure 2: Overview of the mission concept.

The mission concept is depicted in Figure 2 and contains five phases;

Setup where the mission objects are allocated. This is done during start-up of the system and is not considered time-critical.

Initialisation where all object allocations related to the mission or to the entire applications are performed. This phase is not time-critical.

Execution during which all application logic is executed and schedulable entities are set for execution according to a pre-emptive priority scheduler. This phase is time-critical.

Cleanup is entered if the mission terminates and is used for completing the execution of all schedulable entities as well as performing cleanup-related functionality. After this phase, the same mission may be restarted, a new is selected, or the Teardown phase is entered. This phase is not time-critical.

Teardown is the final phase in the lifetime of the application and comprises deallocation of objects and release of locks etc. This phase is not time-critical.

A *mission sequencer* is used for governing the order of the mission objects and can be customised to the application.

3.1.2 Memory Model

SCJ introduces a memory model based on the concept of *scoped memory* from the RTSJ, which circumvents the use of a garbage collected heap to ease verification of SCJ systems. The SCJ memory model is shown in Figure 3 and introduces three levels of memories;

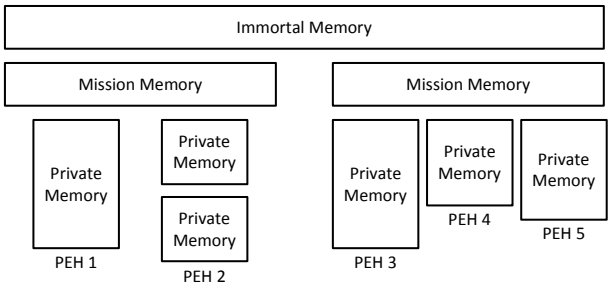


Figure 3: The memory model in SCJ.

Private memory which is associated with each real-time event handler. The private memory exists for the entire duration of the handler. Upon task finish, the memory area is reset.

Mission memory is associated with every mission of the system and as such manages the memories of all real-time handlers part of the mission as well as objects that are shared among the handlers. When the mission completes execution, the mission memory is reset.

Immortal memory is the memory area that exists for the lifetime of the system.

Dynamic class loading is not required. Hence, it is not necessary to reason about classes potentially being loaded over a network which would complicate timing analysis significantly. Furthermore, finalizers will not be executed and we make the reasonable assumption that Java Bytecode verification of class files has been done prior to the time-critical phase. The Predictable Java profile [7], being an alternative Java profile for hard real-time systems development, does allow the use of finalizers and [8] has demonstrated that timing analysis is possible. The contribution in this paper may extend to include finalizers in future work.

3.2 Real-Time Execution Platform

The Hardware near Virtual Machine (HVM) [30, 18] is a lean JVM implementation directed towards use in resource-constrained embedded devices with as low as 256 KB ROM and 20 KB RAM. It features both iterative interpretation, Java-to-C compilation, and a hybrid of the two [17].

The HVM employs *JVM specialisation*; a JVM is produced specifically for hosting the Java Bytecode program. This is done using the ICECAP-TOOLS Eclipse-plugin, which analyses the Java Bytecode program and produces an executable for the target platform. The analyses and transformations can be extended, and it incorporates a number of static analyses for improving performance of the JVM and for reducing its size. This includes receiver-type analysis for potentially devirtualising method calls and intelligent class linking which computes a conservative set of classes and methods that are used in the application. Only this set will be embedded in the final HVM executable. It also conservatively estimates the set of Java Bytecodes that will actually be used. Those that are not, are omitted from the final executable.

The HVM is self-contained and does not rely on the presence of an OS or a C standard library. The overall structure of an SCJ application running on top of the HVM using the accompanying SCJ implementation (HVM-SCJ [30]) as well as the ICECAP-TOOLS SDK is shown in Figure 4. Porting the HVM to new target platforms (including SCJ support) is a matter of implementing the *HW Interface* and the *VM Interface*.

Other distinctive features include support for SCJ level 1 and Hardware Objects [28] that are an object-oriented abstraction of low-level hardware devices such as I/O registers and interrupts which can be handled directly from Java space. A related feature is *native variables*, which allow for access to certain variables in the JVM from Java space.

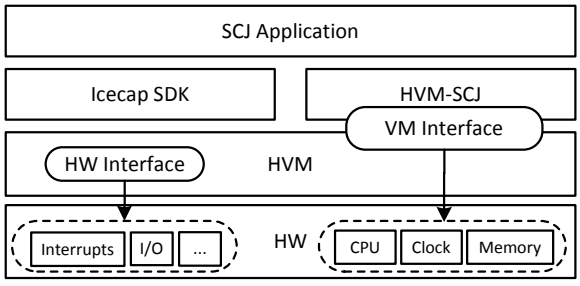


Figure 4: Constituents of an SCJ application on HVM.

3.3 Timing Analysis Tools

We use the open-source TETASARTS collection of timing analysis tools which operate on a timing model amenable to model checking using UPPAAL. Figure 5 shows the major components of the toolchain and their interactions.

Reasoning about the timeliness of the system or any other real-time aspect, requires both the real-time application as well as the underlying execution environment to be modelled. Generating an appropriate model of the latter is the

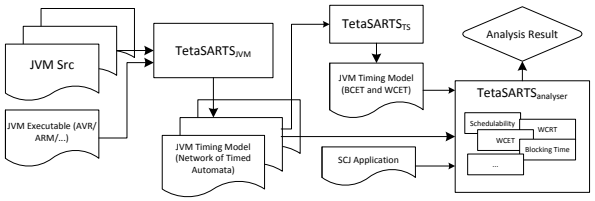


Figure 5: Overview of the TETASARTS toolchain.

purpose of the open-source TETASARTS$_{JVM}$[1] tool. The purpose of this paper is not to detail how the individual tools work, and therefore we only provide an overview.

TETASARTS$_{JVM}$ generates a timing model as a Network of Timed Automata (NTA), which is the modeling formalism of the UPPAAL model checker. A Timed Automaton (TA) is a finite state machine extended with real-valued clocks. An NTA is the parallel composition of a number of TAs sharing clocks and actions (for the semantics, see [5]).

TETASARTS$_{JVM}$ builds a TA that captures the control-flow for each of the supported Java Bytecodes. The tool processes the JVM executable such that compiler optimisations, transformations etc. are accounted for when reconstructing the control-flow. TETASARTS$_{JVM}$ conducts loop identification analysis and expects loop bounds to either be provided interactively at construction time or as comment-style annotations in the source code of the JVM. The tool allows specifying code regions constituting Java Bytecode implementations. The regions are created by embracing the Java Bytecode implementations with macros: BEGIN_JBC(X) and END_JBC(X) mark the beginning and end, respectively, of Java Bytecode X. The macros generate instrumentation code in the binary, which is used by TETASARTS$_{JVM}$ for reconstructing the control-flow. Listing 1 shows the implementation of the `i2l` Java Bytecode and the region specification. Figure 6 shows an excerpt of the corresponding TA gener-

```
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 1: `i2l`.

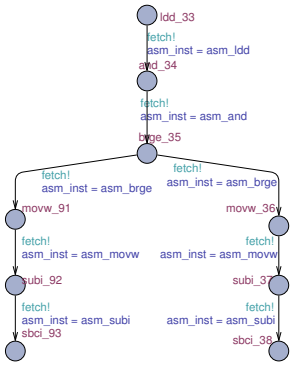


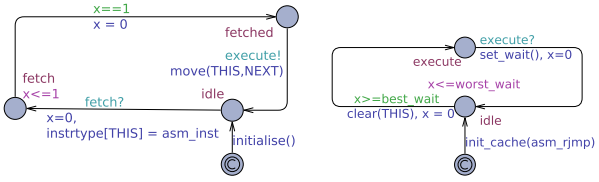
Figure 6: TA excerpt of `i2l`.

ated by TETASARTS$_{JVM}$. As an example, note how the location labelled `brge_35` branches to `movw_91` and `movw_36`. This captures the if-statement in line 6 of Listing 1. Each transition simulates the timing behavior of executing the instruction assigned to the UPPAAL variable `asm_inst`. In

[1]TETASARTS$_{JVM}$, HVM$_{TP}$, and generated models are available on the project website: `http://people.cs.aau.dk/~luckow/hvmtp/`

general, this explicit modeling of system behavior does not scale to large systems due to the inherent problem of state space explosion. However, it is important to note that in our case, the behavior of each Java Bytecode in isolation is sufficiently small to allow model checking to be feasible. We will provide evidence to this later in the paper. The Java Bytecode implementations we are analysing, range from just a few lines of C code in the simplest case, to a few hundred lines of C code producing ∼200 and ∼9000 machine instructions, respectively.

The composition of the generated TAs yields the NTA referred to as the *JVM NTA*. The timing behavior depends on the hardware used; in this particular example the AVR ATmega2560 microcontroller whose behavior is captured by the TAs shown in Figure 7 collectively referred to as the *HW NTA*. TetaSARTS$_{\text{JVM}}$ also supports the ARM7 and ARM9 models from METAMOC [12] and can further be extended to support other hardware. The `fetch` channel is used for



(a) Pipeline fetch stage. (b) Pipeline execute stage.

Figure 7: Hardware TA models from METAMOC [12].

hand-shake synchronisation between the JVM NTA and the pipeline fetch stage in Figure 7a; `asm_inst` communicates the instruction to be simulated in the HW NTA. In a similar way, the `execute` channel establishes the communication means between the fetch stage TA and the execute stage TA in Figure 7b. In both TAs, `x` is a clock variable that simulates the instruction processing time in the respective pipeline stage. In the execute stage TA, `worst_wait` and `best_wait` are set according to the worst and best case clock cycle execution times of the simulated instruction. Note the call `init_cache(asm_rjmp)`; it initialises the pipeline with the temporal behavior of the `rjmp` instruction. This is necessary to ensure a safe timing model, because the pipeline will be filled with this instruction prior to executing the first instruction of any of the Java Bytecodes.

Synthesising the JVM NTA and the HW NTA yields a complete *JVM Timing Model* (see Figure 5) that simulates the timing behavior of the JVM. The JVM Timing Model can subsequently be used directly by further synthesising it with a model of an SCJ application on which various timing related analyses can be applied such as schedulability analysis and WCRT analysis using the TetaSARTS$_{\text{ANALYSER}}$ tool (see [21, 22, 23] for more details).

The JVM Timing Model can also be used for representing the timing behavior of the JVM more abstractly using the TetaSARTS$_{\text{TS}}$ tool (see Figure 5). For all Java Bytecode TAs, the tool determines the WCET and BCET using the sup- and inf-query extensions of the Uppaal model checker. Sup- and inf-queries perform a state space exploration, and outputs the maximum and minimum values, respectively, of the specified clock-variables.

4. RE-DESIGNING THE HVM

In this paper, we focus on the iterative interpreter of the HVM. Handling the Java-to-C compiler and the hybrid of the two is left for future work. Listing 2 shows the structure of the HVM interpreter.

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

Listing 2: Structure of the HVM interpreter.

The interpretation process is comprised of fetching (line 11), analysing and decoding (line 12), and executing (the body of each case statement e.g. line 15); the process is repeated afterwards. The initial stages in the translation process; fetching, decoding, analysing, have constant execution times. Fetching is implemented by reading the opcode pointed to by the instruction pointer from an array of bytecodes of the executing method. Analysing (and decoding) is performed by the switch-case statement; since the case values are close, the structure is translated into a jump-table with constant time look-up. The final executing stage is performed by performing the logic starting from the jump target and is thus the only context-dependent stage.

A time predictable HVM can hence be constructed by redesigning each Java Bytecode implementation to be time predictable, which will be the subject of the following sections.

Many of the Java Bytecodes naturally have constant execution times. This includes the following classes:

Load and Store Instructions used for managing values between the operand stack and the local variables. It is comprised of instructions such as `iload`, `istore`, `ldc`, etc. If `fp` and `sp` denote the frame pointer and the operand stack pointer and `method_code` denotes the instruction pointer, a constant execution time version of `iload` can be implemented as follows:

```
unsigned char index = *(++method_code);
*sp++ = fp[index];
```

Type Conversion Instructions deals with type conversion among the primitive types supported by the JVM. This is done using the instructions with format `x2y` where x is to be converted into type y e.g. `f2d` for a float to double con-

version. The following is a constant time implementation of `i2f`:

```
int32 lsb = *(--sp);
*(float*) sp = (float) lsb;
```

Operand Stack Management Instructions deals with ordinary stack operations such as popping values (`pop`), duplicating values (`dup`) etc. These can intuitively be implemented as constant time operations. The following shows an implementation of `dup`:

```
*sp = *(sp - 1);
sp++;
```

Control Transfer Instructions includes all the conditional instructions such as `ifeq` (if the top of stack element is zero, branch to offset. Otherwise, continue execution at the next instruction). A time predictable version of the `ifeq` instruction is shown below:

```
signed short int offset = 3;
if(*(--sp) == 0) {
  BYTE bb1 = pgm_read_byte(method_code + 1);
  BYTE bb2 = pgm_read_byte(method_code + 2);
  offset = (signed short int) ((bb1 << 8) | bb2);
}
method_code += offset;
```

The remaining classes of Java Bytecodes and concepts related to the workings of the JVM are however, not naturally time predictable, and special care must be taken to make them so. These bytecodes have been categorised into the following:

- Object allocations
- Exceptions
- Method invocations
- Type checking of reference types
- Handling strings
- Platform dependencies
- Handling jump tables

In the following sections, we elaborate on how these have been redesigned to facilitate timing analysis.

4.1 Object Allocation

Object allocations in the JVM are performed using one of the following Java Bytecodes:

`new` creates a new object.

`newarray` allocates a new array of a primitive type.

`anewarray` same as `newarray`, but creates an array of object references.

`multianewarray` similar to `anewarray`, but deals with multi-dimensional arrays.

We are targeting SCJ, and therefore, garbage collection is not a concern; object allocation and deallocation will be achieved using scoped memory. This model enables that (de-)allocations are performed at predictable points in time, but it does not entail that these are performed in a *time predictable manner*.

The HVM implements object allocation by incrementing a pointer by the size of the object that is to be allocated and can hence be performed in constant time. However, the HVM performs zeroing of the memory at allocation time; an operation that takes time linear in the size of the allocated object. While determining an application wide, fixed execution time for zeroing is possible by the bound corresponding to allocating the largest object in the application, it is overly conservative and will be application dependent i.e. a general timing model cannot be constructed. Therefore, a better approach is to harness SCJ: in the initialisation phase of the SCJ safelet, the entire heap structure is zeroed, and subsequent object allocations can then assume proper initialisation of the allocated memory. The technique is inspired by jRate [11], an RTSJ extension to the GNU GCJ compiler. Whenever a scoped memory is exited, the memory area is zeroed (again) which takes time linear in the size of the allocated scope. This operation is done in Java space using the native variables concept [30], which allows to write directly to memory. In this way, the timing model for the HVM does not need to account for zeroing, however, the timing model of the SCJ program does.

4.2 Exceptions

Unchecked exceptions will be thrown implicitly from the JVM in the Java Bytecode implementations and methods are not obliged to establish a catching block. Checked exceptions, on the other hand are explicitly thrown using the `athrow` Java Bytecode and methods are obliged to catch these.

Exceptions are allowed in SCJ and it is permitted to allocate them before entering a time-critical phase. In HVM_{TP}, we exploit this fact and combine it with static information about the exceptions. Before the HVM_{TP} is constructed, the time-critical application code is analysed for conservatively estimating the set of exceptions that may be thrown. This comprises both checked and unchecked exceptions; the former can be determined by the uses of the `athrow` Java Bytecode, while the latter can be determined by examining the application for Java Bytecodes that can throw unchecked exceptions e.g. `idiv`. This analysis is performed at construction time of the HVM_{TP} executable. The set conservatively estimates the exceptions that may be thrown and can hence be considered safe. However, due to the safety measures of the systems we are considering, it is customary to apply static analyses for guaranteeing that certain exceptions are never thrown (e.g. that the denominator cannot become zero in divisions). $\text{TETASARTS}_{\text{JVM}}$ can be instructed to generate a timing model that excludes exceptions to minimize over-approximation.

The exception objects associated with the potentially thrown exceptions are pre-allocated during the initialisation phase. In this way, the remaining issue is to determine and execute the exception handler; whenever an exception is thrown, it

may be caught somewhere in the call stack, which is conducted in time linear in the size of the call stack. For timing analysis, we must assume worst case behavior; in this case it will be the maximum size of the call stack which is determined in ICECAP-TOOLS by reconstructing the call graph and determine the maximum depth. Using this information, the code for finding the exception handler is shown in Listing 3.

```
1 unsigned short handler_pc;
2 unsigned short pc = method_code - method->code;
3 Object* exception = (Object*) (pointer) *(sp - 1);
4 unsigned short classIndex = getClassIndex(exception);
5 //@loopbound = MAX_CALL_DEPTH
6 while((handler_pc = handleAthrow(method, classIndex,
      pc)) == (unsigned short) -1) {
7   sp--;
8   method = popStackFrame(&fp, &sp, method, &pc,code);
9   fp[0] = (int32) (pointer) exception;
10  if(method == 0)
11    return classIndex;
12 }
13 sp = &fp[method->maxLocals];
14 *sp++ = (int32) (pointer) exception;
15 pc = handler_pc;
16 method_code = method->code + pc;
```

Listing 3: The algorithm for finding the exception handler.

In the comment-style annotation in line 5, `MAX_CALL_DEPTH` is a macro, which expands to the value of the maximal depth of the call graph. The annotation will be extracted by TETASARTS$_{JVM}$ when constructing the timing model.

4.3 Method Invocation

The class of Java Bytecode instructions concerned with this issue includes:

`invokestatic` used for invoking static methods.

`invokespecial` used for invoking private instance methods, methods in super classes, and the instance initialisation method.

`invokeinterface` invokes a method declared in an interface.

`invokevirtual` used for dynamic method dispatch.

4.3.1 Redesigning Method Invocation Management

Originally, the HVM employed recursion for dealing with method invocations; whenever executing one of the method invoking Java Bytecodes, the interpreter of the HVM is invoked with the method body of the invoked method as argument (see Listing 4). Once the invoked method finishes, control is returned to the callsite of the method invoking instruction.

```
1 case INVOKEVIRTUAL_OPCODE: {
2   const MethodInfo* mInfo;
3   signed short excep;
4   mInfo = findMethodInfo(&sp[top], &method_code[pc]);
5   //Code for handling native calls...
6   excep = methodInterpreter(mInfo, &sp[top]);
7   //Deal with exceptions...
8 }
```

Listing 4: Recursion in `invokevirtual`.

Recursion in the JVM introduces various issues in relation to time predictability and the approaches we use for timing analyses. First, recursion poses similar problems as unbounded loops. Another inherent issue is that a recursive control flow is difficult to model using TA; a TA is associated with the control flow of a single function (or method) and the semantics of method calls are captured by an action (and corresponding co-action in the callee).

Instead of putting the responsibility on the timing analyses tools, we have addressed the issue by redesigning the implementation of method invocation; instead of recursion, a call stack approach has been implemented. A stack frame is pushed onto the stack whenever a method is invoked and the interpreter continues operating on this (see Listing 5).

```
1 case INVOKEVIRTUAL_OPCODE: {
2   //Code for handling native calls...
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   goto start;
7 }
```

Listing 5: Using stack frames in `invokevirtual`.

The `goto` statement in line 6 jumps to the start of interpreter (see line 5 in Listing 2). When the invoked method returns, execution of the call site continues by popping the stack and restore the context stored in the frame. The push and pop operations on the stack are performed in constant time and are modelled similarly to any other function as a TA.

4.3.2 Method Dispatch

`invokespecial` and `invokestatic` are used for statically dispatching control to the specified method. As such, they do not imply issues in terms of time predictability because determining the callee as well as dispatching control to it is done in constant time.

Virtual method calls as produced by `invokevirtual` and `invokeinterface` are not directly time predictable. The method to call depends on the type of the receiver at runtime, and cannot be linked statically. The HVM has been refactored to make the runtime lookup of the method time predictable. The strategy applied is the following: at compile time ICECAP-TOOLS analyses each virtual call site and determines the set of possible types of the receiver at runtime. This knowledge is a side effect of the computation of the dependency extent; the set of classes and methods that may be called during runtime. The strategy applied to compute this is described in detail in [30]. Because of the devirtualisation performed during the analysis phase only truly virtual callsites are maintained - the rest are treated as `invokespecial`. In the case of Java-to-C compilation each virtual callsite gets translated into a C switch statement, switching on the type of the receiver. Inside each case-statement is a direct call to the correct method. The set of receiver types are calculated differently for `invokevirtual` and `invokeinterface`, but the emitted switch statement looks the same. The HVM now relies on the C compiler to translate this switch statement into efficient machine code.

For simplicity this strategy is simulated in the interpreter. This means that the virtual method dispatch does not utilise a standard virtual table, which would be a constant time lookup. Currently the switch-statement case values (the virtual table) are encoded into the bytecode of the method call.

This is done in the same way for both `invokevirtual` and `invokeinterface`. The interpreter then searches the table for the correct receiver. The time it takes to lookup any method at runtime is proportional to the size of the largest jump table. In some cases the runtime type of the receiver is not in the jump table and it is necessary to get the super class of the receiver and try again. Thus, the time required to perform the `invokevirtual` and `invokeinterface` bytecodes is the size of the largest jump table multiplied by the maximum height of the class hierarchy. Both constants are generated by ICECAP-TOOLS and used for annotating the interpreter loops pertaining to virtual method dispatch. TETASARTS$_{\text{JVM}}$ uses this information when building the timing model.

4.4 Type Checking Reference Types

Runtime type checking between reference types is achieved by the `checkcast` and `instanceof` Java Bytecodes. In a traditional implementation of the JVM where no knowledge about the class hierarchy can be incorporated in the JVM prior to runtime, these operations are performed by consulting the class and subclasses iteratively using the class hierarchy until type compatibility can be concluded. Seen from a static analysis perspective, this implies analysing an unbounded loop. A safe upper bound that captures the maximum number of iterations, can be established but will apply for the entire class hierarchy. Despite being a safe bound, it is overly conservative.

The original implementation of the HVM performs type checking in this way, but we redesigned it in HVM$_{\text{TP}}$ for achieving constant time type checking. This is done by exercising the class hierarchy prior to HVM$_{\text{TP}}$ construction; type compatibility among the classes is stored in a matrix that is incorporated in the final HVM$_{\text{TP}}$ executable. At runtime, type compatibility can be conducted by a constant time look-up using a key constructed from the class to which the object belongs. The details are shown in Listing 6.

```
1 unsigned char isSubClassOf(unsigned short subClass,
      unsigned short superClass) {
2   uint32 bitIndex = (subClass << tupac) + superClass;
3   uint32 byteIndex = bitIndex >> 3;
4   unsigned char b = *(inheritanceMatrix + byteIndex);
5   b = (unsigned char) (b & (1 << (bitIndex & 0x7)));
6   return b != 0;
7 }
```

Listing 6: Using a matrix for determining type compatibility in constant time.

Here `inheritanceMatrix` encodes type compatibility among all the classes, and the computed `bitIndex` is used to extract whether `subClass` is type compatible with `superClass`. A value of 1 represents type compatibility. A trade-off with this solution is that time predictability comes at the expense of quadratic memory overhead; each element in the inheritance matrix occupy one bit.

4.5 Strings

In the SCJ specification, it is assumed that applications do not do extensive text processing and as such, many of the classes in the Java class library such as `String` and `StringBuilder` have disallowed use of instance and class methods. The rationale is to reduce the size and the complexity of the classes to ease verification [20].

We further assume that strings are immutable, and that the `append` method of `StringBuilder` is not used. This also precludes use of the plus operator on strings, which is typically compiled as a `StringBuilder` object on which the `append` method is used for string concatenation.

Strings are stored in the constant pool of the associated type and will upon first reference create a string object. A reference to the string in the constant pool can happen using the `ldc`, `ldc_w`, and `ldc2_w` Java Bytecodes. Creating a string object involves allocating a character array and inserting the characters of the particular string and, furthermore, class initialisers will be executed. The execution time of this is hence dependent on the particular string. To avoid this, HVM$_{\text{TP}}$ deals with strings in a similar way as exceptions; prior to constructing the HVM, the class files are analysed to determine potential references to strings in the constant pool. For this set, HVM$_{\text{TP}}$ constructs the string objects in the initialisation phase of the application. Uses of `ldc`, `ldc_w`, and `ldc2_w` during the time-critical phases can be performed by pushing the string object reference onto the operand stack.

4.6 Platform Dependencies

Many embedded microcontrollers, do not have instruction set support for some arithmetic operations e.g. division and multiplication. Whenever unsupported arithmetic operations are used, the compiler will typically generate code that simulates them. The algorithms used may be compiler dependent, and may be hard to reason about from a timing analysis perspective. To circumvent these issues, HVM$_{\text{TP}}$ contains a generic software implementation of the operations that are hardware dependent and whose execution time can be bounded. All multiplications are conducted using a variant of *shift and add* multiplication whose execution is bounded by the number of bits used for integer representations; this value is used as loop bound. A similar rationale is used for the other operators division and the derived operators modulus and remainder.

4.7 Jump Table

The implementation of the `lookupswitch` and `tableswitch` bytecodes in HVM$_{\text{TP}}$ contain a loop looking for the jump target depending on the actual jump index. This loop has been bounded by ICECAP-TOOLS which calculates the constant value of the largest `lookupswitch` or `tableswitch` jump tables. The size of the jump tables are readily available in the class files. HVM$_{\text{TP}}$ has been annotated with these bounds and this information is subsequently used by TETASARTS$_{\text{JVM}}$ when building the timing model.

4.8 Class Initialisation

Class initialisation happens at unpredictable times e.g. at the first reference to a static field using `getstatic` and `putstatic`. Further, class initialisers have unpredictable execution times, since the initialisation of one class, may lead to the initialisation of another if a static field references a class that is not yet initialised etc. To circumvent the issue, we harness the SCJ specification, which permits class initialisers to be performed after being loaded into immortal memory on start-up. Also the SCJ specification dictates,

that no properly structured SCJ application can have cyclic dependencies in class initialisation.

HVM$_{TP}$ accommodates these changes by performing class initialisers during the initialisation phase together with string and exception object allocations. `getstatic` and `putstatic` have been modified to avoid calling class initialisation code, and therefore have time-predictable execution.

Applying all the methods to the original HVM implementation, resulted in the time-predictable variant, HVM$_{TP}$.

5. RESULTS

We have used TetaSARTS$_{JVM}$ to construct a complete JVM Timing Model of the supported Java Bytecodes of HVM$_{TP}$ on AVR. All experimental results have been obtained on a desktop machine with an Intel Core i7-2620M CPU @ 2.70GHz with 8 GB of RAM. Constructing the JVM NTA from the HVM$_{TP}$ executable takes 16 seconds when exception handling is excluded and 20 seconds when included. The timing model can be integrated with TetaSARTS, but has many other applications. We will here demonstrate the use of TetaSARTS$_{TS}$ for generating corresponding timing schemes. The timing scheme takes approximately 4.5 hours for the model excluding exception handling and approximately 5 days for the model which includes exception. Most of the Java Bytecodes take only a few seconds to process, but a few e.g. the three `ldc_*` bytecodes account for approximately two hours of the total processing time. Note that both the complete timing model and timing scheme need only to be generated once; only application-dependent Java Bytecodes i.e. bytecodes for which the loop bounds are expressed in terms of the properties of the hosting application, need to be processed for each SCJ application to reflect current temporal behavior. This, along with the fact that many Java Bytecodes are not used by the application (and as a consequence are excluded from HVM$_{TP}$ due to JVM specialisation) makes generating the timing scheme a less time consuming process in reality. As an example, we analysed the Minepump control system [6, 14, 22], a representative example of a real-time system written in Java. It uses 49 distinct Java Bytecodes and the initial timing model and timing scheme take 5 seconds and 6 minutes to generate, respectively. The Minepump contains only two application-dependent Java Bytecodes (`invokevirtual` and `invokeinterface`) each of which take $\sim 2s$ to determine BCET and WCET for. Hence, new timing schemes reflecting e.g. further development on the system, can rapidly be generated.

To provide an indication of the validity of the timing scheme, we have compared the results against measurements obtained using Atmel Studio 6 by calculating the difference between the cycle counter prior to executing the first statement and after executing the last statement of the Java Bytecodes. The results of five samples for some of the Java Bytecodes are shown in Table 1.

Note that all results are safe i.e. for all bytecodes, BCET $\leq$ Low and High $\leq$ WCET. The results also demonstrate that some Java Bytecodes have long execution times which is due to the interpreter of HVM$_{TP}$. In [17], it is shown that the AOT compiler of HVM$_{TP}$ produces code that on average is approximately a factor 30 faster than using the interpreter.

Bytecode	TetaSARTS$_{TS}$		Measured		
	BCET	WCET	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

Table 1: Validation of the timing scheme. Times are represented in clock cycles.

The reader is referred to the project website `http://people.cs.aau.dk/~luckow/hvmtp` where all models are available. Furthermore, we provide the complete list of application-dependent Java Bytecodes.

6. CONCLUSION

In this paper, we have presented HVM$_{TP}$; a time-predictable and portable JVM implementation with applications in hard real-time embedded systems. It supports the emerging Safety Critical Java (SCJ) profile level 1. The HVM$_{TP}$ is based on a redesign of the HVM, which combines static knowledge about the hosted SCJ application, the programming and memory model of SCJ, and time-predictable solutions. This paper has presented the redesign.

We have demonstrated that a complete timing model of HVM$_{TP}$ can be constructed using TetaSARTS$_{JVM}$; a JVM timing model generator part of the TetaSARTS timing analysis tool-set for Java. The timing model is represented using the Network of Timed Automata modeling formalism of the Uppaal model checker. Using sup- and inf-queries of Uppaal, we have determined the Best Case Execution Times (BCETs) and Worst Case Execution Times (WCETs) of all Java Bytecode implementations of the HVM$_{TP}$ yielding a complete timing scheme for HVM$_{TP}$ on AVR. Our technique and tools are extensible to other platforms as well. The uses of the timing model and timing schemes are many; they are directly integrable in TetaSARTS to allow reasoning on schedulability and timing properties e.g. WCET, WCRT, and processor utilisation of SCJ systems on the particular platform the timing model is capturing. This opens for new opportunities, e.g., a *write once, run wherever possible* development approach of SCJ systems by evaluating temporal correctness on multiple hardware models. The timing model also opens for easy profiling and performance analysis of the Java Bytecodes described in terms of distributions of execution times – we tested similar ideas in [23].

Future works comprise generating timing models that include the Ahead-Of-Time (AOT) and hybrid between AOT and interpretation capabilities of the HVM$_{TP}$. We also want to further improve the precision of the timing model produced by TetaSARTS$_{JVM}$. This comprises, among others, the adoption of symbolic execution for reducing infeasible paths in the resulting model. Finally, we want to investigate pre-computing timings of Java Bytecodes with application-dependent temporal behavior.

7. REFERENCES

[1] Aicas-GmbH. *JamaicaVM 6.0 - User Manual: Java Technology for Critical Embedded Systems*, 2010.

[2] aJile Systems Inc., 2013. `http://www.ajile.com/`.

[3] A. Armbruster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek. A Real-Time Java Virtual Machine with Applications in Avionics. *ACM Trans. Embed. Comput. Syst.*, 7:5:1–5:49, December 2007.

[4] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — a Tool Suite for Automatic Verification of Real-Time Systems. In *Hybrid Systems III*, Lecture Notes in Computer Science. 1996.

[5] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, page 87–124.

[6] T. Bøgholm, C. Frost, R. Hansen, C. Jensen, K. Luckow, A. Ravn, H. Søndergaard, and B. Thomsen. Towards Harnessing Theories Through Tool Support for Hard Real-Time Java Programming. *Inno. in Systems and Software Engineering*, 2013.

[7] T. Bøgholm, R. R. Hansen, A. P. Ravn, B. Thomsen, and H. Søndergaard. A Predictable Java Profile: Rationale and Implementations. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, 2009.

[8] T. Bøgholm, R. R. Hansen, A. P. Ravn, B. Thomsen, and H. Søndergaard. Schedulability Analysis for Java Finalizers. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, 2010.

[9] T. Bøgholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, and K. G. Larsen. Model-Based Schedulability Analysis of Safety Critical Hard Real-Time Java Programs. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, JTRES '08, 2008.

[10] G. Bollella and J. Gosling. The Real-Time Specification for Java. *Computer*, 33(6), june 2000.

[11] A. Corsaro and D. C. Schmidt. The Design and Performance of the jRate Real-time Java Implementation. In *International Symposium on Distributed Objects and Applications*, 2002.

[12] A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In *10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

[13] C. Erhardt, M. Stilkerich, D. Lohmann, and W. Schröder-Preikschat. Exploiting Static Application Knowledge in a Java Compiler for Embedded Systems: A Case Study. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, 2011.

[14] C. Frost, C. S. Jensen, K. S. Luckow, and B. Thomsen. WCET Analysis of Java Bytecode Featuring Common Execution Environments. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, 2011.

[15] E. Hu, A. Wellings, and G. Bernat. Deriving Java Virtual Machine Timing Models for Portable Worst-Case Execution Time Analysis. *On The Move to Meaningful Internet Systems*, 2003.

[16] E. Hu, A. Wellings, and G. Bernat. XRTJ: An Extensible Distributed High-Integrity Real-Time Java Environment. *Real-Time and Embedded Computing Systems and Applications*, 2004.

[17] S. Korsholm. *Java for Cost Effective Embedded Real-Time Software*. PhD thesis, 2012.

[18] S. Korsholm. Hvm, 2013. `www.icelab.dk`.

[19] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Pearson Education, 2013.

[20] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A. Wellings. Safety-Critical Java Technology Specification, Public draft, 2013.

[21] K. S. Luckow, T. Bøgholm, and B. Thomsen. Supporting Development of Energy-Optimised Java Real-Time Systems using TetaSARTS. In *WiP Proceedings of the 19th Real-Time and Embedded Technology and Application Symposium*, 2013.

[22] K. S. Luckow, T. Bøgholm, B. Thomsen, and K. G. Larsen. TetaSARTS: A Tool for Modular Timing Analysis of Safety Critical Java Systems. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, 2013.

[23] K. S. Luckow, T. Bøgholm, B. Thomsen, and K. G. Larsen. TetaSARTS: Modular Timing and Performance Analysis of Safety Critical Java Systems. *Concurrency and Computation: Practice and Experience*, 2014. In Submission.

[24] Oracle. Sun Java Real-Time System, 2008. `http://java.sun.com/javase/technologies/realtime/index.jsp`.

[25] F. Pizlo, L. Ziarek, and J. Vitek. Real Time Java on Resource-Constrained Platforms with Fiji VM. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, 2009.

[26] A. Plsek, L. Zhao, V. H. Sahin, D. Tang, T. Kalibera, and J. Vitek. Developing Safety Critical Java Applications with oSCJ/L0. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, 2010.

[27] M. Schoeberl. JOP: A Java Optimized Processor. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '03, 2003.

[28] M. Schoeberl, S. Korsholm, C. Thalinger, and A. Ravn. Hardware Objects for Java. In *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, 2008.

[29] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber. Worst-case Execution Time Analysis for a Java Processor. *Software: Prac. and Exp.*, 2010.

[30] H. Søndergaard, S. E. Korsholm, and A. P. Ravn. Safety-Critical Java for Low-End Embedded Platforms. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, 2012.