This is the accepted version of the following article:
Kasper Søe Luckow, Bent Thomsen, and Stephan Erbs Korsholm: *HVMTP: A time predictable and portable java virtual machine for hard real-time embedded systems*. In Concurrency and Computation: Practice and Experience, 2016,
which has been published in final form at `http://dx.doi.org/10.1002/cpe.3828`.
This article may be used for non-commercial purposes in accordance with the Wiley Self-Archiving Policy: `http://olabout.wiley.com/WileyCDA/Section/id-820227.html`.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:2–28
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

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

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

[1] *Department of Computer Science, Aalborg University, Denmark*
[2] *VIA University College, Horsens, Denmark*

SUMMARY

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 all levels of the Safety Critical Java (SCJ) 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 $\text{TetaSARTS}_{\text{JVM}}$. 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. Furthermore, we take a first look at how to support the new Java 8 language feature of Lambda expressions in a SCJ context – we look in particular at how the `invokedynamic` bytecode can be implemented in a time predictable way and integrated in HVM_{TP}. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Java virtual machine, Real-time Java, Time predictability, Model checking

1. INTRODUCTION

Java is widely accepted as a 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) [26] and the Safety Critical Java (SCJ) [36] 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.

*Correspondence to: E-mail: luckow@cs.aau.dk

Copyright © 0000 John Wiley & Sons, Ltd.

Having an appropriate programming model is a necessary, but not the only, step towards use in a hard real-time setting. 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 in most cases be modelled. Hence, the Java Virtual Machine (JVM) is the remaining component in enabling temporal verification of hard real-time systems written in Java.

The JVM adds to the complexity in performing static analysis 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 Processor (JOP) project [48] have both implemented a JVM in hardware thereby 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 analysis. This necessarily demands that the JVM is amenable to static analysis as well, which is difficult, since the JVM specification [35] 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, TetaSARTS_{JVM}, a model-based and highly flexible tool for safe timing analysis of JVM executables part of the TetaSARTS collection of timing analysis tools [37, 38, 39]. Our third contribution, is the TetaSARTS_{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. These contributions were first presented in [41] at the Workshop on Java Technologies for Real-time and Embedded Systems 2014 (JTRES'14). In addition to these contributions, we take a first look at how to support the new Java 8 language feature of Lambda expressions in a Safety Critical Java context – we look in particular at how the `invokedynamic` bytecode can be implemented in a time predictable way and integrated in the HVM_{TP}.

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 undergone as well as providing a treatment of `invokedynamic` in HVM_{TP}. Section 5 outlines the results of analyzing the HVM_{TP} using the TetaSARTS_{JVM} and TetaSARTS_{TS}. Section 6 reports on analysis results of SCJ applications running on the HVM_{TP} on the AVR ATmega2560 processor. In Section 7 we discuss issues relating to the architecture of the JVM, followed by Section 8, which contains conclusive remarks.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

2. RELATED WORK

There are many examples of JVM implementations for embedded real-time systems, e.g., Oracle Java Real Time System (Java RTS) [54], OVM [3], FijiVM [44], KESO VM [22], and JamaicaVM [1]. Java RTS is a Just-In-Time (JIT) compiling JVM supporting the Real-Time Specification for Java (RTSJ) [26]; 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. The OVM is based on AOT-compilation with support for the SCJ [36] specification based on the open-source SCJ implementation, oSCJ [45]. FijiVM has sufficiently low memory demands and is applicable for embedded systems. It does however require an OS, such as RTEMS, Linux or Darwin. 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. 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.

Another direction of research has focused on implementing the JVM in hardware [48, 2]. The purpose of implementing the JVM in hardware is to facilitate time-predictable execution. The JOP comes with accompanying tool support for conducting WCET analysis using the WCET Analyzer tool (WCA) [51], and schedulability analysis using either SARTS [11] or TetaSARTS [38, 37, 39]. The JOP is implemented 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 [31], TetaJ [24] and TetaSARTS. XRTJ presents a framework for portable timing analysis based on the concept of Virtual Machine Timing Models (VMTMs) [30], which allow for expressing the cost of individual Java bytecode instructions for a particular execution environment. The work in [30, 31] proposes 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 [6]. 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.

In this paper we take a first look at lambda expressions in connection with Safety Critical Java. Since March 2014, the standard edition of Java, known as Java SE 8, has support for closures in the form of lambda expressions. Lambda expressions allow for a functional style of programming, for example the elegant MapReduce processing on collections, to be mixed with Java's object oriented programming style. The implementation of lambda expression uses the *invokedymanic* bytecode introduced in the JVM as of version 7. There has, to our knowledge only been a few works looking at Java lambda expressions and other Java SE 8 features in connection with real-time and embedded programming. Chan et al. investigate locality of Java 8 Streams in Real-Time Big Data Applications in the context of RTSJ [14] and mention the use of lambda expression in parallel bulk data operations. Jim Connors introduces an Embedded Java 8 Lambda Expression Microbenchmark in his blog [16]. More broadly lambda expressions and functional style programming is integral in the Erlang Language [55], which however, only support soft real-time programming. The HUME language [28] supports hard-real-time programming in a functional style.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

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

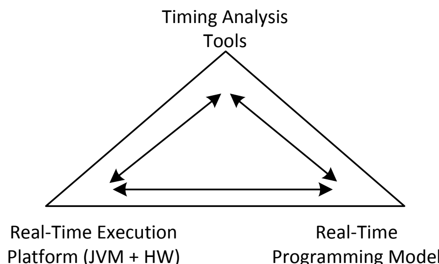


Figure 1. The real-time Java framework.

be used in other contexts, their trinity provides a synergy effect; e.g. timing analysis is made possible by the programming model, 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. The design of all three components have influenced each other. In this section, we review the framework which subsequently will be used for reasoning on the JVM design.

3.1. Real-Time Programming Model

Our programming model is based on the SCJ profile. 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.

Safety critical applications have different complexity levels. To cater for this the SCJ programming model is based on tasks grouped in missions, where a mission encapsulates a specific functionality or phase in the lifetime of the real-time system as a set of schedulable entities. The SCJ specification lets developers tailor the capabilities of the platform to the needs of the application through three compliance levels. Level 0, provides a simple, frame-based cyclic executive model which is single threaded with a single mission. Level 1 extends this model with multi-threading via periodic and aperiodic event handlers, multiple missions, and a fixed-priority preemptive scheduler (FPS). Level 2 lifts restrictions on threads and supports nested missions. The development of SCJ applications at Level 0 is well described in [45]. In the remainder of this section we will focus on the SCJ profile level 1.

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.

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

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

Initialisation where all object allocations related to the mission or to the entire application are performed. This phase is time-critical in applications with mode changes consisting of a sequence of missions.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

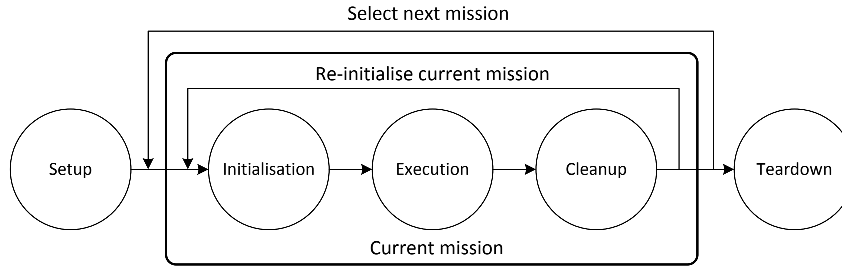


Figure 2. Overview of the mission concept.

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 time-critical in applications with mode changes consisting of a sequence of missions.

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;

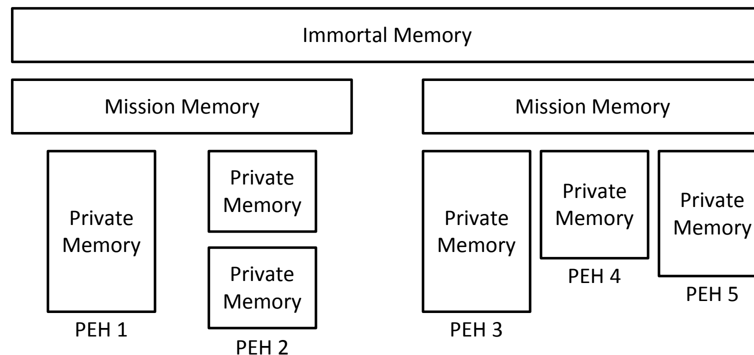


Figure 3. The memory model in SCJ.

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.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Dynamic class loading is outside the scope of the SCJ specification. 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 [9], being an alternative Java profile for hard real-time systems development, does allow the use of finalizers and [10] has demonstrated that timing analysis is possible. The contribution in this paper may be extended to include finalizers in future work.

3.2. Real-Time Execution Platform

The Hardware near Virtual Machine (HVM) [53, 32] is a Java virtual machine for embedded devices. The HVM supports the execution of Java programs on resource-constrained embedded hardware environments with as little as a few kB of RAM and 32 kB of ROM. The HVM supports iterative interpretation and a supplementary AOT Java-to-C translation mechanism or a hybrid of the two execution methods (enabling AOT compiled code to interact with the interpreted units and the other way around). Certain Java annotations can be used for specifying portions of code to be translated directly to C (the default is to do interpretation).

Section 5 contains measurements that show that Java code compiled by the HVM AOT compiler is 2-3 times slower than hand-written native C, and interpreted code is more than 10 times slower. Rudimentary measurements performed using the HVM also indicate that the code size of AOT compiled Java code correlates with the execution time and is 2-3 times higher than for hand-written native C. On the other hand, code size of interpreted code is approximately half that of hand-written native C, since the Java bytecode format is a tight format compared to common RISC instruction set architectures.

The capability to select what to compile and what to interpret enable that e.g. performance intensive parts of the application can be translated to native code thus trading off memory for improving performance.

In either case the HVM produces self-contained, strict ANSI-C code that has been specially crafted to allow it to be embedded into existing C based build and execution environments; environments which may be based on non standard C compilers and libraries. The HVM does not require a POSIX-like OS, nor does it require a C runtime library to be present for the target. The main distinguishing feature of the HVM is to support the stepwise addition of Java into an existing C based build and execution environment for low-end embedded systems. This will allow for the gradual introduction of the Java language, tools and methods into a existing C based development environment. Through program specialization, based on a static whole-program analysis, the application is shrank to only include a conservative approximation of actual dependencies, thus keeping down the size of the resulting Java based software components.

The HVM employs *JVM specialisation*; a JVM is produced specifically for hosting the JBC program of a given application. This is done using the ICECAP-TOOLS Eclipse-plugin, which analyzes the JBC program and produces an executable for the target platform. The analyses and transformations can be extended, and incorporate a number of (static) optimizations 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 JBC that will actually be used. Based on this, only those parts of the interpreter that are required are included. E.g. if the bytecodes for doing long arithmetic are not needed by the program, those parts of the interpreter are not included. In other words a JVM is produced specifically for hosting the Java bytecode program.

3.2.1. Real-Time Library Support An important requirement of the HVM is to support platforms where no OS is available (bare metal platforms). This raises the question of how to

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

support features such as preemptive scheduling and memory management - features which are usually offered by an underlying OS. To make real-time preemptive scheduling and predictable memory allocation available on bare metal platforms the Safety-Critical Java specification (SCJ), Level 0, 1 and 2 has been implemented for the HVM. The SCJ supports real-time preemptive scheduling of asynchronous handlers and predictable memory allocation using the scoped memory concept of SCJ. These high level concepts have been implemented almost entirely in Java. The overall structure of an SCJ application running on top of the HVM using the accompanying SCJ implementation (HVM-SCJ [53]) as well as the ICECAP-TOOLS SDK is shown in Figure 4. The *HW Interface* and the *VM Interface* have a very thin native layer written in assembler. The most important part here is to implement context switching. To port the SCJ implementation to a new platform this part must be written in the instruction set of the target. Only a handful of native code lines are usually required.

SCJ requires the programmer to adopt a specific programming style and program structure. The programmer can also choose to only utilize the core preemptive scheduling and scope memory facilities without using the entire SCJ infrastructure.

On platforms that do include an OS, Java level threads can be mapped to native OS threads, and scheduling can be delegated to the OS. This has been used to implement SCJ Level 2 and multicore support in the HVM†.

The HVM supports well known concepts for device level programming, such as Hardware Objects [52] and 1st level interrupt handling, and it adds some new ones such as *native variables*. The HVM is integrated with Eclipse. More details of the HVM is available elsewhere [33]‡.

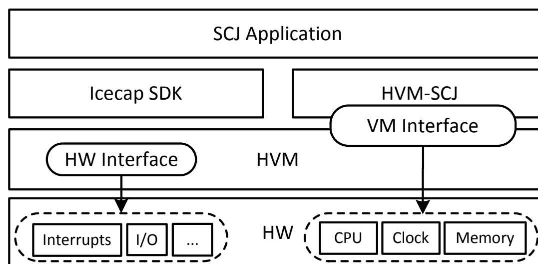


Figure 4. Constituents of an SCJ application on HVM.

3.3. Timing Analysis Tools

The work we present in this paper fits into our TETASARTS collection of open-source timing analysis tools. TETASARTS is model-based in the sense that the analyses are formulated as model checking problems using the modeling formalism of the UPPAAL model checker. Collectively, the timing analysis tools form a toolchain – Figure 5 shows the major components and their interactions.

Conducting the timing analyses requires two fundamental components in our modeling framework; the temporal behavior of the target application as well as the execution environment must be modelled. A model of the temporal behavior of the execution environment can be generated by the TETASARTS$_{\text{JVM}}$§ tool.

The modeling formalism used for capturing the temporal behavior of both the application and the execution environment is Networks of Timed Automata (NTA), which is the input language for the UPPAAL model checker. Here it suffices to say that a Timed Automaton (TA)

†`https://github.com/zs673/Multiprocessor-icecap-SCJ-RTE`
‡the HVM and ICECAP-TOOLS Eclipse-plugin can be downloaded from `http:www.icelab.dk`
§TETASARTS$_{\text{JVM}}$, HVM$_{\text{TP}}$, and generated models are available on the project website: `http://people.cs.aau.dk/~luckow/hvmtp/`

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

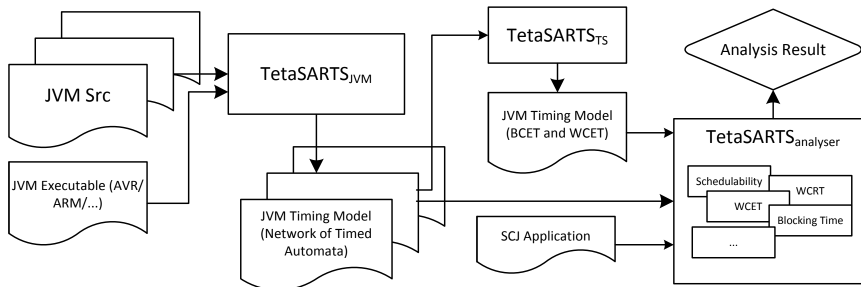


Figure 5. Overview of the TETASARTS toolchain.

is a finite state machine extended with real-valued clocks that progress synchronously as time elapses. A number of TAs can be composed into an NTA by using the (CCS-style) parallel composition operator. As an NTA, the TAs share clocks and action. For a full exposition of the semantics of TAs and NTA the reader is referred to, e.g., [7].

The temporal behavior as a TA is captured from the control-flow of each of the Java bytecodes of the target JVM. TETASARTS$_{\mathrm{JVM}}$ is capable of conducting this process. More specifically, TETASARTS$_{\mathrm{JVM}}$ takes as input the JVM executable to account for compiler optimisations, transformations, and more, that likely influences the temporal behavior of the Java Bytecodes. In addition, during the reconstruction process, the tool performs loop identification analysis. Whenever a loop is identified, the tool currently assumes that loop bounds are provided in the source code using annotations. If the annotations are missing, the bounds can be provided interactively during the analysis.

Reconstructing the control-flow for each Java bytecode is done by defining regions that enclose their implementations. Regions are defined using C macros: BEGIN_JBC(X) and END_JBC(X) denote the start and end of Java bytecode X. The macros expand to code that instruments the binary to allow TETASARTS$_{\mathrm{JVM}}$ to identify regions from which the control-flow must be reconstructed and translated to TAs. As an example of this specification, see Listing 1 which contains the implementation of `i2l` along with the region specification. We will use this as running example. The product of applying TETASARTS$_{\mathrm{JVM}}$ on the produced

```
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`.

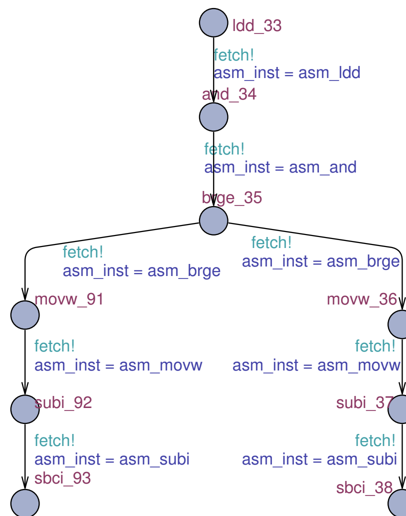


Figure 6. TA excerpt of `i2l`.

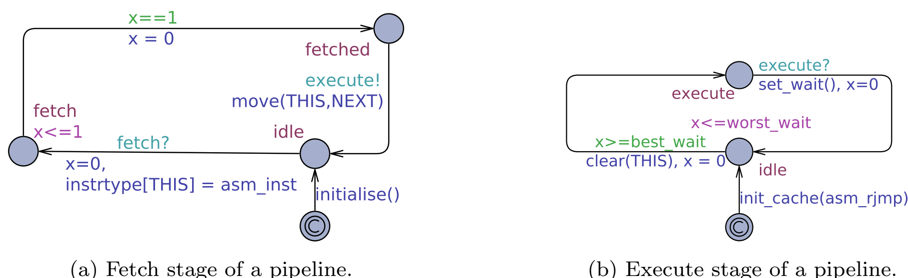
binary is shown in Figure 6. To understand how the control-flow and thus the temporal behavior is captured, note the location in the TA with the label `brge_35` – it branches to two other locations: `movw_91` and `movw_36` essentially capturing the control-flow of the conditional

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

statement in line 6 of Listing 1. Semantically, a transition captures the temporal behavior of executing the instruction stored in to the UPPAAL variable named `asm_inst`. Note that in general, capturing such detailed information of system behavior does not scale to large systems which is attributed the so-called state space explosion problem. In our case, however, each Java bytecode in isolation has sufficiently simple behavior to allow this explicit modeling to be feasible for model checking. Later in the paper, we will provide results that verify this claim. 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 resulting TAs produced from these have the same number of locations and only nondeterminism corresponding to the conditional statements in the code.

Composing all the generated TAs defines an NTA that captures the behavior of the entire JVM. We refer to this composition as the *JVM NTA*. This model, however, only captures the control-flow of the JVM – the actual timing of each Java bytecode depends on the hardware used: the TAs shown in Figure 7 captures the temporal behavior of the AVR ATmega2560 microcontroller. The composition of the TAs shown in the figure will be referred to as the *HW NTA*. We note that additional platforms can be supported by the provision of suitable TAs, e.g., the ARM7 and ARM9 models from METAMOC [20]. The channel labeled `fetch` is



(a) Fetch stage of a pipeline.

(b) Execute stage of a pipeline.

Figure 7. METAMOC hardware TAs [20].

used for 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. Similarly, the channel labeled `execute` is used for establishing communication 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` contains the worst and best case clock cycle execution times of the simulated instruction. The call to the function `init_cache(asm_rjmp)` is used for initialising the pipeline with the temporal behavior of the `rjmp` instruction. This is done to ensure that the timing model is safe since the pipeline will be filled with the `rjmp` instruction prior to executing the first instruction of any of the supported Java bytecodes.

JVM Timing Model refers to the complete timing model of the execution environment synthesised from the HW NTA and the JVM NTA (see Figure 5). As such, it simulates the timing behavior of the target JVM. The JVM Timing Model can be further synthesised with a model of the target application yielding a model where various timing related analyses can be conducted such as analysing schedulability and WCRTs of the tasks. These analyses are performed using the TETASARTS$_{\text{ANALYSER}}$ tool. For details on this tool, the reader is referred to [37, 38, 39].

As an alternative, the JVM Timing Model can also form the basis for representing the temporal behavior abstractly as a timing scheme, i.e., as summaries of execution times. The TETASARTS$_{\text{TS}}$ tool as shown in Figure 5 is used for generating this model. For all TAs corresponding to the Java bytecodes, the tool analyses their respective WCET and BCET using the sup- and inf-query extensions of the UPPAAL model checker. These queries explore the state space to determine the maximum and minimum values of the specified clock-variables.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

4. RE-DESIGNING THE HVM

The JVM Specification does not dictate how interpretation is carried out; typically, JIT compilation is employed for improved average case performance of the application. This, however, is at the expense of added JVM size due to the complexity of the interpreter in turn making JIT-compilation an infeasible choice when considering resource-constrained embedded systems. For real-time systems, another concern is the difficulty in reasoning about timing of code because it is application- and usage-dependent; the specific application and profile information obtained during run-time determines whether a code section is translated into native code or not.

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 are 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:

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
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 conversion. 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 Java bytecode classes not mentioned in the above and certain JVM features are not naturally time predictable. For that reason, special treatment must be done in order to make them time predictable. The Java bytecodes and concepts of concern have been classified as:

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

The purpose of the following sections is to elaborate how we treated each of the classes in order to make the JVM amenable to timing analysis

4.1. Object Allocation

There are four essential Java bytecodes that are used for object allocations:

`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*.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Object allocations are performed in the HVM by incrementing a pointer by the size of the object that is to be allocated. For that reason, object allocation can be performed in constant time. However, the HVM performs zeroing of the memory at allocation time, which is an operation that has execution time linear in the size of the to-be 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. Therefore, a better approach is to harness SCJ: in the setup 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 [17], an RTSJ extension to the GNU GCJ compiler. Thus on first entry to a scoped memory area it is already zeroed. Therefore, whenever a scoped memory is exited, the memory area is zeroed (again) which takes time linear in the size of the allocated scope. We can therefore assume that scoped memory areas are always zeroed a priori on entry. This operation is done in Java space using the native variables concept [53], 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

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 analysis for guaranteeing that certain exceptions are never thrown (e.g., that the denominator cannot become zero in divisions). TetaSARTS_{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 by reconstructing the call graph and determining 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.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

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$_{\mathrm{JVM}}$ 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.

`invokedynamic` used for invoking Java closures (lamda expressions).

4.3.1. Redesigning Method Invocation Management In the original implementation of the HVM, recursion was used for handling method invocations. Specifically, when one of the method invoking Java bytecodes was executed, the HVM interpreter was invoked recursively with the method body of the callee as argument. An overview of this process is shown in Listing 4. Upon returning from the invoked method, control is naturally returned to the callsite of the method invoking instruction. This allows the interpreter to use the C stack for implementing method invoking Java bytecodes, rather than maintaining it own stack implementation.

```
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 interpreter introduces various issues in relation to time predictability and the approaches we use for timing analyses. First of all, any method call in the application is turned into a recursive call of the interpreter, thus it becomes difficult to separate the state of the interpreter from the state of the application it is executing. Recursion also poses similar problems to unbounded loops: the recursion depth can be unbounded. For that reason a bound must either be approximated potentially yielding very conservative results or code annotations can be used to specify the recursion depth.

Another inherent issue of recursion pertains to modeling using TA; in our framework, 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). However, a TA is not capable of receiving on a co-action if it has signalled on the corresponding action – this would intuitively have captured the semantics of a recursive call. Hence, to resolve this, it would be necessary to make inherent changes to the timing analyses approach; to allow recursion one could instantiate a number of the same TA corresponding to the depth of the recursive call and let them synchronise in a sequential manner. This however, brings back the original problem of determining the call depth and will furthermore affect model checking time since the overall state of the model is comprised of the current locations of all instantiated templates.

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

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

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. Redesigning Method Dispatch `invokespecial` and `invokestatic` statically dispatch control to the specified method. Therefore, they do not imply issues in terms of time predictability because determining the callee and dispatching control is done in constant time.

Virtual method calls as produced by `invokevirtual` and `invokeinterface` are usually implemented using well known techniques to perform method constant-time dispatch for single and multiple inheritance. The original implementation of the HVM used vtables generated by Icecap-tools for each class. At execution time a look up based on a simple index into the vtable would yield the actual target method of the invocation. The interpreter source performing the call will in this case be an index into an array of direct or indirect function pointers. Such a call site is not directly time predictable by the offline analysis tools, since the possible targets are not directly referred in the source code.

The HVM has been refactored to make the runtime look up of the possible target methods time predictable. The strategy employed is as follows: 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 [53]. Because of the devirtualisation performed during the analysis phase only truly virtual call sites are maintained – the rest are treated as `invokespecial`. In the case of Java-to-C compilation each virtual call site 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.

```
1 class Polygon {
2   abstract int area();
3 }
4 class Square extends Polygon { ... }
5
6 class Rectangle extends Square { ... }
7
8 class Circle extends Polygon { ... }
9 //...
10 ArrayList<Polygon> figures = new ArrayList<Polygon>();
11 figures.add(new Square(2));
12 figures.add(new Rectangle(2, 3));
13 figures.add(new Circle(3));
14
15 int sum = 0;
16 for (Polygon polygon : figures) {
17   sum += polygon.area();
18 }
```

Listing 6: Translating virtual method invocations (Java Source).

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

The virtual method invocation at line 19 in Listing 6 can be to any of three methods. This gets translated into the C code included in Figure 7.

```
1 switch (classIndex) {
2   case 18:
3     rval_m_85 = test_Circle_area(sp, i_val3);
4     break;
5   case 30:
6     rval_m_85 = test_Rectangle_area(sp, i_val3);
7     break;
8   case 5:
9     rval_m_85 = test_Square_area(sp, i_val3);
10    break;
11 }
```

Listing 7: Translating virtual method invocations (C source).

This switch explicitly mentions the possible targets and lends itself more easily to static analysis. 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 look up. 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 look up 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.

In summary, the strategy described here modifies the `invokevirtual` and `invokeinterface` bytecode by encoding the switch statement case values into the bytecode. The WCET of the interpreter source performing the actual call can be given a conservative estimate of the size of the largest jump table multiplied by the maximum height of the class hierarchy. A drawback to this strategy is that the WCET becomes a global property. Even though the ICECAP-TOOLS knows the possible receiver types for each call site this is not taken into account. Still the WCET of the interpreter source performing the call is as tight as it can be, since this code will interpret all virtual calls in the hosted application.

To make the WCET estimate more accurate the interpreter source must be extended, possibly by adding custom bytecodes handled by different statements in the interpreter, each of which can be given a more tight WCET estimate.

4.3.3. Supporting Java Closures As of Java 8 the concept of closures for Java has been made available to the programmer. Java closures can be used to cast methods to an interface and later invoke that interface (see Listing 8).

```
1 public class TestInvokeDynamic {
2   private interface Action {
3     void doIt();
4   }
5   private static void foo() {
6     System.out.println("do it!");
7   }
8   public static void main(String[] args) {
9     Action action = TestInvokeDynamic::foo;
10    action.doIt();
11  }
12 }
```

Listing 8: Casting a method to an interface.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

The closure facility can also be used as a convenient way to declare anonymous implementors of an interface, as illustrated in Listing 9.

```
1 public class TestInvokeDynamic {
2   private static interface Adder {
3     int add(int x, int y);
4   }
5
6   public static void main(String[] args) {
7     Adder adder = (x, y) -> {
8       return x + y;
9     };
10    int x = adder.add(40, 2);
11    System.out.println("x = " + x);
12  }
13 }
```

Listing 9: Anonymous implementors.

The interfaces `Action` and `Adder` above are called *Functional interfaces* and may only declare one method. This method defines the signature of the closure. The invocation of the closure - see line 10 in Listing 8 and line 10 in Listing 9 - are handled by `invokeinterface` and that bytecode has not been extended in any way because of the introduction of closures. The creation of the closure object itself - see line 12 in Listing 8 and line 8 Listing 9 - is handled by a new bytecode `invokedynamic` introduced to the Java VM. The `invokedynamic` bytecode does not invoke the closure – it creates an instance of the functional interface. So the `invokedynamic` bytecode is similar to `new` in its effect – it creates an instance of a class (that has 1 method declared).

To implement `invokedynamic` in the JVM the following must be known,

- Which functional interface is being instantiated?
- Which method (class name, method name, method signature) implements the functional interface?

Then the JVM calls a bootstrap method with a series of arguments. The bootstrap method is called `LambdaMethodFactory.metafactory` and implemented in the JDK. It generates a `CallSite` instance which has a `MethodHandle` that returns an instance of the functional interface when invoked. So executing `invokedynamic` includes performing a callback from the JVM into a specific JDK method.

In the HVM it is not possible to call `LambdaMethodFactory.metafactory` since the dependency extent of that method is too large. Instead the HVM simulates that functionality. The major part of that simulation is done statically by the compiler. For each occurrence of `invokedynamic` it consults the class file and goes through the following steps:

1. `invokedynamic` is 3 bytes – last 2 bytes is a short index into the constant pool of the enclosing class file
2. This index points to a `ConstantInvokeDynamic` at that index (new in Java 8)
3. That constant contains a `BootstrapMethodIndex` and a `NameAndType` constant
4. The `NameAndType` gives the functional interface method and signature
5. Then the enclosing class file is searched for a `BootstrapMethods` attribute. This contains a single `BootstrapMethod` constant
6. That constant contains a `bootstrapMethodRef` and a list of bootstrap arguments
7. The `bootstrapMethodRef` is a `ConstantMethodHandle` – it has a `referenceKind` and a `ConstantMethodref`
8. The `ConstantMethodref` refers to the bootstrap method `LambdaMethodFactory.metafactory`
9. Then the bootstrap arguments are scanned, looking for a `ConstantMethodHandle`
10. This handle has (as above) a `referenceKind` and a `ConstantMethodref`

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

11. The `ConstantMethodref` refers the closure method (containing class, name and signature)
12. If the method is an anonymous method (like in Listing 9) it will be called `lambda$X` and the byte code for it can be found in the enclosing class file.

We now have (1) the functional interface being implemented, (2) the bootstrap method, and (3) the actual method containing the bytecode of the closure. Now the HVM does the following,

- Throws away the bootstrap method (not used)
- Creates a synthetic class containing one method which is the method with the bytecode of the closure
- Changes the name of that method to be the same as the name declared in the interface

Now this synthetic class is added to the pool of classes managed by the HVM as if it were an ordinary class. When executed at runtime the `invokedynamic` bytecode becomes the same as `new`, since we know which class to instantiate (the synthetic class). The result of this is that doing time analysis of the invokedynamic bytecode requires the same effort as when analyzing the `new` bytecode (see Section 4.1).

This initial support for `invokedynamic` is a drastic over-simplification of what the `LambdaMethodFactory` does. Still, this will be able to handle all uses of the `invokedynamic` bytecode when compiled from Java source by the Java compiler, but the bytecode is intended to be used in other ways as well. An example could be translating other languages (e.g. Scala) into Java bytecode, and in such scenarios this initial support of the `invokedynamic` bytecode is not sufficient.

As support will be extended in the future it will likely turn out that the runtime behavior of `invokedynamic` changes as well, requiring a revisit of the timing analysis of `invokedynamic`.

4.4. Type Checking Reference Types

Runtime type checking between reference types is achieved by the `checkcast` and `instanceof` Java bytecodes. In an 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.

```
1 //...
2 while (subClass != (unsigned short) -1) {
3   if (subClass == superClass) {
4     return 1;
5   } else {
6     subClass = pgm_read_word(&classes[subClass].superClass);
7   }
8 }
9 //...
```

Listing 10: Excerpt of the `checkcast` Java bytecode.

The original implementation of the HVM performs type checking in this way (see Listing 10), but we redesigned it in HVM_{TP} to achieve constant time performance. This is done by exercising the class hierarchy prior to HVM_{TP} construction; type compatibility among the classes is stored in a matrix that is incorporated in the final HVM_{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 11.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

```
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 11: 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 occupies one bit. Another note on this solution is that it precludes the use of features that changes the class hierarchy dynamically such as dynamic class loading. However, dynamic class loading is not supported by the HVM in its current state, and dynamic class loading is also a contributor to temporally unpredictable behaviour due to dependencies to I/O operations etc. Therefore, precluding the use of dynamic class loading is not seen as a great deficiency of this revised design when taking into account that the application domain is embedded real-time systems. It would be possible to implement constant time dynamic subtype tests and even support dynamic class loading, at the expense of some extra space per class, using the R&B subtype test algorithm by Palacz and Vitek [43].

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 [36].

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$_{TP}$ 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$_{TP}$ 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$_{TP}$ 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

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

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_{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_{TP} has been annotated with these bounds and this information is subsequently used by $\text{TETASARTS}_{\text{JVM}}$ 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. TIMING ANALYSIS OF THE HVM

We have used $\text{TETASARTS}_{\text{JVM}}$ to construct a complete JVM Timing Model of the supported Java bytecodes of HVM_{TP} on an AVR ATmega2560 microcontroller. All reported results have been obtained running the tool on a 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 other applications. We will here demonstrate the use of $\text{TETASARTS}_{\text{TS}}$ for generating corresponding timing schemes. The timing scheme takes approximately 4.5 hours and approximately 5 days for the model excluding and including exception handling, respectively. 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. The `ldc_*` bytecodes may be used to load raw character data that is turned into a Java String object. This involves calling a constructor of class String, which in turn may set in motion a non trivial sequence of computations, involving object creation and constructor execution. 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 [8, 24, 38], 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.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

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. Analysis of applications from the JemBench Benchmark suite [50] shows that BenchLift uses only 30 distinct Java bytecodes, BenchKfl uses 40 and BenchUdpIp uses 30. A larger application, TestSCJSingleBoundedBuffer from the HVM distribution, uses 50 distinct Java bytecodes.

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. Table I shows the results of five samples for selected Java bytecodes.

Bytecode	TetaSARTS$_{TS}$		Measured		
	BCET	WCET	Avg	Low	High
i2l	129	136	130	130	130
aload_*	79	79	79	79	79
new	469	1,715	1,568	1,568	1,568
invokedynamic	469	1,715	1,568	1,568	1,568
ireturn	505	1,080	893	865	976
invokespecial	501	977	710	639	772
iinc	191	194	192	192	192

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

Let us first note that all results are safe, i.e. for all bytecodes, BCET $\leq$ Low and High $\leq$ WCET. Also note that the BCET and WCET of `invokedynamic` and `new` are the same since they share the same implementation as outlined in Section 4.3.3. 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.

The results demonstrate that some Java bytecodes have high cycle counts. This is in part due to the interpreter of HVM$_{TP}$, see Table II below, but also in part due to the fact that some Java bytecodes are rather complex and therefore exhibit high cycle counts even when they are implemented in hardware or microcode. On the JOP, the `invokespecial` instruction has a cycle count of $74 + 3 * r + l$ and `ireturn` has a cycle count of $23 + r + l$ where r is additional wait states for memory access and l is the time on a method cache miss or hit and further dependent on the length of the method and the memory access time [49].

	C	KESO (AOT)	FijiVM (AOT)	HVM (AOT)	GCJ (AOT)	JamVM (Int.)	HVMi (Int.)	CACAO (JIT)	HotSpot (JIT)
Quicksort	100	101	136	111	172	697	4,761	147	156
Trie	100	93	54	136	245	772	1,982	294	234
Determinant	100	59	37	96	171	544	1,664	294	48
WordReader	100	251	218	177	328	975	4,979	263	142
Total	100	126	111	130	229	747	3,346	250	145

Table II. Instruction count comparison

Table II presents measurements to compare the execution time of HVM hosted programs with other similar VMs. The measurements are based on 4 benchmark programs - each written in both native C and in Java. The first column in Table II shows the execution time of the program using GCC. The following columns in table II lists the execution times using a range of Java execution environments. The measurements were performed on a desktop Linux host (more details about the benchmarks programs are available elsewhere [33]). Numbers have been normalized, such that the C execution time is index 100. The measurements show that

Copyright © 0000 John Wiley & Sons, Ltd.

for the benchmarks used, the AOT compiler of $\mathrm{HVM}_{\mathrm{TP}}$ (column `HVM`) produces code that on average is approximately a factor 30 faster than using the interpreter (column `HVMi`).

The reason the HVM interpreter is significantly slower than the JamVM is because of portability. The HVM interpreter does not utilize well-known optimization techniques such as *computed gotos* [25] or register allocated variables (the `register` keyword in C), since this would only allow translation of the interpreter source using specific compilers (e.g. GCC). Additionally the HVM allows control to flow from interpreted code to AOT compiled code and vice versa. This feature prevents several obvious optimizations in the interpreter.

6. TIMING ANALYSIS OF SCJ APPLICATIONS ON HVM

The previous section showed that each Java bytecode is implemented time predictably by the $\mathrm{HVM}_{\mathrm{TP}}$. This result can be used when analyzing timing properties of SCJ applications running on the $\mathrm{HVM}_{\mathrm{TP}}$.

Using the timing scheme it would be possible to use the CFG, augmented with loop bounds, of an SCJ application and a classic integer linear programming (ILP) algorithm to estimate WCET by solving the maximum cost circulation problem of the set of constraints that describes program behavior. This is an approach used by many tools for timing analysis of C programs, or rather executables generated from C programs, such as; Ottawa [5], Chronos [34], Heptane [15], TuBound [46], SWEET [42], Bound-T [29], RapiTime [47] and the aiT WCET Analyzer [23]. Armed with WCET estimates for each task of a system, response time analysis [12] is traditionally used for concluding on schedulability.

To the best of our knowledge, the WCA [51] tool is the only tool that supports WCET analysis for Java bytecode using ILP. However, WCA only offers analysis of SCJ applications running on the JOP.

An alternative to WCET analysis based on ILP and schedulability analysis based on response time analysis, is model-based timing analysis. A representation, usually a CFG, of the program under analysis, is translated to a modeling formalism, e.g., Timed Automata, and timing analysis is then formulated as a reachability problem using an appropriate logic such as TCTL. Examples of model-based timing analysis tools are; METAMOC [20] for executables, and TetaJ [24], SARTS [11], TetaSARTS [38] and SymRT [40], a recent extension of TetaSARTS with symbolic execution. WCA also offers WCET analysis using model checking.

We now summarize the WCET estimates of SCJ applications on the $\mathrm{HVM}_{\mathrm{TP}}$ running on the AVR ATmega2560 obtained from TetaSARTS. We also report on the use of TetaSARTS for automated schedulability analysis. Note that an advantage of using model checking for schedulability analysis is that task interactions can be taken into account and thus some systems which might be deemed non-schedulable using a classic response time analysis, might in fact be found schedulable [11].

As examples of WCET estimates, we have used the Java implementations of a subset of the algorithms from the Mälardalen WCET benchmark suite [27]: Binary Search (54 LOC), Bubble Sort (34 LOC), Quick Sort (109 LOC), Insertion Sort (39 LOC), Iterative Fibonacci (40 LOC), and Select Smallest (137 LOC) which selects the nth smallest number in an array. For the sorting algorithms, the array is initialized with symbolic values. For Binary Search, the search key is symbolic. For Fibonacci, the input value, n, is symbolic and constrained such that $1 \leq n \leq 30$. For Select Smallest, an array of size 20 is filled with concrete values.

To give indications of the precision of TetaSARTS we compared with measurements of WCET obtained by using inputs yielding the best and worst case behavior (e.g. for Bubble Sort, a sorted and unsorted list). The measurements have been obtained by using the debugging facilities of Atmel Studio 6. For this set of experiments, we used a laptop with an Intel Core i7-2620M CPU @ 2.70GHz with 8 GB of RAM. The peak memory consumption for symbolic execution is 500-700 MB for all examples. Uppaal peaks at 50-200 MB during model checking.

Table III shows the comparison of estimated and measured WCET on HVM and AVR ATmega2560. First note that all estimates indicate *safety*, that is $\mathrm{WCET}_{symrt} \geq \mathrm{WCET}_{\mathrm{m}}$.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

System	TetaSARTS WCET [cycles]	TetaSARTS Analysis Time [seconds]	Measured WCET [cycles]
Binary Search	70,153	2	23,262
Bubble Sort	287,526	31	37,388
Quick Sort	133,134	228	50,437
Insertion Sort	244,680	4	70,028
Fibonacci	142,764	6	29,850
Select Smallest	3,452,824	134	221,223

Table III. Using TetaSARTS for systems on HVM and AVR.

The pessimistic results of TetaSARTS primarily stem from over-approximating the iterations of nested loops with interdependencies. It might be possible to achieve higher precision using symbolic execution [40]. Also note that for Quick Sort, it is relatively difficult to exercise and measure the path yielding the worst case behavior since it depends on the pivot element selection.

We have used TetaSARTS for schedulability analysis of the Minepump control system [12, 24] (0.5 KLOC), the Real-Time Sorting Machine (RTSM) [11] (0.3 KLOC) and a variant of MD5SCJ [38] with multiple tasks (0.4 KLOC). For this set of experiments, we used an application server with an Intel Xeon X5670 @ 2.93GHz CPU and 32 GB of RAM. The results are shown in Table IV. In all cases, the systems have been deemed schedulable, and the results show the analysis times and memory consumptions.

The TetaSARTS tool failed the MER Arbiter (3.6 KLOC) that models a flight software component for the Mars Exploration Rover (MER) developed at NASA JPL [4] and the Lift real-time system from Jembench [50] with 18 tasks. Even though MER is relatively simple, the dependency extent computed in TetaSARTS for generating the CFG is too large. However, these two systems can be analyzed by SymRT. We refer the reader to [40] for a comparison of TetaSARTS and SymRT (and the WCA tool used on systems running on the JOP).

System	Analysis time	Memory
Minepump	2s	11 MB
RTSM	1m 2s	17 MB
MD5SCJ	8s	17 MB

Table IV. Schedulability analysis using TetaSARTS.

A further use of the timing model is for easy profiling and performance analysis of the Java bytecodes described in terms of distributions of execution times [39].

Clearly there are systems for which analysis is intractable. This is for instance the case for the MER and Lift applications using TetaSARTS. However, note that the number of branches in the code is the limiting factor to scalability since they are modelled as choices in the TA. For the schedulability analysis, the number of tasks also affects scalability since the model grows exponentially in the number of components. However, since schedulability is viewed as a reachability problem, it may be possible to partly alleviate the scalability issue by translating the model into the subset of the Uppaal modeling language supported by the opaal+LTSmin system [18]. In [19] opaal+LTSmin demonstrates a speedup of 40 on a 48 core machine compared to Uppaal. Future work will investigate this direction.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

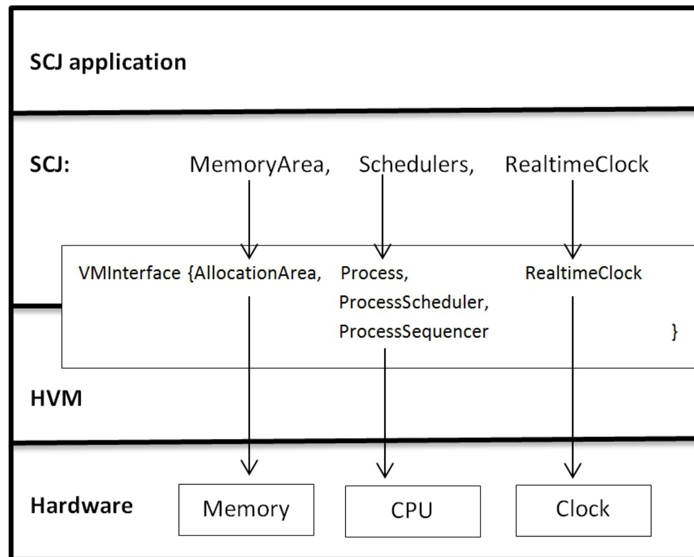


Figure 8. Overview of the HVM SCJ architecture.

7. DISCUSSION

A common architectural pattern is to divide a software system into layers separated by interfaces between the layers. A well known example of this architecture is the Open Systems Interconnection model (OSI) [56]. Another example is the way that SCJ is implemented in the HVM as illustrated in Figure 8.

For both the OSI model and the HVM SCJ architecture the dependencies go from upper layers to lower layers and not vice versa. This means that when implementing, e.g., the HVM it is not required to know which application it is going to host. The same when implementing the VM interface: the `AllocationArea`, `Process` and `RealtimeClock` are implemented as general low level concepts that can be used by any upper layer - not just the SCJ. This direction of dependencies will seem natural to software architects, indeed dependencies in the reverse direction, e.g., requiring intimate knowledge about the SCJ or application layer when implementing the lower layers of the HVM, would be considered unnatural.

When implementing a VM for the Java language it quickly becomes clear that the layered architecture pattern has been broken in some cases: the JVM needs to know about intimate details of the upper layers, more specifically the JDK it is going to host. For a bytecode like `idiv` it is stated that a `java.lang.ArithmeticException` must be thrown, so the VM must know about certain classes being available and their constructor signature. Also allocating a new exception requires interaction with the memory management system. So a seemingly simple bytecode like `idiv` is intimately intertwined with other layers further up the software stack. For the `invokedynamic` bytecode this reverse dependency becomes even clearer: as described in Section 4.3.3 the presence of several classes and APIs are required to implement its behavior. A bytecode like `iadd` is well behaved and does not depend on knowledge about layers further up the software stack.

The presence of these reverse dependencies, of which many others exist than the ones mentioned above, make the implementation of the VM harder and the timing analysis of the resulting implementation harder. Even so Section 5 shows that timing results have been achieved. When designing the next successful programming language we can only encourage the designers not to introduce reverse dependencies - this will make the job of the virtual machine and timing tools developers much easier.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

8. 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 all levels of the emerging Safety Critical Java (SCJ) profile. 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. The techniques presented have been applied to a specific Java VM, but can be applied to achieve time predictability of other Java VMs as well.

We have demonstrated that a complete timing model of HVM_{TP} can be constructed using $\text{TetaSARTS}_{\text{JVM}}$; 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. Although not the case for HVM_{TP} as we have demonstrated, we note that due to the well-known state space explosion problem of model checking, the analysis can become intractable for complex Java Bytecode implementations. 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. 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 [39].

We also took a first look at how to support the new Java 8 language feature of Lambda expressions in a Safety Critical Java context, we looked in particular at how the `invokedynamic` bytecode can be implemented in a time predicatble way and integrated in the HVM_{TP}. Even before the introduction of the `invokedynamic` bytecode the JVM architecture was not well layered, as implementations of lower layers such the `idiv` bytecode needs intimate details of upper layers. This has been further exacerbated with the introduction of the `invokedynamic` bytecode as it requires several classes and APIs to be present to implement its correct behavior. With a time predictable implementation of the `invokedynamic` bytecode it is now possible for SCJ applications to take advantage of lambda abstractions to make application code more concise and elegant. Furthermore, we expect that the SCJ specification itself could be revised and made more concise and elegant using lambda abstractions.

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 want to investigate pre-computing timings of Java bytecodes with application-dependent temporal behavior, which should improve precision and we want to investigate further improvements of the precision of the timing model produced by $\text{TetaSARTS}_{\text{JVM}}$, e.g. by using symbolic execution. Furthermore, the HVM facilitates a tight integration with (legacy) code in C, i.e. handlers in Java can directly be called from handlers in C and visa versa, clearly at the expense of more complex analysis, however, for some systems it is not possible to port all parts of the code to Java. We envision that the techniques of I/O automatas [21] used in the ECDAR tool, analysis of C code using METAMOC [20] and the notion of schedulability abstraction [13] could be combined to provide a framework for analysis of such mixed applications.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

REFERENCES

1. Aicas-GmbH. *JamaicaVM 6.0 - User Manual: Java Technology for Critical Embedded Systems*, 2010.
2. aJile Systems Inc. `http://www.ajile.com/` (Last accessed 9 February 2016).
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. D. Balasubramanian, C. Pasareanu, G. Karsai, and M. Lowry. Polyglot: Systematic analysis for multiple statechart formalisms. In N. Piterman and S. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *Lecture Notes in Computer Science*, pages 523–529. Springer Berlin Heidelberg, 2013.
5. C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In *Proceedings of the 8th IFIP WG 10.2 International Conference on Software Technologies for Embedded and Ubiquitous Systems*, SEUS'10, pages 35–46, Berlin, Heidelberg, 2010. Springer-Verlag.
6. J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal — a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III : Verification and Control: Verification and Control*, pages 232–243, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.
7. J. Bengtsson and W. Yi. *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, chapter Timed Automata: Semantics, Algorithms and Tools, pages 87–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
8. T. Bøgholm, C. Frost, R. R. Hansen, C. S. Jensen, K. S. Luckow, A. P. Ravn, H. Søndergaard, and B. Thomsen. Towards harnessing theories through tool support for hard real-time java programming. *Innov. Syst. Softw. Eng.*, 9(1):17–28, Mar. 2013.
9. 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, pages 150–159, New York, NY, USA, 2009. ACM.
10. 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, pages 1–7, New York, NY, USA, 2010. ACM.
11. 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, pages 106–114, New York, NY, USA, 2008. ACM.
12. A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2001.
13. T. Bøgholm, B. Thomsen, K. G. Larsen, and A. Mycroft. Schedulability analysis abstractions for safety critical java. In C. Hu, G. Karsai, J. Xu, A. Polze, J. Wang, and A. J. Wellings, editors, *ISORC*, pages 71–78. IEEE Computer Society, 2012.
14. Y. Chan, A. Wellings, I. Gray, and N. Audsley. On the locality of java 8 streams in real-time big data applications. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 20:20–20:28, New York, NY, USA, 2014. ACM.
15. A. Colin and I. Puaut. A modular & retargetable framework for tree-based wcet analysis. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, ECRTS '01, pages 37–, Washington, DC, USA, 2001. IEEE Computer Society.
16. J. Connors. An embedded java 8 lambda expression microbenchmark. `https://blogs.oracle.com/jtc/entry/an_embedded_java_8_lambda` (Last accessed 9 February 2016, 2014.
17. A. Corsaro and D. C. Schmidt. *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE: Confederated International Conferences CoopIS, DOA, and ODBASE 2002 Proceedings*, chapter The Design and Performance of the jRate Real-Time Java Implementation, pages 900–921. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.
18. A. E. Dalsgaard, R. R. Hansen, K. Y. Jørgensen, K. G. Larsen, M. C. Olesen, P. Olsen, and J. Srba. *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, chapter opaal: A Lattice Model Checker, pages 487–493. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
19. A. E. Dalsgaard, A. Laarman, K. G. Larsen, M. C. Olesen, and J. Pol. *Formal Modeling and Analysis of Timed Systems: 10th International Conference, FORMATS 2012, London, UK, September 18-20, 2012. Proceedings*, chapter Multi-core Reachability for Timed Automata, pages 91–106. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
20. A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In B. Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 113–123, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.
21. A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Timed I/O automata: A complete specification theory for real-time systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '10, pages 91–100, New York, NY, USA, 2010. ACM.
22. 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, pages 96–105, New

Copyright © 0000 John Wiley & Sons, Ltd.

York, NY, USA, 2011. ACM.

23. C. Ferdinand, R. Heckmann, and B. Franzen. Static memory and timing analysis of embedded systems code. In P. Groot, editor, *Proceedings of VVSS2007 - 3rd European Symposium on Verification and Validation of Software Systems, 23rd of March 2007, Eindhoven*, number TUE Computer Science Reports 07-04, 2007.
24. 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, pages 30–39, New York, NY, USA, 2011. ACM.
25. GNU-Project. Labels as values. `https://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Labels-as-Values.html` (Last accessed 9 February 2016, 2016.
26. J. Gosling and G. Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
27. J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In B. Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 136–146, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.
28. K. Hammond and G. Michaelson. Hume: A domain-specific language for real-time embedded systems. In *Proceedings of the 2Nd International Conference on Generative Programming and Component Engineering*, GPCE '03, pages 37–56, New York, NY, USA, 2003. Springer-Verlag New York, Inc.
29. N. Holsti and S. Saarinen. Status of the Bound-T WCET tool. In *Proceedings of the Euromicro Worst-Case Execution Time Workshop (WCET)*, 2002.
30. 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.
31. E. Y.-S. Hu, A. Wellings, and G. Bernat. *Real-Time and Embedded Computing Systems and Applications: 9th International Conference, RTCSA 2003, Tainan City, Taiwan, February 18-20, 2003. Revised Papers*, chapter XRTJ: An Extensible Distributed High-Integrity Real-Time Java Environment, pages 208–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
32. S. Korsholm. Hvm. `www.icelab.dk` (Last accessed 9 February 2016, 2015.
33. S. Korsholm and A. U. I. for Datalogi. *Java for Cost Effective Embedded Real-time Software: Ph.D. Dissertation*. Publication (Department of Computer Science, The Faculties of Engineering, Science, and Medicine, Aalborg University). Department of Computer Science, The Faculties of Engineering, Science, and Medicine, Aalborg University, 2012.
34. X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A Timing Analyzer for Embedded Software. *Science of Computer Programming*, 69(1):56–67, 2007.
35. T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.
36. 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.
37. 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.
38. 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, pages 11–20. ACM, 2013.
39. 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.
40. K. S. Luckow, C. S. Păsăreanu, and B. Thomsen. Symbolic execution and timed automata model checking for timing analysis of java real-time systems. *EURASIP Journal on Embedded Systems*, 2015(1):1–16, 2015.
41. K. S. Luckow, B. Thomsen, and S. E. Korsholm. Hvmtp: A time predictable and portable java virtual machine for hard real-time embedded systems. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 107:107–107:116, New York, NY, USA, 2014. ACM.
42. MRTC-Center. SWEET (SWEdish Execution Time tool). `http://www.mrtc.mdh.se/projects/wcet/sweet/` (Last accessed 9 February 2016).
43. K. Palacz and J. Vitek. *ECOOP 2003 – Object-Oriented Programming: 17th European Conference, Darmstadt, Germany, July 21-25, 2003. Proceedings*, chapter Java Subtype Tests in Real-Time, pages 378–404. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.
44. 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, pages 110–119, New York, NY, USA, 2009. ACM.
45. 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, pages 95–101, New York, NY, USA, 2010. ACM.
46. A. Prantl, M. Schordan, and J. Knoop. TuBound - A Conceptually New Tool for Worst-Case Execution Time Analysis. In R. Kirner, editor, *8th International Workshop on Worst-Case Execution Time Analysis (WCET'08)*, volume 8 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. also published in print by Austrian Computer Society (OCG)

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

with ISBN 978-3-85403-237-3.
47. RapiTime. RapiTime WCET tool homepage. Website. `http://www.rapitasystems.com` (Last accessed 9 February 2016).
48. M. Schoeberl. *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops: OTM Confederated International Workshops, HCI-SWWA, IPW, JTRES,WORM, WMS, and WRSM 2003, Catania, Sicily, Italy, November 3-7, 2003. Proceedings*, chapter JOP: A Java Optimized Processor, pages 346–359. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.
49. M. Schoeberl. *JOP Reference Handbook: Building Embedded Systems with a Java Processor*. CreateSpace, Paramount, CA, 2009.
50. M. Schoeberl, T. B. Preusser, and S. Uhrig. The Embedded Java Benchmark Suite JemBench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 120–127, New York, NY, USA, 2010. ACM.
51. M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber. Worst-Case Execution Time Analysis for a Java Processor. *Software: Practice and Experience*, 40(6):507–542, 2010.
52. M. Schoeberl, C. Thalinger, S. Korsholm, and A. P. Ravn. Hardware objects for java. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 445–452, May 2008.
53. 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, pages 44–53, New York, NY, USA, 2012. ACM.
54. Sun. Sun Java Real-Time System. `https://docs.oracle.com/javase/realtime/doc_2.2/release/JavaRTSQuickStart.html` (Last accessed 9 February 2016), 2009.
55. R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.
56. H. Zimmermann. Innovations in internetworking. chapter OSI Reference Model — The ISO Model of Architecture for Open Systems Interconnection, pages 2–9. Artech House, Inc., Norwood, MA, USA, 1988.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe