WCET Analysis of Java Bytecode Featuring Common Execution Environments

Christian Frost, Casper Svenning Jensen, Kasper Søe Luckow, Bent Thomsen
Department of Computer Science, Aalborg University
Selma Lagerlöfs Vej 300
DK-9220, Aalborg East, Denmark
{chrfrost,semadk,luckow,bt}@cs.aau.dk

ABSTRACT

We present a novel tool for statically determining the Worst Case Execution Time (WCET) of Java Bytecode-based programs called *Tool for Execution Time Analysis of Java bytecode* (TetaJ). This tool differentiates itself from existing tools by separating the individual constituents of the execution environment into independent components. The prime benefit is that it can be used for execution environments featuring common embedded processors and software implementations of the JVM. TetaJ employs a model checking approach for statically determining WCET where the Java program, the JVM, and the hardware are modelled as Networks of Timed Automata (NTA) and given as input to the state-of-the-art UPPAAL model checking tool.

TetaJ is evaluated through a case study based on the classic text-book example of a hard real-time control system in a mine pump. The system is hosted on an execution environment featuring an interpretation-based JVM, called Hardware near Virtual Machine (HVM), that runs on an Atmel AVR ATmega2560 processor.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Object-oriented languages; D.3.4 [**Processors**]: Run-time environments; C.3 [**Special-Purpose and Application-Based**]: Real-time and embedded systems; I.6.3 [**Simulation and Modelling**]: Applications

General Terms

Verification, Languages, Experimentation

Keywords

Real-time Java, Real-time embedded systems, WCET analysis, Model checking, Software implemented JVM

1. INTRODUCTION

Java in its traditional form, has inherent issues that makes it less suitable for development of hard real-time embedded systems such as the lack of the notion of a deadline and high-resolution real-time clocks. To accommodate these, a variety of initiatives such as the Real-Time Specification for Java (RTSJ) [23] and related profiles: Safety Critical Java (SCJ) [16] and Predictable Java (PJ) [9] have been initiated. These specify certain alterations to the underlying Java Virtual Machine (JVM) and various extensions through libraries that, when adhered to, creates a programming model that can be used for real-time systems development in Java. The SCJ and PJ Java profiles furthermore strive to provide a programming model which is more amenable to program analysis in general and to Worst Case Execution Time (WCET) analysis in particular. This is of utmost importance for hard real-time systems since WCET analysis forms an integral component in schedulability analysis which in turn can be used for determining temporal correctness.

Conducting WCET analysis is not trivial and in respect to Java, the analysis is further complicated by the presence of the JVM, since this introduces an additional layer between the application and the hardware. One approach to overcome this problem has been to implement the JVM directly in hardware, removing this additional layer [25, 2]. The advantage of this approach is that it allows WCET techniques otherwise used for execution environments without a JVM to be used for Java [11, 26].

Including a software implemented JVM in the WCET analysis is still desirable, since it will enable a broader range of common processors to be used. In this regard, the idea of portable WCET analysis based on the concept of Virtual Machine Timing Models (VMTMs) [14, 15] have been proposed which allow for expressing the cost of individual Java Bytecode instructions for a particular execution environment. [14, 15] put forward two strategies for deriving Virtual Machine Timing Models, 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. Fortunately, recent development in model-based analysis has shown how both program behaviour and processor behaviour can be expressed using models and used in conjunction with WCET analysis [26, 13, 11, 22], indicating the possibility of expressing the entire execution environ-

ment using these.

In this paper, we present a novel tool called *Tool for Execution Time Analysis of Java Bytecode* (TetaJ)[1], which is designed to conduct WCET analysis of Java tasks running on a software implementation of a JVM executed on common embedded processors. For this, a model-based approach is taken, allowing TetaJ to be adapted to a variety of different execution environments. The model-based approach is beneficial to use since it is possible to precisely describing the behaviour of programs and hardware. Furthermore, model checking has undergone a variety of optimisation in recent years, thereby making it more amenable than before for solving larger problems [7].

Capturing the behaviour of real-time tasks is cumbersome, and therefore TetaJ is capable of automatically doing this provided the source files and corresponding Java class files in an approach similar to both SARTS [11] and WCA [26]. Modelling the behaviour of the JVM can be done (semi)automatically using similar techniques as for Java tasks to generate a model from the source and executable code for the JVM. The hardware is modelled manually and could in the future be expected to be provided by the hardware vendors.

TetaJ has been evaluated using a case study based on the classic text-book example of a real-time system of a mine pump [12, 20]. The execution environment hosting the mine pump application is based on a software implemented JVM, called Hardware near Virtual Machine (HVM) [18], and the Atmel AVR ATmega2560 [5] processor is used as a representative example of a common embedded processor. The results of the evaluation is twofold: first we demonstrate that the hard real-time Java mine pump application can run on a software implemented JVM on common embedded hardware. Secondly, and most importantly, we show that TetaJ is applicable for determining WCET of the tasks of the mine pump application.

In the following, we introduce the general design of TetaJ in Section 3. Furthermore, we demonstrate how the model-based approach is realised using Networks of Timed Automata (NTA) which can be analysed by the state-of-the-art model checking tool, UPPAAL [7]. Here, we also present some of the optimisations that have been applied for mitigating model checking time. In Section 4, we present the case study. Section 5 presents that evaluation with results showing that TetaJ is indeed an applicable tool for WCET analysis of Java Bytecode-based programs.

2. RELATED WORK

The concepts of TetaJ draw inspiration from a variety of other projects that have focused on WCET analysis for Java and other programming languages.

AbsInt have developed a WCET analysis tool called aiT [1] which uses abstract interpretation for low-level WCET analysis and Implicit Path Enumeration Technique (IPET) for high-level WCET analysis. The tool is build with emphasis on flexibility to accommodate different processors and has currently support for ARM7, LEON2, LEON3, and various PowerPC processors. aiT assumes an executable as input which is subsequently disassembled. Using the disassembled executable, the control flow is reconstructed and subject to different analyses such as value analysis, cache analysis, and pipeline analysis. The result of these analyses is to obtain WCET estimates with little pessimism.

METAMOC [13] is a model-based WCET analysis tool which emphasises flexibility and currently offers support for AVR-based and ARM-based processors. This is achieved by modelling both the hardware and the program using a Network of Timed Automata (NTA) amenable to model-checking using the UPPAAL model checker. Specifically, the pipeline, cache, and main memory are represented using timed automata, whose interconnections effectively simulate an abstract execution of the instructions. To simulate execution of the program, the control flow, represented as a Control Flow Graph (CFG), is reconstructed by disassembling the executable of the program. Subsequently, the CFG is modelled as an NTA which synchronises with the hardware model to simulate execution. Another tool using a modelling approach is SARTS [11, 10] which is targeted at schedulability analysis of applications written in a variant of the SCJ Java real-time profile and requiring the Java Optimized Processor (JOP) [25].

The WCET Analyzer (WCA) [26] is specifically designed for safe WCET estimates of Java programs running on the JOP. Due to the JOP being purposely designed for being amenable to program analysis, very precise WCET estimates have been obtained using WCA. This is for example attributed the fact that the JOP features a deterministic cache model whose behaviour can precisely be described. Similar to SARTS and METAMOC, WCA takes as input an executable, which, in this case, comprises the Java class files, from which the CFG is reconstructed. The WCET can be estimated from the CFG using either a model-based approach using UPPAAL or using Implicit Path Enumeration Technique (IPET). The incentive for supporting both approaches was initially to compare them but it is also argued to be applicable for different purposes when analysing the program. Since the model-based approach yields preciser results at the expense of analysis time, it should be used for smaller, more important parts of the program. The opposite applies for IPET.

eXtensible high-integrity Real-Time Java (XRTJ) [15], is, to our best knowledge, the only project that has attempted to provide WCET analysis of Java programs accommodating software implementations of the JVM. For determining WCET, XRTJ employs a static analysis approach which relies on timing models called VMTMs. These models describe the timings of the individual Java Bytecode instructions in the particular execution environment thereby taking into account the behaviour of hardware, operating system, and JVM. It is suggested that the timing models are derived using a measurement-based approach [14].

3. TETAJ

TetaJ is capable of automatically conducting the WCET analysis of Java Bytecode programs given a JVM model and a hardware model. This entails a number of transformations

[1]TetaJ can be downloaded at `http://tetaj.dk`

which are divided into three individual tools: the *model generator tool*, the *model combiner tool*, and the *model processor tool*. The flow of using these tools is shown in Figure 1.

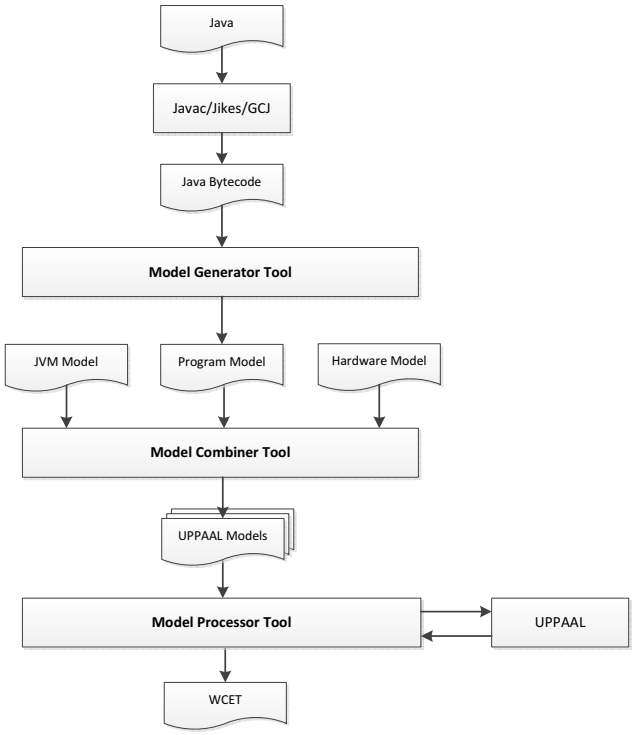

Figure 1: Overview of the individual components in TetaJ and the general usage of the tool.

Initially, the model generator tool is used to construct a model of the analysed task, denoted the *program model*. This in itself requires a number of steps, namely: constructing a CFG of the analysed task, annotating and optimising the CFG, and translating it into a model. The resulting model is combined with a JVM model and a hardware model by the model combiner tool. Finally, the model processor tool is used to estimate the WCET of the modelled task and present the result to the user.

3.1 Model Layers

The analysed task and the target execution environment are modelled using timed automata. These are modelled such that the analysed task, JVM, and hardware are independent and interchangeable. Together they form a layered architecture with predefined interfaces where the program model interacts with the underlying JVM model which further interacts with the hardware model. These are described in the following.

3.1.1 Program Model

The program model represents the analysed task in terms of its control flow. This is done using locations representing the individual Java Bytecode instructions constituting the program with interconnections corresponding to the control flow. Figure 2 illustrates a model with no branches and four sequentially executed Java Bytecode instructions.

A synchronisation on the channel *jvm_execute* is placed at each transition and is used to signal the JVM model to simulate the execution of the instruction assigned to the *jvm_instruction* variable. Note that the channel is marked as *urgent*, meaning that it is fired as soon as possible.

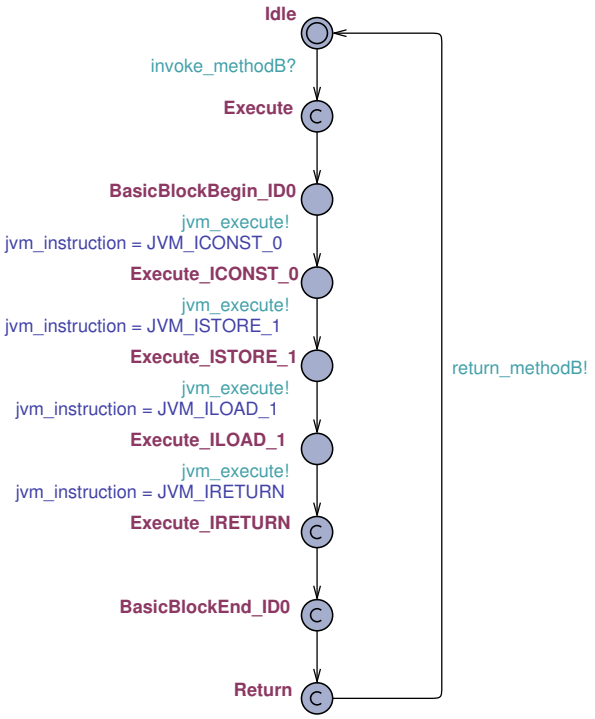

Figure 2: Example of a simple method modelled using timed automata.

Each modelled method is associated with channels used for simulating invocation and return of the particular method. In the example, these channels are called *invoke_methodB* and *return_methodB*, respectively. Synchronising the *invoke_methodB* channel allows the model to enter the *Execute* location from where it continues until reaching the *Return* location. Here, the *return_methodB* channel is fired in order to indicate the completion of the method before entering the *Idle* location, waiting for potentially being invoked again.

An initialisation model is used to initiate simulation of the method for which a WCET estimate is desired. The model essentially conducts the initial synchronisation with the model representing the method of interest, and further synchronises with this model's return channel. Finally, a synchronisation is made on the channel used by the hardware model to indicate that it has successfully simulated execution of all the instructions in the pipeline. The model is shown in Figure 3.

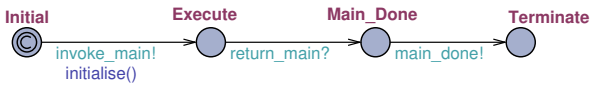

Figure 3: The initialisation model used to start the execution of the modelled method representing the analysed task.

3.1.2 JVM Model

The JVM model represents the target JVM, and is thus dependent on the specific JVM implementation. The interfaces to the program model and the hardware model are predefined such that they can be constructed independent of the particular JVM implementation. A simple JVM model is illustrated in Figure 4. Here, the JVM supports two Java Bytecode instructions: the `iload` instruction and the `istore` instruction. Initially, this model waits in the *Idle* location until the *jvm_execute* channel is fired by the program model which requests simulation of the next Java Bytecode instruction of the program. The transition corresponding to the particular instruction will then be taken. This decision is made by analysing the instruction stored in the *jvm_instruction* variable.

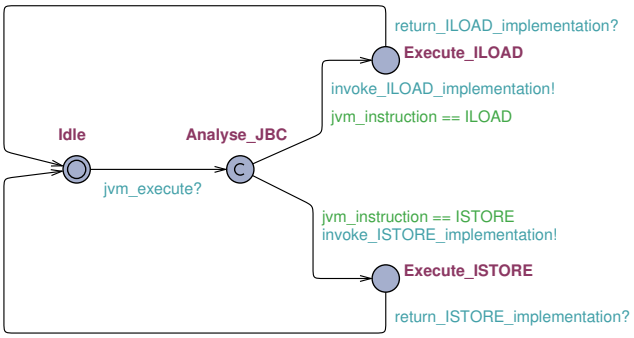

Figure 4: JVM model supporting the execution of two Java Bytecode instructions.

When a transition for a particular Java Bytecode instruction is taken, a synchronisation is used to signal another model which contains the actual implementation of it. This is done similarly to how invocations are simulated in the program model using two channels.

The current version of TetaJ allows automating the process of constructing JVM models by the introduction of a common CFG representation, denoted Tetaj CFG (TCFG). This allows to reuse CFG analyses and model generation for both Java programs and JVMs. This technique has been applied to a timing predictable version of the JVM called HVM and modelling the HVM is therefore a matter of transforming the executable of the JVM into a TCFG.

An example of a model for a Java Bytecode instruction implementation is shown in Figure 5. Each assembly instruction is modelled by conducting a synchronisation on the *assembly_execute* channel and storing the current assembly instruction in the *assembly_instruction* variable. This channel is used by the hardware model to simulate the execution of the given instruction.

3.1.3 Hardware Model

The hardware model represents the behaviour of executing the individual machine instructions defined by the JVM model. This model can vary greatly in complexity depending on the actual target hardware.

We reuse the hardware models provided by METAMOC in TetaJ. This is beneficial since METAMOC is continuously

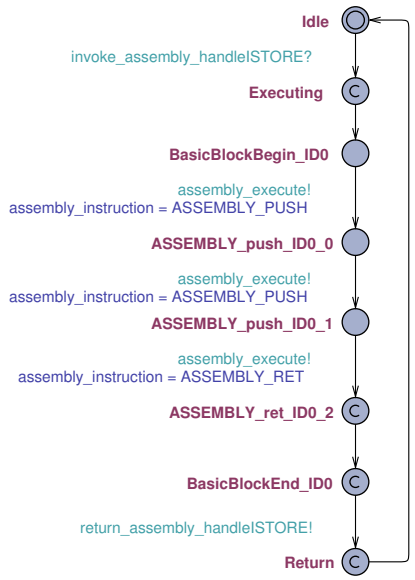

Figure 5: Example of a model representing the `istore` instruction implementation.

extended with new hardware models, thereby implicitly extending the applicability of TetaJ. An example of a two-stage pipeline from METAMOC is shown in Figure 6.

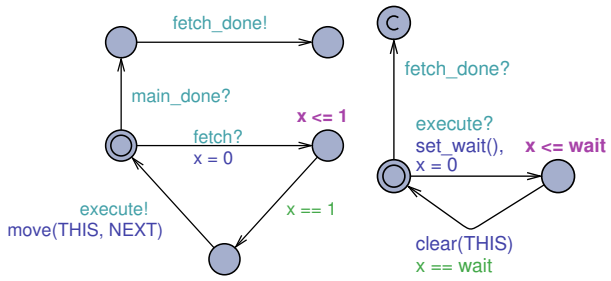

Figure 6: Model of the fetch and execute stages of a two-stage pipeline.[13]

The modelled fetch stage waits for the channel *fetch* to be fired. When this happens, it uses a clock, x, to simulate the fetch process taking one clock cycle. In order to initiate the next stage in the pipeline, the *move(THIS, NEXT)* function is invoked which moves the current from the fetch stage to the execute stage. At the same time, it also fires the *execution* channel, signalling the execute stage to proceed.[13]

The modelled execution stage uses a clock similar to the fetch stage in order to simulate the time used to execute the current instruction. A similar clock, x, is used to force waiting in the location corresponding to the WCET of the instruction. This is achieved by the invariant $x <= wait$ and the guard $x == wait$ where *wait* is set to the WCET of the instruction.[13]

3.2 Model Optimisations

Before conducting the WCET analysis, the models must be processed to avoid potential state space explosion. Therefore, a set of model optimisations are conducted with the

purpose of decreasing the analysis time and memory consumption.[27]

3.2.1 State Space Reduction mode

This optimisation is part of the UPPAAL model checker, and instructs UPPAAL to apply a number of techniques to reduce the memory consumption. Unfortunately, this optimisation is not necessarily exact, and potentially frees memory which is still necessary, resulting in increased estimation time if states must be re-explored.[27]

3.2.2 Progress Measures

Another approach which allows UPPAAL to reduce memory consumption is the use of progress measures. A progress measure describes progress in the model and, hence, must be monotonically increased as the model progresses. Incrementing the progress measure indicates to UPPAAL that it safely can remove knowledge regarding previous states. An example of a UPPAAL model using progress measures is shown in Figure 7. Here, a progress measure, denoted *pm*, is incremented as the model progresses. Whenever all execution traces have crossed a barrier, all memory related to previous states can be removed.

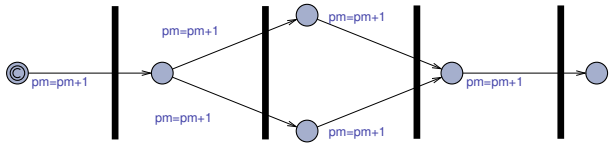

Figure 7: UPPAAL model using progress measures.

3.2.3 Model Reduction

Modelling an entire JVM produces a large number of models if the individual Java Bytecode implementations are encapsulated in individual models. The problem with this approach is that a large number of models increases the state space since each state comprise the current location of all models regardless if they are used or not. This is especially relevant since a program seldom uses all Java Bytecode instructions, and parts of the JVM related to these are thus unnecessary. The program model is therefore statically analysed to detect which Java Bytecode instructions are actually used. The models representing the implementations of the Java Bytecode instructions that are not used are afterwards removed from the JVM model.

3.2.4 Condition Optimisation

The verification time of model checking depends on the size of the models' state space. Especially branches are of concern since these exponentially increase it. Therefore, the final optimisation, partially inspired by [13], aims at safely removing branches from the CFG to simplify the model-checking process. Generally, it is difficult to safely remove branches since it typically cannot be guaranteed which branch edge results in the highest WCET. However, in special cases this can be done. Consider if-statements, which have two branches: one entering the body of the if-statement, and one skipping the body. In this case, it is evident that following the branch into the if-statement body always yields a higher WCET than skipping the body assuming no timing anomalies. Thus, the edge leading around the body can safely be removed. An example of an if-statement for which a branch can be removed is shown in Figure 8.

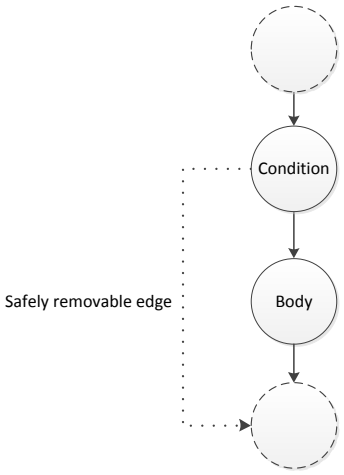

Figure 8: Example of a condition optimisation removing an edge from an if-statement.

4. CASE STUDY

To evaluate the applicability of TetaJ, we have done a case study with a commonly available processor and a JVM targeted at embedded systems.

4.1 The Mine Pump Control System

The embedded hard real-time application used in the case study is a mine pump, which is a classic text-book example [20]. The prime purpose of the mine pump is mine drainage by controlling a water pump. This further requires a number of environmental properties to be monitored such that the mine pump can be safely operated to avoid potentially endangering the lives of the mine workers. The original description of the mine pump is quite elaborate and therefore we have chosen to reduce it following the ideas initially proposed in [8].

The reduced version emphasises the safety control logic responsible for controlling when the water pump is allowed to start and when further operation of the water pump is prohibited. In Figure 9, the mine pump used in this case study is illustrated.

The control system must employ sensors that can be used for determining when the water level has reached a high level, in which case the water pump must be started, and for determining when the low level has been reached, in which case the water pump must stop. Besides monitoring the current water level, the mine pump includes additional sensory equipment for determining the methane concentration. Should this exceed a critical level, the operation of the water pump should be immediately terminated to avoid an explosion. Furthermore, if this condition arises, the water pump is prohibited from starting regardless of water level.

The mine pump control system can be implemented using two periodic tasks: one is responsible for periodically monitoring the current water level and one is responsible for periodically monitoring the current methane concentration.

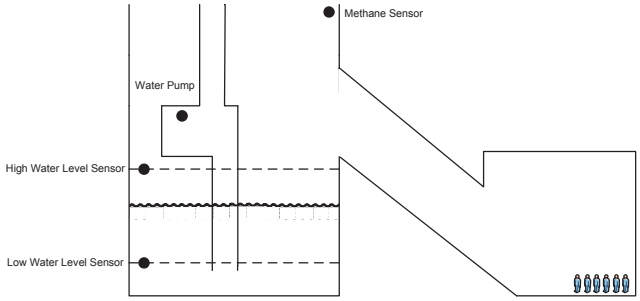

Figure 9: The mine pump featuring sensors for determining water level and methane concentration.

Each of these tasks have temporal requirements which are listed in Table 1.

Task	Period/Deadline
Methane	$56ms$
Water	$40ms$

Table 1: Timing requirements for the mine pump.

In addition to the control software, a physical model of the mine pump has been built using LEGO which is depicted in Figure 10.

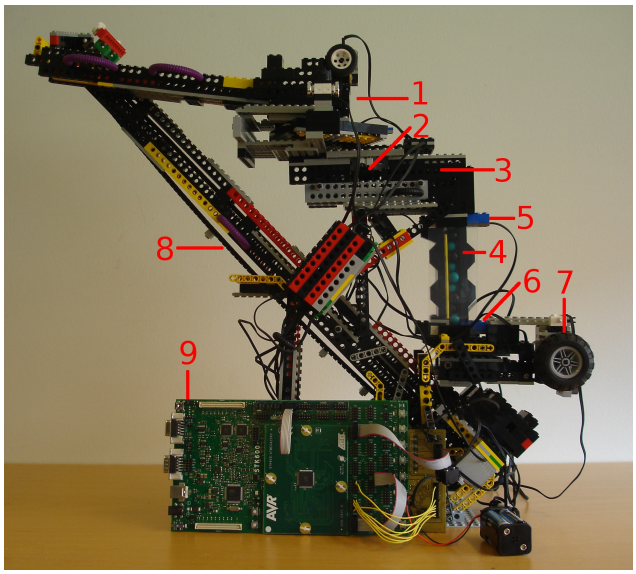

Figure 10: The LEGO construction simulating the behaviour of the mine pump.

In this model, water and methane are represented using two differently coloured bricks. The bricks will enter the conveyor belt, labelled 3, from the feeder, labelled 1. Before entering the mine shaft, labelled 4, a light sensor, labelled 2, is responsible for detecting whether the brick is methane or not. In case it is, the control system will increase the methane concentration accordingly. If this concentration exceeds a predefined limit, the water pump, labelled 7, will stop if it is currently running and be prohibited from starting until the methane concentration is lowered. Due to the feeder continuously adding water to the mine shaft, eventually, the water level will become too high. This is detected using a light sensor, labelled 5. The control system will in this case initiate the water pump, which stepwise removes bricks from the shaft and transfers them to the angled conveyor belt, labelled 8, that will move the bricks back to the feeder. To avoid completely removing all the bricks, a light sensor, labelled 6, is used for determining when the water level becomes low. When this event happens, the water pump is immediately stopped. All sensors and the water pump are connected with the I/O ports to the Atmel STK600 evaluation board, labelled 9. The evaluation board is equipped with an Atmel AVR ATmega2560 processor.

4.2 Hardware near Virtual Machine

The HVM [18], which supports systems with 256 kB flash memory and 8 kB of RAM, is a representative example of a JVM designed for operating on resource-constrained systems. It does not rely on operating system support and can be run on a variety of different processors including Atmel's AVR (specifically the ATmega2560), National Semiconductor's CR16C, and x86. The compilation technique employed is interpretation. Whenever a Java program is to be hosted by the HVM, it is compiled and effectively incorporated into the executable of the HVM itself. This is done using *icecap-tools* which is part of the HVM distribution.

A distinctive feature of the HVM is the notion of *hardware objects* [19] and first-level interrupt handling which essentially are extensions that allow manipulation of hardware registers and assigning Java handlers to hardware level interrupts.

As of the current state, all Java Bytecode instructions are supported except those dealing with float and double types. It is also worth noting that the original implementation of the HVM has not been targeted real-time systems and is therefore not analysable. The following section describes how we have modified the implementation towards being timing predictable.

4.2.1 Time Predictable HVM

The implementation has a number of undesirable constructs such as unbounded loops and recursion. In addition, many of the Java Bytecode implementations have properties that cannot be statically determined. This section presents some of the modifications that have been applied to the implementation of the HVM.

The implementation of the *instanceof* Java Bytecode originally includes an unbounded loop testing whether the object reference is an instance of a given class by consulting its parent classes iteratively. Evidently, a tight loop bound cannot be set statically that applies locally for every class when conducting this process. To circumvent the problem, the provided *icecap-tools* have been modified for analysing the class hierarchies prior to making the final JVM executable. Specifically, a matrix is constructed from which it can be determined whether a particular class is parent to another class. This only requires a look-up to be performed with time complexity $\mathcal{O}(1)$.

The HVM contains a simple interpretation loop which initially fetches and analyses the next Java Bytecode before executing it through a large switch-statement with cases corresponding to the supported Java Bytecode instructions. A problem arises when encountering Java Bytecodes used for invoking methods, such as *invokevirtual* because the interpretation loop will be called recursively thereby giving rise to similar problems as unbounded loops, and, most importantly, recursion is difficult to model using timed automata. To avoid these problems, the interpretation loop has been restructured such that stack frames are pushed to a stack whenever invoking a method. Additionally, the context is saved, such that when the method returns, the registers can be restored.

In the original implementation of the HVM, locating the individual Java Bytecode implementations in the executable is difficult due to being enclosed in individual case-statements. To simplify this task, all implementations have been extracted from the case-statements and placed in corresponding functions with appropriate names denoting the respective Java Bytecode instruction. Evidently, this adds overheads and we envision that employing an annotation strategy to identify the Java Bytecode implementations in the original structure is a better solution and may be subject to future work.

4.3 Atmel AVR ATmega2560

The processor used in this case study is Atmel's AVR ATmega2560 [5] which represents a processor with resources similar to those that can be found in many embedded systems [4]. Specifically, the processor is an 8-bit microcontroller featuring 256 kB flash for program memory, 8 kB SRAM for data memory and 4 kB EEPROM for non-volatile data storage. The clock frequency is variable and has been set to 10 MHz.

4.4 Implementation

We now present excerpts of the mine pump implementation to illustrate that TetaJ applies for non-trivial examples. Listing 1 shows the *main()* method used for initialising various registers on the ATmega2560, and the periodic tasks.

Initially, the objects representing the sensors and actuators of the system are instantiated. Note that all object allocations are done in the *main()* method to avoid time unpredictable behaviour during the time critical phase which is entered as the last part of this method. To allow the two tasks to be executed periodically, a Cyclic Executive Schedule (CES) is constructed. Since the greatest common divisor of the periods of the tasks is 8, this will be the length of the minor cycle. Using TetaJ on the tasks it is clear that the methane task must be split in three such that it can be executed in three successive minor cycles. The schedule is provided to a CES scheduler. This is implemented as a interrupt handler for the hardware interrupts generated by a timer at each minor cycle.

The call to *LegoAVRInterface.initialiseLego()* executes a native method for setting up a number of timers available on the ATmega2560 processor which are used for polling the sensors and control motor speed periodically. Finally, interrupts are enabled, and the scheduler is started.

An essential part of the mine pump is to ensure safe operation of the water pump. This is achieved by disallowing it to run when the methane concentration is critical. A history of measurements is used to keep track of the concentration of methane as shown in Listing 2.

```
public int getMethaneConcentration() {
    int methaneCount = 0;

    //@loopbound = 10
    for (int i = 0; i < this.history.length; i
        ++) {
        if (this.history[i] == METHANE)
            methaneCount++;
    }

    return methaneCount;
}
```

Listing 2: The getMethaneConcentration() method used to determine the amount of observed methane.

As shown, the *getMethaneConcentration()* method contains a loop counting the number of methane measurements in the *history* array. Notice that since TetaJ does not employ automated loop bound analysis, the loop is bounded to 10 iterations by the annotation. This bound corresponds to the length of the history array.

5. EVALUATION

The following evaluates the impact of the presented optimisation techniques, presents that TetaJ upholds the safety criterion, and evaluates the case study using TetaJ.

Conducting WCET analyses using UPPAAL on the models provided by TetaJ, the query $sup : cyclecounter$ is used. Essentially, this returns the suprema of the *cyclecounter* which is a global clock.

5.1 Optimisations

To evaluate the impact of the optimisation techniques, six different experiments have been conducted: one without optimisations, four with one individual optimisation enabled, and, finally, one with all optimisations enabled. Each of the experiments are conducted using a simple Java program containing a loop and were run on an application server with dual Intel Xeon E5420 quad core @ 2.50 GHz and 32 GB RAM. The results are listed in Table 2.

	Analysis time	States explored	Memory usage
None	14h 51m 17s	41854143	3,905 MB
PM	108h 7m 8s	408223029	589 MB
STR	13h 33m 21s	41854143	2,426 MB
CO	1m 16s	53732	294 MB
MR	4h 46m 41s	41854143	3,851 MB
All	19s	57553	144 MB

Table 2: Results using the optimisations: progress measures (PM), state space reduction (STR), condition optimisation (CO), and model reduction (MR).

```
public void main(String[] args) {
    WaterpumpActuator waterpumpActuator = new WaterpumpActuator(ACTUATOR_ID_WATERPUMP);
    MethaneSensor methaneSensor = new MethaneSensor(SENSOR_ID_METHANE, CRITICAL_METHANE_LEVEL,
        BRICK_HISTORY_SIZE);
    HighWaterSensor highwaterSensor = new HighWaterSensor(SENSOR_ID_HIGH_WATER,
        CONSECUTIVE_HIGH_WATER_READINGS);
    LowWaterSensor lowwaterSensor = new LowWaterSensor(SENSOR_ID_LOW_WATER,
        CONSECUTIVE_NO_WATER_READINGS);

    PeriodicMethaneDetectionShared shared = new PeriodicMethaneDetectionShared();
    PeriodicMethaneDetectionStep1 m1 = new PeriodicMethaneDetectionStep1(methaneSensor,
        waterpumpActuator);
    PeriodicMethaneDetectionStep2 m2 = new PeriodicMethaneDetectionStep2(methaneSensor, shared);
    PeriodicMethaneDetectionStep3 m3 = new PeriodicMethaneDetectionStep3(waterpumpActuator, shared
        );
    PeriodicWaterLevelDetection w = new PeriodicWaterLevelDetection(highwaterSensor,
        lowwaterSensor, waterpumpActuator);

    int[] schedule = {W,M1,M2,M3,E,W,E,M1,M2,M3,W,E,E,E,M1,M2,M3,W,E,E,W,M1,M2,M3,E,W,E,E,M1,M2,M3
        ,W,E,E,E};
    SchedulableObject[] tasks = {null,m1,m2,m3,w};

    CyclicExecutive ces = new CyclicExecutive(schedule, tasks, 34);

    InterruptHandler.registerHandler(ces, ATMega2560InterruptHandler.TIMER2_OVF);
    TimerAau.initTimer2();
    ATMega2560InterruptHandler.init();

    LegoAVRInterface.initialiseLego();
    LegoAVRInterface.enableInterrupts();

    ces.startSchedule();
}
```

Listing 1: The implementation of the mine pump's *main()* method.

All the experiments yield a WCET of 17262 clock cycles, however they differ significantly in analysis time and memory consumption. Surprisingly, using the progress measure optimisation yields significantly higher analysis times than the reference. This indicates that for high progress measure values, performance is significantly degraded. With respect to the other optimisations, it can be observed that the results from *all* and *none* are significantly different. Specifically, the analysis time range from 15 hours to 19 seconds, and the memory usage from 3.9 GB to 144 MB. Furthermore, it can be observed that the condition optimisation influences the analysis time significantly by safely removing branches. Finally, the analysis time of model reduction is remarkable. It indicates that the overhead of instantiating unnecessary models in the model is considerable and should be avoided.

5.2 Safety and Precision of WCET Estimates

Evaluating the safety and precision of the WCET estimates produced by TetaJ requires a great effort if done manually. This would require the entire executable to be examined and manually sum the execution times of the individual instructions. This approach is deemed too cumbersome, hence to give indications of whether TetaJ produces safe and precise WCET estimates or not, these are compared with the WCETs obtained by a measurement-based approach. For this, we use the AVR Simulator part of the AVR Studio [6] IDE which is capable of simulating execution of programs on the ATmega2560.

The comparison of measurement-based WCETs and WCETs estimated by TetaJ is based on four algorithms implemented in Java. The results of the experiments are shown in Table 3.

Algorithm	Meas. WCET	TetaJ WCET	Pessimism
Iterative Fibonacci	46,642	46,933	0.6%
Factorial	39,726	40,939	3.1%
Reverse Ordering	64,436	81,919	27.1%
Bubble Sort	907,103	2,270,401	150.3%
Binary Search	54,430	99,301	82.4%
Insertion Sort	849,353	3,740,769	440.4%

Table 3: Comparison of measurement-based WCETs and WCETs obtained by TetaJ in clock cycles.

The measurement-based approach is expected to produce under-estimated WCETs, hence these should always be lower than the results obtained using TetaJ. Referring to Table 3, this is indeed the case for all the experiments which indicates that TetaJ produces safe WCETs. As observed, in some cases, the WCET estimates obtained using TetaJ are very precise. However, for some other cases, particularly Bubble Sort and Insertion Sort, the estimates are overly pessimistic. This is primarily attributed the approach to annotating the loop bounds. For these two algorithms, the iteration count of the inner loop depends on the particular iteration of the outer loop. Such interdependencies are as of this writing not possible to describe and hence the loop bound of the inner

loop is set to the iteration count that applies for the first iteration of the outer loop. For example, if the outer loop is set to ten, the inner loop is set to nine. Furthermore, we hypothesise, that even better results may be possible if various DFAs are applied such as determining and subsequently removing infeasible paths.

5.3 WCET Analysis of the Case Study

Having shown that TetaJ is applicable for WCET analysis of Java programs, we use it to analyse the WCETs of the mine pump application. The results of the WCET analysis using all available optimisations are shown in Table 4.

Task	Analysis time	Memory usage	WCET
Methane 1	1m 19s	140 MB	41,644
Methane 2	1m 16s	105 MB	68,436
Methane 3	29s	75 MB	12,552
Water	6m 40s	271 MB	70,712

Table 4: Results of using TetaJ on the mine pump application.

Recognising that the ATmega2560 is configured with a clock frequency of 10 MHz, all estimated WCETs can be run within a minor cycle of 8 milliseconds, or 80,000 clock cycles. Therefore, running the tasks in the provided schedule both the methane and water tasks will meet their deadlines.

6. CONCLUSIONS

In this paper we have presented TetaJ, a novel tool for statically determining the Worst Case Execution Time (WCET) of Java Bytecode-based programs. TetaJ is designed to conduct WCET analysis of Java tasks running on a software implementation of a JVM executed on common embedded processors.

TetaJ was developed with the purpose of being adoptable in an embedded hard real-time development process because WCET analysis can be conducted on method level thereby allowing for a fine-grained decomposition of the system. The advantage of this is that the WCET analysis can be used for more than being an integral part of the schedulability analysis, since WCET analysis on method level allows for a variety of other applications, such as profiling.

We have shown that TetaJ can be used iteratively in the process since the model checking time is considered sufficiently low as a consequence of applying a series of optimisations.

To examine the concrete effects of applying the optimisations, we conducted a series of experiments which showed remarkable effects in analysis time. For a simple Java program, the analysis time was reduced from 15 hours to 19 seconds without loss of precision in the WCET. Therefore, we conclude that the optimisations improve the feasibility of model checking for estimating WCET.

To provide indications whether TetaJ upholds the safety property, WCETs obtained by it were compared with those from a measurement-based approach. The comparison indicated that TetaJ upholds the safety property and in many cases will be capable of producing very precise estimates. The cases where overly pessimistic results are obtained, is attributed a lack of expressivity in the loop bound annotations which is particularly the cases where the iteration count of inner loops are dependent on the particular iteration count of the outer loop. Furthermore, we hypothesise that even greater precision is possible if various DFAs are employed for e.g. removing infeasible paths that may contribute to higher WCETs.

Having shown that TetaJ is applicable for WCET analysis opens for a variety of new directions which we envision can be subject to future work. Specifically, the flexibility of TetaJ may be further exercised by conducting more case studies in which other well-known JVMs such as JamVM [21], FijiVM [24], or Ovm [3] are used. On a related note, the experiences drawn from modifying the HVM towards time predictability has made us wonder whether a new JVM build with this design goal would be beneficial. In addition, it could be interesting to examine whether concrete guidelines for JVM design can be deduced such that the JVM is more amenable to WCET analysis.

Even though the mine pump is a representative example of a hard real-time system, it may be interesting in future case studies to examine the effects of analysing more elaborate programs with higher complexity such as the Collision Detector (CDj) [17] real-time benchmark suite for Java.

7. ACKNOWLEDGEMENTS

We would like to thank Alexandre David, associate professor at Aalborg University, and Benedikt Huber, research and teaching assistant at Vienna University of Technology, for feedback in UPPAAL technicalities. Furthermore, we would like to thank Mads Christian Olesen, Ph.D. student at Aalborg University, for helping with issues regarding METAMOC. Finally, we thank Stephan Korsholm, lecturer at VIA University, for his commitment and enthusiasm in aiding us in modifying the HVM.

8. REFERENCES

[1] AbsInt. The ait wcet analyser, 2009. `http://www.absint.com/ait/analysis.htm` (9. June 2011).

[2] aJile Systems. aj-100^{TM}real-time low power java^{TM}processor, 2000. `www.ajile.com`.

[3] A. Armbruster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek. A real-time java virtual machine with applications in avionics. *ACM Trans. Embed. Comput. Syst.*, 7:5:1–5:49, December 2007.

[4] Atmel. Atmel's avr microcontroller ships 500 million units, 2004. `http://www.atmel.com/dyn/corporate/view_detail.asp?FileName=Ships500M.html` (26. May 2011).

[5] Atmel. Atmega640/1280/1281/2560/2561 datasheet, 2010. `www.atmel.com/dyn/resources/prod_documents/doc2549.pdf` (3. June 2011).

[6] Atmel-Corporation. Atmel products - atmel avr 8- and 32-bit - megaavr - avr studio 4, 2011. `http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725` (30. May 2011).

[7] G. Behrmann, J. Bengtsson, A. David, K. Larsen,

P. Pettersson, and W. Yi. Uppaal implementation secrets. In W. Damm and E. Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 3–22. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45739-9_1.

[8] T. Bøgholm, C. Frost, R. R. Hansen, C. S. Jensen, K. S. Luckow, A. P. Ravn, H. Søndergaard, and B. Thomsen. Harnessing theories for tool support. *Submitted for publication: Innovations in Systems and Software Engineering*, 2011.

[9] T. Bøgholm, R. R. Hansen, A. P. Ravn, B. Thomsen, and H. Søndergaard. A predictable java profile: Rationale and implementations. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 150–159, New York, NY, USA, 2009. ACM.

[10] T. Bøgholm, H. Kragh-Hansen, and P. Olsen. Model-based schedulability analysis of real-time systems, 2008. `http://sarts.boegholm.dk/cd/Report/thesis.pdf` (10. June 2011).

[11] T. Bøgholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, and K. 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*, pages 106–114. ACM, 2008.

[12] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Educational Publishers Inc., Boston, MA, USA, 4th edition, 2009.

[13] 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.

[14] 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: OTM 2003Workshops*, pages 411–424, 2003.

[15] E. Hu, A. Wellings, and G. Bernat. XRTJ: An extensible distributed high-integrity real-time Java environment. *Real-Time and Embedded Computing Systems and Applications*, pages 208–228, 2004.

[16] JSR302. The java community process^{TM}program - jsrs: Java specification requests - detail jsr# 302, 2010. `http://www.jcp.org/en/jsr/detail?id=302`.

[17] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. Cdx: a family of real-time java benchmarks. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 41–50, New York, NY, USA, 2009. ACM.

[18] S. Korsholm. Hvm lean java for small devices, 2011. `www.icelab.dk`.

[19] S. Korsholm, A. P. Ravn, C. Thalinger, and M. Schoeberl. Hardware objects for java. In *In Proceedings of the 11th IEEE International Symposium on Object/component/serviceoriented Real-time distributed Computing (ISORC 2008*. IEEE Computer Society, 2008.

[20] Z. Liu and M. Joseph. Real-Time and Fault-Tolerant Systems–Specification, Verification, Refinement and Scheduling. Technical report, Technical Report 323, UNU-IIST, PO Box 3058, Macau, 2005.

[21] R. Lougher. Jamvm – a compact java virtual machine, 2010. `http://jamvm.sourceforge.net/`.

[22] A. Metzner. Why model checking can improve wcet analysis. In R. Alur and D. A. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 298–301. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-27813-9_26.

[23] Oracle. RTSJ 1.1 Alpha 6, release notes, 2009. `http://www.jcp.org/en/jsr/detail?id=282`.

[24] 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*, pages 110–119. ACM, 2009.

[25] M. Schoeberl. JOP: A Java optimized processor. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003)*, volume 2889 of *LNCS*, pages 346–359, Catania, Italy, November 2003. Springer.

[26] M. Schoeberl, W. Puffitsch, R. Pedersen, and B. Huber. Worst-case Execution Time Analysis for a Java Processor. *Software: Practice and Experience*, 40(6):507–542, 2010.

[27] UPPAAL. Up4aal, 2010. `http://www.uppaal.com/` (9. June 2011).