# Towards a Real-Time, WCET Analysable JVM Running in 256 kB of Flash Memory

Stephan Korsholm
VIA University College
8700 Horsens, Denmark
sek@viauc.dk

Kasper Søe Luckow
Aalborg University
9220 Aalborg, Denmark
luckow@cs.aau.dk

Bent Thomsen
Aalborg University
9220 Aalborg, Denmark
bt@cs.aau.dk

## 1 Introduction

The Java programming language has recently received much attention in the real-time systems community as evidenced by the wide variety of initiatives, including the Real-Time Specification for Java[8], and related real-time profiles such as Safety-Critical Java[6], and Predictable Java[3]. All of these focus on establishing a programming model appropriate for real-time systems development. The motivation for the profiles has been to further tightening the semantics, for accommodating static analyses, such as Worst Case Execution Time (WCET) analysis that serves an integral role in proving temporal correctness.

Evidently, the presence of the Java Virtual Machine (JVM) adds to the complexity of performing WCET analysis. To reduce the complexity, a direction of research has focused on implementing the JVM directly in hardware[9]. To further extend the applicability of real-time Java, we want to allow software implementations of the JVM executed on common embedded hardware, such as ARM and AVR, while still allowing the system to be amenable to static analyses. This necessarily demands that the JVM is amenable to static analyses as well, which is difficult, since the JVM specification is rather loosely defined. Specifically, the JVM specification emphasises on *what* a JVM implementation must do whenever executing a Java Bytecode, but leaves *how* unspecified. This makes JVM vendors capable of tailoring their implementation to their application domain.

In our recent research, we have developed a WCET analysis tool called Tool for Execution Time Analysis of Java bytecode (TetaJ)[5], which allows for taking into account a software implemented JVM and the hardware. The development of TetaJ has made us explicitly reason about JVM design accommodating WCET analysis. The contribution of this paper is to present our preliminary research efforts in making Java tractable for real-time embedded systems on more common execution environments by elaborating on *how* a real-time JVM must handle certain issues. Constraining the degree of freedom of the real-time JVM vendors is a necessity to ensure, that the application running on the JVM is temporally correct, since the WCET is often obtained using static analyses relying on predictable behaviour.

## 2 TetaJ

TetaJ employs a static analysis approach where the program analysis problem of determining the WCET of a program is viewed as a model checking problem. This is done by reconstructing the control flow of the program and the JVM implementation, and generate a Network of Timed Automata (NTA) amenable to model checking using the state-of-the-art UPPAAL model checker[1]. The NTA is structured such that the model checking process effectively simulates an abstract execution of the Java Bytecode program on the particular JVM and hardware.

TetaJ has proven suitable for iterative development since it analyses on method level, and because analysis time and memory consumption are reasonably low. In a case study[5, 2], an application consisting of 429 lines of Java code was analysed in approximately 10 minutes with a maximum memory consumption of 271 MB. The case study is based on the Atmel AVR ATmega2560 processor and the

Hardware near Virtual Machine (HVM)[1] which is a representative example of a JVM targeted at embedded systems.

Currently, TetaJ provides facilities for automatically generating an NTA representing the JVM by the provision of the HVM executable. The METAMOC[4] hardware models are reused in TetaJ thereby having the desirable effect that TetaJ can benefit from the continuous development of METAMOC.

We have strong indications that TetaJ produces safe WCET estimates, that is, estimates that are at least as high as the actual WCET and TetaJ may therefore be appropriate for analysing hard real-time Java programs. As to the precision, we have results showing that TetaJ produces WCET estimates with as low as 0.6% of pessimism[5].

## 3 Hardware near Virtual Machine

The HVM is a simplistic and portable JVM implementation targeted at embedded systems with as low as 256 kB of flash memory and 8 kB of RAM and is capable of running bare-bone without operating system support. To support embedded systems development, the HVM implements the concept of *hardware objects*[7], that essentially prescribe an object-oriented abstraction of low-level hardware devices, and allow for first-level interrupt handling in Java space.

The HVM employs iterative interpretation for translating the Java Bytecodes to native machine instructions. The interpreter itself is compact, and continuously fetches the next Java Bytecode, analyses it, and finally executes it. The analyse and execute stages are implemented by means of a large switch-statement with cases corresponding to the supported Java Bytecodes.

A special characteristic of the HVM is that the executable is adapted to the particular Java Bytecode program. Specifically, the Java Bytecode of the compiled program is encapsulated in arrays within the HVM itself. This, however, does not affect the behaviour of the interpreter, and is merely a way of bundling the Java Bytecode with the HVM into a single executable.

## 4 Preliminary Design Criteria for a Predictable HVM

During the development of TetaJ, the implementation of the HVM has been inspected and modified according to the needs of WCET analysis. Some modifications simply comprise bounding the number of loop iterations while others require more elaborate solutions to be developed. In the following, we present our experiences with modifying the HVM towards predictable and WCET analysable behaviour.

### 4.1 Eliminating Recursive Solutions

Some Java Bytecode implementations are intuitively based on recursive solutions. Specifically, the Java Bytecodes responsible for method invocations such as *invokevirtual* employ a recursive approach.

The heart of the HVM is the *methodInterpreter* which implements the interpretation facilities. Whenever e.g. *invokevirtual* is executed, the *methodInterpreter* is recursively called to process the code of the invoked method. This, however, is undesirable seen from a static WCET analysis perspective, since it is difficult to statically determine the depth of the recursive call. The problem is circumvented by introducing an iterative approach and introduce the notion of a call stack and stack frames. Using this solution, a stack frame containing the current call context, that is, stack pointer, program counter etc. are pushed onto the stack, and the *methodInterpreter* simply continues iteratively fetching Java Bytecodes from the called method. When the particular method returns, the stack is popped and the context restored to the point prior to the method invocation.

[1]http://www.icelab.dk

### 4.2 Reducing Pessimism of the Class Hierarchy

Since Java is strongly typed, type casts produce the *checkcast* Java Bytecode which is responsible for iteratively checking the class hierarchy to determine whether the type cast is type compatible. Another example is the *instanceof* operator which similarly consults the class hierarchy iteratively. Establishing a tight bound that applies for every part of the class hierarchy cannot be done statically. Instead it is only possible to establish a global bound corresponding to the maximum depth of the class hierarchy. This gives rise to pessimism that affects the resulting WCET extensively.

This problem has been resolved by harnessing that the HVM is adapted to the particular application. Because this process is performed prior to runtime, it is possible to exercise how the class hierarchy is built and construct a matrix specifying how classes are interrelated. The matrix will be incorporated in the final executable, and can be used for determining type compatibility among classes in constant time, by simply looking up the matrix.

### 4.3 Constant Time Analyse Stage

Different compilers and different optimisation levels may or may not implement a sufficiently large switch-statement as a look-up table. Because of this uncertainty, we have replaced the analyse stage in the *methodInterpreter* to ensure that this stage is performed in constant time regardless of compiler and optimisation levels. The replacement consists of extracting the individual Java Bytecode implementations from the switch-statement into respective functions. This also has the desirable side-effect that they are easily located in the disassembled HVM executable. An array of function-pointers to each of these functions substitutes the original switch-statement, thereby allowing for constant access time to each of the Java Bytecode implementations using the opcodes as look-up keys.

## References

[1] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - A Tool Suite for Automatic Verification of Real-time Systems. *Hybrid Systems III*, pages 232–243, 1996.

[2] Thomas Bøgholm, Christian Frost, Rene Rydhof Hansen, Casper Svenning Jensen, Kasper Søe Luckow, Anders P. Ravn, Hans Søndergaard, and Bent Thomsen. Harnessing theories for tool support. *Submitted for publication: Innovations in Systems and Software Engineering*, 2011.

[3] Thomas Bøgholm, René R. Hansen, Anders P. Ravn, Bent Thomsen, and Hans 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.

[4] Andreas E. Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 113–123. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[5] C. Frost, C. S. Jensen, K. S. Luckow, and B. Thomsen. WCET Analysis of Java Bytecode Featuring Common Execution Environments. *Accepted for publication: The 9th International Workshop on Java Technologies for Real-time and Embedded Systems - JTRES 2011*, 2011.

[6] JSR302. The java community process, 2010. `http://www.jcp.org/en/jsr/detail?id=302`.

[7] Stephan Korsholm, Anders P. Ravn, Christian Thalinger, and Martin 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.

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

[9] Martin 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.