A Survey of Worst-Case Execution Time Analysis for Real-Time Java

Trevor Harmon and Raymond Klefstad

University of California, Irvine
Dept. of Electrical Engineering and Computer Science
Irvine, California 92697-2625 USA
{tharmon, klefstad}@uci.edu

Abstract

As real-time systems become more prevalent, there is a need to guarantee that these increasingly complex systems perform as designed. One technique involves a static analysis to place an upper bound on worst-case execution time (WCET). Other techniques aim for new architectures and algorithms that reduce the WCET. At the same time, there is a growing interest in using Java for real-time systems. Several WCET analysis prototypes for Java have been created, and more are under development.

This paper provides a comprehensive survey of research that combines WCET analysis with the Java domain. We begin by explaining the importance of WCET analysis and why it is so difficult to perform adequately. We then examine the features that make Java an attractive platform for WCET analysis, as well as the new challenges it brings. Finally, we provide a survey of prior work on this subject, organized as a simple one-level taxonomy.

1. Introduction

Real-time systems are becoming part of our everyday life. No longer restricted to aerospace and military applications, real-time technology is responsible for keeping our automobiles safe, our surgeons precise, and our food sanitary. Even potato chip factories now rely on real-time systems. As uncooked chips zoom by on a conveyor belt, a digital camera constantly scans them. If it detects any dark spots, it triggers an air hose to blow the chip off the belt and into a compost heap.[1]

1-4244-0910-1/07/$20.00 ©2007 IEEE.

[1]As described in "How It's Made," episode 14: `http://howitismade.net/`

Now that real-time computing is so prevalent, it is perhaps no surprise that the topic is considered mature. University courses and entire books are devoted to the topic. In practice, however, building a real-time system is very much an art, not a science. For instance, a common practice in industry[2] is to design real-time systems with extremely low processor utilization (around 1%), just in case unexpected behavior exceeds CPU resources, resulting in missed deadlines and critical failure.

This over-provisioning of the processor is wasteful. It demands CPU resources two orders of magnitude greater than what is actually required. Worse, it unmasks a more fundamental problem: Despite decades of research, practitioners still cannot trust modern tools and techniques to produce a real-time system that performs as expected. There is often too much uncertainty in the system to provide sufficient confidence in the predictability of real-time tasks.

This is not an immutable situation. Knowing the *worst-case execution time*, or WCET, of tasks can provide great confidence and predictability in real-time systems. Unfortunately, WCET analysis is often neglected or merely "guesstimated" in practice. Research has therefore focused on statically analyzing source code and dynamically measuring system performance to obtain a tight WCET bound. By improving the quality and usability of these techniques, industry use of WCET analysis will likely increase, making real-time systems both safer and more resource-efficient.

2. The Challenge of WCET Analysis

The essential question, then, is why more influential research has not been produced despite two decades of

[2]Based on anecdotal evidence witnessed by the authors.

work in WCET.[3] One explanation comes from Kirner and Puschner [18], who argue that industrial-strength WCET tools are simply too difficult to implement.

2.1. Implementation Difficulties

One of the factors behind this implementation difficulty is the modern CPU. Architectural advancements in RISC processor design, such as very long pipelines and complex multi-level caches, have focused on making the average case as fast as possible. Unfortunately, the shrinking of this average has not come without cost: While the average may be small, its standard deviation has grown large, resulting in large (and overly pessimistic) worst-case execution times.

For example, branch prediction normally results in a fast and efficient pipeline, but when a prediction misses, the pipeline stalls. Handling this kind of unforeseen delay demands one of two things: a very complicated WCET analysis tool (with a complete model of the processor pipeline) or a very conservative WCET bound. Both are unattractive to industry, leading to a neglect of the proper WCET analysis that is necessary for safe and reliable real-time systems.

Hardware is not the only difficulty, of course. The software platform is another reason for the lack of good tools for WCET analysis. Today, most real-time systems are developed in C and, to a lesser extent, Ada. Because the compilers for these languages vary, there is no common intermediate representation (IR) for WCET analysis tools to target. GCC's Register Transfer Language, for example, is incompatible with the Intel compiler's IR. Furthermore, these IRs may change across compiler versions, they may suffer from limited documentation (if any), and they are subject to change with each new compiler version.

The changing nature of IR magnifies the difficulty of building WCET analysis tools. Because there is no clean, consistent separation between high-level source code and low-level machine code, tools must be able to perform a complete top-to-bottom analysis. This includes parsing source code, constructing a control flow graph, mapping the basic blocks to machine code, analyzing each basic block according to a model of the target processor, and so on.

To make this process tractable, most tools offer little flexibility, usually locking themselves to a particular hardware architecture. It is also common to ignore high-level source code altogether, operating at the machine code level. Mapping this machine code analysis back to the high-level source code, which is necessary to act on the information provided by the tool, is typically cumbersome and unintuitive. The aiT tool,[4] for instance, displays its analysis in assembly language, leaving the programmer to mentally map the jumble of mnemonics and hexademical numbers back to the source code language of choice.

All of these complications, at both the hardware and software levels, combine to make WCET analysis tools complex, non-portable, and difficult to implement and to use. It is no wonder, then, that the industry has been slow to adopt the idea of static WCET analysis. This situation has led developers, as well as researchers in the WCET field, to seek a better platform on which to build real-time systems and tools.

2.2. Java as a Catalyst

In recent years, the platform that an increasing number of real-time developers and researchers turn to is Java. At first glance, Java appears to be a terrible match for real-time systems. Its combination of automatic garbage collection, underspecified threading semantics, and pervasive object-orientation (i.e. dynamic dispatch) are all impediments in building time-predictable software.

In the year 2000, however, three near-simultaneous events seemed to open the floodgates for real-time Java: In May, the Real-time Specification for Java (RTSJ) was released [5]. In June, the first paper on applying Java to the problem of WCET analysis was published [4]. And in November, the first processor designed specifically for real-time Java, the aJ-100,[5] became commercially available.

In the years since these innovations, Java has become a viable platform for real-time systems. For example, commercial implementations of the RTSJ are available from aicas[6] [*sic*] and Sun,[7] and real-time garbage collectors, such as Metronome [1], are gaining steam. Boeing is using real-time Java to power drone aircraft, and the United States Navy will use it in next-generation battleships [20].

For a language that offers no temporal guarantees and is designed to be dynamic, these events are surprising. The phrase "real-time Java" may even sound like an oxymoron. Yet, there are a number of reasons why Java is gaining traction in real-time environments:

Bytecode Java's bytecode provides an inherent modularization of static analysis tasks, as illustrated in Figure 1. For example, high-level WCET tools for

[3]Kligerman's and Stoyenko's 1986 paper [19] is generally considered the first publication to address the problem of WCET.

[4]`http://www.absint.com/ait/`

[5]`http://www.jempower.com/ajile/content/view/20/27/`

[6]`http://www.aicas.com/jamaica.html`

[7]`http://java.sun.com/javase/technologies/realtime.jsp`

Java can ignore any timing aspects below the bytecode level. Separate low-level tools, perhaps written by other vendors, can then complete the analysis once the target architecture is known. This sort of modularization helps solve the software complexity problem raised in Section 2.1. Bytecode also acts as a common, well-specified intermediate representation that does not vary with different compiler versions and vendors (unlike the situation with C and Ada).

Java processors Java processors, whose native instruction set is Java bytecode, help solve the hardware complexity problem raised in Section 2.1. These chips eliminate the operating system and virtual machine, making WCET analysis far simpler. In addition, research has produced a Java processor, called JOP, with a short four-stage pipeline and a predictable instruction cache designed for real-time systems [24]. Such chips are easy targets for tight WCET analysis.

Productivity Real-time systems are becoming increasingly complex and widespread. To keep up with the growing demand for these sophisticated systems, developers need to become more productive, and Java is recognized as a more productive language than C. A recent experiment by Nortel Networks found that programmer productivity doubled after switching to real-time Java [20].

Training Technical aspects are not the only consideration when choosing Java for real-time systems. The state of the workforce must be taken into account, as well. Most graduating students of today have been taught Java, not C, and have never touched Ada. Unless Java can be incorporated into the development process for real-time systems, industry will not be able to leverage the existing skills of the new generation of software engineers and will have to retrain them in other languages.

The common theme here is that Java offers a higher level of abstraction. Not only does this make real-time system development more manageable, it also streamlines static timing analysis. Of course, with these advantages come a new set of challenges. Dynamic dispatch, for instance, is still very much an open problem in terms of WCET analysis [17]. Also, existing tools for WCET are grounded in C and Ada, so new tools for Java will have to be written. Despite these obstacles, we believe that Java is an important catalyst in providing safe, accurate, and tight bounds on WCET for real-time programs.

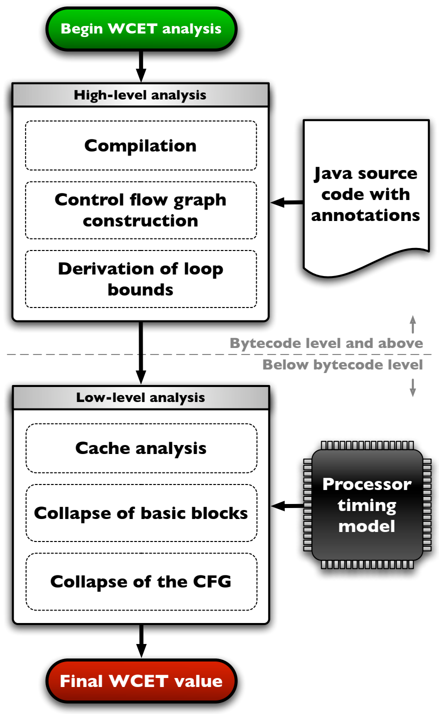

Figure 1. This sketch of the WCET analysis process shows the clean separation between high- and low-level analysis that Java bytecode provides.

2.3. A Survey of WCET Analysis for Java

We are of course not the first to propose bringing WCET analysis techniques to the Java domain. However, while the theory of worst-case execution time has been addressed by hundreds of research papers over the last two decades,[8] fewer than twenty publications spanning only six years have tackled the problem of Java in WCET analysis.

We wish to highlight these efforts and provide a historical record of what has been accomplished in WCET analysis for Java. The following sections therefore provide a survey of the most prominent research in this area. We have categorized the approaches into four basic categories: 1) bytecode as an intermediate representation, 2) high-level WCET analysis, 3) low-level WCET analysis, and 4) miscellaneous work, a catch-all category for research that does not fit cleanly into the first three categories. Where appropriate, we also include a discussion of the strengths and weaknesses of a technique or tool.

[8]Kligerman's and Stoyenko's 1986 paper [19] is generally considered the first publication to address the problem of WCET.

3. Bytecode as an Intermediate Representation

The earliest known work to combine the Java domain with WCET analysis was Bernat's proposal [4] to use Java bytecode in WCET tools. Noting that WCET analysis was not being adopted by industry practitioners, Bernat suggested that a lack of portability in WCET tools was the cause. Existing software for WCET analysis was normally restricted to a single source language, a specific compiler, and a unique configuration of processor, memory, and clock speed. As a result, the usual industry practice was to forgo these tools and rely on *ad hoc* measurement, an inefficient and error-prone technique that often leads to overly optimistic WCET bounds.

To address this problem, Bernat proposed that Java bytecode could serve as an intermediate representation for WCET tools, analogous to register transfer languages acting as intermediate representations in compilers. The assumption was that if WCET tools were to standardize on bytecode, they would be more portable, versatile, and thus more attractive to the industry.

The emphasis, then, was not on Java as a real-time programming language but rather as a catalyst. For example, real-time programs written in C or Ada could be translated to Java bytecode, then translated from bytecode to machine code. This multi-stage process would, in theory, allow a single WCET tool designed only for bytecode to analyze both C and Ada programs without knowledge of either language.

Compared to most instruction sets, Java bytecode contains enough high-level information to perform a full WCET analysis, but this benefit does not come for free. The move to bytecode brings new challenges, such as how to pass WCET annotations from an arbitrary source language to bytecode and how to integrate knowledge of the target hardware into the bytecode analysis.

Bernat offered a somewhat awkward solution to the first problem: Programmers would be required to invoke methods in a predefined class whenever a WCET annotation was required. For instance, the following Ada statement would indicate a maximum loop bound of 10:

```
WCETAn.Loopcount(10);
```

3.1. Discussion

Compared to established techniques, this style of annotation mingles non-functional metadata (that is, the WCET information) with the normal source code statements, making the program more difficult to read. In addition, tools for compiling arbitrary C or Ada source to Java bytecode remain primitive. Most such tools have not progressed beyond the prototype stage or have serious limitations that prevent their use in day-to-day operations. Jazillian, for example, a C-to-Java translator, makes no guarantee that the resulting Java code can even be compiled.

For these reasons, the notion of bytecode as an intermediate representation for WCET tools has failed to reach the industry and has not progressed beyond Bernat's initial research.

4. High-level Analysis for the Java Language

In contrast to low-level analysis, high-level[9] WCET analysis for Java is relatively simpler. The language is similar to existing ALGOL-like imperative languages, so it builds upon a vast body of existing work in compilers and high-level WCET theory. Research in this area is therefore more mature because it has received attention from a greater number of researchers.

The earliest work in high-level WCET analysis for Java comes from Puschner [22]. He noted that significant effort had been expended on making the *functionality* of code portable, but there were no mechanisms for porting or distributing information about the *execution time* of code. To solve this problem, he centered on the idea of "abstract" timing information. The goal was to collect and store as much information as possible about timing, such as the control flow and loop bounds, without knowing the concrete details of the processor, the cache, and so on. This abstract information can then be ported to any processor, saving the work of running a complete analysis for each target architecture.

As a convenient side-effect, this solution also addresses the problem of WCET in third-party libraries. Such libraries make developers more efficient by providing standard functionality—encryption, networking, or graphics processing, for example—in a reusable package. Unfortunately, developers of real-time systems are often cut off from such benefits because these libraries provide no WCET information. End users usually do not wish to perform this analysis themselves, and even if the vendors are willing to perform a WCET analysis and add the necessary source code annotations, they may not wish to expose this code to the outside world. The abstract timing approach advocated by Puschner solves these problems by allowing vendors to bundle WCET information with their code in

[9]We define "high-level" as "above bytecode level."

a portable, reusable format that keeps source code private.

Like Puschner, Hu [14] also developed techniques for making WCET information more portable. Instead of performing an analysis, however, Hu focused on WCET source code annotations, introducing a new format called XAC, or Extensible Annotation Class. Similar in scope to Bernat's WCETAn technique [4], XAC encodes timing hints as source code comments rather than explicit method calls. This improvement over WCETAn eliminates the relatively complex task of re-writing the bytecode to remove the method calls (in order to eliminate their performance penalty). Hu also specified an extensible file format for bundling WCET annotations with their corresponding class files.

Later that same year, Hu extended his XAC format to handle the problem of dynamic dispatch [13]. Typical in object-oriented programs, dynamic dispatch of method invocations (e.g., polymorphism) is simply disallowed in most WCET tools. Such tools are normally designed for procedural languages, such as C, where dynamic dispatch is much less common. In Java, however, almost every method call requires dynamic dispatch. Hu addressed this problem by providing new WCET annotation types designed for class hierarchies. For example, the programmer could specify a subset of child classes that are valid for a particular method invocation on a base class. This simplistic approach dumps most of the work in the programmer's lap, relying entirely on manual annotations to tighten the WCET bounds of dynamic dispatch.

In stark contrast with Hu's style, Guedes [10] dispensed with annotations altogether, explaining how Gustafsson's "abstract interpretation" technique [11] could be applied to Java. The goal was to remove the need for annotations as much as possible, saving the trouble of having to provide WCET parameters (loop bounds in particular) in many cases. This very preliminary work was largely theoretical and has not been pursued.

The remaining work in high-level analysis comes from a European initiative to advance the role of Java in real-time systems. Dubbed HIDOORS (High Integrity Distributed Object-Oriented Realtime Systems) [26], it had many goals, some of which were perhaps unrealistic for a 30-month project: a real-time garbage collector, a graphical UML-based modeling tool, a distributed real-time event manager, and a WCET analysis tool. While the real-time garbage collector has seen new life as part of the Jamaica Virtual Machine [25], the WCET tool never progressed beyond the specification stage.[10]

5. Low-level WCET Analysis for Java Bytecode

WCET analysis of bytecode is only a partial solution. Computation of the actual WCET requires low-level analysis that takes into account the particular timing characteristics of a target processor.

Toward that end, Bate expanded on Bernat's work by developing a framework for low-level WCET analysis of Java bytecode [2]. To remain portable among processors, the framework differs from traditional approaches: Instead of calculating the WCET of each basic block (which is impossible at the bytecode level), it calculates bytecode frequencies. When a particular target architecture is known, the frequency vectors can then be mapped to a concrete timing model. Two years later, Bate integrated this approach into a single framework [3] that combines the high-level [4] and low-level [2] techniques in one vertical package.

Instead of concentrating on a portability solution for WCET analysis tools, Hu targeted the Java platform itself. Citing the growing interest in pure, 100% Java real-time specifications, such as the RTSJ [5] and the Real-time Core Extensions [8],[11] Hu observed that none offered any mechanism for WCET analysis. In addition, existing analysis techniques were exclusive to procedural programming languages, ignoring the dynamic dispatching features of Java.

Hu therefore adapted Bernat's existing WCET framework for the needs of Java, re-branding it the XRTJ (eXtended Real-Time Java) [17]. Essentially a refinement of this existing framework, it added one notable new feature for low-level WCET analysis. Specifically, XRTJ prescribed a measurement-based technique for deriving a timing model of an arbitrary Java virtual machine [15]. This timing model is simply a performance profile, a benchmark of the target processor's ability to interpret bytecodes in the presence of an operating system and virtual machine. The resulting WCET is therefore an estimate and does not provide the hard guarantee of a static analysis. However, it works across all Java systems and requires no modifications to the virtual machine.

[10]Perhaps one reason the HIDOORS project did not prove more successful is that it was based on J Consortium's now-defunct RTCE specification, rather than Sun's RTSJ. In other words, HIDOORS appears to have "bet on the wrong horse."

[11]The RTSJ and the RTCE were two competing real-time specifications for Java. Although they were largely similar, the RTSJ had the support of Sun. As a result, all development of RTCE has ceased, the J Consortium has disbanded, and the web site (j-consortium.org) has been taken over by a domain squatter.

5.1. Discussion

These efforts are the only published work on low-level WCET analysis for Java bytecode. This begs the question of why other groups have not pursued the same challenge. The explanation is manifold:

- Even today, the concept of real-time Java is relatively new. Reliable implementations of the RTSJ, for example, became available only in the last few years. Despite new large-scale projects [20], acceptance of Java for real-time systems is still limited, even among the research community.

- Low-level analysis of bytecode is extremely difficult. Mapping a non-Java language to bytecode is a formidable challenge by itself. In addition, the bytecode must be translated to an arbitrary target architecture, all the while maintaining tight WCET bounds. This requires a detailed analysis to account for pipeline and cache effects, not to mention the overhead of the operating system and the virtual machine. As a result of this complexity, the WCET analysis is often pessimistic, counteracting the benefits that bytecode portability brings.

- The multiple layers of OS, VM, and processor complicate low-level bytecode analysis. One way to mitigate this problem is to adopt a Java-native processor such as Schoeberl's JOP [24] or aJile System's aJ-100 [12]. These processors collapse the vertical stack, removing the OS and VM layers entirely and greatly simplifying low-level analysis. Until recently, however, viable Java-native processors such as these were unavailable, making them even less prevalent in the research community than real-time Java. In addition, restricting low-level analysis to these processors limits the portability, and thus the acceptance, of any analysis technique.

These issues have made low-level bytecode analysis an unattractive research topic.

6. WCET Analysis for Java-specific Processors

Modern CPU architecture is the WCET researcher's worst nightmare. Large pipelines, branch prediction, and sophisticated multi-level caching have greatly improved average throughput, but not without cost. Providing a tight guarantee on worst-case execution time is horrendously difficult on these superscalar processors.

As a result, new processor architectures have emerged that are designed specifically for real-time systems, making them an easier target for WCET analysis. An example from the Java domain is JOP, or Java Optimized Processor [24], a WCET-aware CPU that executes bytecode natively without the need for an OS or virtual machine.

JOP offers three key features that allow bytecode execution time to be predicted tightly:

- JOP translates bytecodes into microcode instructions, each of which executes in a single cycle. And because there are no dependencies between bytecodes, calculating WCET of basic blocks is a simple matter of summing the cycle count of each bytecode.

- JOP has a short four-stage pipeline, allowing branch prediction logic (which complicates WCET analysis) to be discarded with minimal performance loss.

- JOP provides a unique instruction cache specially designed for WCET analysis in Java. It is based on the observation that no branch instructions in Java jump outside of a method; therefore, the "method cache" in JOP [23] is based on whole methods rather than small cache lines. Consequently, hit and miss detection occurs only during method invocation and return, allowing WCET analysis of the cache to be ignored entirely during the execution of individual methods.

In other work on Java-specific processors, Chai [7] developed a technique for pre-processing class files that were destined for embedded systems. Noting that such systems normally prohibit dynamic class loading and garbage collection, Chai relies on these assumptions to replace certain bytecodes with "optimized" variants. These altered bytecodes exchange flexibility for fewer cycles per instruction, leading to a reduced WCET. As such, Chai's proposal is better described as a speed optimization, not a WCET analysis technique.

7. Other Work in WCET Analysis for Java

Portability, low-level analysis, and high-level analysis are where most WCET research for Java has been applied. This section presents work in WCET analysis that does not fall cleanly into one of these three main categories.

Persson describes a development environment for real-time Java that incorporates WCET information

[21]. Called Skånerost, it displays the WCET of a particular method in the margins of its source code editor. The WCET value is updated continuously, as the source code changes, to provide feedback to the developer. (Persson does not describe exactly how this WCET value is obtained; an analysis tool and the appropriate annotations are assumed to be available.)

Hu developed a "gain time" reclamation framework for hard real-time Java [16]. Based on the assumption that real-time tasks often do not follow the worst-case path at run-time, the goal is to reclaim this "gain time" by detecting when a task has completed before its predicted worst-case time. A lower-priority task can then be executed, increasing overall CPU utilization. The novelty of Hu's approach lies in the ability to track object types as they change (via a so-called Object Type Lifetime Graph), thus yielding tighter WCET bounds than would otherwise be possible in dynamic dispatch languages like Java.

Corsaro addressed the problem of obtaining tight WCET bounds on memory allocations in Java [9]. Borrowing principles from UNIX file systems, the approach gains predictable allocation time at the cost of wasted space. The basic idea is to permit fragmentation of memory chunks. Allocation and deallocation of the chunks can then be accomplished in linear time, improving the WCET of memory operations.

Finally, Lei focused on tightening the WCET of RTSJ's asynchronous transfer of control (ATC) mechanism [6]. Conventional ATC implementations rely on a recursive procedure to locate the appropriate catch class, making WCET analysis difficult. Lei solves this problem by performing class resolution and linking at compile-time rather than run-time (if certain assumptions about the run-time environment can be made). ATC then reduces to a simple comparison, and its WCET is more predictable.

8. Conclusion

Given that the entire collective work in WCET analysis for Java can be summarized in little more than three pages, much remains to be done. Several open problems persist:

- Can dynamic dispatch be handled in any automatic way (without a total dependency on manual annotations [13])?
- All of the prototype tools work on a per-method basis only. Is it possible to calculate a whole-program WCET?
- Given the difficulty of performing a fully static analysis on the Java stack, could measurement-based or probabilistic approaches be a sufficient replacement?

For the first two open questions, we believe that a whole-program control-flow graph, combined with new research into program slicing techniques, can address these problems, and we are actively conducting research in this area. For the last question, it may be possible to adapt existing commercial tools for C, such as RapiTime, for measurement-based analysis of Java. Of course, the true solution is unclear, and many challenges still lie on the horizon.

References

[1] D. F. Bacon, P. Cheng, and V. T. Rajan. The Metronome: A simpler approach to garbage collection in real-time systems. In R. Meersman and Z. Tari, editors, *On The Move to Meaningful Internet Systems: OTM 2003 Workshops*, volume 2889 of *Lecture Notes in Computer Science*, pages 466–478. Springer Berlin, November 2003.

[2] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-level analysis of a portable Java byte code WCET analysis framework. In *Proceedings of the Seventh International Conference on Real-Time Computing Systems and Applications (RTCSA 2000)*, pages 39–48, Los Alamitos, CA, USA, December 2000. IEEE Computer Society.

[3] I. Bate, G. Bernat, and P. Puschner. Java virtual-machine support for portable worst-case execution-time analysis. In *Proceedings of the Fifth IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2002)*, pages 83–90, April 2002.

[4] G. Bernat, A. Burns, and A. Wellings. Portable worst-case execution time analysis using Java byte code. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (Euromicro-RTS 2000)*, pages 81–88, Los Alamitos, CA, USA, June 2000. IEEE Computer Society.

[5] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison Wesley Longman, January 2000.

[6] Z. Chai, W. Chen, Z. Tang, Z. Chen, and S. Tu. Asynchronous transfer of control in the RTSJ-compliant Java processor. In *The Fifth International Conference on Computer and Information Technology (CIT 2005)*, pages 764–770. IEEE Computer Society, September 2005.

[7] Z. Chai, Z. Tang, L. Wang, and S. Tu. An effective instruction optimization method for embedded real-time Java processor. In *2005 International Conference on Parallel Processing Workshops (ICPPW'05)*, pages

225–231, Los Alamitos, CA, USA, June 2005. IEEE Computer Society.

[8] J. Consortium. Real-time core extensions, September 2000.

[9] A. Corsaro and C. Santoro. Optimizing JVM object operations to improve WCET predictability. In *Proceedings of the Fourth International Workshop on Worst-Case Execution Time Analysis (WCET 2004)*, pages 15–18, June 2004.

[10] P. A. Guedes and S. V. Cavalcante. On the design of an extensible platform for flow analysis of Java using abstract interpretation. In *Proceedings of the Third International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*, pages 47–50, July 2003.

[11] J. Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Mlardalen University, Vsters, Sweden, May 2000.

[12] D. S. Hardin. Real-time objects on the bare metal: An efficient hardware realization of the Java virtual machine. *Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, pages 53–59, May 2001.

[13] E. Y.-S. Hu, G. Bernat, and A. Wellings. Addressing dynamic dispatching issues in WCET analysis for object-oriented hard real-time systems. In *Proceedings of the Fifth IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2002)*, pages 109–116, Los Alamitos, CA, USA, April 2002. IEEE Computer Society.

[14] E. Y.-S. Hu, G. Bernat, and A. Wellings. A static timing analysis environment using Java architecture for safety critical real-time systems. In *Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 77–84, Los Alamitos, CA, USA, January 2002. IEEE Computer Society.

[15] E. Y.-S. Hu, A. Wellings, and G. Bernat. Deriving Java machine timing models for portable worst-case execution time analysis. In *On the Move to Meaningfull Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems*, volume 2889 of *Lecture Notes in Computer Science*, pages 411–424. Springer, November 2003.

[16] E. Y.-S. Hu, A. Wellings, and G. Bernat. Gain time reclaiming in high performance real-time Java systems. In *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2003)*, pages 249–256, Los Alamitos, CA, USA, May 2003. IEEE Computer Society.

[17] E. Y.-S. Hu, A. Wellings, and G. Bernat. XRTJ: An extensible distributed high-integrity real-time Java environment. In *Proceedings of the Ninth International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2003)*, volume 2968 of *Lecture Notes in Computer Science*, pages 208–228. Springer Berlin, February 2003.

[18] R. Kirner and P. Puschner. Discussion of misconceptions about WCET analysis. In *Proceedings of the Third International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*, pages 61–64, July 2003.

[19] E. Kligerman and A. D. Stoyenko. Real-time Euclid: a language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12(9):941–949, September 1986.

[20] D. Lammers. REAL-TIME JAVA: Reliability quest fuels RT Java projects. *EE Times*, March 2005.

[21] P. Persson and G. Hedin. An interactive environment for real-time software development. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 2000)*, pages 57–68, Washington, DC, USA, June 2000. IEEE Computer Society.

[22] P. Puschner and G. Bernat. WCET analysis of reusable portable code. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS 2001)*, pages 45–52, Washington, DC, USA, 2001. IEEE Computer Society.

[23] M. Schoeberl and R. Pedersen. WCET analysis for a Java processor. In *Proceedings of the Fourth International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, October 2006.

[24] M. Schberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, Vienna, Austria, January 2005.

[25] F. Siebert. Hard real-time garbage-collection in the Jamaica virtual machine. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA 1999)*, pages 96–102, Washington, DC, USA, December 1999. IEEE Computer Society.

[26] J. Ventura, F. Siebert, A. Walter, and J. Hunt. HIDOORS - a high integrity distributed deterministic Java environment. In *Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 113–118, Los Alamitos, CA, USA, January 2002. IEEE Computer Society.