

# An Interactive Environment for Real-Time Software Development

Patrik Persson and Görel Hedin  
Department of Computer Science, Lund University  
Box 118, SE-221 00 Lund, Sweden  
Patrik.Persson@cs.lth.se Görel.Hedin@cs.lth.se

## Abstract

*Object-oriented languages, in particular Java, are beginning to make their way into embedded real-time software development. This is not only for the safety and expressiveness of the source language; the mobility and dynamic loading of Java bytecode make it particularly useful in embedded real-time systems.*

*However, using such languages in real-time systems makes it more difficult to predict the worst-case execution time of tasks. Such predictions are necessary for predictable task scheduling in the developed system. Garbage collection, common in object-oriented languages, must be considered; to schedule garbage collection safely, we must know how much memory it has to handle. Dynamic binding in conjunction with dynamic loading of code also needs treatment.*

*We show how techniques for predicting time and memory demands of object-oriented programs are integrated into the Skånerost development environment. The environment explicitly targets an iterative development process, which is particularly important in real-time software development since time and memory demands cannot be determined until the code is written. Design changes due to timing problems become more costly as development progresses, and Skånerost allows such problems to be detected early.*

## 1. Introduction

In the past, programming of embedded hard real-time systems was often done by hardware engineers primarily to make the hardware work. The programs were small and typically written in assembly or low-level C code.

Today, more microprocessors are sold for use in embedded systems than for desktop computers. Embedded real-time systems become more commonplace, but they also become more complex. Mobile telephones, photocopiers, and industrial robots are today all based on substantial amounts of software. At the same time, many embedded systems (such as aircraft control systems) are safety critical.

To reduce development effort and increase portability and reliability, a number of efforts [16, 22] are made to adapt high-level languages, in particular Java [5], to hard real-time applications. Besides being a modern object-oriented source language, Java is compiled to portable bytecode for execution by a Java virtual machine. This enables dynamic loading of code, which is of particular interest in many embedded systems, such as industrial robots and satellites.

Java is designed to be a safe language. Compile-time checks are used as far as possible, and run-time checks are used in the remaining cases. This safety of the language facilitates early detection of errors and thus implies safety in the developed system. Such safety is important in the dynamically

configured systems mentioned above, where dynamically loaded code must not cause any existing code to fail. In other cases, where predictability requirements rule out dynamic solutions, the need for safety is yet more emphasized.

Real-time software development also places pressure on the software development tools. One of the most fundamental features of a piece of real-time software, its worst-case execution time, is also one of the most difficult ones to predict. Modern object-oriented languages, with mechanisms such as dynamic binding (virtual methods) and garbage collection, complicate the problem further.

Since the correctness of the real-time system as a whole depends on whether timing requirements are met, it is of paramount importance to provide continuous timing feedback to the programmer during development. The *Skånerost*<sup>1</sup> development environment, presented in this paper, is designed to provide such feedback.

### 1.1. Paper outline

In Section 2 we present the aspects of real-time software development that influence our development tool design, including some related work. In Section 3 we present our approach to predictable real-time Java and some associated analysis techniques. In Section 4 we present the Skånerost environment. Section 5 concludes the paper.

## 2. Aspects of real-time software development

The key property of a real-time system is that it performs its computations within some sort of deadlines. The severity of missing a deadline depends on the application; in an engine control system, it may result in permanent engine damage, whereas in a real-time video application, it may merely result in a transient image distortion.

### 2.1. Worst-case execution time

Real-time systems are typically modeled as a set of concurrent, periodic tasks. These tasks must be *scheduled* in a way that ensures that the system can fulfill its timing requirements. More precisely, the task schedule in a real-time system is based on the following parameters:

**Task period:** the interval with which the task is launched for execution.

**Task deadline:** the point in time when a particular execution of the task must be finished.

**Worst-case execution time (WCET):** the largest amount of execution time possibly required for any execution of the task.

Real-time scheduling is a well established research area and a number of scheduling techniques are available, e.g., [13]. However, they all require values for the parameters above (and possibly more).

Whereas the first two parameters (periods and deadlines of tasks) are usually design parameters, the third parameter, the worst-case execution time, cannot be deduced from the requirements. Instead it is a property of the executable code and the hardware it runs on. A *WCET analysis* of the code is required to predict its WCET, based on some model of the target environment. (Our target environment is described in Section 3.) A requirement for the WCET can be stated, of course, but that requirement must still somehow be verified.

---

<sup>1</sup>The name refers to a rather strong coffee blend originating in the Skåne province in southern Sweden.

## 2.2. Supporting an iterative development model

It is desirable to obtain WCET predictions throughout development, not just for the final code. Such predictions may concern the execution time of individual loops, methods, or entire tasks. Should these predictions indicate that the system's timing requirements cannot be met, either the design must be revised or the timing requirements must be reassessed. In any case, such a timing problem should be detected as early as possible during development.

## 2.3. Related work

Most approaches to WCET analysis are targeted towards object code, e.g., [11, 12, 14, 25]. Such analyses have difficulties with object-oriented languages, where invocations of virtual methods are compiled to indirect jumps via pointers. Object code analyses can, in general, not analyze such function pointers.

There are very few publications available on WCET analysis for object-oriented languages besides our own work. Bernat [2] presents an approach to WCET analysis of Java bytecode, but does not discuss dynamic binding or garbage collection. Gustafsson [6] uses abstract interpretation to analyze programs in a special real-time dialect of Smalltalk. He uses type inference to facilitate a WCET analysis of method calls, but does not address the issues of dynamic loading and garbage collection as done in this paper.

The tool we will present allows the programmer to obtain predictions of code portions throughout development. Other approaches typically use less interactive techniques such as integer-linear programming (ILP) [11] or abstract interpretation (symbolic execution) of the object code [6, 14] (which may not terminate). Ko et al. [10] describe an interactive timing analysis environment; however, in that work, the interactive user interface is invoked after a complete analysis of the entire program. They further focus on compiled C code and hardware aspects, rather than the Java bytecode approach in the present work.

## 3. Predictable real-time Java

The target language of our techniques is a subset of Java, supporting classes, inheritance, and dynamic binding. Excluded language features currently include method overloading, interfaces, and exceptions; these will be considered in the future.

We also support the compilation of Java to portable bytecode, a technique originally developed to facilitate code mobility over the Internet in the form of *applets*. Our motivation is somewhat different. Being able to transport code over a network and dynamically load it into a virtual machine in a safe manner is particularly useful in the context of embedded systems, and has enabled interesting consumer technologies such as Jini [23]. The bytecode approach can also add safety to dynamic loading of embedded controller code as previously done in C/C++ [18].

The bytecode language constitutes a clean cut between the high-level language and the execution environment. This division is useful for WCET analysis. By encapsulating the timing properties of the virtual machine into a special *virtual machine timing model*, porting of the WCET analysis to new platforms is made easier. In our current timing model, we ascribe a constant WCET to each bytecode instruction.

As we will discuss more when we treat the analyses, we use *annotations* in the source code to provide information that should be obvious to the programmer but is difficult for a tool to deduce.

These annotations are expressed as source code comments adhering to a special syntax. Such annotations can be identified and used by an analysis tool, yet ignored by a traditional compiler.

One particular case where annotations are sometimes needed is loops. In many important cases, the maximal number of loop iterations can be deduced automatically from the code. For more complicated loops, however, explicit annotations are required to indicate the upper bound. In any case, the halting problem implies that no tool can automatically analyze arbitrary loops.

Although this may at first appear to be a restriction of the language, it is in fact an ubiquitous circumstance of real-time software: if no loop bounds exist, the WCET cannot be bounded either, and the code cannot be scheduled safely. The programmer must thus be aware of these bounds, and it is reasonable to require this knowledge to be stated in the source code.

### 3.1. WCET analysis techniques

A WCET analysis is used to predict the worst-case execution times of tasks. Using object-oriented languages like Java in real-time systems complicates the analysis:

**Garbage collection** requires special treatment. In a real-time system, a classic “stop-the-world” garbage collector would introduce indeterministic response times and invalidate the assumptions of the task scheduling.

**Dynamic binding** (virtual methods) must also be considered. Unlike ordinary function or procedure calls, the code to execute upon a virtual method call is not statically known: it is determined at run-time. Static WCET prediction of such a call requires special techniques.

Our general approach to WCET analysis is based on the timing schema [21] in an attribute grammar context [20]. We will now discuss our approach to the just mentioned issues in WCET analysis for object-oriented languages.

### 3.2. Real-time garbage collection

As shown by Henriksson [9], a garbage collector can be used in a hard real-time system by proper *scheduling* of the garbage collector. The garbage collection work is scheduled into suitable time slots, and high-priority tasks are guaranteed instant access to free memory. We base this discussion on Henriksson’s real-time garbage collection approach.

Predictable garbage collection is not only a matter of execution time. It is also important that the *fragmentation of memory* is predictable (and, of course, preferably small). If it is not, real-time tasks cannot be guaranteed access to memory. To handle fragmentation, a compacting garbage collector is used; that is, the heap is constantly re-organized to keep fragmentation low. (Such a garbage collector may appear to harm the response times of real-time tasks. To the contrary, however, the scheduling approach allows these response times to be *reduced*, since memory management administration is handled entirely in a separate task.)

As the preceding discussion suggests, the amount of garbage collection work that needs to be scheduled depends on the memory consumption of the program at hand. Henriksson’s approach requires information about the amount of *live memory* used by the program, that is, the memory occupied by objects the program can use. Other real-time garbage collectors use similar parameters [17].

Analogously to the WCET analysis, a *live memory analysis* can be used to compute an upper bound on the amount of live memory required by a program. For simple, non-object-oriented

programs this upper bound can be conservatively (but pessimistically) estimated by assuming all references to refer to unique objects. In general, however, some considerations must be made:

**Recursive data structures.** Without additional information, it is impossible to determine the number of elements in a general recursive data structure, and thus the amount of memory occupied by it.

**Aliasing.** The assumption that all references reference unique objects is generally not true; references may be used for traversing data structures (similar to loop induction variables) or to otherwise simplify data structure traversals (such as the back reference in a double linked list). Such references are redundant from a live memory analysis point of view.

**Inheritance.** Inheritance implies that a reference with static qualification  $c$  also may refer to objects of subclasses of  $c$ . Since objects of these subclasses may contain more data and references than those of their superclass, the amount of live memory may be affected.

**Dynamic binding.** Dynamic binding affects control flow, which in turn affects memory use.

We use annotations from the programmer to get information that is known to the programmer but difficult for a tool to deduce accurately, such as shape and size of data structures. The exact form of our annotations will be presented in Section 4, where we present our tool.

Information about the sizes of data structures is, of course, vital to the live memory analysis. Information about the *shape* of data structures is equally important; without it, a doubly linked list of length 50 would be indistinguishable from a binary tree of depth 50. (Both are typically described by a class  $C$  containing two recursive references to  $C$  itself.) Such a tree may contain up to  $2^{50} - 1 \approx 1.12 \cdot 10^{15}$  elements rather than the 50 in a list. Although techniques for shape analysis exist for other purposes, these are too conservative to facilitate an accurate live memory analysis. Other researchers have recognized the need for annotations to improve the accuracy of shape analysis [8].

Based on these rather straight-forward annotations, our live memory analysis algorithm [19] computes an upper bound on the amount of live memory in a program.

### 3.3. Dynamic binding

An automatic WCET analysis of method calls is possible in some cases. If information about all possibly called implementations of a particular method are available for analysis, one safe WCET estimation is the longest WCET of these implementations.

This approach assumes that *any* of the existing implementations may be called from a given call site. Such an approximation may be improved by using existing type analysis techniques (e.g., [24]) developed for optimizing compilers. A similar approach is taken in [6].

However, a global analysis is not always possible. Java's dynamic class loading allows new implementations of a given virtual method to be introduced at run-time. We suggest another approach to handling timing requirements on virtual methods (in the context of dynamic class loading), based on two observations:

- *The WCET of a method call should be known to the programmer implementing that call, even if the implementation is not yet available.* If the execution time of the call is unknown, the programmer has no way of fulfilling the timing requirements.
- *When a new class is loaded, it must not break the timing assumptions of the existing system.* A new implementation of a virtual method should not cause any code *calling* that method to miss its deadline.

These requirements are analogous to those for the type system in any statically typed programming language. For example, assume that the function  $f$  accepts a single integer argument. This information is used both for semantic analysis of calls to  $f$  (to ensure that a call passes an integer argument) and of  $f$  itself (to ensure that it accepts an integer argument).

Timing constraints, such as bounds on WCETs of virtual methods, constitute interface information and should be treated in a manner similar to type information. We thus advocate *expressing timing constraints on a virtual method in the method's signature, along with the types of the parameters and the return value*. This information is expressed as annotations, as discussed in the beginning of Section 3.

To give concrete form to the preceding discussion, Figure 1 shows a small example of an embedded PID controller. The framework allows a method *display* in the operator interface to be called (to display, e.g., the control signal and reference value) upon each iteration of the controller. The controller should be possible to use with a variety of operator interfaces. As an example, an implementation of the operator interface can use *display* to buffer results for use by another task. Regardless of the implementation, we want to bound the WCET of *display* to maintain predictability of the controller.

## 4. The Skånerost environment

In Figure 2 an overview of the Skånerost environment is given. It shows the principles of how the tool, the programmer, and the execution environment (the virtual machine) are interconnected.

Three key properties of the environment are shown in this figure:

- *Integration of analysis and compilation.* Both WCET analysis and live memory analysis gain from using information from the compilation, such as name/type information and generated code. This integration thus reduces the complexity of the analysis implementations.
- *Continuous feedback to the programmer.*
- *Compilation to bytecode.*

Skånerost is based on the APPLAB language tool [3, 4]. We use APPLAB's specification language for implementation of all aspects of compilation and analysis: semantic analysis, code generation, WCET analysis, and live memory analysis. The specification language uses Reference Attributed Grammars (RAGs) [7], a special kind of attribute grammars. RAGs allow attributes to be references to syntax nodes, making it easy to express non-local dependencies (such as those between declarations and uses of variables). Non-local dependencies are especially complex to handle in object-oriented languages, where they do not necessarily follow the block structure of programs.

The syntax-directed editor in APPLAB allows an arbitrary attribute (as defined in an attribute grammar) to be inspected by the programmer at any time during development. The WCET of a part of the developed code (such as a loop or a method) is represented by an attribute in the corresponding syntax node. The amount of live memory possibly referenced by a particular declaration in the program is represented in the same way.

We will now outline the internals of the components of Skånerost: compilation, WCET analysis, and live memory analysis.

### 4.1. Compilation to bytecode

Java source code is compiled to an internal representation of Java bytecode. This internal representation is used both for additional analyses and unparsing to a text representation for use by the

```

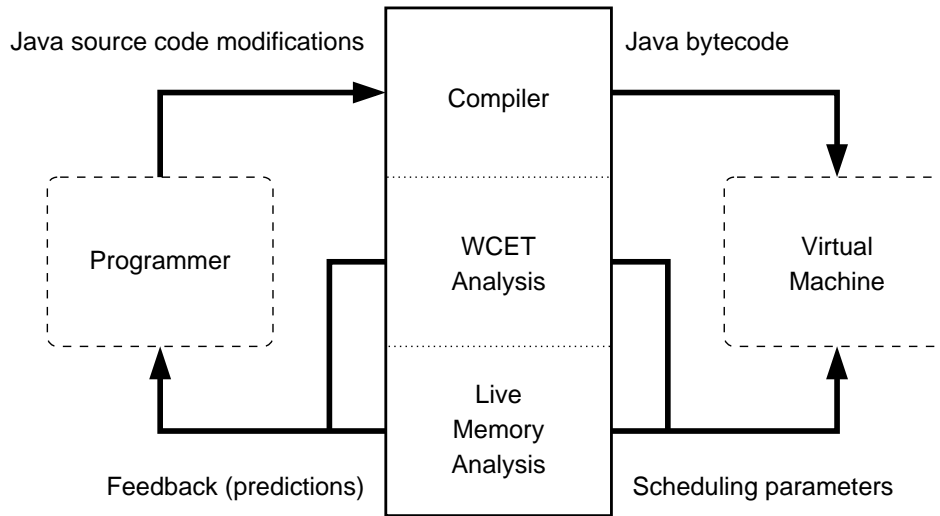
class PIDController extends RealTimeThread {
    private double uc, y, u, v;
    private GUI myGUI = null;
    ...
    public synchronized void setGUI(GUI g) { myGUI = g; }
    public synchronized GUI getGUI()      { return myGUI; }
    public void run() {
        long t = currentTime();
        while (true) {
            y = IO.getY();
            uc = IO.getUc();
            calc_output();
            IO.setU(u);
            update_states();
            GUI g = getGUI();
            if (g != none) g.display(uc, y, u); // ..... (A)
            t += 100;
            waitUntil(t);
        }
    }
}

class GUI {
    ...
    abstract public void display(double uc,
                                double y,
                                double u)
        /*$ time-bound 25ms */; // ..... (B)
}

class BufferedGUI extends GUI {
    ...
    public void display(double uc,
                        double y,
                        double u) {
        // Real-time stuff goes here. .... (C)
    }
}

```

**Figure 1. Expressing WCET bounds for the operator interface in a PID controller. The call at (A) has a bounded WCET, since the top-level declaration of the called method at (B) has a WCET bound associated with it. The implementation at (C) must adhere to this bound.**



**Figure 2. Overall design of Skånerost.**

Jasmin bytecode assembler [15].

One such additional analysis of the internal bytecode representation is used to determine the required stack depth of each method. Since the Java virtual machine is stack-based and an operand stack is allocated in each method frame, the stack depth requirements for each method must be specified in the compiled bytecode. This information is computed by an analysis of the internal bytecode representation.

An example showing Skånerost containing an edited Java class and the compiled bytecode is given in Figure 3.

#### 4.2. Worst-case execution time analysis

The WCET analysis uses the internal bytecode representation just mentioned. It also uses information from the source code, such as annotations from the programmer. In Figure 4, the source code of a polynomial controller is shown. The output of the controller is computed from the equation

$$R(q)u(k) = T(q)u_c(k) - S(q)y(k)$$

where  $u(k)$  is the output at (discrete) time  $k$ ,  $u_c(k)$  is the command signal,  $y$  is the measured process output, and  $R(q)$ ,  $S(q)$ , and  $T(q)$  are polynomials in the forward-shift operator [1, Eq. 5.2].

The predicted worst-case execution time of the `calc_u` method is shown in the  $T$  window. (The exact value depends on the virtual machine timing model mentioned in Section 2.2.) This prediction can be used both as a scheduling parameter and as feedback (regarding the program's ability to meet its deadlines) to the programmer.

#### 4.3. Live memory analysis

As discussed in Section 3.2, recursively defined data structures require special treatment in live memory analysis. The class `PolyTerm` in Figure 4 has an annotation (`/*$ path-bound 10 */`), indicating that at most 10 instances of that class appear in sequence (that is, each polynomial in this controller has at most 10 terms). Without this information, the amount of live memory cannot



**Figure 3. Source code (left) and the corresponding compiled bytecode (right).**

be bounded since the *PolyTerm* class is recursively defined. However, the programmer's domain knowledge about the control algorithm, expressed as an annotation, makes an accurate analysis possible.

The `/*$ redundant */` annotation on the local reference *term* in the *calc\_u* method indicates that any object referenced by *term* is always referenced by another, non-redundant reference. Hence, *term* does not contribute to the amount of live memory.

The tools interactively provides worst-case predictions of the memory demands of the program at hand. Analogously to WCET analysis, the obtained prediction can be used both as a scheduling parameter (to compute the amount of execution time required for garbage collection; the exact form of this computation depends on the garbage collector) and as programmer feedback. However, the use of live memory predictions is not restricted to scheduling of garbage collection. It is often important to know how much memory a program requires, particularly in cost-sensitive embedded systems where memory is scarce.

## 5. Conclusions

The key property of a real-time system is its ability to perform its computations within deadlines. Should such a system turn out to be unable to keep its deadlines, the design or the requirements (or both) must be reassessed. Such changes become more and more expensive as development progresses, and it is thus imperative that such schedulability problems are discovered as soon as possible.

It is highly desirable that the timing properties of the developed program can be continuously observed, and the interactive nature of the tool we have presented allows just that. It provides interactive predictions of time and memory bounds for selected parts of the developed program,

The screenshot shows a window titled "skånerost < Release 1.4.1 Rev 425 >". On the left, there is a sidebar with a tree view containing "ABSTRACT", "CONCRETE", "PARSE", and "PROGRAM". Below this are two windows: "T" with the value "164" and "R" with the value "544". The main area displays the following Java code:

```

class PolyTerm /** path-bound 10 */ {
    double value;
    PolyTerm next;
}

class Polynomial {
    PolyTerm first;
}

class PolyController {
    Polynomial t_poly;
    Polynomial s_poly;
    Polynomial r_poly;
    public double calc_u() {
        double u;
        int k;
        PolyTerm term /** redundant */;
        u = 0;
        term = t_poly.first;
        for (k = 0; k < 10; k++) {
            u = u + term.value * get_uc(k);
            term = term.next;
        }
        term = s_poly.first;
        for (k = 0; k < 10; k++) {
            u = u - term.value * get_yc(k);
            term = term.next;
        }
        term = r_poly.first;
        for (k = 0; k < 10; k++) {
            u = u - term.value * get_ur(k);
            term = term.next;
        }
        return u;
    }
}

```

**Figure 4. Source code of a polynomial controller and predictions for that code. The *T* window shows the worst-case execution time of the *calc\_u* method, and the *R* window shows the amount of memory referenced from a *PolyController* instance.**

along with the generated code.

The target language is Java, a language well suited for embedded real-time systems programming, with respect to both the source representation (a contemporary type-safe object-oriented language) and the portable, dynamically loadable bytecode.

This tool is novel in that these predictions are available directly in the development environment throughout development. It is also novel in its support for a modern, statically typed, object-oriented language with garbage collection and dynamic loading of code.

### 5.1. Future work

The present work is related to the development of the Infinitesimal Virtual Machine (IVM), a real-time Java virtual machine designed for a very small memory footprint (tens of kilobytes). The IVM is currently in development at the Department of Computer Science, Lund University. Future plans include development of a timing model for the IVM on a small embedded system, and possibly extending our work to handling more elaborate timing models.

Work on the Skånerost tool continues to support a larger Java subset and integrate analyses further.

### Acknowledgments

Boris Magnusson, Klas Nilsson, Anders Ive, Roger Henriksson, and the anonymous reviewers provided valuable comments and suggestions. Elizabeth Bjarnason implemented large parts of

## APPLAB.

The work of Patrik Persson was financed by ARTES (A network for Real-Time research and graduate Education in Sweden) and SSF (the Swedish Foundation for Strategic Research). The work of Görel Hedin was partially financed by NUTEK (Swedish National Board for Industrial and Technical Development).

## References

- [1] K. J. Åström and B. Wittenmark. *Computer-Controlled Systems — Theory and Design (Third Edition)*. Prentice-Hall, 1996.
- [2] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. To appear in *Proceedings of the 12th EuroMicro Conference on Real-Time Systems*, Stockholm, June 2000.
- [3] E. Bjarnason. *Interactive Tool Support for Domain-Specific Languages*. Licentiate Thesis, Department of Computer Science, Lund University, December 1997.
- [4] E. Bjarnason, G. Hedin, and K. Nilsson. Interactive Language Development for Embedded Systems. *Nordic Journal of Computing*, Vol. 6, pages 36-55, 1999.
- [5] J. Gosling, B. Joy, and G. Steele. *The Java Language Language Specification*. Addison-Wesley, 1996.
- [6] J. Gustafsson. *Analyzing Execution Time of Object-Oriented Programs with Abstract Interpretation*. PhD Thesis, Department of Computer Systems, Information Technology, Uppsala University, Sweden, May 2000.
- [7] G. Hedin. Reference Attributed Grammars. *Proceedings of WAGA'99: Second Workshop on Attribute Grammars and their Applications*, pages 153-172. Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.
- [8] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*, San Francisco, California, June 1992.
- [9] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD Thesis, Department of Computer Science, Lund University, September 1998.
- [10] L. Ko, N. Al-Yaqoubi, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M. Harmon. Timing Constraint Specification and Analysis. *Software — Practice & Experience*, January 1999.
- [11] Y.-T. S. Li, S. Malik, A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'95)*, December 1995.
- [12] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An Accurate Worst Case Timing Analysis Technique for RISC Processors. *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'96)*, December 1996.
- [13] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM* 20(1), 1973.
- [14] T. Lundqvist and P. Stenström. Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques. *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, Montreal, Canada, June 1998.
- [15] J. Meyer. The Jasmin bytecode assembler. <http://mrl.nyu.edu/meyer/jvm/jasmin.html>
- [16] Newmonics, Inc. <http://www.newmonics.com>
- [17] K. Nilsen. Reliable Real-Time Garbage Collection of C++. *Computing Systems*, 1994.
- [18] K. Nilsson, A. Blomdell, and O. Laurin. Open Embedded Control. *Real-Time Systems*, Vol. 14, No. 3, 1998.
- [19] P. Persson. Live Memory Analysis for Garbage Collection in Embedded Systems. *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*, Atlanta, Georgia, May 1999. ACM SIGPLAN Notices 34(7), pages 45-54.
- [20] P. Persson and G. Hedin. Interactive Execution Time Predictions using Reference Attributed Grammars. *Proceedings of WAGA'99: Second Workshop on Attribute Grammars and their Applications*, pages 173-184. Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.
- [21] A. C. Shaw. Reasoning about Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, Vol. 15, No. 7, 1989.
- [22] Real-Time Specification for Java™ Experts Group. <http://www.rtg.org>
- [23] Sun Microsystems. Jini™ Connection Technology. <http://www.sun.com/jini/>
- [24] V. Sundaresan, C. Razafimahefa, R. Vallée-Rai, and L. Hendren. Practical Virtual Method Call Resolution for Java. Sable Technical Report No. 1998-7, McGill University, Canada, 1998.

- [25] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing Analysis for Data Caches and Set-Associative Caches. *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, June 1997.