Compiling Java for Real-Time Systems

Compiling Java for Real-Time Systems

Anders Nilsson

Licentiate Thesis, 2004

Department of Computer Science
Lund Institute of Technology
Lund University

In dwelling, be close to the land.
In meditation, delve deep into the heart.
In dealing with others, be gentle and kind.
In speech, be true.
In work, be competent.
In action, be careful of your timing.
– Lao Tsu

ISSN 1652-4691
Licentiate Thesis 1, 2004
LU-CS-LIC:2004-1

Thesis submitted for partial fulfillment of
the degree of licentiate.

Department of Computer Science
Lund Institute of Technology
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: anders.nilsson@cs.lth.se
WWW: http://www.cs.lth.se/~andersn

Typeset using $\LaTeX 2_\varepsilon$

Printed in Sweden by Media-Tryck, Lund, 2004

© 2004 by Anders Nilsson

Abstract

Our everyday appliances ranging from toys to vehicles, as well as the equipment used to manufacture them, contain an increasing number of embedded computers. Embedded software often implement functionality that is crucial for the operation of the device, resulting in a variety of timing requirements and resource utilization constraints to be fulfilled. Industrial competition and the ever increasing performance/cost ratio for embedded computers lead to an almost exponential growth of the software complexity, raising an increasing need for better programming languages and run-time platforms than is used today.

Key concepts, such as portability, scalability, and real-time performance, have been defined, which need to be fulfilled for Java to be a viable programming language for hard real-time systems. In order to fulfill these key concepts, natively compiling Java using a revised memory management technique is proposed. We have implemented a compiler and run-time system for Java, using and evaluating new object-oriented compiler construction research tools, which enables a new way of implementing optimizations and other code transformations as a set of transforms on an abstract syntax tree.

To our knowledge, this is the first implementation of natively compiled real-time Java, which handles hard real-time requirements. The new transparent garbage collector interface makes it possible to generate, or write, C code independently of garbage collector algorithm. There is also an implementation of the Java exception mechanism that can be used in conjunction with an incremental real-time garbage collector. Experiments show that we achieve good results on real-time performance, but that some work is needed to get general execution performance comparable to C++. Given our contributions and results, we do see compiled real-time Java, or a similar language such as C#, as industrially viable in a near future.

Acknowledgements

The research presented in this thesis was carried out within the *Software Development Environments* group at the Department of Computer Science, Lund University. This would never have existed had it not been for my supervisors; Professor Boris Magnusson who is the head of the research group, Klas Nilsson who introduced me to real-time Java and has given me invaluable advice and feedback on various real-time issues, and Görel Hedin who introduced me to compiler construction and reference attributed grammars. Thank you!

The Java to C compiler would never have come as far as it has, without all the contributions from people at the department. Special thanks to Torbjörn Ekman for his work on JastAdd, the Java parser, and the compiler front-end, Sven Gestegård-Robertz, Roger Henriksson, Anders Ive, and Anders Blomdell (@control.lth.se), for their work on real-time garbage collection, the garbage collector interface, and various parts of the run-time libraries. Thank you!

I am also grateful to all those students who, in their respective Master’s thesis projects, have contributed implementations in various parts of the compiler and run-time system, as well as pinpointed a lot of bugs which could then be fixed. Francisco Menjíbar, Robert Alm & Henrik Henriksson, Daniel Lindén, Patrycja Grudziecka and Daniel Nyberg, and Lorenzo Bigagli, thank you!

Many thanks also to the rest of you at the department. It has been a pleasure working with you.

Last, but definitely not the least, I am infinitely grateful to Christina and Amanda for their love and support.

The work presented in the thesis has been financially supported by VINNOVA, the Swedish Agency for Innovation Systems.

Contents

1 Introduction **1**
1.1 Real-Time Programming 2
1.2 Compiler Construction 7
1.3 Problem Statement 8
1.4 Thesis Outline 9

2 Preliminaries **11**
2.1 Distributed Embedded Real-Time Systems 11
2.1.1 Portability 12
2.1.2 Scalability 12
2.1.3 Hard Real-Time Execution and Performance . . . 13
2.1.4 Hard Real-Time Communication 13
2.1.5 Applicability 13
2.2 Real-Time Memory Management 14
2.3 Real-Time Operating Systems 16
2.3.1 RTAI 16
2.4 Object-Oriented development 17
2.4.1 Aspect-Oriented Programming 17
2.5 Reference Attributed Grammars 18

3 An Approach to Real-Time Java **19**
3.1 Approach 19
3.2 Simple Example 20
3.3 Memory Management 20
3.4 External Code 22
3.5 Predictability 24
3.5.1 Dynamic Class Loading 24
3.5.2 Latency and Preemption 25
3.6 Findings 28

4 Real-Time Execution Platform **29**
4.1 Garbage Collector Interface 29
4.1.1 User Layer 30
4.1.2 Thread Layer 31
4.1.3 Debug Layer 31
4.1.4 Implementation Layer 31
4.2 Class Library 32
4.2.1 Native Methods 32
4.2.2 I/O 34
4.3 Threads and Synchronization 35
4.3.1 Real-Time Thread Classes 37
4.3.2 Synchronization 38
4.4 Exceptions 40
4.4.1 Exceptions in Compiled Java 42
4.5 Findings 45

5 A Compiler for Real-Time Java **47**
5.1 JastAdd 48
5.2 Architecture and Overview 48
5.3 Simplification Transformations 53
5.4 Optimization Transformations 59
5.4.1 Dead Code Elimination 59
5.5 Code Generation 60
5.6 Evaluation 61

6 Experimental Verification **65**
6.1 Portability 65
6.2 Scalability 66
6.2.1 Low-End Experiment Platform 66
6.3 Hard Real-Time Execution and Performance 70
6.3.1 Hard Real-Time Execution 70
6.3.2 Performance 71
6.4 Hard Real-Time Communication 76
6.5 Applicability 76

7 Future Work **79**
7.1 Optimizations 79
7.1.1 More Efficient GC Locking Scheme 79
7.1.2 Memory Allocation 80
7.1.3 OO optimizations 81
7.1.4 Selective Inlining 81

7.2 Networking 81
7.3 Dynamic Class Loading 82
7.4 Code Analysis 82
7.5 Hybrid Execution Environment 82

8 Related Work 83
8.1 Real-Time Java Specifications 83
8.2 OOVM 85
8.3 Jepes 85
8.4 JamaicaVM 86
8.5 PERC 86
8.6 SimpleRTJ 87
8.7 GCJ 87

9 Conclusions 89
9.1 Real-Time Java 89
9.2 Compiler Construction 91
9.3 Contributions 91
9.4 Concluding Remarks 92

Bibliography 94

A Acronyms 101

B Java Grammar 103

List of Figures

4.1 The four macro layers of the GCI. 30
4.2 The `System.out.print(String)` call chain. 35
4.3 Linking compiled Java with appropriate run-time. 37

5.1 Overview of the Java compiler architecture. 49
5.2 Node class relations in simple JastAdd example. 50
5.3 AST representation of a complex Java expression. 51
5.4 Java code fragment and corresponding AST. 53
5.5 Simplifying names by means of an AST transformation. . 54
5.6 Simplifying a complex method call. 55
5.7 Subtree representing a for-statement. 58
5.8 Subtree representing a simplified for-statement. 58
5.9 Flowchart of compilation process. 62

6.1 Alarm-clock application running on the AVR platform. . 69
6.2 Latencies and response times for three periodic threads. . 73
6.3 Latencies and response times for three periodic threads. . 74

List of Listings

3.1 A small example Java class. 20
3.2 Simple Java method translated into C. 21
3.3 GC handling added to the small Java example class. . . . 23
3.4 Example of using preemption points. 26
3.5 Explicit preemption points may decrease GC overhead. . 27
4.1 Call a legacy function from compiled Java. 33
4.2 Mapping Java monitors on underlying OS. 39
4.3 Example of Java synchronization with compiled code. . . 40
4.4 A simple exception example. 41
4.5 C macros implementing exceptions. 42
4.6 Exception example using exception macros. 44
5.1 JastAdd abstract grammar definition example. 50
5.2 Type checking implemented using semantic equations. . 52
5.3 Pretty-printer implemented using Java aspects in JastAdd. 52
5.4 Simplification transformation example. 55

List of Tables

5.1 Code sizes after dead code elimination. 60
5.2 Source code sizes for the different stages of our compiler. 63
5.3 Java compiler measurements 63

6.1 Implementation of real-time Java runtime environment. . 66
6.2 Measured performance of real-time kernel. 67
6.3 Memory usage for the alarm-clock on the AVR platform. 69
6.4 Timing characteristics of three threads. 70
6.5 Real-time performance statistics. 72
6.6 Performance measurements. 75

640 K ought to be enough for anybody.

Bill Gates, 1981

Chapter 1

Introduction

MAYBE contrary to common belief, the vast majority of computers in the world are embedded in different types of systems. A quick estimate gives at hand that general purpose computers—e.g. desktop machines, file- and database servers—make up less than ten percent of the total, while embedded computers comprise the remaining part. And the numbers are constantly increasing, as small computers are embedded in our everyday appliances, such as TV sets, refrigerators, laundry machines—not to mention cars where computers or embedded processors can sometimes be counted in dozens.

A number of observations can be made regarding software development for embedded systems:

- Object-Oriented (OO) techniques have proved beneficial in other software areas, while development of embedded software is done mostly using low-level programming languages (assembler and C), resulting in extensive engineering needed for development and debugging. Software modules do not become flexible from a reuse point of view since they are hand-crafted for a certain type of application or target system.

- As embedded systems become parts of larger systems that require more and more flexibility, and where parts of the software can be installed or upgraded dynamically, flexibility with respect to composability and reconfiguration will require some kind of safe approach since traditional low-level implementation techniques are too fragile (both the application and the run-time system can crash).

- Embedded systems becomes more and more distributed, consisting of small communicating nodes instead of large centralized ones. It would be very beneficial to make use of available Internet technologies, but with the extension that both computing and communication must enable strict timing guarantees.

Another observation on application development in general, is that programming languages and supporting run-time systems play a central role, not only for the development time, but also for the robustness of the application. These observations all point in the direction that the benefits and properties of Java (further described below) could be very valuable for embedded systems programming.

The languages and tools used for embedded systems engineering need to be portable and easy to tailor for specific application demands. Adapting programming languages, or generation of code, to new environments or to specific application needs (so called domain specific restrictions or extensions) typically requires modifications of, or development of, compilers. However, the construction (or modification) of a compiler for a modern OO language is both tedious and error-prone. Nevertheless, correctness is as important as for the generated embedded software, so for flexible real-time systems the principles of compiler construction deserves special attention.

Thus, both the so call system programming (including implementation language and run-time support) and the development support (including compiler techniques and application specific enhancements) are of primary concern here. A further introduction to these areas now follows, to prepare for the problem statement and thesis outline that conclude this chapter.

1.1 Real-Time Programming

Two of the largest technical problem areas that plague many software development projects are:

Managing System Complexity Given the industrial competition and increasingly challenging application requirements, software systems tend to grow larger and more complex. This takes place at approximately the same rate as CPU performance increases and memory prices decreases, resulting in complexity being the main obstacle for further development. Weak structuring mechanisms in the programming languages used makes the situation worse.

Managing System Development Software development projects are often behind schedule. Software errors found late in the project makes the situation worse since the time needed to correct software errors found late is approximately exponentially related to the point of time in the project when the error was found [Boe81]. Many late hard-to-find programming errors originate from the use of unsafe programming languages, resulting in problems such as memory leaks and dangling pointers.

So, what is the role of programming languages here? A good programming language should help the developer avoid the problems listed above by providing:

- Error avoidance at build time. Programming errors should, if possible, be found at compile time, or when linking or loading the system.
- Error detection at run-time. Programming errors not found at build time should be detected as early as possible in the development process to avoid excessive costs. For instance, run-time errors should, if possible, be explicitly detected and reported when they occur, and not remain in the system making it potentially unstable.

Compared to other software areas, such as desktop computing, development of embedded systems suffer even more from these problems. Errors in embedded software are typically harder to find due to timing demands, special hardware, less powerful debugging facilities, and they are during operation often not connected to any software upgrading facilities. Nevertheless, embedded software projects tend to use weaker programming languages, that is, C has taken over as the language of choice from assembly languages, but the assumption still is that programmers do things right. Since that is clearly not the reality, there is a great need for introducing safe programming languages with better structuring and error detection mechanisms for use in embedded software development.

Object-Oriented Programming Languages

According to industrial practices and experiences, object-oriented programming techniques provide better structuring mechanisms than is found in other paradigms (such as functional languages or the common imperative programming languages). The mechanisms supporting development of complex software systems include:

Classes The class concept provides abstract data structures and methods to operate on them.

Inheritance Classes can be organized, and functionality extended, in a structured manner.

Virtual Operations Method call sites are resolved by parameter type, instead of by name. Method implementations can be replaced by sub-classing.

Patterns Organize interaction between classes.

These concepts can be achieved by conventions, tools, macros, libraries, and the like in a simpler language. Without the built-in support from a true object-oriented language, however, there is a obvious risk that productivity and robustness (with respect to variations in programming skill and style) is hampered. Hence, we need full object-oriented support from the language used.

Implications of Unsafe Programming Languages

Experiences from programming in industry and academia (undergraduate course projects) show that most hard-to-find errors stem from the use of unsafe language constructs such as:

- Type casts, as defined in for example C/C++.
- Pointer arithmetics.
- Arrays with no boundary checks, sometimes resulting in uncontrolled memory access.
- Manual memory management (malloc/free). When to do free? Too early results in dangling pointers, and too late may result in memory leaks.

The first three unsafe constructs usually show up early in the development process. Errors related to manual memory management, on the other hand, does not often show up until very late, sometimes only after (very) long execution times. Because of this time aspect, the origins of these errors can also be *very* hard to locate in the source code. Hence, unsafe language constructs should not be permitted.

Safe Programming Languages

A safe programming language is a language that does not have any of the listed unsafe language constructs. Instead, a safe language is characterized by the fact that *all possible results of the execution are expressed by the source code of the program*. Of course, there can still be programming errors, but they lead to an error message (reported exception), or to bad output as expressed in the program. In particular, an error does not lead to uncontrollable execution such as a "blue screen". If, despite a safe language, uncontrolled execution would occur (which should be very rare), that implicates an error in the platform; not in the application program. Clearly, a safe programming language is highly desirable for embedded systems. Necessary properties of a safe language include:

- Strict type safety. For example, it may not possible to cast between arbitrary types via a type cast to **`void*`** as in C/C++.
- Many programmer errors caught by the compiler. Remaining (semantic) errors that would violate safety are caught by runtime checks, e.g., array bounds and reference validity checks.
- Automatic memory management. All heap memory blocks are allocated when object are created (by calling the operator **`new`**) and automatically freed by a garbage collector when there no longer exists any live references to the object. An object cannot be freed manually.
- From the items above it follows that direct memory references are not allowed.

The characteristics of safe languages usually makes it impossible to directly manipulate hardware in such a language, as safety can not be guaranteed if direct memory references are allowed[1]. Therefore, unsafe languages are still needed for developing device drivers, but the amount of code written in such languages should be kept as small and as isolated as possible. One solution then is to wrote device drivers in C and the application code in Java. There has also been interesting work done trying to raise the abstraction level of hardware drivers using domain specific languages [MRC^{+}00], which can be used to minimize the amount of hand-written "dangerous" code in an application.

[1]A direct memory reference can be unintentionally, or intentionally, changed to reference application data instead of memory-mapped hardware. As a result, type integrity and data consistency are no longer guaranteed, with a potential risk of ending up with dangling pointers and/or memory leaks..

Java

As of today, Java is the only safe, object-oriented programming language available that has reached industrial acceptance. Not just for these previously mentioned qualities, but also for its platform independence[2].

The benefits of security are often referred to as the "sand-box model" , which is a core part of both the Java language and the run-time system in terms of the JVM. The term sand-box refers to the fact that objects cannot refer to data outside its scope of dynamic data, so activities in one sand-box cannot harm others that play elsewhere. This is particularly important in flexible automation systems where configuration at the user's site is likely to exhibit new (and thereby untested) combinations of objects for system functions, which then may not deteriorate other (unrelated) parts of the system. Hence, raw memory access and the like should not be permitted within the application code, and the enhancements for real-time programming should be Java compatible and without violating security.

There exists other programming languages, and run-time systems, which fulfill the technical requirements for a safe language. The most well known Java alternative today is the Microsoft .net environment and the language C#, which is safe except where the keyword `unsafe` is used. In principle one could argue that lack of security is built into that language/platform, but in practice the results of this thesis would be useful for the purpose of creating a 'RT.net' (dot-net for real time) platform. However, due to maturity, availability of source code, simplicity, and cross-platform portability, Java is the natural basis for research in this area.

Considering the rich variety of processors and operating systems used for embedded systems, confronted with the licensing conditions from both Sun and Microsoft, there are also legal arguments for avoiding their standard (desk-top or server oriented) run-time implementations. Luckily, the language definitions are free, and free implementations of run-time systems and libraries are being developed. In the Java case, the availability and maturity of a free GNU implementation of the Java class library [gcj] solves this problem for the Java case. However, standard Java and the GNU libraries are not targeted or suitable for real-time embedded systems, which brings us to the compiler technology issue.

[2]Or rather, its good platform portability, since it takes a platform dependent Java Runtime Environment (JRE), and JREs are not quite fully equivalent on all supported platforms.

1.2 Compiler Construction

Adapting the Java programming language and runtime to meet the requirements for hard real-time execution will inevitably involve the construction of various libraries and tools, including the Java compiler.

Constructing a compiler for a modern OO language, such as Java, using standard compiler construction tools is normally a large, tedious, and error prone task. Since the correctness of the generated code depends on the correctness of the compiler and other tools, it is preferable to have also the tools (except for core well-tested software such as a standard C compiler) implemented in a safe language. Furthermore, focusing on design and build time rather than run time, is is desirable to have a representation of the language and application software that is convenient to analyze and manipulate. Therefore, applicability of real-time embedded Java appears to go hand in hand with suitable compiler constructions tools, preferably written in Java for portable and safe embedded systems engineering.

Work on compiler construction within our research group has resulted in new ideas and new compiler construction tools [HM02], which with the aim of this work represent state of the art. The representation of the language within that tool is based on Attribute Grammars (Attribute Grammar (AG)s). AG-based research tools have been available for a long time, but there are no known compiler implementations for a complete object-oriented language, so this topic also by itself forms a research issue.

Optimizations and Code Generation

Compiling code for execution on very resource-limited platforms consequently involves code optimizations. While many optimizations are best performed on intermediate- or machine code, there are—especially for OO languages—a number of high level optimizations which can only be performed on a higher abstraction level. Examples on such transformations are in-lining and implicit finalization of classes or methods.

With the aim of providing as high level of portability as possible, "Write Once, Run Everywhere" in Java terminology, the code generation phase of a compiler is very important. Should the output be processor-specific assembly language, or would the use of a higher abstraction level intermediate language suit the needs better? Can a standard threading Application Programming Interface (API) such as Posix

[NBPF96] be utilized, and/or what refinements are necessary? Can the code representation and transformation be structured in such a way that tailoring the generated code to specific underlying kernels and hardware configurations can be made simpler and more modular than feasible with currently available techniques?

1.3 Problem Statement

With the aim of promoting flexibility, portability, and safety for distributed hard real-time systems we want to utilize the Java benefits. But, in order to enable practical/efficient widespread use of Java in the embedded systems world there are a number of technical issues that need to be investigated. We then need to identify current limitations and find new techniques to advance beyond these limitations, but also inherent limitations and necessary trade-offs need to be identified and made explicit. In general terms, the topic of this thesis can be stated by posing the following questions:

> *Can standard Java be used as a programming language on arbitrary hardware platforms with varying degrees of real-time-, memory footprint-, and performance demands?*

Here, standard Java means the complete Java language according to Sun's J2SE, and (a possibly enhanced subset of) the standard Java libraries that are fully compliant with J2SE.
If standard Java is useful for embedded systems,

> *what enhancements in terms of new software techniques are needed to enable hard real-time execution and communication, and what are the inherent limitations?*
>
> *If possible, which tools are needed for adapting standard Java to various types of embedded systems? What techniques enable efficient development of those tools, and what limitations can be identified?*

In short, based on the well-known standard Java claim, what we want to accomplish is

> *write once run anywhere, for severely resource-constrained real-time systems*

and find out the resource-related limitations[3].

[3]Note that Sun's J2ME is neither J2SE-compliant nor suitable for (hard) real-time systems as is further discussed in Chapters 3 and 4.

1.4 Thesis Outline

The rest of this thesis is organized as follows:

Chapter 2 presents short introductions to some of the techniques used, as well as some identified important aspects concerning embedded real-time systems development .

Chapter 3 presents a discussion on how to compile Java for usage in real-time systems, possibly with limited resources. This chapter is largely based on the paper *Real Java for Real Time – Gain and Pain* [NEN02], presented at CASES'03 in Grenoble, France.

Chapter 4 presents run-time issues for real-time Java; real-time memory management, the Java standard class library, threads and synchronization, and exceptions.

Chapter 5 gives a description of the Java compiler being developed to accomplish real-time Java.

Chapter 6 presents the experiments performed to see to what extent the ideas are applicable in reality.

Chapter 7 contains the most interesting ideas for further work on the real-time Java implementation and the Java compiler.

Chapter 8 gives short descriptions of some related work.

Chapter 9 presents the conclusions drawn from the work presented in the thesis. A summary of the thesis contributions is also given.

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C. Clarke

Chapter 2

Preliminaries

ADVANCES within three computer science research areas lay the foundation of this work, with the objective to make a modern object-oriented language available for developing hard real-time systems. *(Distributed) Real-Time Systems* is the primary domain for this work, while advances in *Object-Orientation* and *Attribute Grammars* have made possible the construction of the tools used.

2.1 Distributed Embedded Real-Time Systems

Real-time systems can be defined as systems where the correctness of the system is not strictly an issue of semantic correctness, i.e., given a set of inputs, the system will respond with the intended output, but there is also the issue of temporal correctness, i.e., the system must respond with an output within a certain time frame from acquiring the inputs. This time-frame is referred to as the deadline, within which the systems must respond.

One usually makes a distinction between *soft* and *hard* real-time systems, depending on the influence a missed deadline might have on the system behavior. A missed deadline in a soft real-time system results in degraded performance, but the system stability is not affected, e.g., video stream decoding. A missed deadline in a hard real-time system, on the other hand, jeopardizes the overall system stability, e.g., flight control loop in an unstable airplane (such as For example the SAAB JAS39 military aircraft)..

Despite the principal advantages of a safe object-oriented programming language, numerous problems arise when one tries to use the Java

language – and its execution model – for developing real-time systems. More problems arise if one has to consider resource-limited target environments, i.e., small embedded systems with hard real-time constraints such as mobile phones or industrial process control applications.

In the sequel of this section, a number of identified key concepts for being able to use Java in embedded real-time environments are listed. These key concepts are then used to formulate the problem statement for the thesis.

2.1.1 Portability

Portability is important when deciding on the programming language to use for embedded systems development. It might not be clear from the beginning which type of hardware and Real-Time Operating System (RTOS) should be used in the final product. Good portability also makes it much easier to simulate system behavior on platforms better suited for testing and debugging, e.g., workstations. A key concept for retaining as much portability as possible in using Java for embedded and/or real-time systems is:

Standard Java: If possible, real-time programming in Java should be supported without extending or changing the Java language or API. For instance, the special and complex memory management introduced in the Real-Time Specification for Java (RTSJ) specification [BBD$^+$00] needs to be abandoned to maintain the superior portability of standard Java, as needed within industrial automation and other fields.

2.1.2 Scalability

Scalability (both up and down) is also important to consider since non-scalable techniques usually do not survive in the long term. How far towards low-end hardware is it possible to go without degrading feasibility on more powerful platforms?

Since Java has proved to be quite scalable for large systems, the key issue for scalability in this work is:

Memory Footprint: For most embedded devices, especially mass-produced devices, memory is an expensive resource. A tradeoff has to be made between cost of physical memory and cost savings from application development in higher level languages.

2.1.3 Hard Real-Time Execution and Performance

Regarding feasibility for applications with real-time demands, there are a number of issues deserving attention:

Performance: CPU performance, and in some cases power consumption, is also an limited resource. The cheapest CPU that will do the job generates the most profit for the manufacturer. The same tradeoff as for memory footprint has to be made.

Determinism: Many embedded devices have real-time constraints, and for some applications, such as feedback controllers, there might be hard real-time constraints. Computing in Java needs to be as time predictive as current industrial practice, that is, as predictive as when programming in C/C++.

Latency: For an embedded controller, it might be equally important that the task latency, i.e. the time elapsed between the event that triggers a task for execution and when the task actually produces an output, is sufficiently short and does not vary too much (sampling jitter). Jitter in the timing of a control task usually results in decreased control performance and, depending on the controlled process characteristics, may lead to instability.

2.1.4 Hard Real-Time Communication

Embedded real-time systems tend to be more and more distributed. For example, a factory automation system consists of a large number of small intelligent nodes, each running one or a few control loops, communicating with each other and/or with a central server. The central server collects logging data from the nodes and sends new calibration values, and possibly also software updates, to the nodes.

In some cases, it is appropriate to distribute a single control loop over a number of distributed nodes in a network. This places high demands on the timing predictability of the whole system. Not only must each node satisfy real-time demands, the interconnecting network must also be predictable and satisfy strict demands on latency.

2.1.5 Applicability

The applicability of a proposed solution can be defined as the feasibility of using the proposed solution in a particular application. With an application domain including systems ranging from small intelligent

control nodes to complex model-based controllers, such as those found in industrial robots, especially one issue stands out as more important:

External Code: The Java application, with its run-time system, does not alone comprise an embedded system. There also have to be hardware drivers, and frequently also library functions and/or generated code from high-level tools. Examples of such tools generating C-code are the Real-Time Workshop composing Matlab/Simulink blocks [Mat], generation of real-time code from declarative descriptions such as Modelica [Mod] (object-oriented DAE) models, or computations generated from symbolic tools such as Maple [Map]. Note that assuming these tools (resembling compilers from high-level restricted descriptions) are correct, programming still fulfills the safety requirement.

2.2 Real-Time Memory Management

Automatic memory management has been well-known ever since the appearance of function-oriented and object-oriented languages with dynamic memory allocation, such as Lisp [MC60] and Simula [DMN68, DN76] in the 1960's. However, most garbage collection algorithms are not suitable for use in systems with predictable timing demands. This is caused by the unpredictable latencies imposed on other threads when the garbage collector runs.

Two slightly different Garbage Collect(ion | or) (GC) algorithms are used in the work described in this thesis; *Mark-Compact* and *Mark-Sweep*. Both algorithms work in two passes, starting with the *Mark* pass where all live memory blocks are marked. Then follows the *Compact* or *Sweep* pass, depending on which algorithm is used, where unused memory is reclaimed and is available for future allocations. In our implementations, both algorithms depend on the application maintaining a list of references to heap-allocated objects, a *root stack*. The root stack is used by the GC algorithm as the starting point for scanning the live object graph in the marking phase.

During the *Compact* phase in a Mark-Compact GC, all objects which were marked as live during the marking phase are moved to form a contiguous block of live objects in the heap. After the compact phase has finished, the heap consists of one contiguous area of live objects, and one contiguous area of available free memory. The *Sweep* phase in a Mark-sweep algorithm, on the other hand, does not result in live objects being moved around in the heap. Instead, memory blocks which are no

longer used by any live objects are reclaimed by the memory allocator, similar to the `free()` call in a standard C environment.

The GC can be run in two ways. The simplest way of running the GC algorithm is the batch, or stop-the-world. When the memory management system determines it is time to reclaim unused memory, the application is stopped and the GC is allowed to run through a full cycle of *Mark* and *Compact* or *Sweep*. When the GC has finished its cycle, the application is allowed to continue. Naturally, this type of GC deployment is utterly inadequate for use in hard real-time systems since the time needed for performing a full GC cycle varies greatly, and the worst case is typically much larger than the maximum acceptable delay in the application.

In order to lessen the delay impact of the GC on the application, the deployment of the GC can be made incremental instead, in which case the GC may give up execution after each increment if the application wants to run.

In 1998, Henriksson [Hen98] showed that by analyzing the application, it is possible to schedule an incremental mark-compact garbage collector in such a way that the execution of high priority threads is not disturbed. This is accomplished by freeing high priority threads from doing any GC work during object allocation, while having a medium priority GC thread performing that GC work and letting low priority threads perform a suitable amount of GC work during allocations. The GC increments are then chosen sufficiently small so as not to introduce too much worst-case latency to high priority threads.

The analysis needed for computing GC parameters, so it can be guaranteed that the application will never run out of memory when a high priority thread tries to allocate an object, is rather complex and cumbersome. The complexity is equal to calculating Worst-Case Execution Time (WCET) for all threads in the application. In 2003, Gestegård-Robertz and Henriksson [GRH03] presented some ideas and preliminary results on how scheduling of a hard real-time GC can be achieved by using adaptive and feedback scheduling techniques. Taking that work into account, it appears reasonable to accomplish real-time Java without compromising the memory allocation model, in contrast with what is done in for example the two real-time Java specifications [BBD+00, Con00].

2.3 Real-Time Operating Systems

Real-Time Operating Systems (RTOSs) differs from more general purpose desktop- and server operating systems, such as Windows, Solaris or GNU/Linux, in a number of ways, relating to the different purpose of the Operating System (OS). Whereas a main purpose of a general purpose OS is to make sure that no running process is starved, i.e., no matter the system load, all processes must be given some portion of CPU time so they can finish their work, RTOSs functions in a fundamentally different way. RTOSs are generally strict priority based. A thread may never be interrupted by a lower priority thread, and a thread is always interrupted if a higher priority thread enters the scheduler ready queue.

Despite this difference in process scheduling between general purpose OSs and RTOSs, a lot of work has been done trying to combine th strengths of both types, since general purpose OSs usually have better support for application development.

2.3.1 RTAI

The Real-Time Application Interface for Linux (RTAI) project [Me04], which originated as an open-source fork of the RT-Linux project [FSM04], aims at adding hard real-time support to the GNU/Linux operating system. Real-Time Application Interface for Linux (RTAI) manages to achieve hard real-time in the otherwise general purpose GNU/Linux OS, by utilizing the modularity of the Linux kernel. By applying a patch to the Linux kernel, the RTAI kernel module is able to hook into the kernel as a Hardware Abstraction Layer (HAL) intercepting the kernel's communication with the hardware. This means that all hardware interrupts have to pass through the RTAI module before being communicated to the Linux kernel, and the effect is a two-layered scheduler with the Linux Kernel running as the idle task in the RTAI scheduler.

RTAI threads are scheduled by the strict priority based RTAI scheduler, and as they are not disturbed by Linux processes, and therefore very good timing predictability can be achieved. A side effect is, obviously, that RTAI threads may starve the Linux kernel, loosing responsiveness to user interaction and resulting in a locked-up computer, but that is no different from any other RTOS.

2.4 Object-Oriented development

OO languages have over the years proven to be a valuable programming technique ever since the first object-oriented language, Simula [DMN68, DN76]. Since then, many object-oriented languages have been constructed, of which C++ [Str00], Java [GJS96], and C# [HWG03] are the best known today.

The object-oriented technology has, however, had very little success when it comes to developing software for small embedded and/or real-time systems. The widespread apprehension that OO languages introduce too much execution overhead is probably the main reason for this. If this apprehension could be contradicted, there would probably be much to gain in terms of development time and software quality if OO technology finds its way into development of these kinds of systems. Many groups, both inside and outside academia, are working on adapting OO technology and programming languages for use in small embedded systems. Most groups work with Java, for example [Ive03, SBCK03, RTJ, VSWH02, Sun00a], but there are also interesting work being done using other OO languages, such as the OOVM [Bak03] using Smalltalk.

2.4.1 Aspect-Oriented Programming

In 1997, Kiczales et al. published a paper [KLM+97] describing Aspect-Oriented Programming (AOP) as an answer to many programming problems, which do not fit well in the existing programming paradigms. The authors have found that certain design decisions are difficult to capture—in a clean way—in code because they cross-cut the the basic functionality of the system. As a simple example, one can imagine an image manipulation application in which the developer wants to add conditional debugging print-outs just before every call to a certain library matrix function. Finding all calls is tedious and error-prone, not to mention the task of, at a possible later time, removing all debug print-outs again. These print-outs can be seen as an *aspect* on the application, which is cross-cutting the basic functionality of the image manipulation application.

By introducing the concept of programming in *aspects*, which are woven into the basic application code at compile-time, two good things are achieved; the basic application code is kept free from disturbing add-ons (conditional debugging messages in the example above), and,

the aspects themselves can be kept in containers of their own with good overview by the developers of the system.

The tool *aspectj* [KHH$^+$01] was released in 2001 to enable Aspect-Oriented Programming (AOP) in Java. There is also a web site for the annual aspect oriented software development conference[1] where links to useful information and tools regarding AOP are collected.

2.5 Reference Attributed Grammars

Ever since Donald Knuth published the first paper [Knu68] on Attribute Grammar (AG) in 1968, the concept has been widely used in research for specifying static semantic characteristics of formal (context-free) languages. The AG concept has though never caught on for use in production code compilers.

By utilizing Reference Attribute Grammars (RAGs) [Hed99], it is also possible to specify in a declarative way the static semantic characteristics of object-oriented languages with many non-local grammar production dependencies.

The compiler construction toolkit, *JastAdd*, which we are using for developing a Java compiler, further described in Chapter 5, is based on the Reference Attribute Grammar (RAG) concept.

[1] `http://aosd.net`

I have yet to see any problem, however complicated, which, when you looked at it in the right way, did not become still more complicated.

Poul Anderson

Chapter 3

An Approach to Real-Time Java

WITH the objective to use Java in embedded real-time systems, one can quickly see that standard Java as defined by Java2 Standard Edition (J2SE) or Java2 Micro Edition (J2ME), including their run-time systems as defined by the Java Virtual Machine (JVM), is not very well suited for these kinds of systems.

This chapter will discuss the suitability of different execution strategies for Java applications in real-time environments. Then, specific details on the chosen strategy, in order to obtain predictability in various situations, will be discussed.

3.1 Approach

Given a program, written in Java, there are basically two different alternatives for how to execute that program on the target platform. The first alternative is to compile the Java source code to byte code, and then have a—possibly very specialized—JVM to execute the byte code representation. This is the standard interpreted solution used today for Internet programming, where the target computer type is not known at compile time. The second alternative is to compile the Java source code, or byte code, to native machine code for the intended target platform linking the object files with a run-time system.

A survey of available JVMs, more or less aimed at the embedded and real-time market, reveals two major problems with the interpreted

Listing 3.1: *A small example Java class.*

```
class AClass {
  Object aMethod(int arg1, Object arg2) {
    int locVar1;
    Object locVar2;
    Object locVar3 = new Object();

    locVar2 = arg2.someMethod();

    return locVar2;
  }
}
```

solution, see also Chapter 7 on page 79 for the survey. JVMs are in general too big, in terms of memory footprint, and they are too slow, in terms of performance. A better approach is to use the conventional execution model, with a binary compiled for a specific CPU, and, if one wants to use a JVM, it can be used as a special loadable module.

One thing in common for almost all CPUs, is that there exists a C compiler with an appropriate back-end. In the interest of maintaining good portability, using C as an intermediate language seems like a good idea. In the sequel, C is used as a portable (high level) assembly language and as the output from a Java compiler.

3.2 Simple Example

Consider the Java class in Listing 3.1, showing a method that takes two arguments (one of them a reference), has two local variables, and makes a call to some other method before it returns. , Compiling this class into equivalent C code yields something like what is shown in Listing 3.2 on the facing page. Note that the referred structures that implement the actual object modeling are left out.

The code shown in Listing 3.2 on the next page will execute correctly in a sequential system. However, garbage collection, concurrency and timing considerations will complicate the picture.

3.3 Memory Management

The presence, or absence, of automatic garbage collection in hard real-time systems has been debated for some years. Both standards pro-

Listing 3.2: *The method of the previous small Java example class translated to C, neglecting preemption issues.*

```
ObjectInstance* AClass_Object_aMethod(
                       AClassInstance* this,
                       JInt arg1,
                       ObjectInstance* arg2) {
  JInt locVar1;
  ObjectInstance* locVar2;
  ObjectInstance* locVar3;

  // Call the constructor
  locVar3 = newObject();

  // Lookup and call virtual method in vTable
  locVar2 = arg2->class->methodTbl.someMethod();

  return locVar2;
}
```

posals for real-time Java [BBD+00, Con00] assume that real-time GC is impossible, or at least not feasible to implement efficiently. Therefore they propose a mesh of memory types instead, effectively leaving memory management into the hands of the application programmer. Some researchers, on the other hand, work on proving that real-time GC actually is possible to accomplish in a useful way.

Henriksson [Hen98] has shown that, given the maximum amount of live memory and the memory allocation rate, it is possible to schedule an incremental compacting GC in such a way that we have a low upper bound on task latency for high priority tasks.

Siebert [Sie99] chooses another strategy and has shown that, given that the heap is partitioned into equally sized memory blocks, it is possible to have an upper (though varying depending on the amount of free memory) bound on high priority task latency using an incremental non-moving GC. The varying task latency relates to the amount of free memory in such a way that the task latency increases dramatically in a situation when there is almost no free memory left. In a system where the amount of free memory varies over time, the jitter introduced may hurt control performance greatly.

Example with GC

Using an incremental compacting GC in the run-time system, the C code in Listing 3.2 on the preceding page will not suffice for two reasons. The GC needs to know the possible root nodes, i.e. references outside the heap (on stacks or in registers) peeking into the heap, for knowing where to start the mark phase. Having the GC to find them by itself can be very time-consuming with a very bad upper bound, so better is to supply them explicitly. Potential root nodes are reference arguments to methods and local reference variables. Secondly, since a compacting GC will move objects in the heap, object references will change. Better than searching for them, is to introduce a read barrier (an extra pointer between the reference and the object) and pay the price of one extra pointer dereferencing when accessing an object. The resulting code is shown in Listing 3.3 on the next page.

The `REF(x)` and `DEREF(x)` macros implement the needed read barrier while the `GC_PUSH_ROOT(x)` and `GC_POP_ROOT(n)` macros respectively register a possible root with the GC, and pops the number of roots that was added in this scope.

If using a non-moving GC, on the other hand, references to live objects are never changed by the GC, and the read-barrier is just unnecessary performance penalty. A simple redefinition of the GC macros, as is seen in Listing 3.3, is all that is needed to remove the read-barrier while leaving the application code independent of which type of GC is to be used.

3.4 External Code

Every embedded application needs to communicate with the surrounding environment, via the kernel, hardware device drivers, and maybe with various already written library functions and/or generated code blocks from high level programming tools (such as Matlab/Real-Time Workshop from The MathWorks Inc.). As mentioned, native compilation via C simplifies this interfacing. Sharing references between generated Java code and an external code module, e.g. a function operating on an array of data, has impact on the choice of GC type and how it can be scheduled.

When using a compacting GC, one must make sure that the object in mind is not moved by the GC while referred to from the external code since that code can not be presumed to be aware of read barriers. If the execution of the external function is sufficiently fast, we may consider

Listing 3.3: *GC handling added to the small Java example class.*

```
/* Include type definitions and GC macros.
 * Omitted in following listings
 */
#include <jtypes.h>
#include <gc_macros.h>

#ifdef COMPACT_GC
/* Compacting GC */
#define REF(x)   (x **)
#define DEREF(x) (* x)
#else
/* Non-moving GC */
#define REF(x)   (x *)
#define DEREF(x) (x)
#endif

REF(ObjectInstance) AClass_Object_aMethod(
            REF(AClassInstance) this, JInt arg1,
            REF(ObjectInstance) arg2) {
  JInt locVar1;
  REF(ObjectInstance) locVar2;
  REF(ObjectInstance) locVar3;
  GC_PUSH_ROOT(arg2);
  GC_PUSH_ROOT(locVar2);
  GC_PUSH_ROOT(locVar3);

  locVar3 = Object();

  locVar2 =
      DEREF(arg2)->class->methodTbl.someMethod();

  GC_POP_ROOT(arg2);
  GC_POP_ROOT(locVar2);
  GC_POP_ROOT(locVar3);
  return locVar2;
}
```

it a critical section for memory accesses and disable GC preemption during its execution. More on this topic in Section 3.5.2. A seemingly more pleasant alternative would be to mark the object as read-only to the GC during the operation. Marking read-only blocks for arbitrarily long periods of time would however fragment the heap and void the deterministic behavior of the GC.

For non-moving GCs, the situation at first looks a lot better as objects once allocated on the heap never move. However, as a non-moving GC

depends on allocating memory in blocks of constant size to avoid external memory fragmentation in order to be deterministic, objects larger than the given memory block size (e.g. arrays) have to be split over two or more memory blocks. Since we can never guarantee that these memory blocks are allocated contiguously, having external non GC-aware functions operate on such objects (or parts thereof) is impossible.

However, if we do not depend on having really hard timing guarantees, the situation is no worse (nor better) than with plain C using `malloc()` and `free()`. Memory fragmentation has been argued by Johnstone et al. [JW98] *not* to be a problem in real applications, given a good allocator mechanism. Using a good allocator and a non-moving GC, the natively compiled Java code can be linked to virtually any external code modules. The price to pay is that memory allocations times are no longer strictly deterministic, just like in C/C++.

3.5 Predictability

The ability to predict timing is crucial to real-time systems; an unexpected delay in the execution of an application can jeopardize safety and/or stability of controlled processes.

Predictability and Worst-Case Execution Time (WCET) analysis in general is by now a mature research area, with a number of text books available [BW01], and is not further discussed in this thesis. However, adapting Java for usage in real-time systems requires considerations about dynamic loading of classes, latency, and preemption.

3.5.1 Dynamic Class Loading

In traditional Java, every object allocation (and calls to `static` methods or accesses to static fields) pose a problem concerning determinism, since we can never really know for sure if that specific class has already been loaded, or if it has to be loaded before the allocation (or call) can be performed. In natively compiled and linked Java applications, all referred classes will be loaded before execution starts since they are statically linked with the application. This ensures real-time performance from start. However, there are situations—such as software upgrades on-the-fly—where dynamic class loading is needed.

Application-level class loading does not require real-time loading, but when a class has been fully loaded, it should exhibit real-time behavior just like the statically linked parts of the application. This is related to ordinary dynamic linking, but class loaders provide convenient

object-oriented support. That can, however, be provided also when compiling Java to C, using the native class loading proposed by Nilsson et al. [NBL98]. Using that technique, we can let a dedicated low-priority thread take care of the loading and then instantaneously switch to the cross-compiled binaries for the hard real-time parts of the system. Dynamic code replacement can be carried out in other ways too, but the approach we use maintains the type-safety of the language.

3.5.2 Latency and Preemption

Many real-time systems depend on tasks being able to preempt lower priority tasks to meet their deadlines, e.g. a sporadic task triggered by an external interrupt needs to supply an output within a specified period of time. Allowing a task to be preempted poses some interesting problems when compiling via C, especially in conjunction with a compacting GC. How can it be ensured that a task is not preempted while halfway through an object de-referencing, by the GC? The GC then moves the mentioned object to another location, leaving the first task with an erroneous pointer when it later resumes execution. And what about a "smart" C compiler that finds the read-barrier superfluous and stores direct references in CPU registers to promote performance?

Using the **`volatile`** keyword in C, which in conjunction with preemption points would ensure that all root references exist in memory, is unfortunately not an answer to the latter question since the C semantics does not enforce its use but merely recommends that **`volatile`** references should be read from memory before use. Though many C compilers for embedded systems actually enforce that **`volatile`** should be taken seriously.

One possible solution is to explicitly state all object references as critical sections during which preemption is disallowed, see the example code in Listing 3.4 on the following page.

This can be a valid technique if the enabling/disabling of preemption can be made cheap enough. On the hardware described in Section 6.2 on page 66, for example, it only costs one clock cycle. Using this technology, the only possible way to ensure the read barrier will not be optimized away, is to not allow the C compiler to perform optimizations which rearrange instruction order. It may seem radical but the penalty for not performing aggressive optimizations may be acceptable in some cases. As shown by Arnold et al. [AHR00], the performance increase when performing hard optimizations compared to not opti-

Listing 3.4: *Preemption points implemented by regarding all memory accesses to be critical sections.*

```
REF(ObjectInstance) AClass_Object_aMethod(
            REF(AClassInstance) this, JInt arg1,
            REF(ObjectInstance) arg2) {
  JInt locVar1;
  REF(ObjectInstance) locVar2;
  REF(ObjectInstance) locVar3;
  GC_PUSH_ROOT(arg2);
  GC_PUSH_ROOT(locVar2);
  GC_PUSH_ROOT(locVar3);
  ENABLE_PREEMPT();

  DISABLE_PREEMPT();
  locVar3 = Object();
  ENABLE_PREEMPT();

  DISABLE_PREEMPT();
  locVar2 = DEREF(arg2)->class->methodTbl.someMethod();
  ENABLE_PREEMPT();

  DISABLE_PREEMPT();
  GC_POP_ROOT(3);
  return locVar2;
}
```

mizing at all is in almost all cases less than a factor of 2. Whether this is critical or not, depends on the application.

However, there are still many possibilities to optimize the code. The optimizations that will probably have the greatest impact on performance are mostly high-level, operating on source code (or compiler-internal representations of the source code). They are best performed by the Java to C compiler, which can do whole-program analysis (from an OO perspective), and perform object-oriented optimizations. Some examples which have great impact on performance are:

Class finalization A class which is not declared `final`, but has no subclasses in the application is assumed to be final. Method calls do not have to be performed via a virtual methods table, but can carried out as direct calls.

Class in-lining Small helper classes, preferably only used by one or a few other classes, can be in-lined in their client classes to reduce reference following. The price is larger objects which may be an issue if a compacting GC is used.

Listing 3.5: *Using explicit preemption points may in many cases decrease the GC synchronization overhead.*

```
REF(ObjectInstance) AClass_Object_aMethod(
            REF(AClassInstance) this, JInt arg1,
            REF(ObjectInstance) arg2) {
  JInt locVar1;
  struct {
    REF(AClassInstance) this;
    REF(ObjectInstance) arg2;
    REF(ObjectInstance) locVar2;
    REF(ObjectInstance) locVar3;
  } refStruct;
  refStruct.this = this;
  refStruct.arg2 = arg2;
  GC_PUSH_ROOT(&refStruct,sizeof(refStruct)/sizeof(void*));

  PREEMPT(&refStruct);
  refStruct.locVar3 = Object();

  PREEMPT(&refStruct);
  refStruct.locVar2 =
            refStruct.arg2->class->methodTbl.someMethod();

  GC_POP_ROOT();
  return refStruct.locVar2;
}
```

A more in-depth discussion on optimizations implemented in the Java compiler can be found in Section 5.4, while a more comprehensive listing of object-oriented optimizations can be found in for example [FKR+99].

In the last example, Listing 3.4, we assumed that preemption of a task is generally allowed except at critical regions where preemption is disabled for as short periods of time as possible. If one considers overturning this assumption and instead have preemption generally disabled, except at certain "preemption points" which are sufficiently close to each other in terms of execution time, some of the previous problems can be solved in a nicer way, see Listing 3.5. To ensure that all variable values are written to memory before each preemption point, all local variables (including the arguments of the method) are stored in one local structure, the `struct refStruct`. By taking the address of this struct in each call to the `PREEMPT` macro, the C compiler is forced to write all register allocated values to memory before the call is made. To handle scoped variable declarations, the names are suf-

fixed in order to separate variables in different scopes that can share the same name. Registration of GC roots (with the **GC_PUSH_ROOT**(x,n) macro) is simplified to passing the address of the struct and the number of elements it contains, compared to registering each root individually.

The **PREEMPT**(x) macro checks with the kernel if a preemption should take place. Such preemption point calls are placed before calls to methods and constructors, and inside long loops (even if the loop does not contain a method call). By passing the struct address, we utilize a property of the C semantics which states that if the address of a variable is passed, not only must the value(s) be written to memory before executing the call, but subsequent reads from the variable must be made from memory. Thus we hinder a C compiler from performing (to us) destructive optimizations.

To prevent excessive penalty from the preemption points, a number of optimizations are possible. After performing some analysis on the Java code, we may find that a number of methods are short and final (in the sense that that they make no further method calls), and a preemption point before such method calls may not be needed. Loops where each iteration executes (very) fast, but have a large number of iterations, may be unrolled to lower the preemption point penalty.

Since reference consistency is a much smaller problem with non-moving GCs, the situation is simplified. In fact, no visible changes have to be made to the code in Listing 3.3 on page 23 for maintaining reference integrity, and therefore average performance will be improved. However, when dynamically allocating several object sizes the allocation predictability will be as poor as in C/C++.

3.6 Findings

Inclusion of external (non GC-aware code in a real-time Java system raises a tradeoff between *Latency* and *Predictability*. For hard real-time, a compacting GC should be used, and no object references may be passed to non GC-aware functions. If we need to pass object references to non GC-aware code functions, a compacting GC is not applicable since calls to non GC-aware functions must be considered critical sections, and task latencies can no longer be guaranteed.

Using a good allocator and a non-moving GC, the natively compiled Java code can be linked to virtually any external code modules. The price to pay is that memory allocations are no longer strictly deterministic, just like in C/C++.

Frenchmen, I die guiltless of the countless crimes imputed to me. Pray God my blood fall not on France!

Lois XVI, 1793

Chapter 4

Real-Time Execution Platform

THE execution platform—scheduler, GC, class library, etc.—is very important for the behavior of a Real-Time (RT) Java system. Compiled Java code will need to cooperate with the RT multi-threading system on the underlying run-time platform. It will also need to cooperate closely with the RT memory management system in such a way that timing predictability is accomplished, while memory consistency is maintained at all times.

This chapter will first describe the concept of Real-Time Garbage Collect(ion | or) (RTGC) for a compiled Java application, and the generic Garbage Collector Interface (GCI). Then follows considerations concerning on the Java class library, threads and synchronization, and Exceptions, for some different hardware platforms and operating systems.

4.1 Garbage Collector Interface

Different types of (incremental) GC algorithms need different code constructs. For example, to guarantee predictability, a mark-compact GC requires all object references to include a read-barrier, while a read-barrier would only be unnecessary overhead with a mark-sweep GC. These differences makes it error-prone and troublesome to write code generators supporting more than just one type of GC algorithm, and it gets even worse considering hand-written code that needs a complete rewrite for each supported GC type.

The GCI [IBE+02] is being developed within our group to overcome these problems. The GCI is implemented as a set of C preprocessor macros in four layers, as seen in figure 4.1, from the user layer via threading and debug layers to the implementation layer. The two mid-

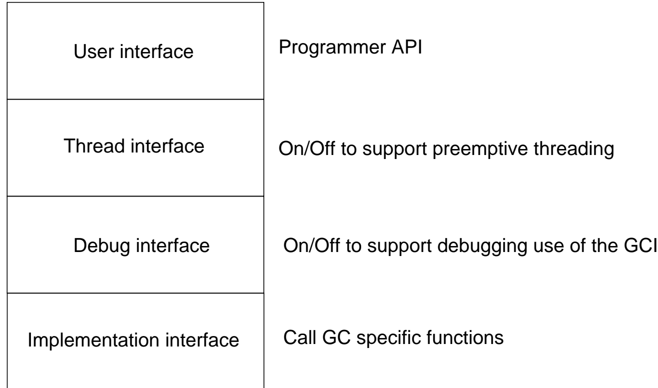

Figure 4.1: *The four macro layers of the GCI.*

dle layers can be switched on/off to support GCI debugging and/or multi-threaded applications.

4.1.1 User Layer

The user layer contains all macros needed for the synchronization between an application and any type of GC. The macros can be divided into eight groups based on functionality.

One time: Macros used to declare static GC variables, and to initialize the heap.

Object layout declaration: Used for declaring object type structs, struct members, and struct layouts.

Reference declaration: Declare reference variables, push/pop references on GC root stacks.

Object allocation: A macro representing the language construct `new`.

Reference access: Reference assignment and equality checks.

Field access: Get/set object attribute values.

Function declaration: Macros for handling function declarations, parameter handling, and return statements.

Function call: Macros for different function calls, and for passing arguments.

None of the macros in the user layer have a specific implementation, but just passes on to the corresponding thread layer macro.

4.1.2 Thread Layer

In a multi-threaded environment, where preemption is allowed to occur at arbitrary locations in the code, all reference handlings become critical sections concerning the GC.

The GCI thread layer adds GC synchronization calls to those macros handling references, i.e.,

```
GC__THREAD_<macro> = gc_lock();
                     GC__DEBUG_<macro>;
                     gc_unlock();
```

4.1.3 Debug Layer

The debug layer macros, if debugging is turned on, adds syntactic and consistency checks on the use and arguments of the GCI macros. While not adding functionality, the debug layer is very useful when manually writing code using the GCI. For instance, consistency of the root stack is checked so that roots are popped in reversed order to the order they were pushed on the stack.this functionality is of great help, not only when implementing a code generator as part of a compiler, but also when implementing native method implementation where GC root stack administration is handled manually.

4.1.4 Implementation Layer

The implementation layer macros, currently there are about 60 of them, finally evaluate to GC algorithm specific dereferencing and/or calls to GC functions, e.g., allocating a memory block on the GC controlled heap.

4.2 Class Library

The standard Java class library is an integral part of any Java application. Most of the standard classes pose no timing predictability or platform dependency problems, and will thus not be discussed here. With the scalability aspect in mind, some adjustments may be needed so as to lower the memory demands. The Java thread-related classes, and the related thread synchronization mechanisms, are of such importance, that they will be treated specially in section 4.3.

When implementing a Java class library for natively compiled Java, intended to execute on (possibly very limited) embedded systems, there are especially two areas needing special care; native methods and I/O.

4.2.1 Native Methods

The Java language was designed from the very beginning not to be able to use direct memory pointers, for good programming safety reasons. There are, though, many good reasons for a Java application to make calls to methods/functions implemented in another programming language:

- Accessing hardware.
- External code modules, as mentioned in section 3.4 on page 22.
- For efficiency reasons, some algorithms may have to be implemented on a low abstraction level using, for instance, C or assembler.
- Input/output operations, as is further discussed next in section 4.2.2.

It is important to note that native method implementations are seldom truly platform independent. If the compiled Java applications is supposed to be executable on more than one platform[1], platform specific versions of all native method implementations for all intended platforms must be supplied. This is analogous to standard Java as defined by the J2SE.

[1]Which is often the case when developing software for embedded systems. First debug on a workstation, e.g. Intel x86 & Posix, then recompile for the target platform, e.g. Atmel AVR & home-built RTOS. See also section 6.2.

Method Calling Convention

There is a standardized calling convention for making calls from Java classes to native method implementations, Java Native Interface (JNI) [Lia99]. To be able to cross the boundary between Java code executing in a virtual machine sandbox and natively compiled code—such as method call-back from a native function—, JNI specifies additional parameters in the call, as well as complicated methods for accessing fields and methods in Java objects.

Considering natively compiled Java code, the situation changes drastically. The overhead created by the JNI no longer fills any function, as there is no language- or execution model boundaries to cross. Straight function calls using the C calling convention provides the best performance, and since all code share the same execution model, native methods may access Java objects, attributes, and methods in a straightforward way.

Memory Management

It is important to note that all external code must access Java references in the same way as the compiled Java code, in order to ensure correctness—also in cases where a compacting garbage collector is used—and timeliness of the application. For legacy code, all code which is not GC-aware, it may be necessary to implement wrapper functions for handling object dereferencing.

The example in listing 4.1 shows what a call to a legacy function may look like, using a wrapper method for object dereferencing.

Listing 4.1: *Example of making a call to a legacy function from compiled Java.*

```
/*
 * Java code
 */
public static native int process(byte[] arg);

public void doSomething() {
  byte[] v = new byte[100];
  int result;
  result = process(v);
}

/*
 * Generated C code from Java code above
 * Most GC administration code left out
 * for clarity.
 */
```

```
JInt Foo_process_byteA(
          GC_PARAM(JByteArray,arg));

GC_PROC_BEGIN(Foo_doSomething,
              GC_PARAM(Foo,this))
  GC_VAR_FUNC_CALL(j2c_result,
                   Foo_process_byteA,
                   GC_PASS(j2c_v));
GC_PROC_END(Foo_doSomething)

/*
 * Hand-written wrapper function
 */
GC_VAR_FUNC_BEGIN(JInt,Foo_process_byteA,
                       GC_PARAM(JByteArray,arg))

  byte[] array;
#ifdef COMPACT_GC
  /* Have to make copy to ensure integrity */
  make_copy_of_array(array,arg);
#else
  /* Objects will not move, so just get a pointer */
  array =  &GC___PTR(arg.ref)->data[0];
#endif

  // Perform the call
  return process(array);

GC_VAR_FUNC_END(JInt,Foo_process_byteA)

/*
 * Legacy (non GC-aware) C function
 */
int process(byte[] arg){
  // Code that does something
}
```

4.2.2 I/O

All no-nonsense applications will, sooner or later, have to communicate with its environment. On desktop computers, this communication takes place in some kind of user interface, e.g. keyboard, mouse and graphics card, via operating system drivers.

Embedded systems typically have much more limited resources for performing I/O. They often have neither normal user interface, nor a file system. The Java streams based I/O (**package** `java.io`) then becomes more a source of unnecessary execution- and memory overhead,

than the generic, easy to use, class library it serves as in workstation- and server environments.

One solution to handle this class library overhead for embedded systems is to flatten the class hierarchy of the Java I/O classes. As an example, consider the widely used method `System.out.print(arg)` which, in an embedded system, could typically be used for logging messages on a serially connected terminal. As is seen in figure 4.2, printing a string on `stdout` starts a very long call chain before the bytes reach the OS level. Clearly, the overhead imposed by an imple-

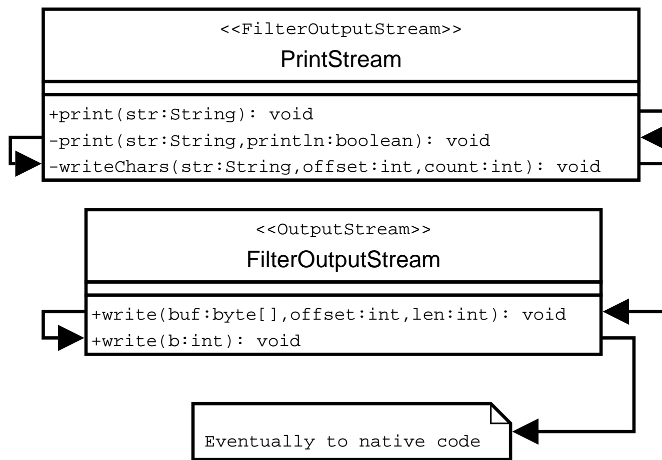

Figure 4.2: *The `System.out.print(String)` call chain, as implemented in the GNU javalib.*

mentation such as in listing 4.2 can not be motivated on a resource-constrained platform. On such platforms, the call chain can be cut in the `PrintStream` class by declaring **`native`** `print` methods.

Aggressive inlining of methods may shorten the call chain substantially, and is an interesting issue for further investigation.

4.3 Threads and Synchronization

One of the benefits of using Java as a programming language for real-time systems is its built-in threading model. All Java applications are executed as one or more threads, unlike C or C++ where multi-threading and thread synchronization is performed using various li-

brary calls (such as Posix). In an environment running natively compiled Java applications, there are two choices on how a Java multi-threading runtime can be implemented:

- One general Java thread runtime for all supported platforms.
  - + One consistent thread model interfacing the Java class library.
  - - May introduce unnecessary overhead on platforms that are already thread-capable (such as Posix).
- For each supported platform, map thread primitives to native methods.
  - + More efficient.
  - - Implementation less straight-forward.

For efficiency reasons, the native implementation of Java threads is best done by providing mappings from Java thread primitives to the underlying OS as native methods implementations, one for each supported platform, as mentioned in section 4.2.1.

The technique with providing the mapping from Java thread classes to underlying OS primitives by using native methods renders the compiled Java application portable between all supported runtime platforms. Recompiling the generated C code and link with the appropriate set of native methods implementations is all that is needed, see figure 4.3 on the facing page.

In order to adhere to the Java thread semantics, the application start-up needs a special twist. Instead of assigning the `main` symbol to the application main class `main`-method, `main` is a hand-coded C function performing the following to start an application:

- Initialize the GC controlled heap.
- Initialize Java classes, i.e., fill in virtual method tables and static attributes.
- Start the GC thread.
- Create a main thread.
- Start the main thread, with the main class `main` method as starting point.

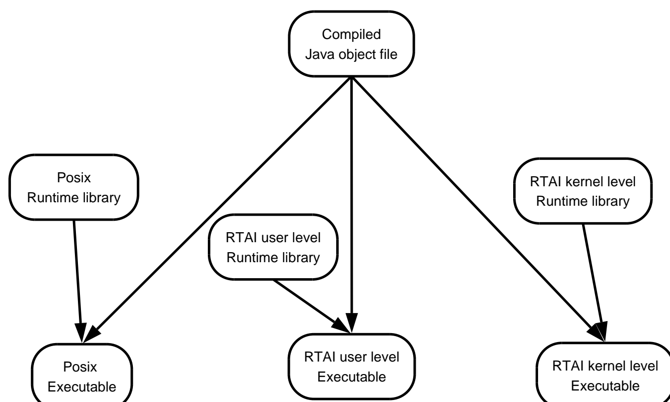

Figure 4.3: *A compiled Java object file can be linked to an appropriate runtime library without recompilation.*

4.3.1 Real-Time Thread Classes

The multi-threading and synchronization semantics in regular Java are quite flexibly specified. Though good for OS portability in a general purpose computing environment, it is not very well suited for hard real-time execution environments.

In order to enhance the thread semantics, a set of new classes for real-time threads in the package `se.lth.cs.realtime` has been developed within our research group [Big98]. A brief description of the most important classes follow below.

FixedPriority Any class implementing the `FixedPriority` interface may not change its runtime priority after the thread has been started. The FixedPriority property can be used in a compile time program analysis to apply directed optimization for code which is only executed by a high priority thread. See also section 7.1 for some examples of such directed optimizations.

RTThread The real-time threads, `RTThread` and its subclasses such as `PeriodicThread` and `SporadicThread`, classes are the extended real-time counterparts to the standard Java thread classes. In order not to inherit any thread semantics from the standard Java threads, the real-time threads do not extend the `java.lang.Thread` class or implement the interface `java.lang.Runnable`, but form an inheritance hierarchy of

their own. This way, the thread semantics for `RTThreads` can be kept suitable for hard real-time systems, if needed.

RTEvent The `RTEvent` is an abstract super class for time-stamped messages objects which can be passed between instances of the `RTThread` class.

RTEventBuffer All instances of the `RTThread` class has an `RTEventBuffer` attribute buffer, serving as a mailbox in inter-thread communication. Both blocking and wait-free message passing is supported.

The real-time thread classes currently have no native implementations in our class library, but implementations are planned for in a near future since some important optimizations rely on these classes, see chapter 7.

4.3.2 Synchronization

The ability to synchronize execution of two or more threads is fundamental to multi-threading applications, for instance monitors and synchronous inter-thread communication. In Java, thread synchronization is built into the language with the **`synchronized`** keyword, and the `wait()`, `notify()`, and `notifyAll()` methods in the `java.lang.Object` class.

The common way of implementing Java thread synchronization is to let each (synchronized) Java object comprise one monitor, where the monitor keeps track of the thread locking the object and which threads are blocked by this lock. This model is fairly simple and it is what is currently implemented in the prototype. There are, though, disadvantages with this model regarding scalability, since all objects in the system must have a monitor object reference even if it will never be used.

An important observation on virtually any real-world Java application is that the number of objects in the application by far outnumbers the number of threads. Blomdell [Blo01] has presented an alternative lock object implementation, where the monitor associated with locked objects is stored in the thread owning the lock instead of in each object. This way, substantial memory overhead may be saved.

Similar to thread implementation, thread synchronization is best implemented in natively compiled Java as native methods, mapping the Java semantics on the underlying OS thread synchronization primitives. Depending on the OS support for monitors, the thread synchro-

Listing 4.2: *Mapping Java monitors on underlying OS.*

```
/**
 * Posix implementation
 */

GC_PROC_BEGIN(monitor_enter,GC_PARAM(java_lang_Object,this))
    pthread_mutex_t *lock;
    gc_lock();
    GC_GET(lock,this);
    gc_unlock();
    pthread_mutex_lock(lock);
GC_PROC_END(monitor_enter)

GC_PROC_BEGIN(monitor_leave,GC_PARAM(java_lang_Object,this))
    pthread_mutex_t *lock;
    gc_lock();
    GC_GET(lock,this);
    gc_unlock();
    pthread_mutex_unlock(lock);
GC_PROC_END(monitor_enter)

/**
 * RTAI implementation
 */

GC_PROC_BEGIN(monitor_enter,GC_PARAM(java_lang_Object,this))
    pthread_mutex_t *lock;
    gc_lock();
    GC_GET(lock,this);
    gc_unlock();
    rt_sem_wait(lock);
GC_PROC_END(monitor_enter)

GC_PROC_BEGIN(monitor_leave,GC_PARAM(java_lang_Object,this))
    pthread_mutex_t *lock;
    gc_lock();
    GC_GET(lock,this);
    gc_unlock();
    rt_sem_signal(lock);
GC_PROC_END(monitor_enter)
```

nization implementation is more or less straight-forward. Example implementations for Posix threads and RTAI kernel threads are shown in listing 4.2.

Using this mapping of synchronization primitives makes it possible to generate portable code as output from the Java compiler, as can be seen in code listing 4.3 on the following page.

Listing 4.3: *Example of Java synchronization with compiled code.*

```
public synchronized void synch() {
  HelloWorld hello;
  hello = foo();
  synchronized(hello) {
    bar();
  }
}

/////////////////////////////////////////////

GC_PROC_BEGIN(HelloWorld_synch,GC_PARAM(HelloWorld,this))
  GC_REF(HelloWorld,j2c_hello);
  GC_PROC_CALL(monitor_enter,GC_PASS(this));

  GC_REF_FUNC_CALL(j2c_hello,foo,GC_PASS(this));

  GC_PROC_CALL(monitor_enter,GC_PASS(j2c_hello));
  {
    GC_PROC_CALL(bar,GC_PASS(this));
  }

  GC_PROC_CALL(monitor_leave,GC_PASS(j2c_hello));
  GC_PROC_CALL(monitor_leave,GC_PASS(this));
GC_PROC_END(HelloWorld_synch)
```

4.4 Exceptions

The exception concept in Java is a structured way of handling unexpected execution situations. When such a situation arises, an Exception object is created and thrown, to be caught somewhere upstream in the call chain. There, the exception object may be analyzed, and proper actions taken.

The Java semantics states that at most one exception at a time can be thrown in a thread. As a consequence, it is natural to implement exceptions, in a natively compiled environment, using the `setjmp()` and `longjmp()` C library functions. These functions implement non-local goto, where `setjmp()` saves the current stack context, which can later be restored by calling `longjmp()`.

An example implementation, also considering memory management issues, is shown in listing 4.4 on the next page. A few notes may be necessary for the comprehension of this example:

{store | get}ThreadLocalException Since only one exception at a time can be thrown in a thread, the simplest way to pass an exception

object from the **throw** site to the **catch** statement is by a thread local reference. All Java exceptions must be sub-classed from the `java.lang.Throwable` class.

{push | pop}ThreadLocalEnv The execution environment is pushed on a thread local stack at each **try** statement executed. A thrown exception is checked at the nearest **catch** statement and, if it does not match, the next environment is popped from the environment stack and the exception is thrown again.

{save | restore}RootStack In order to keep the thread root stack consistent when an exception is thrown, the root stack must be saved when entering a **try** block. If an exception is thrown in a call chain inside the **try** block, and caught by a subsequent **catch** statement, the root stack state can then be restored to the same state as just before entering the **try** block.

Listing 4.4: *A simple exception example.*

```
/*
 * Java code exemplifying Exceptions
 */
void thrower() throws Exception {
  throw new Exception();
}

void catcher() {
  try {
    thrower();
  } catch (Exception e) {
    doSomething();
  }
}

////////////////////////////////////////////////////////

/*
 * More or less equivalent C code
 */
void thrower() {
  ex_env_t *__tmp_env;
  // Create new exception object
  Exception __e = newException();
  // Store reference
  storeThreadLocalException(__e);
  // Get the stored environment,
  // from latest try()-statement
  __tmp_env = popThreadLocalEnv();
  // Restore context, jump to catch block
```

```
  longjmp(__tmp_env->buf, Exception_nbr);
}

void catcher() {
  ex_env_t __env;
  volatile int __ex_nbr;
  volatile int __ex_throw = 1;
  // Save current status of GC root stack
  saveRootStack();
  // save environment
  pushThreadLocalEnv(__env);
  // try
  if ((__ex_nbr=setjmp(__env.buf)) == 0) {
    thrower();
    __ex_throw = 0;
  } else if (isCompatException(__ex_nbr,Exception_class)) {
    // Matching exception caught
    Exception e;
    // Restore previously saved GC root stack
    restoreRootStack();
    // Fetch Exception object reference
    e = getThreadLocalException();
    __ex_throw = 0;
    doSomething();
  }
  if(__ex_throw) {
    // No matching exception caught,
    // Pass upwards in call chain
    ex_env_t *__tmp_env;
    __tmp_env = popThreadLocalEnv();
    longjmp(__tmp_env->buf, Exception_nbr);
  }
}
```

4.4.1 Exceptions in Compiled Java

From a compiler writer's point of view, the exception implementation shown in listing 4.4 poses no really hard problems, he/she just have to get it right once and for all. The compiler user might have an alternative view though, since the generated code tend to get messy and hard to read. To facilitate code readability and decrease the risk of entering bugs, C macros, as shown in listing 4.5 are introduced.

Listing 4.5: *C macros for more understandable exception implementation.*

```
#define EXCEPTION_THROW(__nbr) \
{ \
  ex_env_t *__tmp_env; \
  EXCEPTION_POP(__tmp_env); \
```

```
  if(__tmp_env) { \
    longjmp(__tmp_env->buf, __nbr); \
  } else { \
    UNCAUGHT_EXCEPTION(__nbr); \
  } \
}
#define EXCEPTION_TRY \
{       SAVE_ROOT_STACK(exception); \
  {\
    ex_env_t __env; \
    volatile int __ex_nbr; \
    volatile int __ex_throw = 1;\
    EXCEPTION_PUSH(__env); \
    if ( (__ex_nbr = setjmp(__env.buf) ) == 0) {

#define EXCEPTION_CATCH(__catch_nbr)    \
     __ex_throw = 0; \
     EXCEPTION_POP_DISCARD; \
   } else if (__ex_nbr == __catch_nbr) { \
     RESTORE_ROOT_STACK(exception); \
     __ex_throw = 0; \

#define EXCEPTION_CATCH_MORE(__catch_nbr)       \
   } else if (__ex_nbr == __catch_nbr) { \
     RESTORE_ROOT_STACK(exception); \
     __ex_throw = 0; \

#define EXCEPTION_FINALLY } {
#define EXCEPTION_AFTER_CATCH\
   }\
   if(__ex_throw) EXCEPTION_THROW(__ex_nbr);\
 }\
}
#define EXCEPTION_PUSH(__env) \
   __env.next = gc_thread_get_current()->env; \
   gc_thread_get_current()->env=&__env
#define EXCEPTION_POP(__env_out) \
   __env_out = gc_thread_get_current()->env;\
   if(__env_out) \
      gc_thread_get_current()->env=
               gc_thread_get_current()->env->next
#define EXCEPTION_POP_DISCARD \
   gc_thread_get_current()->env;\
   gc_thread_get_current()->env=
               gc_thread_get_current()->env->next
```

The reader may notice that there are many fragile parentheses in the macro implementations, implicating a very strong dependence between the macros. Using these macro definitions from listing 4.5, applied to the C code in listing 4.4 yields far more comprehensible code, as shown in listing 4.6.

Listing 4.6: *Equivalent C code from listing 4.4, but using exception macros from listing 4.5.*

```
// Java code exemplifying Exceptions
void thrower() throws Exception {
  throw new Exception();
}

void catcher() {
  try {
    thrower();
  } catch (Exception e) {
    doSomething();
  }
}

////////////////////////////////////////////////////////

// More or less equivalent C code
void thrower() {
  Exception e = newException();
  storeThreadLocalException(e);
  EXCEPTION_THROW(Exception_nbr)
}

void catcher() {
  EXCEPTION_TRY
    thrower();
  EXCEPTION_CATCH(Exception_nbr)
    Exception e = getThreadLocalException();
    doSomething();
  EXCEPTION_AFTER_CATCH
}
```

Due to the rather fragile nature of the exceptions macros, it is not recommended to write C code utilizing exceptions by hand, although possible and even inevitable in some situations.

Some execution environments, such as RTAI kernel threads, lacks a working implementation of `setjmp()` and `longjmp()` (for policy reasons in the RTAI case). In these situations, one must supply an appropriate implementation of these functions. In the case of RTAI, the `setjmp()` and `longjmp()` implementations can copied from the standard C library, but on other platforms one may have to implement these functions from scratch.

4.5 Findings

The work presented in this chapter brings forward a number of interesting findings in the various topics described:

GCI: The GC algorithm transparent GCI works very well as a generic interface to different GCs. It is, though, best suited for use in code generators where strict control of the code can be maintained. Manually writing code using the GCI is rather error-prone, due to the complexity of the interface. The debug support in the GCI is of very good use in situations where manually written code is inevitable, such as wrapper functions to external code.

Also non RT applications may benefit from using the GC algorithm transparent interface and debugging facilities of the GCI.

Class Library: In natively compiled Java, since there is no execution environment barrier to pass, native method calls are much simplified compared to JNI.

Calling external non GC-aware functions imply the need for declarations of wrapper functions to resolve symbol names and GCI references.

The I/O model in Java (`java.io.*`) is excessively flexible and bulky for use in resource-constrained embedded systems, with limited I/O capabilities. In such systems, a constrained implementation of the I/O package can be used without inappropriately changing the semantics and decrease portability.

Threads and Synchronization: The thread semantics are enhanced by providing new thread classes more suited for use in real-time environments. By mapping the Java thread and synchronization APIs on OS supplied implementations, we can achieve very good portability of compiled Java code. The same Java application code may be executed on many platforms (including a JVM) without alterations, and often without recompiling the Java source code to C.

Exceptions: Java type exceptions can be implemented on the C level using C macros, in a way such that RTGC is not jeopardized, while retaining readable code. The Exception implementation was also found not to have any significant impact on code size. In a typical application, enabling exceptions only increased code size by approximately 2%.

Never put off till run-time what
you can do at compile-time.

D. Gries

Chapter 5

A Compiler for Real-Time Java

THE tools used for constructing compilers have not changed much during the last decade. Typically, the concrete grammar for the to-be-compiled language is specified according to the parser generator of preference (for example lex/yacc, or bison). The generated parser parses the source code and builds an Abstract Syntax Tree (AST). Hand-crafted code is then typically used for performing static-semantic analysis on the AST, generate some kind of intermediate code, perform various optimizations on the intermediate code, and finally generate assembly- or machine code—possibly with a final optimization pass. Due to the large amount of hand-crafted code needed for the analysis-, code generation-, and optimization phases, the task of constructing a compiler, from scratch, for a modern OO language can be overwhelming.

Using the *JastAdd* [HM02] compiler construction toolkit, developed at our department, we are developing a compiler for real-time Java. The back-end generates C code according to the ideas described in Chapter 3, which is compiled and linked against the GCI and runtime, as described in Chapter 4, to produce a time predictable executable suitable also for hard real-time-systems.

5.1 JastAdd

The JastAdd system [HM02, EH04] is based on current research on Reference Attribute Grammars (RAGs) and Aspect-Oriented Programming (AOP). The goal of the JastAdd system is to provide compiler developers with a better tool for AST manipulations than those available today.

Using the RAG technique makes it possible to declare semantic equations that state how to compute attributes from an AST. These equations present a convenient way of implementing name- and type analysis, and can also be used for rewriting subtrees of the AST on demand while computing attributes.

AOP, as described in Section 2.4.1, is a good help for separation of concerns in the compiler implementation. Different aspects can be kept in separate source code modules which enhances readability, and also makes it possible to add or remove specific aspect modules during implementation or debugging. It is also possible to integrate ordinary Java code modules, if desired, using the JastAdd system.

Input to JastAdd is divided in two parts; an abstract grammar definition of the language, and a set of aspects which will be woven into the AST node classes. The abstract grammar defines both the context-free grammar of a language, and the inheritance hierarchy of the node classes comprising an AST. The other part of a JastAdd system is a set of aspects, usually a mix of ordinary Java code with semantic equations, which are woven in as Java code into the node classes.

The JastAdd tool does not include support for building a concrete parser which generates the AST. Any parser generator capable of constructing a parser which can build Java ASTs may be used as a front-end. JavaCC [Met] is used in our Java compiler implementation, but CUP [HFA+99] has also been used in other JastAdd experiments.

5.2 Architecture and Overview

The architecture of our compiler differs from most available compilers, in that there is no explicit symbol table, nor will it generate internal intermediate code. Instead, all operations are implemented as methods on the AST nodes using the JastAdd system. A concrete grammar description is used to create a parser, while an abstract grammar describes the AST node class hierarchy. A collection of aspects, including name- and type analysis, optimizations and code generation, are woven into the node classes. The parser, node classes, and auxiliary hand-written Java code makes up the compiler, see Figure 5.1.

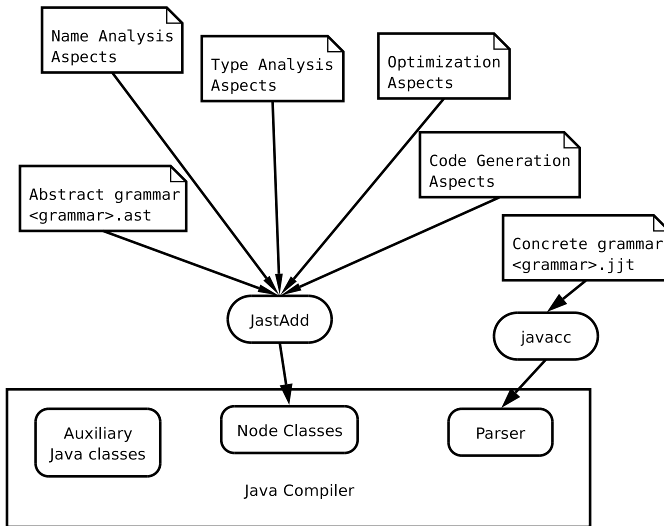

Figure 5.1: *Overview of the Java compiler architecture.*

Abstract Grammar

The purpose of the abstract grammar definition is twofold; at the same time as it specifies the node relations of an AST, it also specifies the class hierarchy for the node classes. As an example, consider the small excerpt from our Java grammar in Listing 5.1 (the complete abstract grammar for our Java compiler is listed in appendix B). The corresponding node class inheritance hierarchy is shown in Figure 5.2.

As can be seen in the listing and figure, the abstract class `Expr` is inherited by all other expression classes, which enables elegant implementations of methods common to all expressions, as exemplified later in this section. The abstract classes `Binary` and `Unary` are analogously inherited by all respective concrete classes.

Using the grammar shown in Listing 5.1, we can build an AST representation of the code snippet

```
a.b = c.d + e - 2;
```

as shown in Figure 5.3. Taking advantage of the inheritance hierarchy of the AST node classes, we can now define aspects operating on this AST representation of a program.

Listing 5.1: *A small example of the JastAdd abstract grammar definition.*

```
abstract Expr;
abstract Binary:Expr ::= Left:Expr Right:Expr;
abstract Unary:Expr ::= Expr;

AssignExpr : Expr ::= Dest:Expr Source:Expr;

AddExpr:Binary;
SubExpr:Binary;

MinusExpr:Unary;

Primary:Expr ::= Name;
Name ::= IdUse*;

IdUse ::= <ID>;

abstract Literal:Expr;
IntLiteral:Literal ::= <INT>;
FloatLiteral:Literal ::= <FLOAT> ;
```

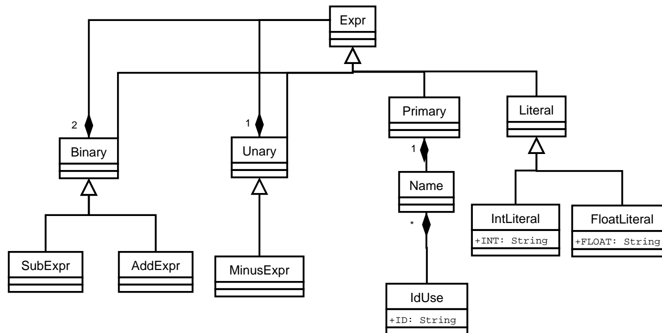

Figure 5.2: *Node class relations in simple JastAdd example.*

JastAdd Aspects

Aspects in the JastAdd system are used for implementing the operations to be performed on the generated AST. They can be implemented either as normal Java code, methods and attributes woven into the AST node classes, or as RAG semantic equations which are translated into Java code by JastAdd, and then woven into the AST classes.

Considering the Java assignment expression above, the two types of aspects can be illustrated with two of the operations a compiler would

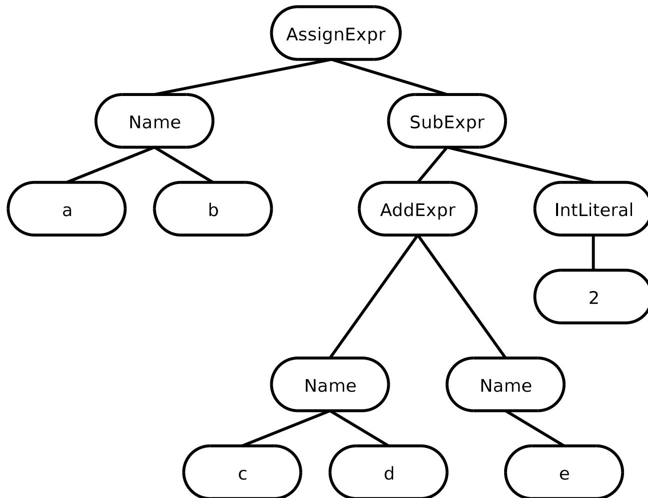

Figure 5.3: *AST representation of the Java expression `a.b=c.d+e-2`, according to the grammar in Listing 5.1.*

typically perform on code; type checking, and code generation. Implementing type checking for this subset of Java expressions is conveniently done using semantic equations, and is shown in Listing 5.2 below.

The **`class`** `TypeCheck` declaration does not, in this example, have any other semantic meaning other than being a syntactic placeholder for declarations (similar to the `aspect` declaration in AspectJ). A synthetic attribute, **`syn`** `TypeDecl type`, is declared in the `Expr` and `Name` classes with default values, and overridden in some subclasses of `Expr`. Semantic equations are written as assignments, and will be transformed to Java methods by the JastAdd system as can be noted in, for example, the declaration of `Unary.type` where the `type` attribute in its `Expr` child is evaluated by calling the generated `type()` method.

Analogously to the synthetic attribute shown in this example, there are also inherited attributes which are used to propagate information downwards in the AST.

Code generation from an AST representation is not equally suited to describe in the form of semantic equations, as is for example type checking. Printing is operational in nature, and it is thus more convenient to implement a code generator using imperative code, even though it would be possible to generate code by evaluating one large string at-

Listing 5.2: *Type checking implemented using semantic equations in JastAdd.*

```
class TypeCheck {
  syn TypeDecl Expr.type = null;
  syn TypeDecl Name.type = lookupTypeDecl();
  Binary.type = LeastCommonType(getLeft().type(),
                                getRight().type());
  Unary.type = getExpr().type();
  Primary.type = getName().type();
  IntLiteral.type = lookupType("int");
  FloatLiteral.type = lookupType("float");
}
```

tribute. As an example of using imperative code in JastAdd, consider the pretty-printer example in Listing 5.3 below. As can be noted in the example, one can mix use of semantic equations with imperative code.

Listing 5.3: *Pretty-printer implemented using Java aspects in JastAdd.*

```
class PrettyPrint {
  abstract void Expr.prettyPrint(PrintStream out);
  syn String Binary.operator = "";
  String AddExpr.operator = "+";
  String SubExpr.operator = "-";

  void Binary.prettyPrint(PrintStream.out) {
    getLeft().prettyPrint(out);
    out.print(operator());
    getRight().prettyPrint(out);
  }
  void MinusExpr.prettyPrint(PrintStream.out) {
    out.print("-");
    getExpr().prettyPrint(out);
  }
  void Name.prettyPrint(PrintStream.out) {
    for (int i=0; i<getNumIdUse(); i++) {
      getIdUse().prettyPrint(out);
      if (i<getNumIdUse()) out.print(".");
    }
  }
  void IdUse.prettyPrint(PrintStream out) {
    out.print(GetID());
  }
  void Literal.prettyPrint(PrintStream out) {
    out.print(GetLITERAL());
  }
}
```

5.3 Simplification Transformations

Generating code from an AST representation can be rather cumbersome , depending on the AST topography[1] and the complexity of the parsed language. Especially, expressions in Java may be rather complex, as for example in the code fragment with corresponding AST in Figure 5.4.

```
a.b().c().d = e().f.g();
```

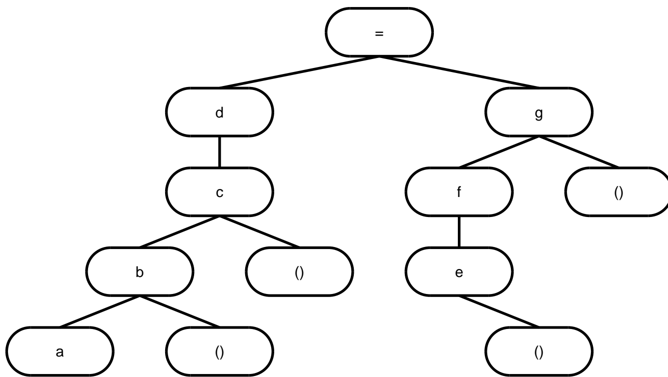

Figure 5.4: *Java code fragment and corresponding AST.*

Building a code generator capable of handling arbitrary expressions tends to be a very complex and error-prone task. Instead, by defining the simplest possible Java language subset while not restricting the semantics, the code generator becomes much simpler and less error-prone, see [Men03] for a definition of such a Java language subset.

The mapping from the full Java language specification [GJS96] to the simpler subset can be conveniently described as a set of transformation on the AST, as will be shown in the following sections.

Names

Most of the simplifying transformations needed to perform on the AST are consequences of real-time memory management, see Section 4.1 for details. Memory operations on references are performed via side-effect

[1] The parser usually does not have all the information needed for building a semantically "good" AST, but instead builds an AST syntactically close to the source code, see Figure 5.4.

macros, only allowing one level of indirection at each step. It is therefore necessary to transform all Java expressions with more than one level of indirection into lists of statements each containing at most one level of indirection. For example, the Java statement

```
a.b = c;
```

contains one indirection, whereas

```
a.b = c.d;
```

has two indirections, and must therefore be transformed into something like

```
tmp_1 = c.d;
a.b = tmp_1;
```

or, described as a transformation on the AST in Figure 5.5, to meet the indirection level requirements.

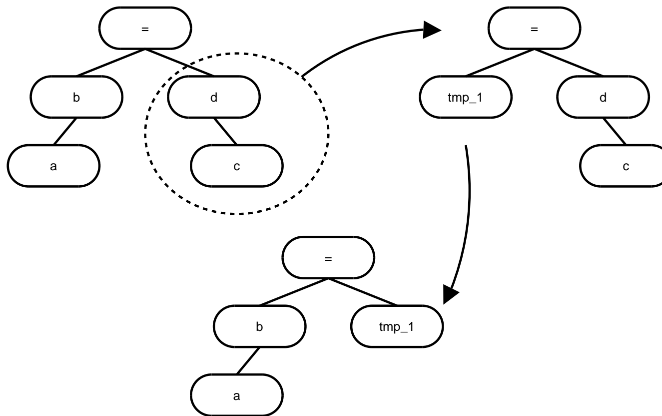

Figure 5.5: *Simplifying names by means of an AST transformation.*

The situation becomes a little more complicated with method calls, since arguments passed in the call may contain arbitrarily complex expressions. By studying the method call

```
a(b(c()),d());
```

we may soon see that the evaluation order of the method calls must be

```
c(), b(c()), d(), a(b(c()),d())
```

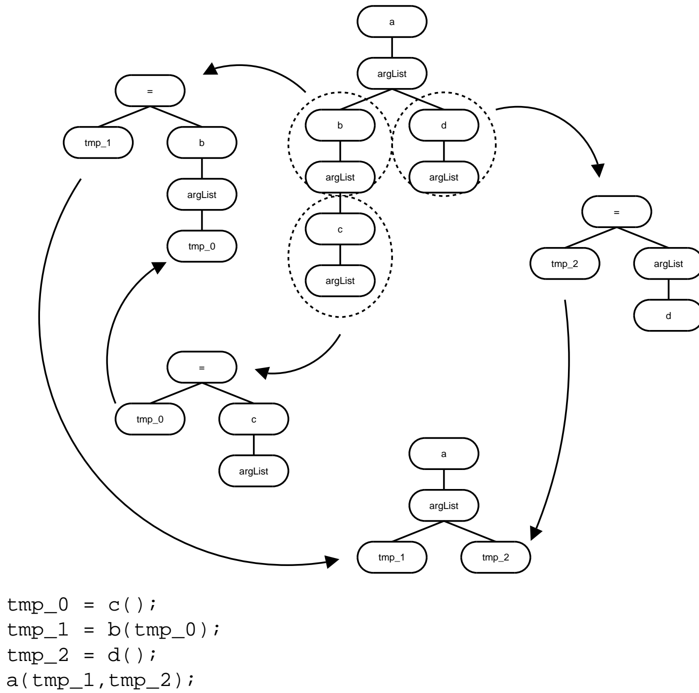

Figure 5.6: *Simplifying a complex method call.*

A suitable simplifying transformation for the above expression, to meet the indirection requirements, could then be expressed as AST transformations or as code as in Figure 5.6.

The aspect code needed for performing the simplification transformations shown in Figures 5.5 and 5.6 is shown below in Listing 5.4.

Listing 5.4: *JastAdd aspects performing simplification transformations for Java names .*

```
class Simplify {
  void Stmt.simplify() {
    if (stmt.needsRewrite()) {
      setStmt(stmt.rewrite(),stmtIndex);
    }
  }

  syn boolean Stmt.needsRewrite = false;
  syn boolean Expr.needsRewrite = needsRewrite(0);
```

```
  ExprStmt.needsRewrite = getExpr().needsRewrite();

  boolean AssignExpr.needsRewrite {
    return getSource.needsRewrite(0) || getDest.needsRewrite(0);
  }

  boolean Access.needsRewrite(int level) {
    return nbrOfDeref() > level;
  }

  syn int Expr.nbrOfDeref = 0;
  VarAccess.nbrOfDeref = 1+getEnv().nbrOfDeref();

  void AssignExpr.rewrite(List l) {
    int sLevel=0,dLevel=1;
    VariableDeclaration varDecl = createTempVar(type());
    l.add(varDecl);
    Expr source = getSource().rewrite(l,sLevel);
    Expr dest = getDest().rewrite(l,dLevel);
    l.add(new ExprStmt(new AssignSimpleExpr(
                         accessVar(varDecl),dest)));
    l.add(new ExprStmt(getSpecialAssignExpr(
                         accessVar(varDecl),source)));
    l.add(new ExprStmt(new AssignSimpleExpr(
                         dest,accessVar(varDecl))));
  }

  Expr Access.rewrite(List l, int level) {
    if (nbrOfDeref() > level) {
      Expr e = getEnv().rewrite(l,0);
      if (level == 0) {
        VariableDeclaration varDecl = createTempVar(type());
        setEnv(e);
        l.add(varDecl);
        l.add(new ExprStmt(
              new AssignSimpleExpr(accessVar(varDecl),this)));
        return accessVar(varDecl);
      } else {
        setEnv(e);
      }
    }
    return this;
  }

}
```

Unary Expressions

Unary expressions which as a side effect changes the value of the operand, may need to be simplified in order to meet indirection requirements. For example, the simple statements

```
a++;
b.a++;
```

should be read as

```
a = a+1;
b.a = b.a+1;
```

which poses no problem in the first statement, with zero indirections, but the latter statement now has two indirections and must be simplified to something like

```
tmp_0 = b.a;
b.a = tmp_0+1;
```

However, things get more complicated as such unary expressions may be used inside other expressions. For example, the seemingly simple statement

```
a[k.i++] = b[++k.i];
```

has a non-trivial evaluation order. A simplification of the above statement which meet indirection requirements can be written as:

```
tmp_0 = k.i;
++tmp_0;
k.i = tmp_0;
tmp_1 = b[tmp_0];

tmp_2 = k.i;
k.i = tmp_2 + 1;

a[tmp_2] = tmp_1;
```

Note that the evaluation of a *PreIncrement* expression differs from the evaluation of a *PostIncrement* expression to maintain semantic correctness.

Control-Flow Statements

The expression—or expressions—which is an important part of all control-flow statements require special care in the simplification process, so as not to alter the semantics of the program. Only the for-statement will be described here, as it is—semantically—the most complicated control-flow statement in Java.

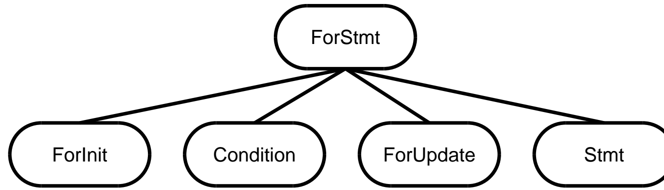

Figure 5.7: *Subtree representing a for-statement.*

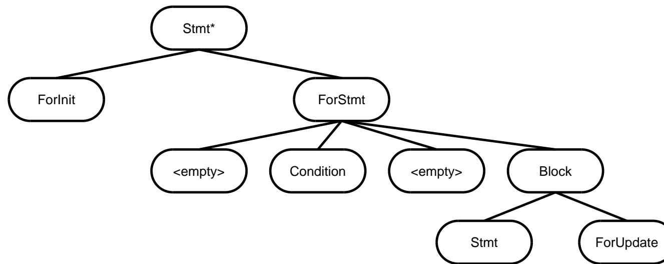

Figure 5.8: *Subtree representing a simplified for-statement.*

A Java for-statement, as defined by the abstract grammar in appendix B, is represented by the AST subtree in Figure 5.7. As defined in the Java language specification [GJS96], the *ForInit* and *ForUpdate* nodes may hold arbitrary lists of *StatementExpressions* or, in the case of *ForInit*, a *variableDeclaration*. An example of a complex **for**-statement could be

```
for(a=b(c(1),d),e=f[g()];a[h++]<i;a=b(c(h++)),d)
  // code
```

The solution to simplifying complex for-statements is to, in fact, create **while**-statements by moving the *ForInit* ahead of the statement and move the *ForUpdate* last inside the *Stmt* node (which has been transformed to a *Block*). A simplified for-statement subtree is shown in Figure 5.8. The resulting code after simplifying the example for-statement above would then be

```
tmp_0 = c(1);
a = b(tmp_0,d);
tmp_1 = g();
e = f[tmp_1];

tmp_2 = a[h++];
for ( ; tmp_2<i ; ) {
  // code

  tmp_3 = c(h++);
  a = b(tmp_3,d);

  tmp_2 = a[h++];
}
```

Similar techniques are used to simplify the other Java control-flow statements.

5.4 Optimization Transformations

Also in cases when compiling to some kind of pseudo-high-level intermediate language (such as C), there is need for some optimizations at the higher abstraction level which can not be taken care of by the intermediate language compiler. Examples of such optimizations are typical OO optimizations, such as implicit finalization of method calls, class in-lining, but also, depending on the object model, (high level) dead code elimination. Of these optimizations, only dead code elimination is currently implemented in our compiler.

5.4.1 Dead Code Elimination

Constructing an AST based on static dependencies between classes in an application clearly results in a set of type declarations including a subset of the J2SE standard classes. However, the J2SE is so designed that, for any application, this subset will include >200 type declarations. A static analysis of all possible execution paths of the application reveals that there exist a set of type declarations, possibly referenced during execution, which includes much fewer classes than static dependencies would suggest. It has also been shown by Tip et al. [TSL03] that there is much to gain regarding the application size if also referenced type declarations are stripped of unused code, such as attributes, methods, and constructors.

Dead-code elimination requires static compilation of the program to be optimized, as dynamically loaded code may try to reference methods or fields which were previously unreachable. It should also be

performed using whole-program analysis, since otherwise only private methods and fields may be analyzed.

Implementation

We have implemented dead code elimination in our Java compiler using JastAdd aspects to calculate the transitive closure of an application, starting from the application `main` method and all `run` methods found in thread objects. Encountered methods and constructors are marked as *live*, as are type declarations with referenced constructors, methods, or fields. During the code generation pass only code for *live* types, constructors, and methods will be generated.

Evaluation

The dead code optimization algorithm has been tested on a couple of applications, with good results, as seen in Table 5.1 below. The two applications are described in Section 5.6.

Application	**Without opt. (kB)**	**With opt. (kB)**
HelloWorld	316	218
Robot Controller	1059	759

Table 5.1: *Code size results from utilizing dead code optimization on some applications.*

5.5 Code Generation

When the AST has been transformed, as described in Section 5.3, to reflect the simplest possible Java coding style, the task of generating intermediate code—in this case C code—becomes relatively simple.

First, a C header file is generated for each used class in the AST, containing the type declarations of the object model. Handwritten C code, such as native method implementations, can then include appropriate class headers. Then, one C file containing the actual implementations of all constructors and methods, as well as class initialization code.

Header Files

The organization of the header files is sketched below as:

<class>_ClassStruct A C struct representing the class. Has pointers to the class's super class struct, and a pointer to this class' virtual methods table. Only one instance of this struct exist in run-time.

<class>_StaticStruct A struct containing static fields of this class, and all ancestors. Only one instance exist in run-time.

<class>_ObjectStruct A struct representing an instantiated object of this class. Contains a pointer to the class struct and all non-static fields of this class (including ancestors).

<class>_MethodStruct The virtual methods table associated with objects instantiated from this class. Contains function pointers for all methods of objects of this class. One instance of this struct exist in run-time.

C code file

The organization of the generated C code files is sketched below as:

- Include necessary header files
- Declare the static object model structs for each class/interface; class, class static, object layout, object static layout, vtable, interface table (if applicable).
- Declare function prototypes for all constructors and methods. This is needed since declare/use order of these is free in Java.
- All function (methods and constructors) implementations.
- The Java classes init function. Pushes layouts on the GC root stack, fill in virtual method tables, and initialize static attributes.

The process of compiling a Java program to an executable machine code image is sketched in Figure 5.9 on the following page.

5.6 Evaluation

The use of a modern RAG-based compiler construction toolkit, JastAdd, lead to a rather compact—yet modular and easy to read—compiler specification. Our Java compiler, as of today, comprises only about

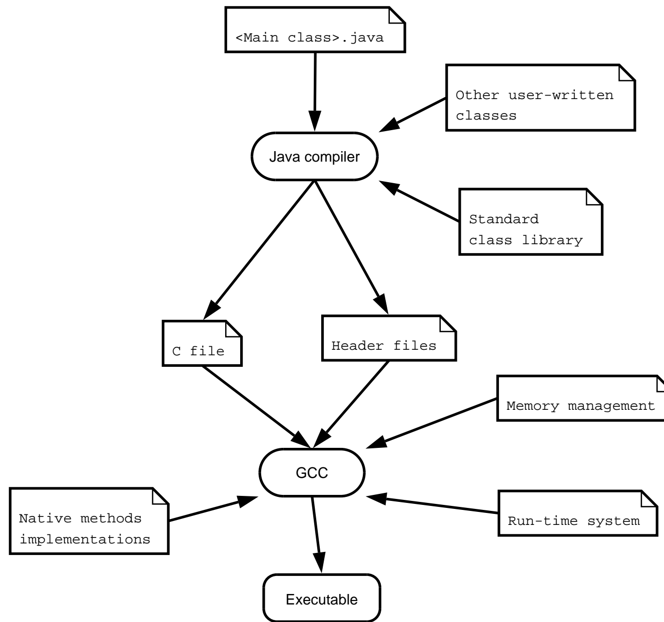

Figure 5.9: *Flowchart of compilation process.*

10000 lines of source code including abstract grammar, concrete grammar, and all aspects needed for semantic analysis, simplifications, optimizations, and code generation needed for generating real-time capable C code. The sizes of the modules of our compiler are listed in Table 5.2.

The current version of the compiler front-end (parser and static semantic analysis) is fully compatible with the current Java standard, version 1.4. Code generation still lacks support for some features of the Java language, most notably inner- and anonymous classes, but the implementation of these features is quite straight-forward and will not add more than, at most, some hundred lines of aspect code to the compiler.

Preliminary Benchmarks

Our Java compiler is still very much in development and very little effort has been spent on compiler speed, but to get an idea on how much slower it is compared to available Java compilers, some very preliminary benchmarks have been performed.

	Lines of code
Front-End	
Abstract Grammar	181
Concrete Grammar	1044
Semantic Analysis	
Name- and Type Analysis	1458
Transformations and Optimizations	
Simplifications	901
Dead Code Optimization	154
Code Generation	
Code generation	5745

Table 5.2: *Source code sizes for the different stages of our compiler.*

The test platform was an ordinary PII 300MHz workstation with 128 MB of RAM. The operating system was Debian GNU/linux, kernel version 2.4.19, and the Java environment is the Sun J2SE version 1.4.1. As reference Java compilers we used javac version 1.4.1 and gcj version 3.3.3.

Two applications were used to benchmark our compiler against the references. *HelloWorld* is a very small one-class application, basically just instantiating itself and printing the words "Hello World" on the terminal. The *RobotController* is a much larger application consisting of about 25 classes, implementing one part of a network-enabled controller for an ABB industrial robot. For some reason, possibly due to the use of native methods, it was not possible to compile the robot controller application using gcj.

	Our compiler	gcj	javac
HelloWorld			
Memory usage (MB)	14	<5	21
Time (s)	26	0.65	3
RobotController			
Memory usage (MB)	34	-	30
Time (s)	160	-	9

Table 5.3: *Java compiler measurements*

As can be seen in Table 5.3, our Java compiler is substantially slower than the other tested compilers. One main reason is the two-pass nature of our compiler (see Figure 5.9), the time needed for gcc to com-

pile the generated C file exceeds 90 s itself. Another reason for the large difference in compilation times is simply that compiler performance has been, and still is, of low priority in the compiler development process. Nevertheless, separate compilation of Java classes, and using more modern computers, would surely decrease compilation times significantly.

Observations

The modularization of a computer achievable using JastAdd benefits compiler development in a number of ways. Some examples include:

Instrumentation The compiler can be instrumented with code for debugging, for example an aspect to dump information in AST nodes.

Measurements Code can be added for measuring, for example the effect of various optimizations.

Experiments A compiler developer can try experimental code, which is easy to remove later. For example, a new optimization can be written as a JastAdd aspect and tested. If it turns out to be useful, the aspect stays, otherwise it goes.

Marge, I agree with you – in theory. In theory, communism works. In theory.

Homer Simpson

Chapter 6

Experimental Verification

IN order to verify the validity of the solutions described in Chapters 3 and 4 with respect to the identified important concepts for embedded real-time Java, as described in Chapter 1, practical experiments are needed.

In this chapter we will present a couple of Java test applications, and how they are compiled and linked for relevant run-time platforms. Results from executing these test applications are used to validate our real-time Java solution with respect to the identified key concepts.

6.1 Portability

Java byte code, and also the generated C code which is the output from our Java compiler, is in itself platform independent. The adaptation to different platforms comes with the run-time systems, or the JVM in the Java byte code case.

The real-time Java runtime, as described in Chapter 4, has been implemented with support for 5 different threading models on 4 different hardware platforms. Table 6.1 shows a matrix covering the current available implementations. Of course there are no timing guarantees in the Posix thread model on a standard Linux or Solaris OS, but it is the best suited runtime for verifying the semantic correctness and concurrency behavior of an application.

Table 6.1: *Current implementation status of the real-time Java runtime environment.*

	AVR	PPC	i386	SPARC
CSRTK[a]	X			
STORK[b]		X		
Linux RTAI(k)[c]		X	X	
Linux RTAI(u)[d]		X	X	
Linux and Solaris Posix			X	X

[a] Small real-time kernel for the Atmel AVR developed at the department of CS.
[b] real-time kernel for PPC, developed at the department of automatic control.
[c] Kernel level threads.
[d] User level threads.

The available implementations span a range of quite different types of CPUs (Harvard RISC micro-controller, RISC, and CISC) and very different threading models. Given this diversity, porting the runtime to a new CPU and/or threading model should be affordable, and the portability of the proposed solution can be considered agreeable.

6.2 Scalability

Experimentally verifying the scalability of compiled real-time Java effectively boils down to trying to find the lower limit for resource-constrained hardware, on which it is possible to deploy a useful application. Finding the upper limit is not equally interesting, since the main development platform is a standard Intel PC (≈2GHz, ≈512MB RAM).

6.2.1 Low-End Experiment Platform

As a low-end experimental platform, we have a small experimental board, see Figure 6.1 on page 69, equipped with an Atmel AVR 128, a two row LCD display, 6 buttons, and a summer.

Hardware

The Atmel AVR ATmega 128 [Atm03] is a modern 8-bit RISC micro-controller, with many features on-chip (not all listed here):

- 32x8 general purpose registers plus peripheral control registers.
- Up to 16 MIPS throughput at 16 MHz.
- 128K Bytes of in-system re-programmable flash memory with 10,000 write/erase cycles endurance.
- 4K Bytes E^2PROM. Endurance is 100,000 write/erase cycles.
- 4K Bytes internal SRAM. Up to 64K extended memory.
- SPI interface for in-system programming.
- Two 8-bit timers/counters and two 16-bit timers/counters. One real-time counter with separate oscillator.
- 8-channel, 10-bit ADC.
- Dual programmable serial USARTs.

The experimental platform is equipped with an additional 128 KB SRAM chip, of which 61184 bytes are reachable from the AVR, making a total of 64K bytes SRAM available to the running application.

RTOS

A very small real-time kernel has been developed at the department, for use on the Atmel AVR. The fully preemptive kernel has a footprint of less than 10 kbytes of ROM and 1 kbyte of RAM. Worst-case execution times of operations in the kernel are summarized in Table 6.2. See also [Ekm00, NE01].

	Execution time in CPU cycles	
Operation	*Worst*	*Best*
Context switch due to timer interrupt	$963 + 358 \cdot k$	$889 + 346 \cdot k$
Context switch due to voluntary suspension	$740 + 12 \cdot k$	728
Take a mutex	113	113
Give a mutex	$1024 + 12 \cdot k$	1014
Create an object	$234 + 78 \cdot i + 54 \cdot n$	$234 + 78 \cdot i + 54 \cdot n$

Table 6.2: *Measured performance with k priority levels and object size s bytes with n pointers divided into i groups. 1 CPU cycle is 0.25 μs.*

Experimental Application

For the purpose of testing the scalability of compiled real-time Java, a suitable application is needed. It should, at least, contain two (communicating) threads and a reasonable number of classes.

A suitable application is found in the first programming assignment of the undergraduate course in *Concurrent and Real-Time Programming*[1]. This application is a simple implementation of an alarm-clock with basic functionality. The alarm-clock application in itself—not considering Java classes and threads in the Graphic User Interface (GUI)—consists of at least two threads sharing a critical resource—the representation of time—, and four user-written classes. A typical implementation would contain these four classes:

AlarmClock The application main class, initializes the application and starts the two threads.

ClockStatus A passive class containing the critical resources; the time- and alarm time representations.

TimeHandler Contains a periodic thread which updates the time once every second. If the alarm conditions match, a beep is also emitted.

ButtonHandler Contains a thread which waits on a semaphore for user interaction, i.e. a button in the user interface has been pressed. Depending on the sequence in which buttons are pressed, time or alarm time is set in ClockStatus.

All User Interface (UI) specific code is placed in a separate Java package, which greatly enhances the portability of the alarm-clock application. In order to run this application on the experimental AVR platform, the UI package is substituted with a new implementation—with an identical API—which communicates with the LCD display and hardware buttons, via native methods, instead of using the `java.swing` or `java.awt` packages, see Figure 6.1.

Table 6.3 on the facing page shows the memory usage of the alarm-clock application, when compiled for the AVR. Worth notice is the significant decrease in ROM footprint when compiling with dead code elimination. As a comparison, running the alarm-clock as an applet requires about 22M bytes of RAM.

[1]See the EDA040 course at `http://www.cs.lth.se/Education/Courses/`.

Compilation flags	ROM (bytes)	RAM (bytes)
-w	89k	<32k
	89k	<32k
DCE -w	61k	<32k
DCE	61k	<32k
DCE -Os	61k	<32k

Table 6.3: *Memory usage for the alarm-clock on the AVR platform. DCE stands for dead code elimination turned on.*

One interesting observation from Table 6.3 is that the size of the ROM flash image does not depend on whether debug (-w) or size optimization (-Os) flags were given to the C compiler. The reason may be related to some of the tricks used in the GCI to disallow dangerous C code optimizations, but this will need more investigation.

Results

Considering the alarm-clock to be of adequate complexity for a typical real-world application in this type of hardware platform, it still leaves half of the amount of RAM and ROM in our hardware platform unused. We can then argue that natively compiled real-time Java is a viable solution also for systems of this size.

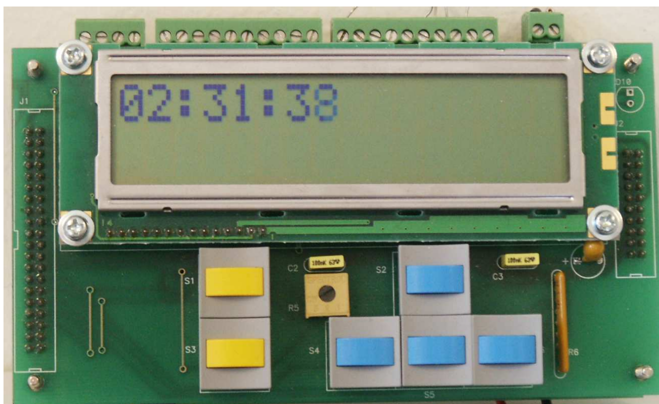

Figure 6.1: *Alarm-clock application running on the AVR platform. The platform consists of two stacked cards, with a user interface card on top of a generic CPU card.*

6.3 Hard Real-Time Execution and Performance

Hard real-time performance (predictability) and general performance (speed) are not necessarily coupled. Instead, in order to guarantee timing predictability it is often necessary to sacrifice execution speed for increased timing predictability. Experiments verifying the timing predictability of our real-time Java execution environment are presented in Section 6.3.1, while general performance execution experiments are presented in Section 6.3.2.

6.3.1 Hard Real-Time Execution

In order to try to verify the alleged hard real-time capabilities of our proposed real-time Java, we implemented a simple multi-threaded application and executed in a RTOS. The test application creates and starts three periodic threads with different priorities and periods, as shown in Table 6.4. This application resembles the execution pattern of a typical embedded real-time application, with a couple of periodic threads controlling a physical process.

The purpose of this experiment is to see if our ideas on how real-time Java can be implemented, without forsaking automatic memory management, hold up to the reality. The system is quite heavily loaded, ≈65% utilization from the threads, and then another ≈30% GC utilization from cleaning up the garbage generated by the threads. We then have a system where the GC thread uses practically all idle time there is left, trying to free unused memory blocks. The three threads will then always need to preempt the GC thread, to be able to execute (when not waiting for a higher priority thread to finish).

Thread	**Period (μs)**	**Workload (μs)**
T_1	100	≈30
T_2	300	≈50
T_3	500	≈90
GC_1	NA	NA

Table 6.4: *Characteristics of the three threads in the timing experiment. Thread T_1 has highest priority and the GC thread GC_1 has lowest priority. Neither period, nor workload is applicable for the GC thread.*

Test Platform

The benchmarks were performed on a 333MHz Pentium II workstation running Debian/GNU Linux, kernel version 2.4.19, RTAI version 24.1.11.

Compilation Configurations

The test application was compiled and linked against two different GC implementations; one *Mark-Sweep* GC, and one *Mark-Compact* GC. All threads allocate garbage in their run loops, forcing the GC thread to clean up the heap during idle time.

Results

Results from the experiments are shown in Figures 6.2 and 6.3, and some statistics are found in Table 6.5. From these results, we can draw a number of conclusions regarding the real-time behavior:

- Latency and response times are quite good. The amount of jitter is well within margins for thread T_1, bearing in mind that we have a sampling frequency of 10 kHz and a system with more than 90% CPU load.
- There is a slight, but notable, difference in performance between the two GC algorithms used in the experiment. The reason for the *Mark-Sweep* GC giving a slightly better performance is here due to that GC synchronization can be made more efficient. The performance differences between the two GC algorithms will be further discussed in Section 6.3.2 below.

From these results, there seem to be no reason why natively compiled Java could not be a feasible programming language for hard real-time systems.

6.3.2 Performance

Although determinism and the ability to guarantee that deadlines are met are absolutely crucial for hard real-time systems, general execution performance must not be forgotten. This is especially important in small, resource constrained, embedded systems where a faster processor may not be a viable alternative due to common processor constraints including power consumption (if run on battery power), heat dissipation, and cost.

		Mark-Compact	Mark-Sweep
Latency μs			
T_1	min	3	2
	max	12	14
T_2	min	41	36
	max	53	48
T_3	min	43	36
	max	159	139
Response μs			
T_1	min	32	27
	max	43	41
T_2	min	95	83
	max	153	95
T_3	min	175	142
	max	291	254

Table 6.5: *Real-time performance statistics.*

In order to investigate the efficiency of the code generated by our Java compiler, described in Chapter 5, and to try to investigate what kind of impact the GCI implies, we implemented a couple of small applications. These test applications were then compiled with our Java compiler, using different GC configurations, as well as using Sun's `javac` and GNU's `gcj` [gcj] java compilers for reference. As a fairly realistic estimate of the best possible performance, equivalent applications implemented in C were compiled with `gcc`. The benchmark applications are as follows:

fibonacci A simple recursive implementation of the Fibonacci algorithm. Implemented both as a virtual method and as a static method to investigate the possible performance impact of looking up virtual methods.

scalar A method which calculates the scalar product of two vectors, implemented as Java arrays. Linear algebra calculus is very common in automatic control, and efficiency is very important since these calculations often take place in high frequency control threads.

One could argue that these small benchmark applications do not reflect the behavior of realistic embedded and real-time applications, which is true. However, they do bring forward the code constructs where we believe RTGC synchronization imposes the largest impact on performance.

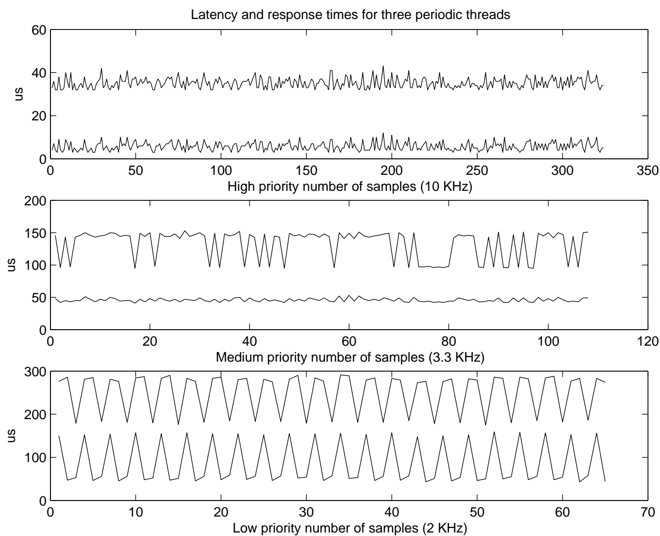

Figure 6.2: *Three periodic threads with top-down increasing priority. Mark-Compact GC used.*

Test Platform

The benchmarks were executed on a 333MHz Pentium II workstation running Debian/GNU Linux, kernel version 2.4.19. Involved software include GNU Compiler Collection (GCC) version 3.3.3 and Sun J2SDK version 1.4.1.

Compilation Configurations

The different compilation (and runtime) configurations used in the benchmark tests are briefly described below:

Our compiler The compiled Java code was linked against both against a *mark-compact* GC and a *mark-sweep*, to see how much the more complex GC synchronization for a mark-compact GC hurts performance. As a reference, the code was also compiled without any GC support at all. However only usable for applications with only static memory allocation, compiling without GC synchronization reveals the the total cost of GC synchronization, and it also serves as an indication of overall code efficiency compared to other Java compilers, which lack hard real-time support.

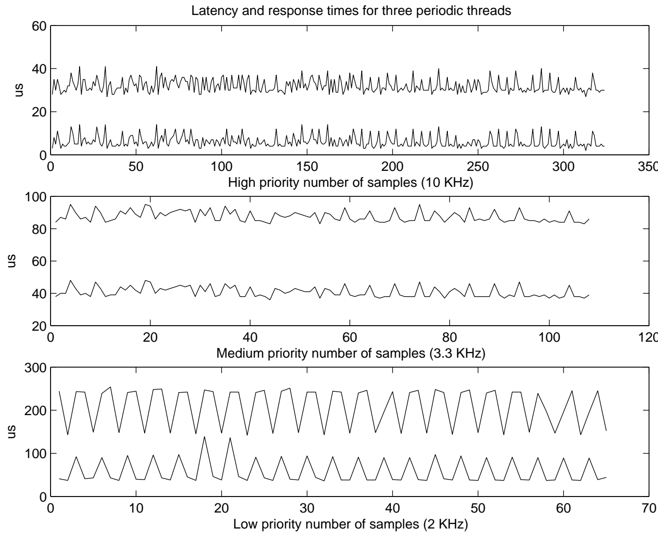

Figure 6.3: *Three periodic threads with top-down increasing priority. Mark-Sweep GC used.*

Sun JVM The Sun JVM was run with three different configurations. First, the default client configuration which dynamically compiles the byte codes using the Sun HotSpot Just-In-Time (JIT) compiler. Then, using the `-server` option which tries to optimize more for speed. Last, since JIT compilation is often impossible to to in hard real-time systems, the JVM was run using the `-Xint` option which turns off all dynamic optimizations, and run the application in the interpreted fashion.

GCJ GNU Compiler for Java (GCJ) was used to compile and link the applications to native static binaries. Compilation was done utilizing no optimizations at all, and with the most aggressive speed optimizations.

GCC GCC was used to compile and link the applications to native static binaries. Compilation was done utilizing no optimizations at all, and with the most aggressive speed optimizations.

Results

Results from executing the benchmark applications are shown in Table 6.6 below.

	fibonacci (virtual)	fibonacci (static)	scalar
Our compiler			
mark-compact GC	10050	7012	146400
mark-sweep GC	7002	6904	7760
no GC	753	586	5402
Other			
Sun JVM	271	251	5085
Sun JVM -server	270	245	3910
Sun JVM -Xint	3302	3120	52500
GCJ	360	567	10098
GCJ -O3	328	504	2249
C code			
GCC	NA	280	6810
GCC -O3	NA	293	761

Table 6.6: *Performance measurements. Execution times in milliseconds.*

These results reveal a number of interesting properties of the current compiler and runtime implementation. The most obvious finding is that, at least in these small benchmark examples, we achieve really poor performance using our Java compiler together with a RTGC. The fibonacci application executes 20-30 times slower than when using the Sun JVM (with JIT compiling) or the GCJ compiler, and 2–3 times slower than pure interpreted java byte codes. However, when we disable GC synchronization we achieve results that are just 1–2 times slower than the JVM with JIT compiler and GCJ, which is not too bad for a compiler with just about no performance optimizations implemented. We can thus draw the conclusion that the observed performance problem does not relate to the generated C code as such, but is almost solely related to the GC synchronization mechanisms.

The *scalar* application shows more mixed results. Running with a *Mark-Compact* GC results in very poor performance while running with the *Mark-Sweep* almost matches the performance of the Sun JVM or the GCJ. Also in this case, it is quite clear that the GC synchronization mechanism is the main culprit for poor performance.

One intuitive conclusion to draw from these results would be that using a *Mark-Sweep* GC is always preferable to using a *Mark-Compact* al-

ternative due to much higher performance. That is not always true, and depends on the type of application being executed. A *Mark-Compact* GC implies both read- and write barriers in order to ensure consistent object references, while only the write barrier is needed for ensuring consistency with a *Mark-Sweep* GC, see also Chapter 3. This difference is very clear in the *scalar* application, which is focused on reading values from arrays, where the *Mark-Compact* case suffers from the read-barrier penalty. The same phenomenon is found in the *fibonacci* application, although not as evident. For an application with few live objects, and which generates a lot of garbage (short-lived objects), the *Mark-Compact* GC may result in significantly better performance than the *Mark-Sweep* GC, since considerably less work is needed to copy a few (small) live objects than adding all memory blocks occupied by garbage to free-lists.

Although results from timing experiments suggest good real-time characteristics with short latencies and small amount of jitter, there is certainly future work to do in order to get acceptable general performance.

6.4 Hard Real-Time Communication

As has been shown in Section 6.3.1, our Java execution environment provides very predictable and stable response times, also in highly loaded systems with plenty of GC work going on.

Hard real-time communication in a compiled real-time Java environment has been verified in a Masters thesis project done at our department [GN04]. Future work is planned to further investigate and develop real-time communication in this environment, see Chapter 7.

6.5 Applicability

The proposed solution to natively compile Java for real-time systems has been tested in experiments on various hardware platforms. Tested applications range from very small with soft real-time demands—the alarm-clock application in Section 6.2—, to industrial robot control systems with hard real-time demands and workstations running real-time Linux. A large number of testing applications have also been executed on standard Linux workstations, with and without hard real-time support using the RTAI.

There are general performance issues that need to be dealt with, but nevertheless we feel that Java will very soon be a viable programming language for most types of embedded and/or real-time systems.

Perfection is reached, not when
there is no longer anything to add,
but when there is no longer
anything to take away.

Antoine de Saint-Exupéry

Chapter 7

Future Work

THE Java to C compiler and associated run-time system framework is, as of current status, capable of handling most of the Java language, generating semantically correct C code. Apart from the fact that neither the compiler, nor the runtime system and class library, are complete, with regard to the Java specification and the Java Development Kit (JDK), there are many interesting problems to look into.

7.1 Optimizations

Generating code that will function properly in all possible executions will result in conservative code, with sometimes unnecessary overhead degrading application performance[1]. We are therefore looking at several ways of enhancing general performance, without sacrificing real-time performance.

7.1.1 More Efficient GC Locking Scheme

The technique used for controlling GC critical sections is very conservative regarding threads with higher priority than the GC thread. A high priority thread can not be preempted by the GC, and thus all GC locking/unlocking in code executed by such a thread is unnecessary overhead.

[1] e.g., The wanted sampling rate of a high priority regulator thread can not be accomplished due to GC overhead

Hypothesis

Under certain conditions, a static analysis of the AST can reveal methods which are only called from high priority threads. GC-locking can then be omitted in the generated code for those methods, resulting in a significant performance gain for the highest priority thread.

Prerequisites

All threads must implement the `FixedPriority` interface, see Section 4.3.1, and all thread priorities must be determinable at compile time.

Method

- Traverse the reachable AST subtree starting from the main class `main` method and all `run()` methods found in thread classes, searching for thread activations.
- For each thread activation found, traverse the call graph, marking each processed method declaration with the $Min(current, called)$ priority.
- During code generation, disable GC locking for those methods which are only called in high priority threads.

7.1.2 Memory Allocation

When using a mark-sweep GC, and not partitioning the heap in blocks of constant size, there is a critical real-time performance bottleneck when high priority threads allocate objects from the heap. The time needed for a memory allocator to find a suitable free block is not deterministic, and may cause the thread to miss deadlines, or introduce unacceptable jitter. this may be a serious problem in an application where high priority threads need to allocate memory, and, for various reasons, it is not feasible to use a mark-compact GC or block-allocate from the heap.

A possible solution to this problem would be to let high priority threads allocate objects from a maintained pool, which is guaranteed to contain free blocks of appropriate sizes at all times. The cost for maintaining the memory block pool is added to the GC overhead paid by low priority threads.

7.1.3 OO optimizations

There are a number of OO optimization techniques which could be used to increase general performance of an application. To this class of optimizations belong such well-known techniques, see for example [AHR00, FKR+99, TSL03], as method call de-virtualization and class in-lining.

7.1.4 Selective Inlining

In conjunction with "normal" in-lining, it would be very interesting to investigate the possible benefits from more aggressive in-lining of code which is called from the highest priority thread `run()` method.

Hypothesis

Under certain conditions, a static analysis of the AST can reveal methods which are only called from high priority threads. Aggressive class- or method in-lining could then be used to increase performance of the highest priority thread, by omitting indirection- and function call overhead. Similar to the GC locking optimization above.

Prerequisites

All threads must implement the `FixedPriority` interface and all thread priorities must be determinable at compile time.

Method

Traverse the AST call graph originating from the highest priority thread `run()` method. Classes or methods found during the tree traversal is then in-lined in the highest priority thread class, if they are explicitly or implicitly final, and are not used by any other class in the application.

7.2 Networking

The current trend towards distributed automation systems, and to close control loops over distributed nodes in a network, introduces some interesting issues in the programming language and run-time environment domain. Two concepts which would be very interesting to study closer are Quality of Control and Constant Bandwidth Server [HCÅÅ02].

7.3 Dynamic Class Loading

Although dynamic loading of classes is not of very much interest for small embedded real-time systems, it can be useful in larger real-time systems, e.g in industrial robot control systems where a software upgrade on-the-fly can save a lot of money.

Method

Implement dynamic class-loading using the technique described in [NBL98]. The new class is loaded and initialized in a low priority thread, and the time needed for activation can be kept very short so as not to disturb high priority threads.

In the RTAI runtime environment, the concept of loadable Linux kernel modules may present one way of achieving dynamic loading of code.

7.4 Code Analysis

Persson [Per00], has published work on using the JastAdd tool to implement worst-case memory usage and WCET analysis on Java applications. His work should be continued and implemented in our Java compiler, not only as an aid to the programmer, but the analysis results should be possible to use in some optimizations.

7.5 Hybrid Execution Environment

In some situations a hybrid execution model, mixing code executed in a JVM with natively compiled code, can be preferred to choosing one of the execution models. Since the IVM [Ive03] uses the same object model as our Java compiler, it should be possible to integrate these execution environments.

A thing is not necessarily true
because a man dies for it.

Oscar Wilde, "The Portrait of Mr. W.H."

Chapter 8

Related Work

THE concept of natively compile Java code and/or making Java viable for use in systems with hard timing constraints is not new. There is plenty of both academic and industrial work published, and this chapter will present some of the more interesting projects.

8.1 Real-Time Java Specifications

There exist two committee driven standards for adapting Java to hard real-time systems; the *Real-Time Specification for Java* (RTSJ) from *The Real-Time for Java™ Expert Group* (`http://www.rtj.org`), and the *Real-Time Core Extensions for Java* (RTCE) from the *JConsrtium* (`http://www.jconsortium.org`).

RTSJ

The RTSJ identifies seven areas where the Java specification must be enhanced in order to facilitate the use of java in real-time systems. The intended execution model is that of an enhanced JVM. The seven areas, with a brief description of the needed enhancement, are as follows:

Thread Scheduling and Dispatching: *Schedulable objects* are scheduled by the instance of a `Scheduler`. The scheduler is priority-based, but the actual implementation of the scheduler may be replaced.

Memory Management: GC controlled heaps are seen as an obstacle to achieve good real-time performance. A normal heap is supplied, but real-time threads must not hold references to objects within it.

instead, the RTSJ defines three additional memory areas; scoped memory, physical memory, and immortal memory, in conjunction with the heap. Objects in scoped memory may only be accessed from other scoped memory object or local variables, if in visible scope (same, outer, or shared).

Synchronization: The Java synchronization semantics is strengthened by mandating priority inversion control by the means of priority inheritance or priority ceiling algorithms. Wait-free communication between real-time threads and regular Java threads is also supplied.

Asynchronous Event Handling: The enhanced support for asynchronous event handling is not real-time specific, but a more efficient way of handling external events in Java applications.

Asynchronous Transfer of Control: A more efficient (and data consistent) way to make a thread abandon its execution, than can be done using the `interrupt()` method and the regular exception mechanism.

Asynchronous Thread Termination: Through a combination of asynchronous event handling and asynchronous transfer of control, threads may be forced to terminate in a clean an ordered way.

RTCE

The Real-Time Core Extensions (RTCE) has a slightly different approach to real-time Java than does RTSJ. Instead of enhancing the regular Java specification, the RTCE defines a set of *Core* (real-time Java) components, which can, but must not, be used together with *Baseline* (regular Java) components, in an application. Another difference is that the RTCE allows for a Core Native Compiler and the possibility to use a conventional execution model instead of a JVM.

Objects allocated in core memory may not access baseline objects, and baseline objects may only access core objects via special method calls. There is also support for stackable (short-lived) core objects.

Findings

The two standards differ in many details, but they do have one large drawback in common. Instead of focusing on how to solve the real-time garbage collection problem, they both resort to introducing additional

memory types which can be used by high priority threads. This will, in effect, return memory management to the error-prone programmer, who will have to figure out which objects may reference which other objects without violating the various memory access rules.

8.2 OOVM

Although not based on Java technology, but on Smalltalk, the OOVM [Bak03] is nevertheless a very interesting project. It is from the beginning designed for compactness and speed, and preliminary results show the OOVM speed to be on par, or slightly faster than, the HotSpot virtual machine, while ROM footprint is considerably smaller than for the JVM.

The current implementation runs on StrongARM or IA32, and needs about 128 kB of RAM for the system itself. There is no hard real-time support currently, but a RTGC is said to be on its way.

8.3 Jepes

The JEPES project [SBCK03] aims at being a high-performance, customizable platform for Java in small embedded systems. The target platforms range from low-end 8-bit micro-controllers with 512 bytes of RAM, 4KB of ROM, up to 32 bit microprocessors with more than 1MB of RAM. JEPES hence places itself covering the range from javaCard [Sun00b] environments to J2ME [Sun00a] environments.

The authors of JEPES introduce a nice idea, called *Interface Directed Configuration*, to specify per-class compile-time configurations in a non-intrusive way. By implementing an interface, it is possible to e.g. specify a method of a class as an interrupt handler, and the compiler can then generate appropriate prolog/epilog code for that handler.

A feature of the JEPES compiler is the use of optimizations to minimize memory (ROM and RAM) usage. By performing a context-insensitive whole-program data-flow analysis on the application, it has been shown to reduce some applications to $\sim 20\%$ of the original size. Optimizations include among others virtual dispatch elimination, method in-lining, and dead code elimination.

Findings

JEPES was not originally intended to be used in hard real-time environments, and thus lacks real-time memory management. JEPES applications can, though, have a predictable behavior if one does not use any dynamic memory, all objects must be statically allocated at initialization time.

8.4 JamaicaVM

Aicas GmBH and IPD Universität Karlsruhe have implemented a combined JVM and Java bytecode-to-native compiler called Jamaica that is said to comply with hard real-time constraints, see [Sie00, Sie99, SW01]. The Jamaica VM is always responsible for garbage collection and the task scheduling, while some classes may be natively compiled and call the VM for services such as memory allocation. The GC principle used is a non-moving type with fixed memory block size for eliminating external fragmentation. The amount of GC work to do at each object allocation is scheduled dynamically with respect to the current amount of free memory, and task latency (also for high priority tasks) will vary accordingly.

There is some new work going on, involving JamaicaVM, called HIDOORS, which aims at being a fully fledged Java integrated development environment for embedded and real-time systems. No results have been published, except for a short objectives paper [VSWH02] and a web-site.

Findings

Not only the varying task latency, but the need for a "bloated"[1] VM and the fact that the fixed size memory block scheme makes linking with non-GC-aware code modules complicated, make the Jamaica system inappropriate for small embedded systems and for flexible hard real-time systems.

8.5 PERC

The PERC Java platform from Newmonics Inc. [NL98] is another example of a hybrid platform with alleged hard real-time capabilities. but

[1] The memory footprint of the Jamaica VM is around 120KB.

it has a footprint of at least 256KB ROM and 64 KB RAM. Linking in external code is also not feasible.

8.6 SimpleRTJ

SimpleRTJ [RTJ] is a clean-room implementation of a Java JVM, intended to run on devices with limited amount of memory. There is support for multi-threaded applications and a GC-controlled heap, and the typical memory footprint for the VM is around 20KB.

However, the included GC is of the ordinary three color mark-and-sweep stop-the-world batch type, and there can thus be no timing guarantees in an application running on the SimpleRTJ.

8.7 GCJ

The GCJ is the Java compiler and class library part of the GNU GCC project [gcj]. It is capable of taking Java source code or byte code as input, producing Java byte code or a native binary as output. The GCJ runtime provides the core class library, a garbage collector, and a byte code interpreter, which makes it possible to run an application in mixed mode (compiled/interpreted) and to use dynamic loading of classes.

Since GCJ share back-end with the rest of the GCC, it can be configured as a cross-compiler for many types of CPUs, making it suitable for embedded systems development. The included memory management is though not intended for use in hard real-time applications, and thus lacks strict timing guarantees.

"Contrariwise," continued Tweedledee, "if it was so, it might be, and if it were so, it would be; but as it isn't, it ain't. That's logic!"

Lewis Carroll, "Through the Looking Glass"

Chapter 9

Conclusions

MOTIVATED by the needs to shorten development times, and to improve software quality, in embedded systems development, we have investigated and experimentally verified the possible benefits from using more modern programming languages. Modern, safe OO languages with built-in support for multi-threading, distributed environments, and platform independence have been shown to be beneficial in terms of software quality and development time in other software areas. We have chosen to use Java as an example of a modern OO language, but our results should be valid for virtually any, safe, OO programming language, such as C#.

Important aspects, crucial for the viability of real-time Java, have been identified. A Java compiler, and accompanying run-time libraries, have been developed and experimentally verified against these identified aspects. In order to use safe languages as much as possible, and to increase efficiency in the compiler development, we used and evaluated a new OO compiler construction tool, which also enabled us to implement optimizations in a new way.

9.1 Real-Time Java

Many of the problems embedded systems developers are faced with, such as memory leaks and dangling pointers, originate from the use of unsafe low-level programming languages used. As the complexity of embedded systems software is constantly increasing (more functionality, more distributed, more flexible), there is clearly need for language

support to manage the complexity (encapsulation), and to detect programming errors as early as possible (safe languages).

Using Java for developing embedded real-time systems can shorten development time, and also improve the quality of the resulting software application. In order to make Java a viable programming language for embedded real-time systems development, the following key aspects have been addressed:

Portability We have successfully ported the run-time environment to five different thread models (of which four are hard real-time), executing on four different CPU types ranging from small 8-bit CPUs, such as the Atmel AVR, to high-end embedded CPUs, such as the PowerPC and the x86. Due to the diversity of both thread models and CPU types, we find portability being accomplished and that porting the run-time environment to yet more platforms is quite easily done.

Scalability We have shown, by experimental verification, that our compiled real-time Java scale down pretty well. Multi-threaded Java applications have been run successfully on platforms with such severe resource constraints as having only 128 KB ROM and 32 KB RAM (AVR).

Hard Real-Time Execution and Performance By natively compiling Java, and adding support for RTGC, hard real-time determinism has been achieved, and verified experimentally. The current GCI prototype implementation does impose significant overhead, hampering general execution performance. This overhead is particularly due to frequent GC synchronization in the compiled code. GC synchronization overhead can be decreased by a combination of; synchronizing less frequently, more efficient synchronization implementation, and utilizing compile-time knowledge about the threads' priorities and run-time behavior, to generate tailored synchronization schemes. Some ideas on how to improve general performance is proposed as future work.

Hard Real-Time Communication For distributed embedded real-time systems, the ability to communicate with other nodes with timing guarantees is by definition very important. By using a real-time network protocol in conjunction with a real-time Java application, hard real-time communication can be achieved, as has been shown in [GN04].

Applicability Experiences clearly indicate that, by providing flexibility in the choice of GC algorithm and run-time libraries, real-time Java can be made applicable for many different applications. Linking a Java application with external non GC-aware code modules is feasible also in hard real-time systems, if some care is taken when choosing a GC algorithm.

We have found an inherent limitation associated to linking the Java application to legacy, non GC-aware, code modules, where a tradeoff must be made between thread latency and timing predictability. In all other cases are our initial requirements fulfilled, and we can argue that Java can be made feasible for implementing hard real-time systems.

9.2 Compiler Construction

For the construction of a Java compiler, academic state-of-the-art tools based on RAGs and AOP techniques were used. The compiler was constructed in a modular fashion, with a number of aspects for the JastAdd tool, comprising the normal phases of a compiler; static semantic analysis, optimizations, and code generation.

Having implemented a compiler for a complete modern OO programming language, using the JastAdd tool, we have drawn the following conclusions:

- The OO fashion of the generated AST and the use of semantic equations renders a very compact, yet apprehensible, compiler implementation.
- Code analysis, refactorings, and optimizations, can be conveniently described as aspects performing computations and transformations on an OO AST.
- Although substantially slower to compile Java applications than other Java compilers (javac and gcj), it is still fast enough to build embedded software using standard workstations.

9.3 Contributions

The research contributions in this thesis are related to the two different research areas, compiler construction and real-time Java. The listing below is thus divided into two sections to reflect the different research areas.

Compiler Construction

- A compiler for a complete real-world object-oriented programming language, Java, has been constructed, and experimentally verified, using reference attributed grammars and aspect-oriented programming tools.
- A new way of implementing high-level code optimizations is devised. Using RAG and AOP techniques, code transformations can be very conveniently implemented as node transformations on the AST.
- Especially for object-oriented languages, where very complex expressions are possible, intermediate or back-end code generation can be very difficult. We have implemented simplifying code transformations using RAGs and AOP techniques, which appears to be easier and more elegant than creating a complex code generator.

Real-time Java

- A prototype implementation of real-time Java, showing that Java (based on J2SE) is a viable programming language, also for severely resource-constrained, real-time systems. This is, to our knowledge, the very first implementation of compiled (efficient) hard real-time Java.
- The transparent Garbage Collector Interface (GCI), which makes it possible to generate, or write, code independent of which type of GC algorithm should be used.
- An implementation of the Java exception mechanism which can be used together with an incremental RTGC.
- The *Latency* versus *Predictability* tradeoff, concerning the use of non-GC-aware (legacy) code in a real-time Java application, is brought forward.

9.4 Concluding Remarks

With these conclusions and contributions, we can now look back at the problem statement in Section 1.3 with more confidence and rephrase the original question as a statement:

> *Standard Java can be used as a programming language on arbitrary hardware platforms with varying degrees of real-time-, memory footprint-, and performance demand.*

To make this possible, enhancements to the Java semantics need to be made. The Java application should be natively compiled in order to meet memory footprint and performance requirements. In order to meet real-time requirements, support for (and synchronization with) RTGC is needed.

An inherent limitation of real-time memory management raises a tradeoff to be made between latency and predictability when external, non GC-aware code is needed.

During the implementation of a prototype compiler, we have found that compiler construction tools based on AOP and RAGs are very beneficial to the compiler development in terms of encapsulation and expressiveness, and thus also increasing code readability and quality.

> And then, finally, due to our contributions and experiences from a prototype implementation, there are strong reasons to believe that our main goal is well within reach:
>
> *Write once run anywhere, for resource-constrained real-time systems, is feasible and can become mature enough for industrial usage in a near future.*

Bibliography

[AHR00] Matthew Arnold, Michael Hind, and Barbara G. Ryder. An empirical study of selective optimization. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing. (LCPC '00)*, August 2000.

[Atm03] *ATmega128(L) Preliminary (Complete)*, December 2003. *http://www.atmel.com*.

[Bak03] Lars Bak. Object-oriented virtual machine for embedded systems., 2003. `http://www.oovm.com`.

[BBD+00] Greg Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[Big98] Lorenzo A. Bigagli. Real-time java, - a pragmatic approach. Master's thesis, Department of Computer Science, Lund Institute of Technology, October 1998.

[Blo01] Anders Blomdell. Efficient Java Monitors. In *The Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, pages 270–276. IEEE Computer Society, IEEE Computer Society, May 2001.

[Boe81] W. Boehm, Barry. *Software Engineering Economics*. Prentice Hall PTR, October 1981. ISBN: 0138221227.

[BW01] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages (Third Edition) Ada 95, Real-Time Java and Real-Time POSIX*. Addison Wesley Longmain, March 2001. ISBN: 0201729881.

[Con00] J Consortium. Real-Time Core Extensions. P.O. Box 1565, Cupertino, CA 95015-1565, September 2 2000.

[DMN68] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. SIMULA 67 Common Base Language. Technical report, Norwegian Computing Center, 1968.

[DN76] Ole Johan Dahl and Kristen Nygaard. *SIMULA – A language for Programming and Description of Discrete Event Systems*. Norwegian Computing Center, Oslo, Norway, 5th edition, September 1976.

[EH04] Torbjörn Ekman and Görel Hedin. Rewritable Reference Attributed Grammars. Olso, June 2004. Presented at the 18th European Conference on Object-Oriented Computing (ECOOP) 2004.

[Ekm00] Torbjörn Ekman. A real-time kernel with automatic memory management for tiny embedded devices. Master's thesis, Department of Computer Science, Lund Institute of Technology, November 2000.

[FKR+99] Robert Fitzgerald, Todd B. Knoblock, Erika Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for java. Technical report, Microsoft Research, 1 Microsoft Way Redmond, WA 98052, June 1999.

[FSM04] FSMLabs. FSMLabs - the RTLinux Company. `http://www.fsmlabs.com/`, 2004.

[gcj] The GNU compiler for the Java programming language.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1st edition, August 1996.

[GN04] Patrycja Grudziecka and Daniel Nyberg. Real-Time Network Communication in Java. Master's thesis, Department of Computer Science Lund University, 2004.

[GRH03] Sven Gestegård Robertz and Roger Henriksson. Time-triggered garbage collection — robust and adaptive real-time gc scheduling for embedded systems. In *Proceedings of the ACM SIGPLAN Languages, Compilers, and Tools for Embedded Systems - 2003 (LCTES'03)*, pages 93–102. ACM SIGPLAN, ACM Press, June 2003.

[HCÅÅ02] Dan Henriksson, Anton Cervin, Johan Åkesson, and Karl-Erik Årzén. On Dynamic Real-Time Scheduling of Model Predictive Controllers. In *Proceedings of the 41st IEEE Conference on Decision and Control*, Las Vegas, Nevada, 2002.

[Hed99] Görel Hedin. Reference attributed grammars. Presented at WAGA99, March 1999.

[Hen98] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, Lund Institute of Technology, July 1998.

[HFA^{+}99] Scott E. Hudson, Frank Flannery, C. Scott Anaian, Dan Wang, and Andrew W. Appel. Cup parser generator for java. *http://www.cs.princeton.edu/ appel/modern/java/CUP/*, 1999.

[HM02] Görel Hedin and Eva Magnusson. The JastAdd system - an aspect-oriented compiler construction system. *SCP - Science of Computer Programming*, 1(47):37–58, November 2002. Elsevier.

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley Pub Co, 1st edition, October 2003. ISBN 0-321-15491-6.

[IBE^{+}02] Anders Ive, Anders Blomdell, Torbjörn Ekman, Roger Henriksson, Anders Nilsson, Klas Nilsson, and Sven Gestegård-Robertz. Garbage collector interface. In *Proceedings of NW-PER 2002*, August 2002.

[Ive03] Anders Ive. Towards an embedded real-time java virtual machine. Licentiate thesis, Department of Computer Science, Lund Institute of Technology, 2003.

[JW98] Mark S. Johnstone and Paul R. Wilson. The Memory Fragmentation Problem: Solved? In *International Symposium on Memory Management, Vancouver B.C. (ISMM 98)*, pages 26–36. ACM, October 1998.

[KHH^{+}01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001. *http://eclipse.org/aspectj/*.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Published by Springer-Verlag New York Inc.

[Lia99] Sheng Liang. *The java Native Interface*. Addison-Wesley, 1999. ISBN 0-201-32577-2.

[Map] Waterloo maple inc. *http://www.maplesoft.com*.

[Mat] Mathworks inc. *http://www.mathworks.com*.

[MC60] John Mc Carthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4), April 1960.

[Me04] Paolo Mantegazza et.al. Rtai - the real-time linux application interface for linux.
`http://www.aero.polimi.it/ rtai/`, 2004.

[Men03] Fransisco Menjíbar. Portable Java compilation for Real-Time Systems. Master's thesis, Dep. of Computer Science Lund University, September 2003.

[Met] Java-cc parser generator.
Metamata Inc. *http://www.metamata.com*.

[Mod] Modelica. *http://www.modelica.org*.

[MRC+00] "F. Merillon, L. Reveillere, C. Consel, R. Marlet, and G. Muller". Devil: An idl for hardware programming. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 17–30, San Diego, California, October 2000. USENIX.

[NBL98] Klas Nilsson, Anders Blomdell, and Olof Laurin. Open Embedded Control. *Real-Time Systems*, 14(3):325–343, May 1998.

[NBPF96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly, 1st edition, September 1996. ISBN: 1-56592-115-1.

[NE01] Anders Nilsson and Torbjörn Ekman. Deterministic Java in Tiny Embedded Systems. In *The Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, pages 60–68. IEEE Computer Society, IEEE Computer Society, May 2001.

[NEN02] Anders Nilsson, Torbjörn Ekman, and Klas Nilsson. Real java for real time – gain and pain. In *Proceedings of CASES-2002*, pages 304–311. ACM, ACM Press, October 2002. To be presented.

[NL98] Kevin Nilsen and Steve Lee. PERC real-time API, July 1998. *http://www.newmonics.com*.

[Per00] Patrik Persson. Predicting time and memory demands of object-oriented programs. Licentiate thesis, Department of Computer Science, Lund Institute of Technology, April 2000.

[RTJ] Simplertj. *http://www.rtjcomm.com*.

[SBCK03] Ulrik Pagh Schultz, Kim Burgaard, Flemming Gram Christensen, and Jørgen Lindskov Knudsen. Compiling java for low-end embedded systems. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 42–50. ACM, ACM Press New York, NY, USA, June 2003.

[Sie99] Fridtjof Siebert. Hard real-time garbage collection in the jamaica virtual machine. In *The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA '99)*, Hong Kong, December 1999. IEEE.

[Sie00] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for java. In *Compilers, Architectures and Synthesis for Embedded Systems (CASES 2000)*, San José, November 2000.

[Str00] Bjarne Stroustrup. *The C++ Programming Language (Special Edition)*. Addison-Wesley Pub Co, February 2000. ISBN 0-201-70073-5.

[Sun00a] Java 2 platform micro edition (j2me) technology for creating mobile devices, May 2000. Sun Microsystems Inc. White Paper. *http://www.java.sun.com*.

[Sun00b] Javacard 2.1.1 runtime environment specification, May 2000. Sun Microsystems Inc. *http://java.sun.com/products/javacard/*.

[SW01] Fridtjof Siebert and Andy Walter. Deterministic execution of java's primitive bytecode operations. In *Java Virtual Machine Research & Technology Symposium '01*, Monterey, CA, April 2001. Usenix.

[TSL03] Frank Tip, Peter F. Sweeney, and Chris Laffra. Extracting library-based java applications. *Communications of the ACM*, 46(8):35–40, August 2003.

[VSWH02] João Ventura, Fridtjof Siebert, Andy Walter, and James Hunt. HIDOORS - A high integrity distributed deterministic Java environment. In *7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 113–118. IEEE, April 2002. *http://www.hidoors.org*.

Appendix A

Acronyms

AG Attribute Grammar

AOP Aspect-Oriented Programming

API Application Programming Interface

AST Abstract Syntax Tree

GC Garbage Collect(ion | or)

GCC GNU Compiler Collection

GCJ GNU Compiler for Java

GCI Garbage Collector Interface

GUI Graphic User Interface

HAL Hardware Abstraction Layer

J2ME Java2 Micro Edition

J2SE Java2 Standard Edition

JDK Java Development Kit

JIT Just-In-Time

JNI Java Native Interface

JRE Java Runtime Environment

JVM Java Virtual Machine

OO Object-Oriented

OS Operating System

RAG Reference Attribute Grammar

RT Real-Time

RTAI Real-Time Application Interface for Linux

RTCE Real-Time Core Extensions

RTGC Real-Time Garbage Collect(ion | or)

RTSJ Real-Time Specification for Java

RTOS Real-Time Operating System

UI User Interface

WCET Worst-Case Execution Time

Appendix B

Java Grammar

Below is listed the Java context-free grammar used by the JastAdd tool to produce appropriate AST node types.

```
CompilationUnit ::= PackageDecl:IdDecl* ImportDecl* TypeDecl*;

abstract Access : Expr ::= [Env:Expr];// Transformed from Name
TypeAccess : Access ::= [Env:Expr] <Name:String> <Decl:TypeDecl> ;
ThisAccess : TypeAccess;
SuperAccess : TypeAccess;
ArrayAccess : Access ::= Env:Expr Expr <Decl:TypeDecl> ;
VarAccess : Access ::= [Env:Expr] <Name:String> <Decl:Variable> ;
MethodAccess : Access ::= [Env:Expr] Arg:Expr* <Name:String>
                          <Decl:MethodDecl>;
ConstructorAccess : Access ::= [Env:Expr] Arg:Expr* <Name:String>
                               <Decl:ConstructorDecl>;

Name : Access ::= [Expr] ParseName*;
ParseName ::= IdUse Dims*;
ParseMethodName : ParseName ::= IdUse Arg:Expr* Dims*;

ImportDecl ::= IdDecl* [Wildcard];
Wildcard ::= ;

abstract TypeDecl ::= Modifier* IdDecl BodyDecl*;
ClassDecl : TypeDecl ::= Modifier* IdDecl
                         [SuperClassAccess:Access]
                         Implements:Access* BodyDecl*;
PrimTypeDecl : ClassDecl ::= Modifier* IdDecl
                             [SuperClassAccess:Access]
                             Implements:Access* BodyDecl*;
InterfaceDecl : TypeDecl ::= Modifier* IdDecl
                             SuperInterfaceId:Access* BodyDecl*;
```

```
ArrayDecl : ClassDecl ::= Modifier* IdDecl
                          [SuperClassAccess:Access]
                          Implements:Access* BodyDecl*
                          <ElementType:TypeDecl> <Dimension:int>;

abstract BodyDecl;
InitializerDecl : BodyDecl ::= Modifier* Block;
ConstructorDecl : BodyDecl ::= Modifier* IdDecl Parameter*
                               Exception:Access* Block;

FieldDecl : BodyDecl ::= Modifier* TypeAccess VariableDecl*;
FieldDeclaration : BodyDecl ::= Modifier* TypeAccess IdDecl
                                [AbstractVarInit];
                                // Simplified FieldDecl

VarDeclStmt : Stmt ::= Modifier* TypeAccess VariableDecl*;
VariableDeclaration : Stmt ::= Modifier* TypeAccess IdDecl
                               [AbstractVarInit];
                               // Simplified VarDeclStmt

VariableDecl ::= IdDecl EmptyBracket* [AbstractVarInit];

Parameter ::= Modifier* TypeAccess IdDecl EmptyBracket*;
ParameterDeclaration : Parameter ::= Modifier* TypeAccess
                                     IdDecl;
                                     // Simplified Parameter

EmptyBracket;

abstract AbstractVarInit;
VarInit : AbstractVarInit ::= Expr;
ArrayInit : AbstractVarInit ::= AbstractVarInit*;

MethodDecl : BodyDecl ::= Modifier* TypeAccess IdDecl Parameter*
                          EmptyBracket* Exception:Access*
                          [Block]; // Create simplified version

abstract InnerType : BodyDecl ::= TypeDecl;
InnerClass : InnerType ::= ClassDecl;
InnerInterface : InnerType ::= InterfaceDecl;

IdDecl ::= <ID>;
IdUse ::= <ID>;

abstract Expr;

abstract AssignExpr : Expr ::= Dest:Expr Source:Expr;

AssignSimpleExpr : AssignExpr ;
AssignMulExpr : AssignExpr ;
AssignDivExpr : AssignExpr ;
```

```
AssignModExpr : AssignExpr ;
AssignPlusExpr : AssignExpr ;
AssignMinusExpr : AssignExpr ;
AssignLShiftExpr : AssignExpr ;
AssignRShiftExpr : AssignExpr ;
AssignURShiftExpr : AssignExpr ;
AssignAndExpr : AssignExpr ;
AssignXorExpr : AssignExpr ;
AssignOrExpr : AssignExpr ;

abstract PrimaryExpr : Expr;
abstract Literal : PrimaryExpr ::= <LITERAL>;
IntegerLiteral : Literal ;
FPLiteral : Literal ;
BooleanLiteral : Literal ;
CharLiteral : Literal ;
StringLiteral : Literal ;
NullLiteral : Literal ;

ParExpr : PrimaryExpr ::= Expr;

StringLiteralExpr : PrimaryExpr ::= StringLiteral;

PrimTypeClassExpr : PrimaryExpr ::= <ID>;

abstract InstanceExpr : PrimaryExpr;
ClassInstanceExpr : InstanceExpr ::= Access Arg:Expr* [ClassDecl];
ArrayInstanceExpr : InstanceExpr ::= TypeAccess Dims* [ArrayInit];
Dims ::= [Expr];


abstract Unary : Expr ::= Operand:Expr;
PreIncExpr : Unary ;
PreDecExpr : Unary ;
MinusExpr : Unary ;
PlusExpr : Unary ;
BitNotExpr : Unary ;
LogNotExpr : Unary ;

CastExpr : Unary ::= TypeAccess Expr;

abstract PostfixExpr : Unary;
PostIncExpr : PostfixExpr ;
PostDecExpr : PostfixExpr ;

abstract Binary : Expr ::= LeftOperand:Expr RightOperand:Expr;

abstract ArithmeticExpr : Binary;
MulExpr : ArithmeticExpr ;
DivExpr : ArithmeticExpr ;
ModExpr : ArithmeticExpr ;
AddExpr : ArithmeticExpr ;
```

```
SubExpr : ArithmeticExpr ;

abstract BitwiseExpr : Binary;
LShiftExpr : BitwiseExpr ;
RShiftExpr : BitwiseExpr ;
URShiftExpr : BitwiseExpr ;
AndBitwiseExpr : BitwiseExpr ;
OrBitwiseExpr : BitwiseExpr ;
XorBitwiseExpr : BitwiseExpr ;

abstract RelationalExpr : Binary;
LTExpr : RelationalExpr ;
GTExpr : RelationalExpr ;
LEExpr : RelationalExpr ;
GEExpr : RelationalExpr ;
EQExpr : RelationalExpr ;
NEExpr : RelationalExpr ;

InstanceOfExpr : RelationalExpr ::= Expr TypeAccess;


abstract LogicalExpr : Binary;
AndLogicalExpr : LogicalExpr ;
OrLogicalExpr : LogicalExpr ;

QuestionColonExpr : Expr ::= Condition:Expr TrueExpr:Expr
                             FalseExpr:Expr;

Modifier ::= <ID>;

// Statements

abstract Stmt;
Block : Stmt ::= Stmt*;
EmptyStmt : Stmt;
LabelStmt : Stmt ::= Label:IdDecl Stmt;
ExprStmt : Stmt ::= Expr;

SwitchStmt : Stmt ::= Expr Case*;
abstract Case;
ConstCase : Case ::= Value:Expr Stmt*;
DefaultCase : Case ::= Stmt*;

IfStmt : Stmt ::= Condition:Expr Then:Stmt [Else:Stmt];

WhileStmt : Stmt ::= Condition:Expr Stmt;

DoStmt : Stmt ::= Stmt Condition:Expr;

ForStmt : Stmt ::= InitStmt:Stmt* [Condition:Expr]
                   UpdateStmt:Stmt* Stmt;
```

```
BreakStmt : Stmt ::= [Label:IdUse];
ContinueStmt : Stmt ::= [Label:IdUse];

ReturnStmt : Stmt ::= [Result:Expr];

ThrowStmt : Stmt ::= Expr;

SynchronizeStmt : Stmt ::= Expr Block;

TryStmt : Stmt ::= Block Catch* [Finally:Block];
Catch ::= Parameter Block;


abstract TypeDeclStmt : Stmt ::= TypeDecl;
ClassDeclStmt : TypeDeclStmt ::= ClassDecl;
InterfaceDeclStmt : TypeDeclStmt ::= InterfaceDecl;
```