# Tailoring native compilation of Java for real-time systems

# Tailoring native compilation of Java for real-time systems

**Anders Nilsson**

Doctoral Dissertation, 2006

Department of Computer Science
Lund University

ISBN 91–628–6830–6
ISSN 1404–1219
Doctoral Dissertation 26, 2006
LU–CS–DISS:2006–3

Department of Computer Science
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 $2_\varepsilon$

Cover artwork by Helena Persson

Printed in Sweden by Tryckeriet i E-huset, Lund, 2006

## 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 systems 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. Java was chosen as an example of a safe, object-oriented programming language that could benefit embedded systems development.

Defined key concepts, such as portability, scalability, and real-time performance, 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. A new transparent garbage collector interface makes it possible to generate, or write, C code independently of garbage collector algorithm. Experiments show that we achieve good results on real-time performance. Given our contributions and results we do see compiled real-time Java, or a similar language such as C#, as industrially viable.

We also propose a technique for multi-level deployment of embedded applications, by taking advantage of the fact that Java source code has a well defined meaning.

The same compiler construction techniques and tools used for implementing our Java compiler also appears to be suitable on a higher abstraction level. This is exemplified with with a prototype of a compiler-compiler on the ontology level.

# Acknowledgements

First of all, I wish to thank my supervisors; Klas Nilsson who introduced me to real-time Java and has given me invaluable advice and feedback on various real-time issues, Görel Hedin who introduced me to compiler construction and reference attributed grammars, and professor Boris Magnusson who is the head of the research group.

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

A very special thank you goes to Anders Blomdell at the Department of Automatic Control. Your expert knowledge in real-time operating systems, compilers, device drivers, and other nasty low-level implementation details has been invaluable when I have done stupid things.

I am also grateful to all those students who in various projects have contributed code to various parts of the compiler and run-time system, as well as pinpointed a lot of bugs which could then be fixed. Emma Nyman, 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.

The work presented in the thesis has been financially supported by VINNOVA[1], the Swedish Agency for Innovation Systems, and SMErobot, a project funded under the EU's sixth framework programme.

[1]*FLEXCON* and *Network Programming* projects.

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

*Lund, May 2006*

*Anders*

# Contents

# List of Figures

# List of Listings

# List of Tables

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 — comprise less than ten percent of the total. That is, more than 90% of all computers are embedded. This percentage is 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 such as C, resulting in extensive engineering needed for development and debugging as the programs grow bigger. Software modules do not become flexible from a reuse point of view since they are handcrafted for a certain type of application or target system.

- As embedded systems software 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 in an uncontrollable manner.

- Embedded systems become more and more distributed, consisting of several communicating nodes instead of one central processor. It would be very beneficial to make use of available Internet technologies, but with the extension that both computing and communication must comply with strict timing requirements.

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 require 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 deserve 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 decrease, resulting in complexity being the main obstacle for further development. Weak structuring mechanisms in the programming languages used make the situation worse.

**Managing System Development** Software development projects tend to be behind schedule. Software errors found late in the project make 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 more suitable structuring mechanisms than are 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 using simpler language conventions, tools, macros, libraries, and the like. 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, full object-oriented support from the programming language is needed.

### Implications of unsafe programming languages

Experiences from programming in industry and in 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 (using malloc/free). When should `free()` be called? 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, do often not 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:

- Type safety. For example, it is 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 objects are created (by calling the operator **`new`**) and automatically freed by a garbage collector when there no longer exist any live references to the object. An object cannot be freed manually.

The characteristics of safe languages usually make 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, platform neutral, object-oriented programming language available that has reached industrial acceptance. Not only for the previously mentioned qualities, but also for its platform independence[2].

The benefits of language safety 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. Also, the enhancements for real-time programming should be Java compatible and without violating the security of the programming language.

There exist 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 an 'RT.Net' (dot-net for real time) platform. One can also argue that Ada qualifies as a safe object-oriented programming language and run-time, but there is no strict sand-box model and the acceptance of Ada is very low outside the aerospace and military industry. So, 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,

[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.

standard Java and the GNU libraries are not targeted or suitable for real-time embedded systems, which brings us to the compiler technology issue.

## 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, 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 (AGs). 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 optimizations which can only be performed on a higher abstraction level. Examples of such transformations are method in-lining, and implicit finalization of classes and 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 Tailoring deployment and systems

The perhaps most important parts of embedded real-time systems perform sequencing or feedback control. By far the most control functions today are implemented in software. In the robotics research community for instance, most attention is paid to the implementation of the algorithmic parts of robot control systems. An industrial robot control systems, including all supervision, mode changes, error handling, support for modular testing, user interaction, shop-floor path adjustments, etc., consists of more than 1 million lines of code. Today this is typically written in C in a rather static manner. For future robot control, the demands on flexibility and safety is increasing and contradictory, so we may seek other means of implementation such as code generation from engineering tools. However, it is a small (albeit important) part of the system that is suitable for being efficiently generated from tools such as Simulink/RTW; we still need supporting tools and methods for developing and deploying hand-written control software (including the building blocks of the control engineering tools).

Another dimension of engineering, typically applied to each level of control, is the stepwise refinement of embedded control functions. As many engineers have experienced, if you try to implement and deploy the complete control system in one step it will not work. Instead, you have to be able to test and run smaller core parts of the system, and then gradually activate additional features. This applies to both control and software engineering of the system. Recall that a feedback control system depends on the true process dynamics and standard software testing techniques alone are not sufficient to ensure proper operation. While also stepwise *development* today is well understood in both the control and software engineering communities, we want to address the additional following issue:

- How can stepwise *deployment* of embedded control software be carried out such that the control properties (due to timing, latencies, etc.) are verified in multiple smaller stages?

Finally, assuming we can efficiently develop and deploy our control software, there is the issue of interconnecting and integrating our flexible systems. Then we have to deal with existing hardware, vendor-specific configurations, and legacy or proprietary software. This leads to a situation where we need to not only compile embedded software, but also system definitions on a meta-level may need compiler support. More as a prospect but not as the core of this work, we have looked at combining the concepts known as *Ontology* and the *Semantic Web* with our compiler construction tools. We believe this combination can be valuable for efficient handling of different kinds of system definitions.

## 1.4 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 core 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].

## 1.5 Thesis outline

The rest of this thesis is organized as follows:

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

**Chapter 3: An approach to real-time Java** presents our approach to compiling Java for use in real-time systems, possibly with limited resources.

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

**Chapter 5: A compiler for Java and real-time** describes the Java compiler being developed to accomplish real-time Java.

**Chapter 6: Experimental verification** presents some experiments performed using the Lund Java-based Real Time (LJRT) platform in order to see to what extent the ideas are applicable in reality.

**Chapter 7: Prospects** describes a few other projects where the tools and techniques presented in the thesis are used. These projects are still in an early state, but preliminary results look very promising.

**Chapter 8: Future work** contains the most interesting ideas for further work with the tools and techniques presented in the thesis.

**Chapter 9: Related work** gives short descriptions of some related real-time and embedded Java implementations.

**Chapter 10: Conclusions** presents conclusions drawn from the work presented in the thesis. A summary of the thesis contributions is also given.

[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.

## 1.6 Publications

This thesis largely consists of material from published papers. The ideas about how to approach real-time Java presented in Chapter 3 were originally published as a proof of concept in

> 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, May 2001.

and then in more developed and generalized form in

> 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.

The Garbage Collector Interface (GCI), which is briefly described in Chapter 4, was first published in

> Anders Ive, Anders Blomdell, Torbjörn Ekman, Roger Henriksson, Anders Nilsson, Klas Nilsson, and Sven Gestegård Robertz. *Garbage Collector Interface*. In Proceedings of NWPER 2002, August 2002.

The description of JastAdd and the LJRT compiler in Chapter 5 is a revised and extended version of the paper

> Anders Nilsson, Anders Ive, Torbjörn Ekman, and Görel Hedin. *Implementing Java Compilers Using ReRAGs*. Nordic Journal of Computing, 11(3):213–234, 2004.

The descriptions and experimental results from various optimizations in the run-time system and the LJRT compiler, found in Chapters 4 and 5, were published in

> Anders Nilsson and Sven Gestegård Robertz. *On Real- Time Performance of Ahead-of-Time Compiled Java*. In The Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005), pages 372–381. IEEE Computer Society, IEEE, May 2005.

The ideas and strategies about multi-stage deployment, presented in Chapter 7 are also published in

> Sven Gestegård Robertz, Anders Nilsson, Mathias Haage, and Klas Nilsson. *Multi-Stage Deployment of Robot Control Software*. To appear in Proceedings of the 8th International IFAC Symposium on Robot Control, SYROCO, September 2006.

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C. Clarke

# Chapter 2

# Preliminaries

ADVANCES within three research areas of computer science lay the foundation of this work, which has 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 when, given a set of inputs, the system will respond with the intended output. 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 of an unstable airplane.

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 promotes simulating 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, and especially for devices produced in large quantities, memory is an expensive resource. A trade-off 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 trade-off as for memory footprint has to be made.

**Determinism:** Many embedded devices need to fulfill 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

The concept of 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 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 on 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 decrease 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, to guarantee 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 involved in calculating the maximum memory allocation rate of an application is similar to calculating the Worst-Case Execution Time (WCET) for all threads in the application. In 2003, Gestegård-Robertz and Henriksson [RH03] 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. This is 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) differ 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 and Xenomai

The Real-Time Application Interface for Linux (RTAI) project [Me04], which originated some years ago as an open-source fork of the RT-Linux project [FSM04], aims at adding hard real-time support to the GNU/Linux operating system. In October 2005 Xenomai [Xen06], another open source project which had earlier joined forces with the RTAI developers in the RTAI Fusion branch, again broke lose from RTAI as a result of the respective lead developers not agreeing upon how to continue development. The differences between the two projects are as of yet mostly hidden deep inside the low-level implementation, and is barely visible in the structure or the API.

The implementation of Xenomai is modelled as three layers stacked on top of each other, see Figure 2.1:

**HAL** The Adeos Hardware Abstraction Layer (HAL) module consists of a patch to the Linux kernel adding hooks into the interrupt handling routine, and a loadable kernel module that uses the hooks to intercept interrupts before they arrive at the kernel. By controlling the interrupt handling mechanism, Xenomai gains full control over the execution of the Linux kernel.

**Nucleus** The nucleus module is an abstract real-time kernel implementation providing a scheduler and all the RTOS primitives that are

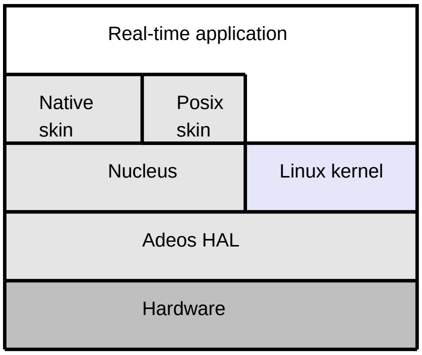


**Figure 2.1:** *Schematic view of the Xenomai structure.*

needed. The scheduler runs the Linux kernel as it's idle task. Functions in the nucleus are not called directly from real-time tasks, but should only be called from the different skins on top of the nucleus.

**Skins** On top of the nucleus there can be several "skins" each defining an API for the real-time tasks to use. Currently available in Xenomai are one Xenomai skin and one Posix skin. The skin concept makes it easier to port applications from other run-time environments to Xenomai.

There are two execution environments available for Xenomai tasks. The traditional way to execute RTAI or Xenomai tasks is in form of loadable Linux kernel modules. The real-time tasks execute alongside the Linux kernel in the same address space, with no memory protection available to stop a badly behaving task from thrashing the Linux kernel's state. Other problems with executing as a kernel module are that only a subset of the standard C library functions is available, and debugging is much more awkward. The RTAI project then developed a way to execute real-time tasks in user-space, called LXRT. Now normal memory protection is used and the real-time tasks have access to the same libraries as any other Linux process (but care must then be taken, so that not task execution times become unpredictable). Originally, tasks running in LXRT experienced slightly longer latencies than kernel-space tasks, but the difference has decreased since then. Running Xenomai tasks in user-space is now the preferred way, but the option to run in kernel-space is still present.

Xenomai threads are scheduled by the strict priority based Xenomai 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 Xenomai 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

Object-Oriented (OO) languages have over the years proven to be a valuable programming technique ever since the concept was first implemented in the 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 a possible solution to many programming problems that 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.

## 2.5 Reference attributed grammars

Ever since Donald Knuth published the first paper [Knu68b] 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) [Hed00], 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 RAG concept.

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 Java for real-time

In Section 2.1, we have identified a number of key concepts which need to be fulfilled in order to make Java a viable programming language for embedded real-time systems.

In this chapter, we will discuss the suitability of different execution strategies for Java applications in real-time environments. We will then present and discuss the approach we have chosen, and the rationales for why we believe it is the best strategy in order to use Java in embedded real-time systems.

## 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 — Java Virtual Machine (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.

A survey, see Chapter 9, of available JVMs, concerning their usefulness for embedded and real-time systems, reveals two major problems with the interpreted solution. First, JVMs are in general too big in

**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;
  }
}
```

terms of memory footprint. Second, they are too slow, in terms of performance. This leads to the alternative approach to use a conventional execution model with a binary compiled for a specific CPU. Of course, there are advantages with the JVM alternative but 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 next page. Note that the referred structures that implement the actual object modeling are left out.

The code shown in Listing 3.2 on the facing 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 severely.

### Example with GC

Using an incremental compacting GC in the run-time system, the C code in Listing 3.2 on the previous 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 facing 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

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

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 would move 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 next page.

This can be a valid technique if the enabling/disabling of preemption can be made cheap enough. On the hardware described in Section 6.3 on page 104, 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 acceptable 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 LJRT 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.

A more in-depth discussion on optimizations implemented in the Java compiler is found in Section 5.3.3, 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 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 suffixed 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. Listing 3.5.

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, numerous 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. No read barrier is needed, and average performance will typically be improved. However, when dynamically allocating several object sizes the allocation predictability will be as poor as in C/C++.

**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;
}
```

## 3.6 Findings

Inclusion of external (non GC-aware code in a real-time Java system raises a trade-off 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!

Last words of 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 of the underlying run-time platform. It will also need to cooperate closely with the memory management system in such a way that timing predictability is accomplished, while memory consistency is maintained at all times.

Due to external requirements, we need our system to operate in an uncooperative environment; we want to be able to use an off-the-shelf C compiler and RTOS as well as external, legacy, or automatically generated C code. That means that we cannot rely on detailed assumptions on the behavior of the back-end C compiler or the thread scheduler, which makes implementation of a real-time GC more challenging. For instance, it means that any synchronization required between collector and application, or *mutator*[1], needs to be done explicitly. It also means that the generated C code must be written so that it ensures, in a portable way, that no back-end optimization causes interference with the GC.

In particular, the combination of uncooperative compiler, uncooperative scheduler, and tight real-time requirements (low latency) makes the problem challenging. Without control over the scheduling, some

[1]From the GC's point of view, the application is a process that changes, or mutates, the object-reference graph, causing objects to become garbage.

compiler optimizations cannot be allowed since threads may be preempted at any time. For instance, if we are using a copying or compacting GC algorithm, pointers must always be read from memory, and not kept in registers, as the collector may move objects at (from the mutator's point of view) any time.

Thus we need to investigate how the Java constructs, including automatic memory management, could be implemented in an efficient and portable way on a standard (RT)OS. This chapter will first describe the concept of real-time garbage collection for a natively compiled Java application, and the generic garbage collector interface. Then follows considerations concerning 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, see for example Jones and Lins [JL96], 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. It ranges from the top user layer, via threading and debug layers, to the implementation layer. The two middle layers can be switched on/off to support GCI debugging and/or multi-threaded execution.

### 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.

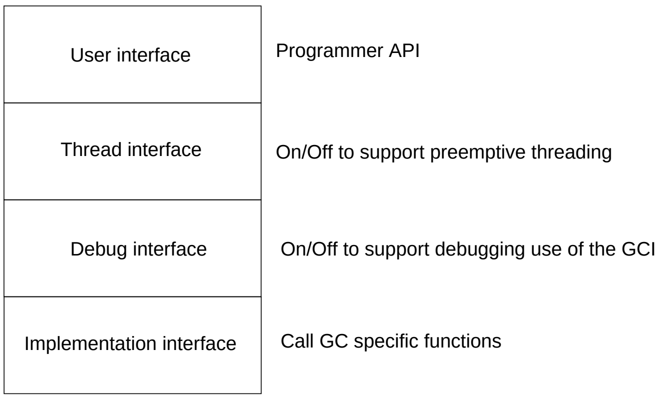

User interface
Programmer API
Thread interface
On/Off to support preemptive threading
Debug interface
On/Off to support debugging use of the GCI
Implementation interface
Call GC specific functions

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

**Object layout declaration:** Macros used for declaration of object type structs, struct members, and struct layouts.

**Reference declaration:** Declaration of reference variables, push/pop references on the 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 operations 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.

There are currently three different GC algorithms implemented in the run-time environment: one *incremental mark-copy* (compacting) and one *incremental mark-sweep* (non moving) that both support hard real-time, and one traditional *batch-copy* algorithm.

## 4.2 Performance issues with the GCI

The key requirements on a run-time system for real-time applications are predictability and low latency, and the real-time properties of our approach have been previously verified [Hen98, NEN02, Rob06]. However, for a system to be practically feasible, the inlined overhead must not be unacceptably large. This section discusses how to achieve good general execution performance while maintaining the hard real-time properties.

As our Java system needs to operate in an uncooperative environment, it must ensure that correct behavior and real-time performance is not jeopardized by compiler optimizations, concurrency issues or interference from external code. In isolation, each of these aspects do not pose a problem; the difficulty comes from the combination, which gives conflicting requirements. According to our experience, the main bottleneck is the synchronization between mutator and collector.

Under an uncooperative scheduler, preemption can occur at any instant. Therefore, all reference operations must be protected to ensure mutual exclusion between collector and mutator and, if we want low latency, the critical sections must be small. However, this means a lot of synchronization, which may add up to a significant execution time overhead. It should be noted that this problem is due to the uncooperative scheduler, and not the ahead-of-time compilation; a JVM using native threads would face the same problems if a non-intrusive concurrent GC was desired.

The execution time overhead can be reduced in two basic ways: decrease the number of operations that require synchronization or using less expensive synchronization primitives.

### 4.2.1 Reducing the need for synchronization

The level of required synchronization is affected both by the choice of GC algorithm (e.g., if a read barrier is required or not) and by different implementation decisions in the compiler and run-time system. This section gives examples of how those issues can be addressed in the run-time system. Compiler related GCI synchronization optimizations are addressed in Section 5.5.

**Function calls:** For function calls, the level of locking required depends on how reference arguments are passed — as references or as actual pointers (i.e., if the read barrier is executed in the caller or in the callee). In our implementation, reference structures are stack allocated and thus will not be moved by the GC. Therefore, if references are called by reference (i.e., a pointer to the reference structure is passed) no new roots are pushed in the callee and no heap locking is required. As the caller will always out-live the callee, parameters to functions are known to be rooted in the calling context and do not have to be rooted again in the called context. Similarly, we know that the return value of a function will be used in the calling function (or not at all). Therefore, the variable that will receive the return value must already be rooted so if we pass a reference to this variable to the called function, it can be assigned before the return which removes the need to protect the return value. If function arguments and return values are handled in this way, no locking is required for function calls.

**Root stacks in multi-threaded programs:** Another example of overhead caused by an uncooperative environment is the presence of root stacks. In multi-threaded programs, each thread has its own root stack, and therefore, all root operations (i.e. push and pop) requires a pointer to the root stack of the current thread. In a system where the thread scheduler is Java-aware, the root stack pointer is part of the execution context of each thread and is saved and restored automatically.

In systems which cannot rely on scheduler cooperation, this has to be handled in the application code. As the root operations are part of the application code, and the current thread is not known at compile time, this must be looked up at run time. However, looking up the root stack at each root operation is quite inefficient so this should be done once for each function call and cached. Similarly, if no root operations are done in a function (like in e.g. a typical math function of the standard library), such lookup is unnecessary. Therefore, lookup of the thread root stack is done lazily at the first root operation of each function and the result is cached. This can be implemented quite efficiently.

### 4.2.2 Reducing the cost of synchronization

With fine-grained memory operations and heap-intensive applications, such as Java programs, the heap is almost always locked, so whenever preemption occurs, the probability that the heap is locked is high. Assume that a thread (`T1`) is executing and is in the middle of a memory operation. Then, a context switch occurs; the thread that is scheduled to run (`T2`) will probably try to lock the heap very soon after the context switch and be blocked. Then `T1`, which is holding the heap lock, is scheduled to run again until it releases the heap lock, allowing `T2` to continue its execution. This means that there will be three context switches instead of one, increasing the execution time overhead due to such *context switch chatter*.

Low latency due to locking is a requirement, so just increasing the size of the critical sections is not a viable solution. Therefore, we need a solution that allows very fine-grained preemption without the overhead of frequent unlocking and re-locking. We also need to make sure that context switches are not performed when the heap is locked.

Three possible solutions were found based on turning off interrupts, preemption points, and lazy locking.

**Turning off interrupts:** The straight forward solution is to implement `gc_lock()` by turning off (clock) interrupts and `gc_unlock()` by turning them on again. On most architectures, interrupt requests that arrive when interrupts are masked are latched, so that when the interrupts are turned back on, any missed interrupt will be generated and the corresponding interrupt routine is executed. On such an architecture, this will give the desired semantics that if a time-slice ends, and preemption should take place, when the heap is locked, the context switch is delayed until the heap lock is released. Turning off interrupts may, however, not be allowed by the OS, or have negative effects on other parts of the system, such as interrupt-based drivers for peripherals, etc.

**Preemption points:** By using a scheduler which only allow preemption at certain pre-determined points in the execution flow, frequent locking/unlocking can be avoided. In fact, if the memory accesses are taken into account when placing preemption points so that preemption is only allowed when the heap is in a consistent state, no additional housekeeping or synchronization is needed in order to ensure correct GC operation.

However, preemption points are problematic for two reasons. The first is that most standard real-time operating systems don't support them. The second one is that calling external native code (that doesn't have preemption points) may cause priority inversion. An illustrating example is a background thread calling an external routine with a long execution time. As external code doesn't have preemption points, high priority threads may be delayed indefinitely. One solution is switching to "native" preemption when calling external code and then switching back to preemption-points when executing known code. However, calling external code would then have a performance penalty due to the additional housekeeping required and scheduler implementation would be more complex.

**Lazy locking:** Since turning off interrupts or using preemption points turned out to be less suitable, an alternative strategy for reducing the locking overhead was proposed. This is based on the observation that, while the frequent locking and unlocking is required in order to achieve low latency, in the common case the heap is unlocked and then shortly re-locked by the same thread. Thus, most of the locking operations are really unnecessary and could be removed without changing the behavior of the program (other than reduced overhead). The problem is just determining which lock and unlock operations that need to be performed. This could be done statically, but the analysis would be difficult

```
          gc_lock();
          ...
-->       gc_unlock();
-->       gc_lock();
-->       ...
-->       gc_unlock();
-->       gc_lock();
          ...
          gc_unlock();
```

**Figure 4.2:** *Locking example: Small atomic operations cause frequent locking.*

and highly dependent on the low-level scheduling, control flow based on input data, etc. Therefore, a dynamic, on-line approach is preferable.

For example, take a code sequence like in Figure 4.2. If we are executing in the marked region, and no clock interrupt has arrived (i.e., the thread will not yet be preempted), it is unnecessary to perform the unlocking and re-locking operations. Thus, if we could dynamically decide whether to perform the unlock/lock operations (in a way that is much cheaper than actually performing the locking), the overhead could be reduced. Then, when a clock interrupt occurs, the heap should really be unlocked at the next unlock instruction and the context switch performed.

One way of implementing this is by having two versions of the operations: the actual lock/unlock operations (which are executed when the locking is required) and "NOP" versions that are used when unlocking and re-locking isn't necessary. Then, the run-time system ensures that the correct version is run at each time to both guarantee the correct semantics and achieve the best performance. In principle, an implementation of this scheme looks like in Figure 4.3. This method gives similar behavior as preemption points with regard to heap accesses, but without requiring additional housekeeping in order to allow external native code to be run with real-time guarantees.

If modifying the scheduler is not possible, or practically feasible, much of the benefit of lazy locking can still be obtained if the OS has a call-back hook for a method to be called at context switches. In fact, this is the method used in our Linux/RTAI prototype, and it gives the same reduction of the number of locking operations, but does not address context switch chatter. That may, however, be a reasonable trade-off for not having to modify the scheduler.

There are, of course, many other small details that must be taken

```
void (*gc_lock)(void);
void (*gc_unlock)(void);

void gc_lock_real(void)
{   lock(heap_mutex);
    gc_lock   = f_nop;
    gc_unlock = f_nop;
}
void gc_unlock_real(void)
{   unlock(heap_mutex);
    yield();
}
void f_nop(void) { return; }
void reschedule(void)
{   if(is_locked(heap_mutex)) {
       gc_lock   = gc_lock_real;
       gc_unlock = gc_unlock_real;
      } else {
         /* perform actual context switch */
    }
}
```

**Figure 4.3:** *Lazy locking implementation sketch*

care of when implementing such a scheme; e.g., the system must ensure that the heap is always unlocked before a blocking call is made or before a thread dies; otherwise there is a risk of deadlock.

### 4.2.3 Evaluation

The experiment setup used in sections 4.2.3 and 4.2.3 was a low level servo controller for an ABB IRB-2000 industrial robot. Given a desired motor angle for each of the six joints, suitable torque values and the corresponding AC motor currents are calculated. Both servos executed on a 350 MHz PowerPC G3 with 32 MB RAM running Linux/RTAI.

#### Lazy locking

This experiment, consisting of one part of an industrial robot controller, investigates the impact of lazy locking on the number of lock operations that are actually performed. Figure 4.4 shows the frequencies of locks in the vanilla version and real and lazy locks in the lazy version. This shows that only a small fraction of the locks actually need to be performed and thus that the locking overhead can be significantly re-

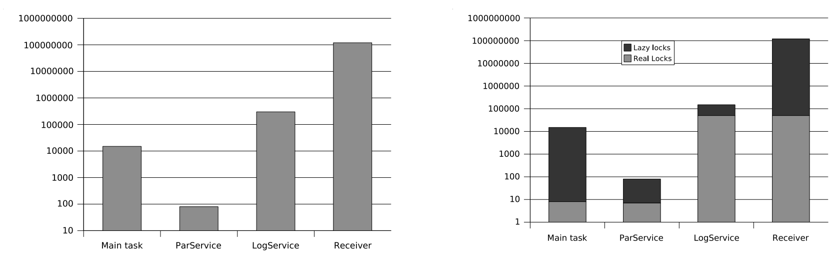


**Figure 4.4:** *Comparing the vanilla version (left) to the one with lazy locks (right), showing the frequencies of real and lazy locks for each of the application threads. Please note that the scale is logarithmic. In this experiment, the mark-compact collector was used. The application was run for a fixed amount of time, so the numbers should not add up.*

duced. For instance, in the receiver thread, which performs most of the computations, only $0.03\%$ of the lock instructions in the code actually cause a mutex operation.

### GC algorithm, synchronization mechanism and root alias optimization

Figure 4.5 shows how the choice of locking primitive and the root alias optimization affects total throughput, i.e. the maximum possible sample rate. The big difference between the mark-sweep and the mark-compact collector is caused by the extra synchronization required for the read barrier in the mark-compact case. With synchronization turned off[2], there is no big difference between a moving and a non-moving collector. In this example, the overhead of the read barrier is compensated by the cheaper allocation[3].

In this experiment, lazy locking was implemented with the call-back method, instead of modifying the scheduler, so it only shows the savings from doing fewer locks. It remains to be investigated how large the effects of context switch chatter are. The call-back method also adds to the overhead of each context switch, so an implementation inside the scheduler would yield a much bigger improvement.

[2]Of course, running without synchronization is not safe and may cause race conditions and memory corruption, so this is done for reference only and is not a practically usable configuration.

[3]In the mark-compact collector, allocation is done by simply incrementing a pointer, whereas in the mark-sweep case, free-list search and block splitting is done.

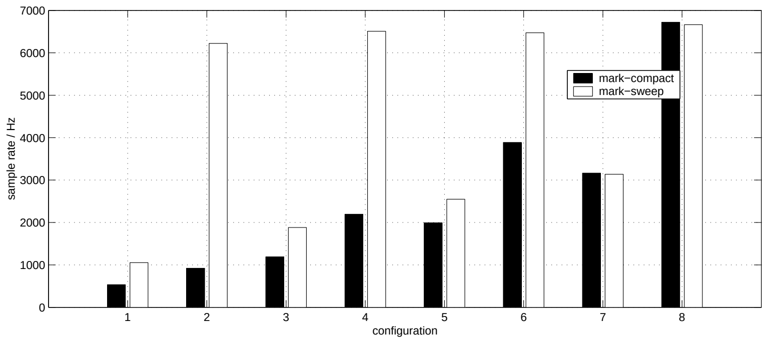


**Figure 4.5:** *The effect on throughput of different locking primitives and root optimization. The configurations are 1) Mutex locking, 2) Mutex locking with root alias optimization, 3) Lazy mutex locking, 4) Lazy mutex locking with root alias optimization, 5) Interrupt masking (cli/sti), 6) Interrupt masking with root alias optimization, 7) No locking and 8) No locking with root alias optimization*

### 4.2.4 Compiler optimization effects

Another problem with locking is that the lock/unlock operations are function calls or inline assembler, and that tend to break basic blocks and interfere with compiler optimizations. This is, partly, intentional, as many optimizations are not safe in the general case. E.g., we must make sure that pointers (gotten through the read barrier) to objects are always read from memory as objects may have moved since the last access, etc., when we enter the next critical section, and such race conditions will lead to memory corruption.

However, this is really only needed when a context switch actually has taken place; as long as the same thread is executing, any optimization is legal, as long as the heap and all references in memory are consistent at the next context switch. Thus, performance could be improved significantly if it was possible to implement lazy locking in a way that the fast case did not break basic blocks. We believe that this could be done with self-modifying code, injecting the lock/unlock operations into the code where they are needed and modifying the lock/unlock instructions so that they ensure heap consistency. This, of course, requires detailed information about the inner workings of the optimizing back-end and target architecture and can hardly be done in a simple or portable way.

## 4.3 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 4.4.

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.

### 4.3.1 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.
- For efficiency reasons, some algorithms are better implemented on a lower level of abstraction where knowledge about data storage is utilized..
- Input/output operations, as discussed next in section 4.3.2.

As known, native method implementations are seldom truly platform independent. If the compiled Java applications is supposed to be executable on more than one platform[4], 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 Java2 Standard Edition (J2SE).

[4]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 6.3.

### Method calling convention

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

The situation is different for natively compiled Java code. The overhead created by the JNI no longer needs to be there, 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 straight-forward way.

### Memory management

To ensure correctness and timeliness, all external code must access Java references in the same way as the compiled Java code — also in cases where a compacting garbage collector is used. For legacy code and 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));
```

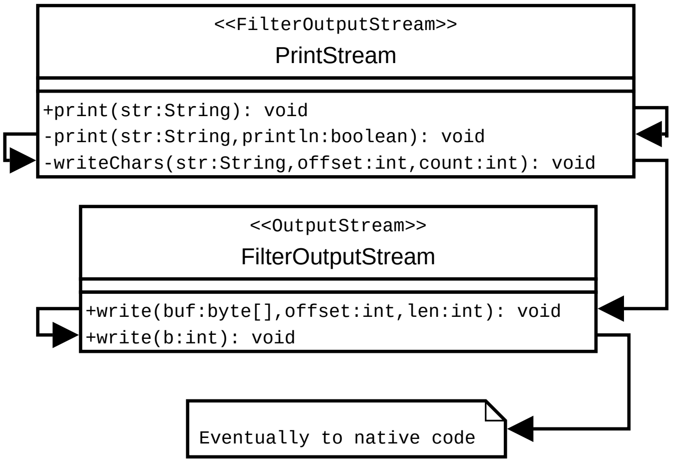
```
GC_PROC_BEGIN(Foo_doSomething,
              GC_PARAM(Foo,this))
  GC_VAR_FUNC_CALL(j2c_result,
                   Foo_process_byteA,
                   GC_PASS(j2c_v));
GC_PROC_END(Foo_doSomething)

/*
 * 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
}
```

### 4.3.2 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 4.6, 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-



<<FilterOutputStream>>
PrintStream
+print(str:String): void
-print(str:String,println:boolean): void
-writeChars(str:String,offset:int,count:int): void
<<OutputStream>>
FilterOutputStream
+write(buf:byte[],offset:int,len:int): void
+write(b:int): void
Eventually to native code

**Figure 4.6:** *The* `System.out.print(String)` *call chain, as implemented in the GNU javalib.*

mentation such as the one schematically shown in Figure 4.6 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 future investigation.

## 4.4 Threads and synchronization

One of the benefits of using Java as a programming language for realtime systems is its built-in threading model. All Java applications are executed as one or more threads, unlike the lack of language support in C or C++ where multi-threading and thread synchronization is performed using various library 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 the Java thread primitives on native threads.
  - + More efficient.
  - - Implementation less straight-forward.

The technique with providing the mapping from Java thread classes to underlying OS primitives by using native methods gives both good performance, and makes 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 4.7.

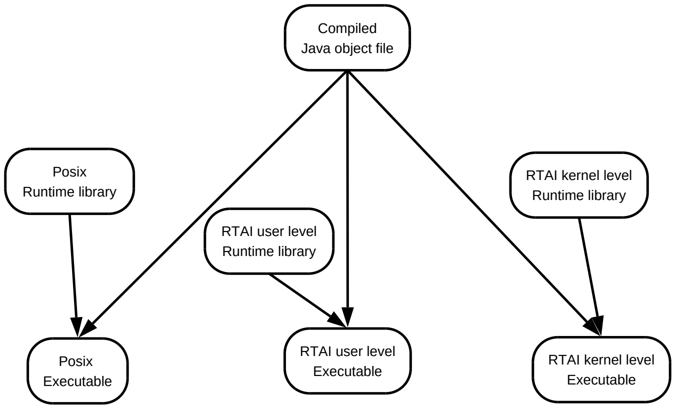

Compiled Java object file
Posix Runtime library
RTAI user level Runtime library
RTAI kernel level Runtime library
Posix Executable
RTAI user level Executable
RTAI kernel level Executable

**Figure 4.7:** *A compiled Java object file can be linked to an appropriate runtime library without recompilation.*

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. Run static blocks and static initializers.
- Start the GC thread.
- Create a main thread.
- Start the main thread, with the main class `main` method as starting point.

### 4.4.1 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 promoting 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.

**RTThread** The real-time threads, `RTThread` and its subclasses, such as `PeriodicThread` and `SporadicThread`, are the extended real-time counterparts to the standard Java thread classes. In order not to inherit any unwanted thread semantics from the standard Java threads, the real-time threads do not extend the standard `java.lang.Thread` class or implement the `Runnable` interface, but form an inheritance hierarchy of their own. This way, the thread semantics for `RTThread`s can be kept suitable for hard real-time systems, if needed.

**JThread** The `JThread` class is supplied as a compatibility class, providing the event services of the `RTThread` while still being a subclass of `java.lang.Thread`. It is thus not suitable for real-time tasks, but may aid porting existing Java applications to the LJRT environment.

**Semaphore** The `Semaphore` interface represents the well-known basic primitive for synchronization and mutual exclusion within the field of concurrent and real-time programming, which is not included in standard Java. Supplied implementing classes include; `BinarySem`, `CountingSem`, and `MutexSem`.

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

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

**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 8.1 for some examples of such directed optimizations.

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, as will be further treated in Chapter 8.

### 4.4.2 Synchronization

The ability to synchronize execution of two or more threads is fundamental to multi-threaded applications, for instance monitors and synchronous inter-thread communication. In Java, thread synchronization is built into the language with the **`synchronized`** keyword, and the methods `wait()`, `notify()`, and `notifyAll()` declared 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 synchronization implementation is more or less straight-forward. Example implementations for Posix threads and RTAI kernel threads are shown in listing 4.2.

**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)
```

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.

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

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

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

GC_PROC_BEGIN(HelloWorld_sync,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_sync)
```

### Finalization of native resources

The native OS resources, such as threads and semaphores, used in the Java thread synchronization often need to be explicitly released when the related Java objects die. There are a couple of reasons for this. When using Xenomai for hard real-time support, the OS resources are never automatically reclaimed when the process that created them dies, as is the case in normal Posix-compatible operating systems. Not only will one run out of available resources if they have limited lifetimes, since OS resources in a Xenomai application all have unique names it will not be possible to stop and restart an application unless the OS is reloaded because of name conflicts. To avoid OS resource starvation and name conflicts, all Java objects have been given low level finalizers on the GCI level, which are used by the GC to release all allocated OS resources before the object is deallocated.

## 4.5 Exceptions

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

The Java standard 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()`.

### 4.5.1 Exceptions based on `setjmp/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 (simplified)
 */
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);
  }
}
```

## LJRT compiler implementation

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, an alternative exception implementation is of interest.

### 4.5.2 Exceptions based on `goto`

Implementing the exception mechanism using `setjmp()/longjmp()` may seem natural and straight-forward, but there is a possible non-negligible performance penalty connected to the use of `setjmp()`. The use of exceptions for handling unusual situations in Java program execution leads to many executions of **`try`** statements that need to be cheap, and much fewer executions of **`throw`**`()` and **`catch`**`()` statements that can then be more expensive in terms of CPU cycles without harming program performance. The problem with a `setjmp()` implementation of the **`try`** statement is that it saves the CPU context, which on a CPU architecture with many registers might be a rather expensive operation. That is, when running programs on a CPU with few registers, such as the Intel x86, it will not matter much, whereas when running on a register-rich CPU, such as a PowerPC, the performance penalty can be significant.

One idea that might reduce the cost of **`try`** statements is to implement exceptions by using a combination of a thread-local exception flag, and the C **`goto`** statement in order to abort normal execution flow and jump to the nearest **`catch`** clause or to return from the function of no matching **`catch`** clause is to be found. This idea is most certainly best explained by looking at an example. Listing 4.7 below shows a corresponding exception example to that found in Listing 4.4 earlier in this section. Note the extra `void *__eFlag__` argument to all functions that is used as a pass-by-reference variable indicating both that an exception has occured, and which class the thrown exception instantiates.

**Listing 4.7:** *A simple exception example using goto.*

```
/*
 * 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 (simplified)
 */
void thrower(void **__eFlag__) {
  // Create new exception object
  Exception __e = newException();
  // Store reference
  storeThreadLocalException(__e);
  // Assign flag and jump to end of function
  *__eFlag__ = &Exception_Class;
  goto catch_0;
catch__0: return;
}

void catcher(void **__eFlag__) {
  thrower(__eFlag__);
  if (*__eFlag__) goto catch__1;
catch__1:
  if (*__eFlag__) {
    if (isCompatibleClass(*__eFlag__, &Exception_Class)) {
      // Matching exception caught
      *__eFlag__ = 0;
      Exception e;
      // Fetch Exception object reference
      e = getThreadLocalException();
      doSomething();
    } else {
      // No matching catch found
      goto catch__0;
    }
  }
catch__0: return;
}
```

### Throwing exceptions

The task of throwing an exception is quite simple. After storing the exception object for the running thread, set the exception flag and jump to the end of the function, or to the first following **`catch`** statement if currently inside a **`try`** block.

### Catching exceptions

Catching exceptions is a little bit trickier, when trying to obey the Java semantics. Every function call must be directly followed, or, at least before trying to use the return value or side effects from the call, by a check of the exception flag. If an exception occurred inside the called function, the exception flag will hold the address of the class of the thrown exception and we should jump immediately to the nearest following **`catch`** statement (or to the end of the function if no **`catch`** is to be found).

The **`catch`** statements are implemented as a sequence of **`if`** statements checking if the thrown exception matches the current **`catch`**. If the **`catch`** matches, the flag is reset, the exception object reference is fetched, and execution of the **`catch`** block contents begin. If no matching **`catch`** can be found in the list, execution jumps to the end of the function and exception handling is transferred to the calling function. To maintain the Java semantics, there must be a catch-all check in the thread main (or run) function for reporting uncaught exceptions and other throwables.

### Implementation in the LJRT compiler

Compared to the `setjmp/longjmp` exception implementation, this exception implementation is even harder to get right if one is to code all details manually. To make it somewhat easier for the programmer, the exception macros as of Listing 4.5 are redefined as shown in Listing 4.8 below.

**Listing 4.8:** *C macros for more understandable exception implementation.*

```
#define CHECK(__catch_lbl) \
  if (*__eFlag__) goto catch_##__catch_lbl

#define EXCEPTION_THROW(__catch_lbl, __nbr) \
  *__eFlag__ = __nbr; goto catch_##__catch_lbl;

#define EXCEPTION_TRY \
{

#define EXCEPTION_CATCH(__catch_lbl,__catch_nbr) \
  catch_##__catch_lbl: \
if (*__eFlag__) { \
  if (isCompatibleClass(*__eFlag__, __catch_nbr)) { \
    *__eFlag__ = 0;

#define EXCEPTION_CATCH_MORE(__catch_nbr) \
  } else if (isCompatibleClass(*__eFlag__, __catch_nbr)) { \
    *__eFlag__ = 0;

#define EXCEPTION_FINALLY } {

#define EXCEPTION_AFTER_CATCH(__catch_lbl) \
  }} \
  if(*__eFlag__) {EXCEPTION_THROW(__catch_lbl,__eFlag__)}; \
  }
```

Finally, Listing 4.9 below shows the same code as Listing 4.7, but using the macros from Listing 4.8 for better readability.

**Listing 4.9:** *Equivalent C code from listing 4.7, but using exception macros from listing 4.8.*

```
// 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(void **__eFlag__) {
  Exception e = newException();
```

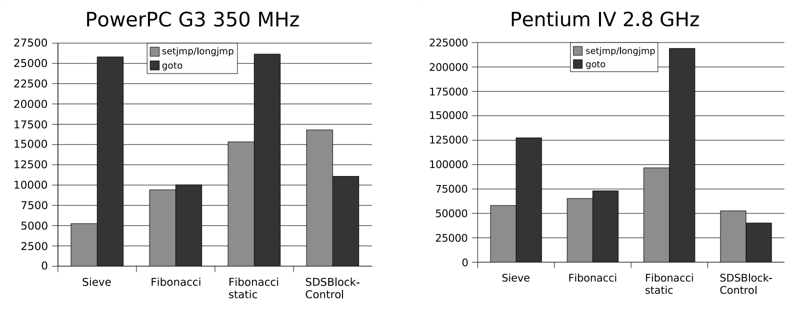


**Figure 4.8:** *Comparing exception implementations.*

```
  storeThreadLocalException(e);
  EXCEPTION_THROW(0,&Exception_Class)
catch__0: return;
}

void catcher(void **__eFlag__) {
  EXCEPTION_TRY
    thrower(__eFlag__); CHECK(1)
  EXCEPTION_CATCH(1,Exception_Class)
    Exception e = getThreadLocalException();
    doSomething();
  EXCEPTION_AFTER_CATCH
catch__0: return;
}
```

## 4.5.3 Exception implementation evaluation

It is not intuitively clear from looking at the code that the **goto** implementation will give the best performance. One could also argue that the opposite could be true due to the exception flag checks after each and all method calls in a program. Running a couple of benchmark applications, see results in Figure 4.8, does not give any clear answers. For the small benchmarks the **goto** implementation yields better performance, while the SDSBlockControl application performs better with the `setjmp/longjmp` implementation.

One conclusion to draw from these results is that it depends on the application which exception implementation will give the best performance. Both implementations are included in the LJRT system, and developers can choose at compile time which one to use. Before making the final decision which one to use in deployed software, one should try both and see which one gives better performance.

### 4.5.4 Finally

The semantics of the optional **`finally`** clause in the Java exception is not intuitionally clear at first glance, and is also a little bit tricky to implement on a platform which is not a stack machine. The Java semantics state that the contents of a **`finally`** clause shall be executed regardless of what takes place in the **`try`** clause, or in any of the **`catch`** clauses. Looking at the example method `foo()` in Listing 4.10[5], one could draw the premature conclusion that there are five possible return values from the method. In fact, "42" is the only possible return value since the **`finally`** clause contains a return statement.

**Listing 4.10:** *Example of* **`finally`** *semantics.*

```
int foo() {
  try {
    int i = bar();
    if ( i > 1 ) return 1;
  } catch (AException a) { return 2; }
  catch (BException a) { return 3; }
  finally { return 42; }
  return 0;
}
```

The semantics of **`finally`** can be enforced in a Java to C translator by inserting the contents of the **`finally`** block just before every possible exit point of the **`try`** and **`catch`** blocks. Even though that means multiplicating identical blocks of code, the resulting code size overhead is not expected to be significant. These **`finally`** blocks are typically rather short, and a good C compiler have the opportunity to optimize away duplicated code.

An analogous situation to that shown in Listing 4.10 occurs if a new exception, which is not caught locally, is thrown in the **`finally`** statement. The above described implementation works here also.

## 4.6 Finalizers

The finalization mechanism in Java is a way to run some code in an object when it has become unreachable, but before it is removed from

[5]Which, by the way, will not pass the *javac* compiler since it contains unreachable statements. It is still a good example, though.

the heap by the GC. Finalization is implemented by overriding the `finalize()` method from `java.lang.Object`. This method is then called from within the JVM before the object's heap memory is freed by the GC. Typical reasons for using the finalization mechanism include releasing OS resources, such as open files and/or sockets, that were handled by the object in question. The `finalize()` method is not restricted in any way compared to other virtual methods in Java, so in theory any code (with unlimited complexity in time and space) can be executed in this method.

The finalization mechanism does pose a problem for those trying to use Java in hard real-time applications. The `finalize()` method must be run in the same context as the GC (or, the execution of a `finalize()` method must be synchronized with GC execution), which means that, since the finalization method may be arbitrarily complex, we add arbitrarily long execution times to the GC thread, sporadically. These added computations may then lead to the system becoming temporarily overloaded, or worse, memory exhaustion jeopardizing the stability of the system.

Due to the potential impact finalization may have on the GC and memory management, finalization should be used very carefully, if at all, in hard real-time Java applications.

In the LJRT, there is also a lower level concept of finalization in the run-time system. Some Java constructs, such as threads and monitors, are mapped onto native target system OS resources, which are not visible on the Java level. When the Java objects die, also these allocated OS resources need to be freed in order to avoid memory exhaustion or resource starvation.

## 4.7 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 Real-Time Garbage Collection (RTGC) is not jeopardized, while retaining readable code. Two different exception implementations have been made and evaluated. Experiments suggest that performance depends on the application at hand, and that tests should be made before deciding on which one to use.

With the techniques presented in this chapter, we are confident that the run-time solutions for all Java constructs could be implemented, using C and being portable with regard to the underlaying (RT)OS. Hence a natural next step is to look at how the Java compiler could be constructed.

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

D. Gries

# Chapter 5

# A compiler for Java and real-time

COMPILERS are complex systems, and implementing them in a modular way is a challenging task. In this chapter we will describe our experiences from using Rewritable Reference Attributed Grammars (ReRAGs) [EH04] for generating Java compilers. ReRAGs is a conditional rewrite formalism that is based on object-orientation, aspect-orientation, and reference attributed grammars.

A major challenge in implementing compilers in a modular way is how to deal with contextual information. For example, a syntactic source construct may have several different interpretations depending on context, and should result in correspondingly different code instruction sequences. Many optimizations also depend on contextual information. For example, a reference to a constant variable may be replaced by a constant literal.

ReRAGs support rewriting of an abstract syntax tree (AST) using contextual information. An AST for a program can be transformed in a series of steps, allowing each computation to be expressed on the most suitable form of AST. The Java front end uses these techniques to transform the syntax-oriented source AST (produced by the parser) into an attributed AST, which reflects the static semantics of the program. Our back-ends use ReRAGs to further transform the AST in a series of steps, to prepare, optimize, and generate code.

The rest of this chapter is structured as follows. Section 5.1 gives an introduction to ReRAGs. Section 5.2 describes the architecture of our compilers. Section 5.3 describes the general parts of the back end, and section 5.4 the specific parts of the LJRT back-end. Section 5.7 evaluates our approach and compares it to compilers handwritten in Java.

## 5.1 ReRAGs

ReRAGs is a declarative conditional rewrite formalism based on reference attributed grammars. There are two features of ReRAGs that are particularly important and that distinguish them from many other rewriting systems. The first is that the rewrite conditions can make use of attributes to easily describe complex contextual conditions for the rewrites. The second is that the formalism is declarative: attributes and rewrites are evaluated automatically, driven by the dependencies, rather than by explicitly given evaluation orders. This makes the evaluation transparent: whenever a syntax node or attribute is inspected, it will automatically be in its final form according to the grammar.

ReRAGs have been implemented in a compiler compiler tool, JastAdd [Ekm04a]. The specification language used is an extension to Java. There is strong support for separation of concerns through the combined use of object-orientation, aspect-orientation, reference attributed grammars, and rewrites.

The JastAdd tool does not include any support for parsing, but relies on the use of an external parser. Any parser generator that can generate Java code will do. We have used JavaCC [Met] and CUP [HFA$^+$99] for different versions of our compilers.

### 5.1.1 Object orientation

The AST is described in an object-oriented fashion by a class hierarchy, as in the Interpreter pattern [GHJV95]. Behavior common to a group of language constructs can be specified in a common super-class and specialized in subclasses. An AST class corresponds to a non-terminal or a production (or a combination thereof) and may define a number of descendants and their declared types. The AST nodes must be *type consistent* according to the normal type checking rules of Java. Support for lists, optionals, and lexical items are also provided. An example of some AST classes is shown in Listing 5.1.

### 5.1.2 Aspect orientation

Rather than defining the AST behavior directly in the AST classes, it is defined in *aspect modules*, supporting static aspect-oriented program-

**Listing 5.1:** *The grammar is expressed as an object-oriented class hierarchy.*

```
// Expr is an abstract AST class and corresponds to a nonterminal
abstract ast Expr;
// Id is a subclass of Expr and corresponds to a production
// id is a String-valued token
ast Id : Expr ::= <String id>;
// BinExpr is an abstract subclass of Expr representing
abstract ast BinExpr : Expr ::= Expr left, Expr right;
// ArithmeticBinExpr is an abstract subclass of BinExpr
abstract ast ArithmeticBinExpr : BinExpr;
// AddExpr is a subclass of ArithmeticBinExpr
ast AddExpr : ArithmeticBinExpr;
// RelationalBinExpr is an abstract subclass of BinExpr
abstract ast RelationalBinExpr : BinExpr;
// LessThanExpr is a subclass of RelationalBinExpr
ast LessThanExpr : RelationalBinExpr;
```

ming similar to open classes or static introduction as described in AspectJ [KHH$^+$01]. These modules allow behavior that crosscuts the AST class hierarchy to be specified together. For example, name analysis, type analysis, error checking, code generation, can be specified in different modules, although they add attributes and rewrites that belong to the same classes. In addition to enhancing readability, this makes it possible to add or remove specific aspect modules during implementation or debugging, and to reuse modules for different versions of a compiler.

The behavior defined in the aspect modules is typically in the form of rewrites, attributes, and equations. However, it is also possible to add ordinary Java code that make use of the attributes. If desired, such Java code can be added in aspect modules that extend the AST classes with variables and methods. This is useful, e.g., for printing the generated code to a file.

Aspects are much more powerful than the Visitor design pattern [GHJV95]. First, they allow instance variables and interface implementation clauses to be modularized, and not only methods. Second, the aspects allow the types of method arguments and return values to be retained, whereas the Visitor pattern requires these to be up-cast to some common type, typically `Object`. Third, aspects can group declarations in an arbitrary manner. For example, if a new language construct is added, its combined behavior concerning name analysis, type analysis, etc. can be grouped together in an aspect if desired. Visitors can only group implementations of the same method.

### 5.1.3 Reference attributed grammars

Rewritable reference attributed grammars (ReRAGs) are based on the Reference Attributed Grammars (RAGs) [Hed00], which is an object-oriented extension to Attribute Grammars (AGs) [Knu68a]. RAGs extend plain AGs by allowing attributes to be references to nodes in the AST. This allows simple and straight-forward representation of non-local relationships in the AST, such as cross references, superclass subclass relations, etc.

Attributes are defined by equations residing either in the node itself (synthesized attributes) or in an ancestor node (inherited attributes). The attributes defined by a certain module constitutes an API that can be used by other modules. In particular, the code generation modules use attributes defined by the name and type analysis modules.

### 5.1.4 Rewrites

ReRAGs extend RAGs with rewrite rules that automatically and transparently rewrite nodes. The rewriting of a node is triggered by the first access to it. Such an access could occur either in an equation in an ancestor node, in some imperative code traversing the AST, or even during other rewrites. In either case, the access will be captured and a reference to the final rewritten tree will be the result of the access. This way, the rewriting process is transparent to any code accessing the AST.

A rewrite step is specified by a rewrite rule that defines the conditions when the rewrite is applicable, as well as the resulting tree. After the application of one rewrite rule, more rewrite rules may become applicable. This allows complex rewrites to be broken down into a series of simple small rewrite steps. Since rewrite rules are declarative, their lexical order is irrelevant.

A rewrite rule has the following form:

```
rewrite N {
  when (cond)
  to R result;
}
```

This specifies that a subtree of type *N* will be replaced by the subtree *result* of type *R*, if the condition *cond* holds. To maintain type consistency, *R* must be the same type or a subtype of *N*. The rewrite may be destructive in that it may reuse nodes from the original *N* tree to build the result tree. Therefore, a rewrite that involves changes to several nodes must be placed in a common ancestor of these nodes. The expression computing *result* may be written using imperative code, but must in that case contain no side effects apart from changing the structure of the rewritten tree.

For a given node, there may be several rewrite rules that apply at the same time. The rewrite rules should be written so that they are confluent, i.e., that the order of evaluation is irrelevant. If the rules are not confluent, their conditions should normally be made more specific so that they do in fact not apply at the same time. This is discussed in more detail in [Ekm04b].

### 5.1.5 Node specialization

A particularly common way of rewriting in ReRAGs is to *specialize* a node, i.e., to replace the node of a class *A* by a node of another class *B*, where *B* is a subclass to *A*. This is useful for turning an AST generated from a context-free grammar into a context-sensitive AST, i.e., one where the nodes are selected based on their context. For example, a use of a name can be specialized into a variable use or a method use, depending on the meaning of the name. Because this rewrite is so common we have introduced a special shorthand syntax for it:

```
specialize N {
  when (cond)
  to result;
}
```

This is a shorthand for a normal rewrite rule with an additional condition that the node is *exactly* of type *N* (and not of any subtype of *N*), and where the result is a subtype of *N*.

## 5.2 Compiler architecture

The architecture of our compiler differs from traditional compilers in that there is no explicit symbol table or other large external data structures. Instead, the attributed AST is itself used as the main representation.

A concrete grammar description is used to create a parser, while an abstract grammar describes the AST class hierarchy. A collection of aspect modules, including name- and type analysis, optimizations and code generation, are woven into the AST classes. The generated parser, the woven AST classes, and auxiliary hand-written Java code make up the compiler, see Figure 5.1.

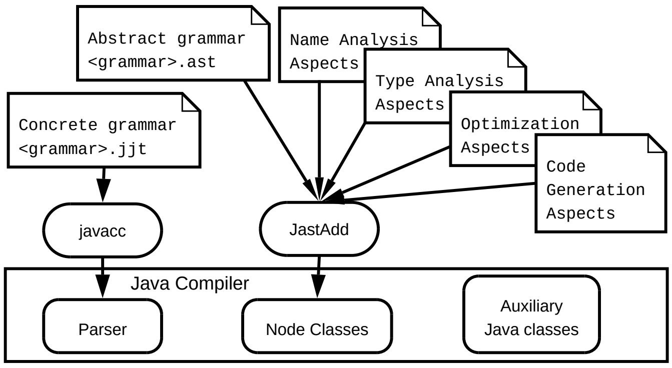


**Figure 5.1:** *Compiler generation.*

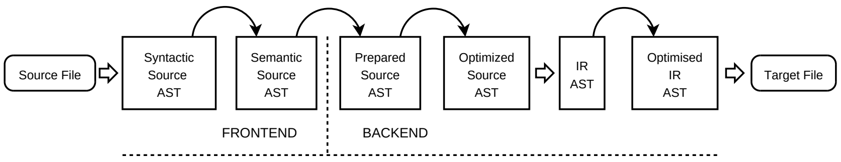


**Figure 5.2:** *Typical ReRAG architecture showing different conceptual states of the compilation resulting from rewrites (arched arrows) or generation (hollow arrows).*

The compilation behavior is divided into a front end, which performs parsing and static-semantic analysis, and a back end, which performs optimizations and generates code. Figure 5.2 shows a typical ReRAG compiler architecture. The boxes represent APIs defined by ReRAG aspects, and the arrows dependencies: either generational (hollow arrows) or rewrites (arched arrows). The boxes can also be interpreted as conceptual states and the arrows as phases. Thus, we can think of the figure as showing a source file that is parsed into a syntactic AST; then semantic analysis is performed resulting in a semantic AST; then the AST is prepared for code generation and optimized; then an intermediate representation AST is generated which is further optimized; and finally the target code is output. In reality, all the steps (except for the first and the last) are applied on demand rather than monolithically, and different parts of the program may be in different states at a given point during evaluation.

In the front end, the parser reads the source file and produces a *syntactic source AST*, i.e., an undecorated AST that closely follows the context-free grammar used by the parser. The static-semantic analyzer is a ReRAG that transforms the syntactic AST into a *semantic source AST* that reflects the static semantics of the program. The semantic AST is attributed, and it is rewritten to capture context-sensitive properties that could not be detected by the context-free grammar. For example, each name node (an AST node representing the use of an identifier) is bound directly to its corresponding declaration node through a reference attribute. Furthermore, each syntactic name node is rewritten to a specialized node, e.g., corresponding to a variable name or a method name. The AST is also rewritten to be simplified, eliminating various "shorthand" constructs in the language. For example, a declaration clause listing many variables of the same type can be rewritten to a list of declaration clauses, one for each variable.

The backend uses ReRAGs to go from the semantic AST and optimize it and generate code. This is done in several conceptual phases. First, the semantic AST is prepared for code generation by further specializing and rewriting some AST nodes. The goal of this *preparation phase* is to allow straightforward code generation, and results in a *prepared source AST*. For example, the semantic AST contains general variable nodes, while the prepared AST has semantically specialized them into instance variables, class variables, local variables, parameters, etc., since the code will differ for accessing these different kinds of variables. Second, the prepared AST is optimized into an *optimized source AST* by performing high-level transformations of the AST. In our LJRT backend, the C code is generated directly from the optimized source AST. In our Java2Bytecode backend, we first generate a bytecode AST (as the intermediate representation). The bytecode AST initially contains a basic set of bytecodes. A subsequent phase replaces some of the basic bytecodes with more efficient ones, resulting in an *optimized bytecode AST*, which is traversed and printed to a target class-file.

The overall view of the architecture given in Figure 5.2 is simplified: each of the phases actually consists of several smaller aspects. For example, the semantic analysis phase consists of both name and type analysis aspects, and the preparation phase consists of both general preparation aspects used for both backends, and target specific preparation aspects. The linear arrangement of the phases is also a simplification, aimed only at giving an overall picture of the compiler architecture. For example, some of the optimization aspects depend only on the semantic AST and not on the preparation phase, and could as well be placed before the preparation phase.

## 5.3 General parts of the back end

Many parts of the back end are general and applicable regardless of the kind of target code generated. This includes general preparations of the AST in order to simplify the code generation, and also many optimizations.

### 5.3.1 Removing redundant variety

The Java language permits many different ways of expressing the same behavior. This variety is desirable for programmers, but not for the compiler implementer. One way of preparing the AST for code generation is to eliminate such redundant variety. This can be done by rewriting the AST to express the programs in a simplified and more homogeneous form, while preserving the semantics.

#### Simplifications in the front end

Some of the simplifications are done already in the front end in order to simplify the semantic analysis and to provide a simpler API for subsequent phases like back ends and analysis tools. Figure 5.3 shows some typical simplifications carried out in the front end.

| *Rewrite* | *Example* |
|---|---|
| Compound declaration splitting | `int x, y;` ➔ `int x; int y;` |
| Default constructor insertion | `class A{}` ➔ `class A{A(){}}` |
| Default super insertion | `A(){}` ➔ `A(){ super(); }` |
| Implicit typecast insertion | `9*3.14F;` ➔ `(float)9*3.14F;` |
| Implicit `this` insertion | `f(p);` ➔ `this.f(this.p);` |

**Figure 5.3:** *Typical simplifications in the front end.*

#### Splitting declarations and initialization code

The back end contains additional simplifications. An example is the splitting of declarations from their initialization code as shown in figure 5.4. Note that the initializations of instance variables are moved into those constructors that directly call a super constructor.

Listing 5.2 shows the rewrite rule that splits the declaration and initialization code for local variables. The relevant part of the abstract

| *Rewrite* | | *Example* |
|---|---|---|
| Declaration and initialization splitting | local var. | `int i=9; → int i; i=9;` |
| | instance variable | `class A {int i=9; A() {super();}} →` `class A {int i; A() {super(); i=9;}}` |
| | class variable | `class A {static int i=9;} →` `class A {static int i; static {i=9;}}` |

**Figure 5.4:** *Examples of splitting declarations and initializations.*

grammar is shown on lines 1–4 (but are actually part of a separate file). The rewrite rule (starting on line 6) applies to VariableDecl objects that are in the context of a Block's statement list (line 6), and where there is an initialization part (line 7). When the rewrite is applied, the VariableDecl is replaced by a list of two statements (line 18): one which is the old VariableDecl (`this`) without its initialization part, and one which is a new assignment, initializing the variable.

### 5.3.2 Differentiating nodes

In many cases, the same kind of language construct will generate quite different code, depending on context. In order to prepare for the code generation, these different cases can be split up by rewriting them to more specialized nodes. Subsequent code generation can then be specified separately for the different cases, associated with the appropriate AST type, thus allowing simpler and more modular specification of the code generation.

As was discussed earlier, the front end specializes the AST nodes for names, so that different AST types will be used depending on if a name refers to, e.g., a variable or a method. Further semantic specialization is carried out in the back end in order to differentiate between accesses to different kinds of variables and methods. For example a general VarAccess node (representing variable access) can be specialized to ClassVariableAccess if its declaration is a class variable, and to InstanceVariableAccess if its declaration is an ordinary instance variable. Subsequent code generation can then be specified separately for the different cases, associated with the appropriate AST type.

In order to specialize these nodes, the abstract grammar needs to be extended with new subclasses. This is done simply by adding an additional abstract grammar module, as shown in Listing 5.3.

**Listing 5.2:** *Rewrite rule specifying splitting of variable declaration and initialization.*

```
ast VariableDecl: Stmt ::= Type type, IdDecl idDecl, [Expr init];
ast AssignExpr  : Expr ::= Expr dest, Expr source;
ast Block       : Stmt ::= Stmt stmt*;
ast ExprStmt    : Stmt ::= Expr expr;
...
rewrite VariableDecl in Block.stmt() {
  when (hasInit())
  to List {
    Expr init = init();
    removeOptInit();
    Stmt assign =
      new ExprStmt(
        new AssignExpr(
          new VarAccess(new IdUse(idDecl().id())),
          init
        )
      );
    return new List().add(this).add(assign);
  }
}
```

Listing 5.4 shows the rewrite rule that performs the specialization of variable access nodes. The API of the semantic AST is used for specifying the appropriate rewrite conditions. This is very simple: the reference attribute `decl` refers to the appropriate declaration node in the AST. That node in turn has attributes `isClassVariable`, etc., in order to find out what kind of variable it is.

A similar semantic specialization is done for assignment statements. Depending on the assignment destination (local, instance, or class variable), different code will be generated. To prepare the AST, the assignment nodes are specialized to differ between these kinds of storage locations. The rewrite rule is shown in Listing 5.5. The rewrite rule uses the API from the aspect specifying the variable access specialization, with methods like `isClassVarAccess`, etc., to specify the appropriate rewrites.

Method accesses (i.e., method calls) are specialized in the same way as the variable accesses in order to differentiate between different kinds of calls.

**Listing 5.3:** *Additional AST classes that are used for semantic specialization in the preparation phase.*

```
ast ClassVarAccess    : VarAccess;
ast InstanceVarAccess : VarAccess;
ast LocalVarAccess    : VarAccess;

ast ClassVarAssignExpr    : AssignExpr;
ast InstanceVarAssignExpr : AssignExpr;
ast LocalAssignExpr       : AssignExpr;

ast ClassMethodAccess    : MethodAccess;
ast InstanceMethodAccess : MethodAccess;
```

**Listing 5.4:** *Rewrite rule for specializing variable accesses.*

```
specialize VarAccess {
  when (decl().isClassVariable())
  to new ClassVarAccess(idUse());

  when (decl().isInstanceVariable())
  to new InstanceVarAccess(idUse());

  when (decl().isLocalVariable())
  to new LocalVarAccess(idUse());
}
```

### 5.3.3 Optimizations on the source AST

By describing source level optimizations in the form of aspect modules with rewrites, it should be possible to easily combine and select different optimizations. So far, we have only done experiments with some small local optimizations using this technique. An example is constant-expression evaluation (constant folding). Consider the following fragment of Java code:

```
final int a=1;
x=a+2;
```

The expression `a+2` operates only on constants and can therefore be evaluated at compile time and replaced by the expression `3`. The rewrite rule in Listing 5.6 specifies constant folding for the primitive types in Java. The rule makes use of the semantic AST API with attributes like `isConstant()`, `constantIntegerValue()`, etc. These attributes are already defined in the semantic AST API since they are

**Listing 5.5:** *Rewrite rule for specializing assignment statements.*

```
specialize AssignExpr {
  when ((dest().isClassVarAccess())
  to new ClassVarAssignExpr(dest(), source());

  when ((dest().isInstanceVarAccess())
  to new InstanceVarAssignExpr(dest(), source());

  when ((dest().isLocalVarAccess()
  to new LocalVarAssignExpr(dest(), source());
}
```

**Listing 5.6:** *Constant folding.*

```
rewrite Expr {
  when (isConstant() && type().isInteger())
  to IntegerLiteral new IntegerLiteral(constantIntegerValue());
  when (isConstant() && type().isLong())
  to LongLiteral new LongLiteral(constantLongValue());
  when (isConstant() && type().isFloat())
  to FloatLiteral new FloatLiteral(constantFloatValue());
  when (isConstant() && type().isDouble())
  to DoubleLiteral new DoubleLiteral(constantDoubleValue());
  when (isConstant() && type().isBoolean())
  to BooleanLiteral new BooleanLiteral(constantBooleanValue());
  when (isConstant() && type().isString())
  to StringLiteral new StringLiteral(constantStringValue());
}
```

needed to check that the cases in a switch statement have constant disjoint values.

## 5.4 The LJRT backend

The development of our LJRT compiler is motivated by our research on languages for small embedded systems with hard real-time constraints. In order to run Java on such systems, special support is needed for memory management, in particular garbage collection. Depending on the system constraints, different memory management schemes can be used, e.g., different kinds of usual (non-real-time) garbage collection, or garbage collection for systems with hard real-time constraints [Hen98] or combinations thereof. In order to separate these constraints from the

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

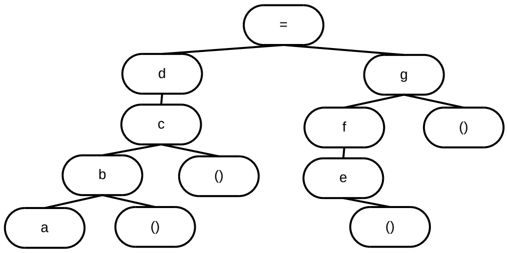


**Figure 5.5:** *Java code fragment and corresponding AST.*

compilation, we have previously developed a Garbage Collection Interface (GCI) to the runtime system [IBE+02]. Before generating the C code, we have an additional preparation phase that rewrites the AST to make code generation to this interface simple.

### 5.4.1 Simplifying to a Java subset

In the runtime system for the LJRT compiler, the GCI is implemented as a set of macros for assignment and referencing variables, involving at most one indirect reference for each macro. To simplify the generation of code for this runtime system, we have defined a Java language subset [Men03] and added an additional preparation phase to the compiler that rewrites an arbitrary Java AST to this Java subset. For example, a complex expression will be broken down into a sequence of simple assignments, using temporary variables, similar to three-address code.

Expressions in Java may be rather complex, as for example in the code fragment with corresponding AST in Figure 5.5.

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 and the GCI, see Section 4 or [IBE+02] for details. Memory operations on references are performed via side-effect 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 the following.

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

Figure 5.6 shows the corresponding AST transformation.

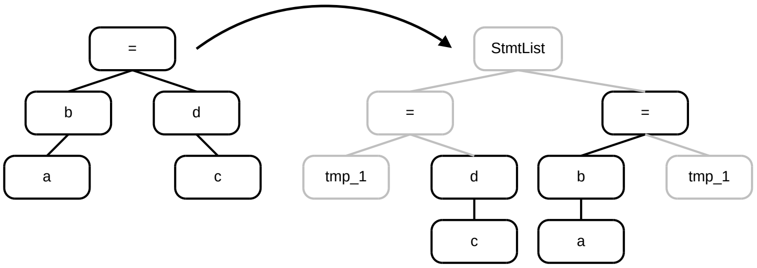


**Figure 5.6:** *Simplifying names by means of an AST transformation. Grey nodes and edges are inserted as result of the transformation.*

The situation becomes a little more complicated with method calls, since arguments passed in the call may contain arbitrarily complex expressions. For example, for the following method call

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

the evaluation order must be

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

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 5.7.

The aspect code needed for performing the simplification transformations shown in Figures 5.6 and 5.7 is shown below in Listing 5.7. For better understanding of the aspect examples it may help to look at the complete abstract grammar for the LJRT compiler which is included in Appendix B.

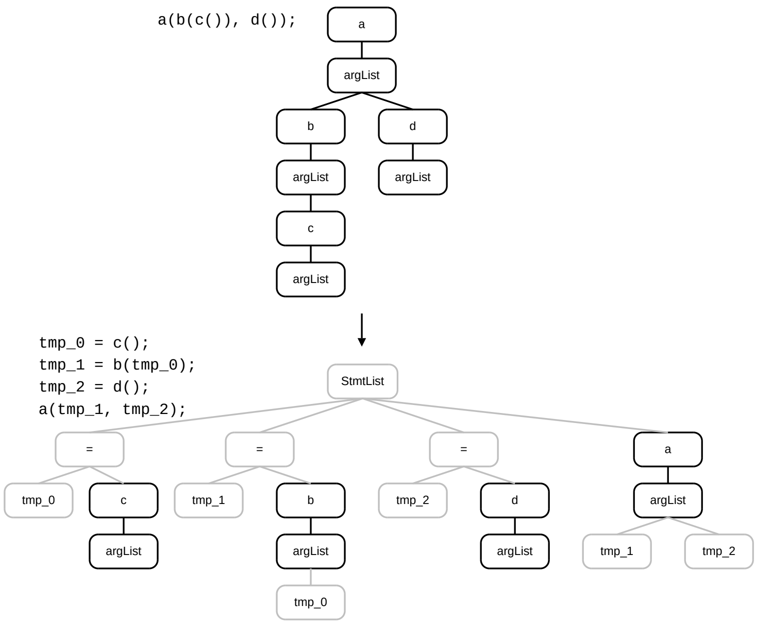


**Figure 5.7:** *Simplifying a complex method call. Grey nodes and edges are inserted as result of the transformation.*

**Listing 5.7:** *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 that 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;
```

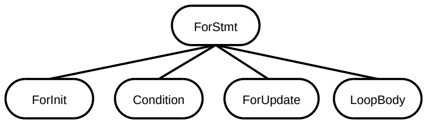


**Figure 5.8:** *Subtree representing a for-statement.*

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

Expressions that are part of control-flow statements require special care in the simplification process, so as not to alter the semantics of the program. The for statement is the most complicated loop statement in Java, and serves well to illustrate these issues.

A Java for-statement, as defined by the abstract grammar for Java, is represented by the AST subtree in Figure 5.8. As defined in the Java language specification [GJS96], the *ForInit* and *ForUpdate* nodes may hold a list of StatementExpressions, and the *ForInit* may alternatively contain a local 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))
  // ... loop body ...
```

The solution to simplifying for-statements is to replace them with while-statements by moving the *ForInit* ahead of the statement and then move the *ForUpdate* last inside the *Stmt* node (which has been transformed to a *Block*). A simplified for-statement subtree is shown in Figure 5.9.

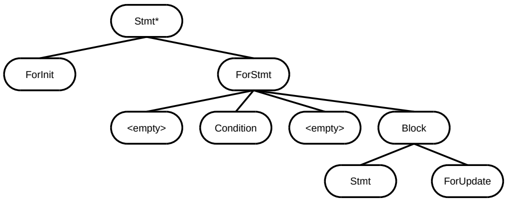


**Figure 5.9:** *Subtree representing a simplified for-statement.*

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 ; ) {
  // Loop body

  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.

## 5.5 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.

### 5.5.1 Dead code elimination

Constructing an Abstract Syntax Tree (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. See also Appendix B.2 for the JastAdd code for performing reachability analysis.

#### 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 applications are described in Section 6.1.4.

| **Application** | **Without opt. (kB)** | **With opt. (kB)** |
|---|---:|---:|
| HelloWorld | 458 | 79 |
| EmbjBench | 559 | 148 |
| SDSBlockControl | 1016 | 430 |

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

For these example applications, ranging from extremely small to very moderate, the savings in code footprint were substantial, with resulting code size ranging from 17% to 42% of the original size. If 79 kB looks a little high for a simple *HelloWorld* application, have in mind that this includes the footprint of an incremental GC, and that a `java.lang.String` object must be instantiated for printing a simple message on the standard output.

## 5.5.2 Implicit finalization

In a Java program, most methods are declared virtual and can be overridden by new declarations in subclasses. Virtual methods calls can not be fully resolved until run-time, when the call target is found through virtual method tables.

The same technique used for code liveness analysis in the dead code elimination can also be used for determining if methods are implicitly final. That is, if a (virtual) method declaration has no live shadowing declarations in subclasses of the method's declaration class, the extra cost for making virtual calls is unnecessary overhead and should be avoided.

As for dead code analysis, implicit finalization is obviously depending on static full-program compilation and can not be used in systems where dynamic loading of classes is used.

### Implementation

Implicit finalization of methods has been implemented in our Java compiler utilizing the already existing code for liveness analysis. Methods, which are found to be implicitly final, are marked as being final in the AST, and the code generation module will treat them as any other (explicit) final method generating straight function calls instead of virtual calls.

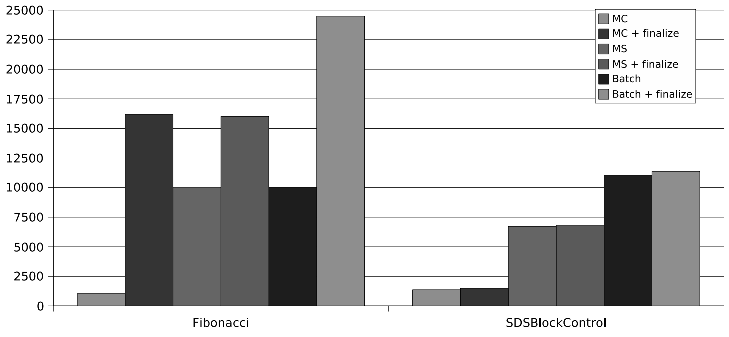


**Figure 5.10:** *Results of implicit finalization optimization.*

### Evaluation

Benchmark tests have been performed using some of the test applications described in Section 6.1.4 and the results are presented in Figure 5.10. The benchmarks were executed on a 350 MHz Motorola PowerPC 603. Three different GC algorithms; *Mark-Compact*, *Mark-Sweep*, and *Batch-Copy* were used.

As can be seen in the figure, the performance gain is substantial for the simple Fibonacci application, which is not very surprising since this application does virtually nothing else but performing virtual method calls. The Mark-Compact case achieves equal performance to the Mark-Sweep case since the temporary reference variable, with it's associated read barrier, is no longer needed for a direct function call.

The optimization effect is much less evident on the SDBlockControl application, with a performance improvement between 1% and 3% depending on which GC algorithm was used. This application evidently does not spend very much CPU time performing optimizable virtual methods calls. The source code also shows that the called methods have multiple implementations in the class hierarchy, which makes it hard to perform this kind of optimization.

It can the be concluded that, although it is possible to implement an application in such a way so that the implicit finalization optimization has little to no effect on the performance, application performance can benefit substantially from using implicit finalization.

```
GC_REF(Type1, tmp1);          // Type1 tmp1;
GC_REF(Type2, tmp2);          // Type2 tmp2;
GC_PUSH_ROOT(tmp1);
GC_PUSH_ROOT(tmp2);
GC_GET_REF(tmp1, a, b);       // tmp1 = a.b;
GC_GET_REF(tmp2, tmp1, c);    // tmp2 = tmp1.c;
GC_GET_REF(foo, tmp2, d);     // foo = tmp2.d;
GC_POP_ROOT(tmp1);
GC_POP_ROOT(tmp2);
```

**Figure 5.11:** *Example of how GCI requires temporary reference variables*

### 5.5.3 Root alias analysis

A *GC root* is a reference from outside the garbage collected heap to an object on the heap, typically a global variable or a local variable on a stack. The root references are used as starting points when the GC traverses the reference graph to identify live objects, and a GC cycle in a tracing collector typically starts with a stack scanning phase, where the root references are identified [JL96].

As our implementation is constrained by an uncooperative environment, we cannot scan the C stacks directly, as they contain no type information which means that we cannot discriminate between pointers and data. Therefore, we use an auxiliary *root stack* for each thread to keep track of the set of live local reference variables [Hen98, RH05].

In a typical object oriented program, a large part of local variables will be of reference types, and thus there will be many root references. The use of GCI also makes it necessary to introduce many temporary variables as complex constructs, such as `foo = a.b.c.d`, has to be split up into simple attribute accesses as shown in Figure 5.11. This means that a lot of roots has to be pushed on and popped from the root stack, causing a significant execution time overhead, primarily from the required synchronization.

It can, however, be observed that in order to ensure correct GC behavior, it is enough that each live object is reachable from one root[1]. This means that the amount of necessary root operations, and thereby the overhead, can be reduced; if it can be statically determined that a variable will only reference objects that are also referenced by another

[1]This does not hold for copying collectors that use forwarding pointers in the objects, as the roots are used for updating pointers as well as for finding live objects; for this optimization to work, the read barrier must be implemented using an indirect table outside the object.

```
void main() {
     Foo f; Bar b;
     ...
     f = new Foo();
     b = new Bar();
     ...
     proc(f,b);
}
void proc(Foo foo, Bar bar) {
     Test t1, t2; Bar b1;
     ...
     t1 = foo.test1;
     t2 = foo.test2;
     b1 = bar.x();
     ...
}
class Foo {
   Test test1, test2;
   ...
}
class Bar {
   Bar b;
   ...
   public Bar x() { return b; }
}
```

**Figure 5.12:** *Root alias example*

variable with longer lifetime, the "inner" variable does not have to be registered as a root. We call this *root alias* analysis, and the compile-time analysis is trivial, as we do whole-program compilation. With this optimization, the push and pop operations in Figure 5.11 would be removed, which means that there will be no additional overhead of having the temporary variables explicitly in the code. In a typical Java program, the amount of "root duplication" is, in our experience, very high, as the associativity between objects tend to be high — between 50 % and 70 % of roots (including temporaries) were redundant in our experiments. A large portion of the required roots are temporary references required to keep a newly allocated object live before its constructor has completed. This is needed to keep latency low; as the constructor can be of arbitrary length it cannot be treated as atomic.

As an example of how the root alias analysis works, we take the code fragment in Figure 5.12. There, `f` and `b` will (or may) reference objects that are allocated in the context of `main`, so these variables must be registered as roots, as they are the only references to the new objects.

On the other hand, in `proc`, we know that the parameters have been registered as roots in the calling scope. Local analysis in `proc`, can statically determine that `t1` and `t2` only reference objects that are reachable from (the attributes of) the parameters, and therefore it is not necessary to register these variables as roots. In contrast, we cannot tell if `b1` is an alias for something already rooted, or not. However, by analyzing the method `Bar.x()` it is seen that `x` only returns an object reachable from an attribute. Therefore, `b1` does not need to be registered as a root.

If we are doing whole-program compilation, all calls to functions returning references can be analyzed and will finally boil down to either an attribute access (which doesn't require rooting) or an allocation (which does). In a separate compilation context, it is not generally possible to perform the whole-program root alias analysis, but the local analysis may still be used to get rid of unnecessary roots caused by temporary variables.

### Implementation

The implementation of the root alias analysis is quite simple, and the majority of the code is shown in Listing 5.8. In the case of class overloading, the analysis of whether a method call may return a new root must analyze all overloaded implementations of the method which may be executed, which may yield a conservative result. For the sake of readability, that code has been left out from the figure.

**Listing 5.8:** *Root alias analysis in the front-end.*

```
boolean VariableDeclaration.isNewRoot() {
  boolean result = false;  Stmt stmt = null;
  ASTNode scope = getSurroundingScope();
  foreach stmt in scope {
    result |= stmt.isNewRoot(this);  }
  return result;
}
boolean ExprStmt.isNewRoot(VariableDeclaration varDecl) {
  if (getExpr() instanceof AssignSimpleExpr) {
    AssignSimpleExpr expr = (AssignSimpleExpr) getExpr();
    return expr.getDest().isUse(varDecl) &&
           expr.getSource().isNewRoot();  }
  return false;
}
boolean MethodAccess.isNewRoot(){return decl().isNewRoot();}
boolean VarAccess.isNewRoot(){return decl().isNewRoot();}
boolean MethodDecl.isNewRoot(){ return returnsNewRoot();}
boolean InstanceExpr.isNewRoot(){return true; }

boolean Block.returnsNewRoot() {
  boolean result = false;
  for (int i=0; i<getNumStmt(); i++) {
    result |= getStmt(i).returnsNewRoot(); }
  return result;
}
boolean ReturnStmt.returnsNewRoot() {
  boolean result = false;
  if (hasResult()) { result = getResult().isNewRoot(); }
  return result;
}
boolean MethodDecl.returnsNewRoot() {
  // Native methods do not have bodies, so let's be conservative
  boolean result = true;
  if (hasBlock()) { result = getBlock().returnsNewRoot(); }
  return result;
}
```

### Evaluation

The effect of root alias optimization is shown in Figure 4.5 on page 43. Although the performance of the specific benchmark application is doubled in the Mark-Compact case, the really dramatic performance improvement is seen when root alias optimization is used together with a Mark-Sweep GC. In that case, the performance increases more than sixfold. The main reason for the dramatic improvement lies in the fact that an incremental Mark-Sweep GC does not rely on a read barrier in order to maintain reference integrity. Since reads are typically much more common than writes, it seems that the unnecessary root registration operations for the temporary variables occupied a large percentage of the CPU time in the Mark-Sweep case. The Mark-Compact case still suffers from the overhead induced by the read barrier and is not affected in the same dramatic way.

## 5.6 Code generation

When the AST has been prepared to fulfill the indirection requirement, the task of generating the C code becomes straight-forward.

For each used class in the AST, a C header file is generated. The header file contains the type declarations of the object model. That is, structs representing the class, its instances, and its virtual method tables (vtables). The actual code is generated into a single C implementation file that contains the implementations of all constructors and methods,

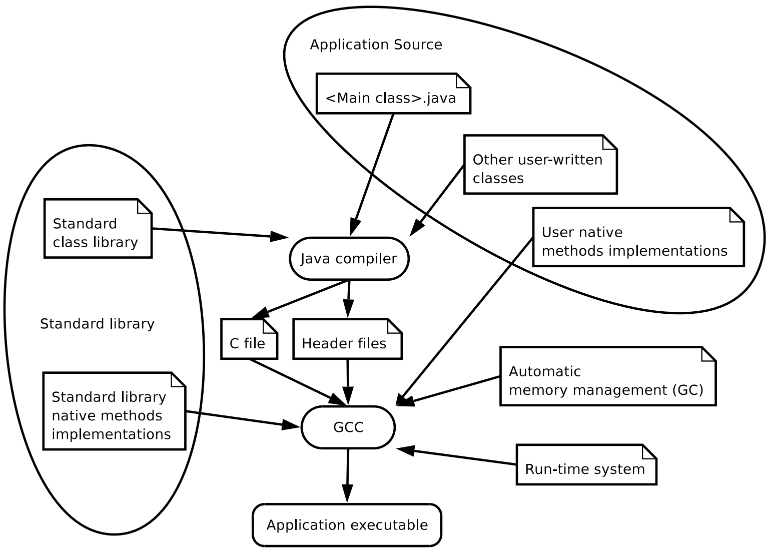


**Figure 5.13:** *Flowchart of compilation process.*

as well as class initialization code. Handwritten C code, such as native method implementations, can include appropriate class header files.

The process of using the LJRT compiler to compile a Java program to an executable machine code image is sketched in Figure 5.13.

### 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
- Declarations of the static object model structs for each class/interface; class, class static, object layout, object static layout, vtable, interface table (if applicable).
- Declarations of 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.

## 5.7 Evaluation

To evaluate our compiler architecture we compare our Java compiler to two compilers and one front-end; javac included in j2sdk 1.4.2, the Java compiler in Eclipse 2.1.3, and the polyglot 1.3 extensible Java frontend. All four compilers implement version 1.4 of the Java programming language.

### 5.7.1 Compiler architecture

We first give a brief description of the other compiler architectures to prepare for a comparison with our architecture from a modularization and extensibility perspective.

#### J2SDK 1.4.2 javac

Javac builds an AST modeled using the interpreter pattern and consists of approximately 40 node types. In order to model the entire language with only 40 node types, some nodes are fairly abstract and contain constants to describe the actual language element. The nodes only model the structure of the language. Contextual information is passed around as parameters to the methods accessed during tree traversal.

Four main visitors are used to perform various computations on the AST: attribution, data-flow analysis, flattening, and code generation. Attribution performs the main contextual computations and calls utility functions in separate modules to add implicit tree nodes, build a symbol table, resolve syntactically ambiguous names, constant folding, and type checking. The various activities are implemented in separate modules but their invocation is scheduled imperatively in the attribution visitor. The data-flow analysis visitor handles exceptions, definite assignment, and reachability computations. These computations are mixed in one module. A full Java AST with nested classes is then transformed to a flat Java AST using a tree translator. This simplifies code generation in that the tree structure better fits the target bytecode structure.

### Eclipse Java compiler

The Java compiler in Eclipse builds an AST similar to javac but has approximately twice as many node types. Visitors are used but to a lesser extent than in javac: some computations are placed directly in the node types and thus scattered over the type hierarchy. Examples include code generation, type evaluation, and generic data flow computations. The generic data flow analysis computations are then, however, used by visitors that modularize computations such as exception targets, definite assignment and various reachability computations.

The code generation is performed on an AST supporting nested types. This, in combination with nodes that represent multiple semantically different language elements, results in quite a few case blocks in the code generation to choose the appropriate code generation.

### Polyglot

Polyglot differs from the other compilers in that it is merely an extensible front-end that performs semantic analysis and does not emit bytecode but merely Java source. This is mainly used to experiment with source code analysis and language extensions that can be transformed into plain Java. The AST is constructed from approximately 95 node types. Rigorous use of interfaces and factories makes it easy to extend the language including new language elements, changed type system, modified scoping rules etc. The various computations in the compiler are separated using a set of visitors. The tree is also transformed in several steps to simplify later computations, e.g., to add implicit nodes and to resolve contextually ambiguous names.

### 5.7.2 Modularization techniques

The architectures in the described compilers, including our own, are similar in that they are based on an object-oriented AST modeled using the interpreter pattern. Visitors are used to separate the various computations on the tree structure except for our solution that is based on AOP techniques and open classes. Another difference is that our approach is declarative and need thus not order the computations in a list of traversals over the AST using visitors. The declarative attributes are computed on-demand according to their dependencies.

All solutions rewrite the AST, to some extent, during compilation to simplify later computations. Examples include the addition of implicit nodes such as default constructors, and translation of nested Java classes to flat Java classes. Polyglot and our compiler also rewrites the tree to reflect the semantic meaning of names, e.g. variable name, field name, or type name. Our compiler takes this approach one step further and tries to create a semantic AST where each node maps to a single semantic language entity. This results in an AST class hierarchy of over 200 node types that in combination with open classes enables excellent modularization through object-oriented techniques such as virtuality and overriding. Another difference is that our solution is declarative and need thus not be scheduled explicitly but is demand driven and transparent to other computations.

### 5.7.3 Extensibility

The visitor pattern is a common technique to add computations on an existing class hierarchy in a modular fashion. All the described compilers can thus easily be extended with a new computation module by adding a new visitor. The same is, however, not true for new language elements. It is not easy to both extend the data model and the computations performed on it in a modular fashion as also noted by others [Rey94].

Polyglot is designed to allow both added computations and language elements using a technique based on extensive use of interfaces and factories. Our compiler does not need a similar framework, but can easily be extended directly due to its declarative nature where equations and rewrites can be combined freely and placed in modules as desired.

### 5.7.4 Compiler size and speed

We have compared the size of the implementation for the Java compilers. While this comparison is not completely fair since they are somewhat different. The LJRT compiler has a C backend, Polyglot has no backend but includes an extensible framework, and the Eclipse compiler is somewhat incremental, it gives some understanding of how much code that is needed for a Java compiler. We took the source folders and removed some utility code not concerned with the compilers and gzip:ed the source code. This technique was chosen instead of lines of code to remove differences in formatting. The specifications from smallest to largest:

- LJRT compiler 120 kbyte
- javac 225 kbyte
- polyglot 295 kbyte
- Eclipse Java compiler 460 kbyte

To get an impression on where the size is spent in our compiler, the sizes of the modules of our compiler are listed in Table 5.2.

| | **Lines of code** |
|---|---|
| **Front-End** | |
| Abstract Grammar | 260 |
| **Semantic Analysis** | |
| Name- and Type Analysis | 9200 |
| **Transformations and Optimizations** | |
| Simplifications | 1275 |
| Dead Code Optimization | 401 |
| **Code Generation** | |
| Code generation | 5745 |

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

Our compiler development has focused on modularity, and so far we have done little effort to improve the compiler speed. We have, however, run some initial benchmarks to compare the compilation speed of our compiler to two other Java compilers. 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.

| | **LJRT 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, the LJRT compiler is substantially slower than the other tested compilers. One main reason is the two-pass nature of our compiler (see Figure 5.13), the time needed for gcc to compile 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 would decrease compilation times significantly in most cases.

## 5.7.5 Observations

We have presented our experiences from generating Java compilers using ReRAGs. So far, our experience is very positive. We have been able to successfully divide the compilers into modules that separate the different concerns and make each module simple to write and understand on its own:

- The preparation modules contain a few simple rewrites that make the AST streamlined for code generation.
- The optimization modules are optional and more optimization modules can be added later without affecting other parts of the backend.
- The generation modules consist of simple, straight-forward code.

The rewrites in the preparation and optimization modules are easy to write due to the information available in the attributed AST, where the attribute API from the semantic analysis can be used to specify the context-dependent rewrite conditions in an easy way. The declarative approach provides automatic rewrite scheduling and thereby allows the different modules to be written independently of each other.

From our experience of compiler implementation, we have drawn the following conclusions:

- The use of ReRAGs promotes compact compiler implementation. The size of our ReRAG specifications is significantly smaller than the size of the Java code of handwritten compilers based on visitors.
- The use of ReRAGs enables the compiler to be better modularized , for instance compared to handwritten Java code. This is because of the aspect-oriented and declarative approach, which permits the compiler implementer to group attributes, equations, and rewrites that address the same concern, without having to take evaluation order into account, and without having to follow the class hierarchy.
- Our generated compilers are around four times slower than compilers handwritten in Java, like javac. Even if there is ongoing work on improving this performance, we find the present performance sufficient for practical use, since modularity and implementation efficiency of compiler development is of primary importance.
- The use of node specialization and rewriting to simpler forms results in more straight-forward code generation that is easier to maintain and extend.

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. To that end, this chapter presents 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 Equipment and applications

Here follows descriptions of the various execution platforms used for experimental verification of the LJRT, as well as descriptions of the software applications used in experiments.

### 6.1.1 Development platform

For the development and debugging of the LJRT, standard workstations with OSs supporting the Posix threading model are used. This usually nowadays means a PC with an Intel x86 compatible CPU running some GNU/Linux distribution, but Sun workstations with UltraSPARC Central Processing Unit, microprocessors (CPUs) running Solaris have also been used. Executing on this kind of platform will obviously not provide any form of hard real-time timing guarantees, but is nevertheless very useful for verifying logical and concurrency correctness of an application. Compiler and run-time system debugging is also greatly facilitated using the standard debugging tools available in desktop environments.

The configuration used for producing some of the results presented in the thesis is a standard PC:

- Pentium IV 2.8 MHz CPU with hyper-threading.
- 1024 MB RAM.
- Running Debian GNU/Linux with a version 2.6 kernel.

### 6.1.2 Real-time control platform

For all real-time experiments, and many other, we are using VME bus based systems equipped with Motorola PowerPC CPUs, models G3 and G4. These systems are also, equipped with AD/DA boards, used for controlling industrial robots and other research systems at the Department of Automatic Control. The specific example used for producing the results presented in the thesis has the following configuration:

- PReP MVME 2400 system.
- Motorola PPC G3 350 MHz.
- 32 MB RAM.
- Running small home-made GNU/Linux with a 2.6 kernel and BusyBox.
- Xenomai 2.1 [Xen06] real-time support added to the Linux kernel.

### 6.1.3 Low-end platform

As a low-end experimental platform, we have a small experimental board, see Figure 6.2 on page 106, 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 microcontroller, 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.

- 128 kBytes in-system re-programmable flash memory with more than 10,000 write/erase cycles endurance.
- 4K Bytes E$^2$PROM. 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 6.1. 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 6.1:** *Measured performance with $k$ priority levels and object size $s$ bytes with $n$ pointers divided into $i$ groups. 1 CPU cycle is 0.25 $\mu s$.*

### 6.1.4 Test applications

A couple of different applications has been used in the experiments presented in the thesis. They range from the minimalistic *HelloWorld*, to a real-time controller implementation.

#### HelloWorld

The classical small test application, which in Java version may look like Listing 6.1. Although quite minimalistic, the HelloWorld application still tests the basics of both compiler and run-time system.

**Listing 6.1:** *Java version of the HelloWorld application.*

```
public class HelloWorld{
    public static void main(String[] args) {
        System.out.println("Hello␣World!");
     }
}
```

#### AlarmClock

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.

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

**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 6.2.

### EmbjBench

Developed by Martin Schoeberl, originally for benchmarking Java in tiny embedded systems, the EmbjBench [Sch06] test suite consists of two sets of small benchmark applications. Each application is run iteratively for at least one second, and then number of iterations per second is calculated and presented as the result.

In the first set, each benchmark application measure the execution time of one bytecode. Such results are of interest when benchmarking a bytecode interpreter implementation, bu less so when the Java source code is compiled to native machine instructions using optimizing compilers. The second set of benchmark applications consist of three small applications:

**Sieve:** Calculate prime numbers. Relies heavily on array indexing.

**Kfl:** Implementation of one of the nodes in a distributed motor control system, with simulated environment (sensors and actuators) and communication (commands from the master node).

**UdpIp:** Measures the performance of UDP communication using a tiny TCP/IP stack for embedded Java. This application was never used in our experiments.

In addition to the applications originally in the EmbjBench suite, we have added two additional application:

**Fibonacci:** Calculate the Fibonacci series using a double recursive algorithm on a virtual method.

**StaticFibonacci:** The same application as *Fibonacci*, but explicitly declared as being **`static`**.

### RTPerf

*RTPerf* is a small artificial application used for measuring latency, jitter, and response times. The application consists of three periodic threads where period, priority, and workload is easily tunable in order to vary memory consumption and overall system load. Measured latencies and response times for the three threads are logged and later offline processing.

### SDSBlockControl

Sampled Dynamic System block, see Figure 6.1, have properties permitting composition such that regulator objects or other sampled dynamic systems comprise entities with the same compositional properties as their aggregated parts. SDS blocks are mathematically causal (outputs are well defined and depend on former and current inputs, and time, as defined by the timestamps of the dynamics input, is monotonously increasing). The inputs/outputs of an SDS block are briefly explained as follows:

**DynamicsInput (DI)** comprises the sampled data driving the advancement in time and is the most recently obtained measured value.

**DesiredReference (DR)** is the most recently requested setpoint (sometimes called the reference) value. If no back-propagation of saturation is done to the set-point, i.e. restricting the admitted setpoint, this object is identical to *AdmittedOutput*.

**AdmittedReference (AR)** is the set-point that is accepted to be used by this regulator. When being different from the *DesiredOutput*, this signal should be propagated backward as a argument to the update of the SDSblock.

**DesiredOutput (DO)** signal is the nominal control output from this regulator, before effected by any saturation or tracking of manual control.

**AdmittedOutput (AO)** is the control output after being effected by saturation or tracking of manual control.

**LoadDisturbance (LD)** represents uncontrollable input that has been possible to measure or compute (externally to this block). Regulation with good disturbance rejection should as much as possible make the effect of the *LoadDisturbance* small. This input is

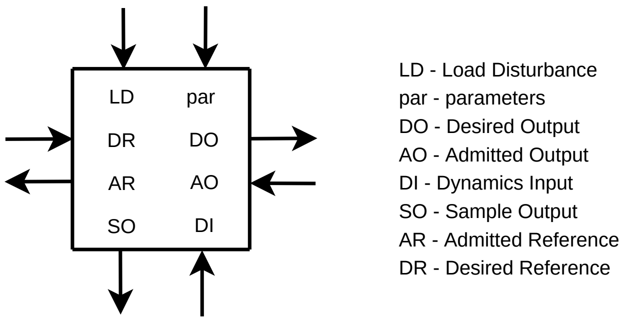


**Figure 6.1:** *Schematic picture of an SDS block.*

assumed to be piece-wise constant and changing asynchronously with the computing of the internal states, and has therefore no direct (time critical) computations based on it, and there are (as seen from the outside) no internal saturations effecting the value.

**SampleOutput (SO)** is the output from this block which is read by outer blocks.

**Parameters (par)** represent the tunable parameters of an SDS block.

The *SDSBlockControl* application used in experiments is an implementation of a simple controller for all five joints of an ASEA IRB6 industrial robot, utilizing SDS blocks. In different experiments, it is used in slightly different ways. In the hard real-time execution experiments, latency of the high-priority sample-and-control thread is measured while lower priority threads use CPU time to generate garbage at a sufficiently high rate to ensure that the GC will complete several full cycles per minute. For other experiments, regarding execution performance, the sample-and-control thread is changed from periodic to free-wheeling, and the number of iterations executed in a fixed period of time is measured.

## 6.2 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 6.2 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 6.2:** *Current implementation status of the real-time Java runtime environment.*

| | AVR | PPC | i386 | SPARC |
|---|---|---|---|---|
| CSRTK[a] | X | | | |
| STORK[b] | | X | | |
| Linux/RTAI[c] | | X | X | |
| Linux/Xenomai | | 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. Deprecated, replaced with Linux/Xenomai.

[c] Both kernel and user level threads. Deprecated, replaced with Linux/Xenomai.

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.

## 6.3 Scalability

Experimentally verifying the scalability of natively compiled Java using the LJRT 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 ($\approx$3GHz, $\approx$1024MB RAM).

Table 6.3 on the next 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.

| 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.

## 6.4 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 6.4.1, while general performance execution experiments are presented in Section 6.4.2.

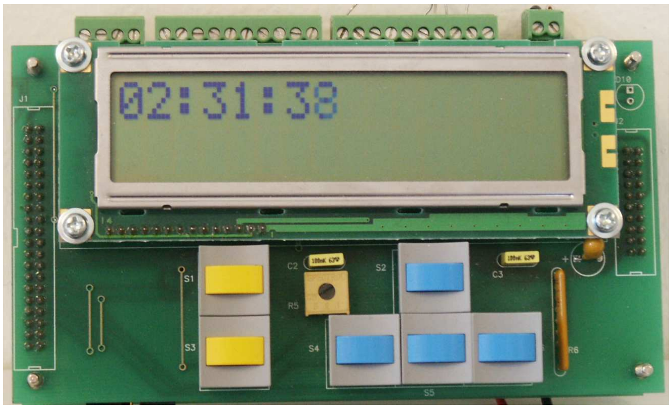


**Figure 6.2:** *Alarm-clock application running on the AVR platform. The platform consists of two stacked cards, with a user interface card on top of a generic CPU card.*

### 6.4.1 Hard real-time execution

In order to experimentally verify the alleged hard real-time capabilities of our proposed real-time Java, we will present results from two different experiments. First, a quite heavily loaded multi-threaded application where latency and task response time is logged for three periodic threads, then a long time test with a robot control application measuring maximum latency. Both experiments were run on a VME-based Motorola PowerPC running GNU/Linux with Xenomai real-time extensions, as described in 6.1.2.

#### Jitter

The small multi-threaded *RTPerf* application, see Section 6.1.4, was configured with one fast (4 KHz) thread and two slower threads, each one performing enough memory allocation work to force frequent GC execution and to obtain a high system load, see Table 6.4. The test application was compiled and linked against two different GC implementations; one *Mark-Sweep* GC, and one *Mark-Compact* GC.

The system is quite heavily loaded with ≈65% utilization from the threads, and then another ≈30% GC utilization from cleaning up the garbage generated by the threads. The difference in workload when using different GCs come from quantization effects in the workload tuning due to different execution overhead for the different GC variants. 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 ($\mu$s)** | **Workload ($\mu$s)** | |
|---|---|---|---|
| | | **Mark-Compact** | **Mark-Sweep** |
| $T_1$ | 250 | $\gtrsim$35 | $\gtrsim$50 |
| $T_2$ | 400 | $\gtrsim$140 | $\gtrsim$160 |
| $T_3$ | 1000 | $\gtrsim$160 | $\gtrsim$180 |
| $GC_1$ | NA | 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.*

Results from the experiments are shown in Figures 6.3 and 6.4, 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 4 kHz and a heavily loaded system.
- 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 6.4.2 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.

### Worst-case latency

Calculating the theoretical Worst-Case Execution Time (WCET), which include worst-case task release latencies, is a truly gargantuan undertaking considering the complexity of modern CPUs, operating systems, and memory management. Running the application for a very long time, measuring the release latency, will give ab estimate of the worst-case scenario. If the execution is let to run for long enough, the estimate will also get statistical significance.

| | | Mark-Compact | Mark-Sweep |
|---|---|---|---|
| **Latency** $\mu s$ | | | |
| $T_1$ | min | 0 | -4 |
| | max | 13 | 8 |
| $T_2$ | min | 0 | -4 |
| | max | 91 | 80 |
| $T_3$ | min | 69 | 58 |
| | max | 326 | 224 |
| **Response** $\mu s$ | | | |
| $T_1$ | min | 34 | 47 |
| | max | 54 | 66 |
| $T_2$ | min | 130 | 155 |
| | max | 225 | 227 |
| $T_3$ | min | 473 | 462 |
| | max | 738 | 708 |

**Table 6.5:** *Real-time performance statistics.*

The *SDSBlockControl* application was instrumented to log the maximum measured latency of the periodic sample-and-control thread. The thread's period was set to 5 ms (running at 200 Hz), while a low priority thread was set to allocate garbage object at a sufficient rate so as to force the GC thread to complete several full cycles per minute on a 10 MB heap.

The System was left to run on the real-time control system, see Section 6.1.2 for a little more than 8 days (210 hours), with more than 151 million invocations of the sample-and-control thread. The measured latency varied between $5\,\mu s$ and $12\,\mu s$ (the higher values when the GC was active), with an all-time-high measured latency of $19\,\mu s$. It should be noted that these latency measurements are conservative, the execution time needed for measuring latency is included in the measured values.

### 6.4.2 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 con-

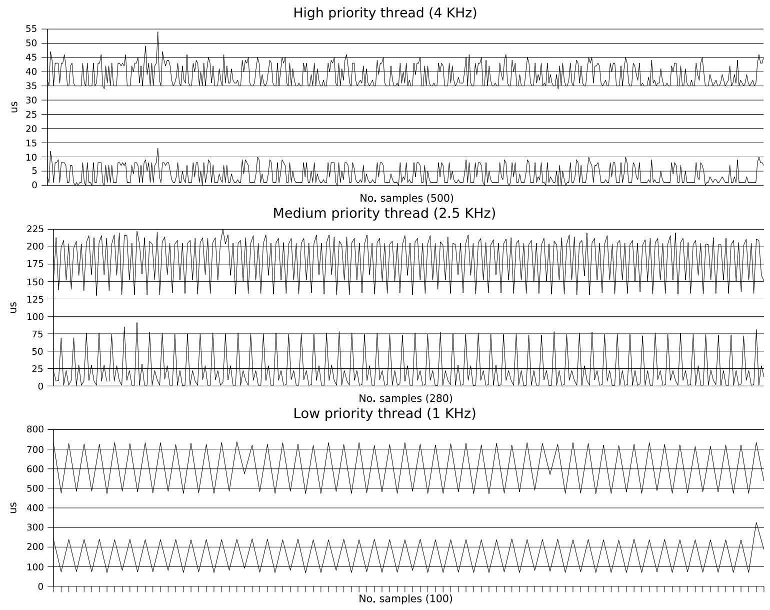


**Figure 6.3:** *Latency for three periodic threads with top-down decreasing priority. Mark-Compact GC used.*

straints including power consumption (if run on battery power), heat dissipation, and cost.

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 have run tests on a few different applications, see Section 6.1.4. These test applications were then compiled with our LJRT 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 (when possible) implemented in C were compiled with `gcc`. The benchmark applications are as follows:

**EmbjBench** The two *Fibonacci* implementations and two of the benchmark applications from the *EmbjBench* suite, see Section 6.1.4.

**SDSBlockControl** For the purpose of measuring the general execution performance, the period of the sample-and-control thread is set

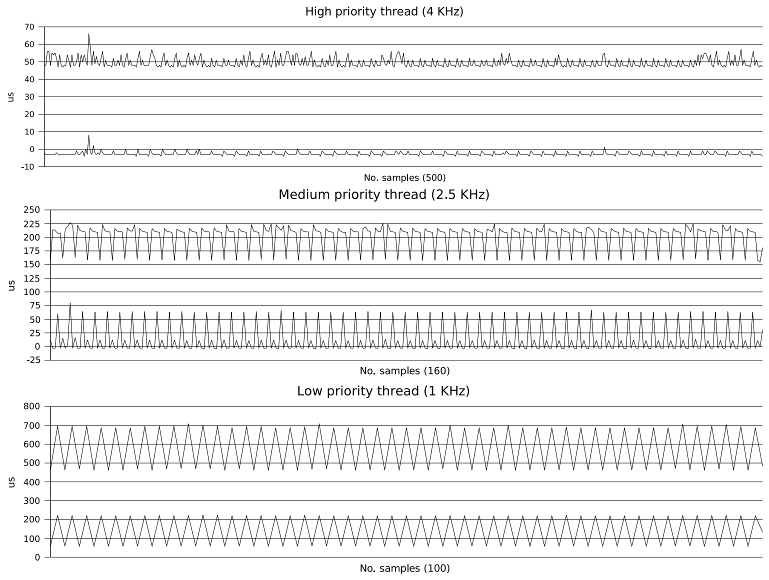


**Figure 6.4:** *Latency for three periodic threads with top-down decreasing priority. Mark-Sweep GC used.*

to zero, effectively making it free-wheeling. The number of iterations during a fixed time interval is measured.

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.

## Test platform

Ideally, we would have liked to run the performance benchmarks on the same real-time control platform as was done with the real-time benchmarks. Unfortunately, Sun Microsystems do not support a Java runtime for the GNU/Linux OS on the PowerPC, and we could not have compared the performance of the LJRT with that of the leading Java runtime.

The benchmarks were executed on a typical LJRT development platform (see Section 6.1.1), a 2.8 MHz Pentium IV workstation running

Debian/GNU Linux, kernel version 2.6.16. Involved software include GNU Compiler Collection (GCC) version 4.0.4 and Sun J2SDK version 1.5.

### Compilation configurations

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

**LJRT** 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.

**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.

These results reveal a number of interesting properties of the current compiler and runtime implementation. First, we conclude that the

| | Sieve | Fibonacci (virtual) | Fibonacci (static) | Kfl | SDSBlock Control |
|---|---|---|---|---|---|
| **LJRT** | | | | | |
| Mark-Compact | 12 355 | 5 385 | 81 209 | 73 926 | 6 943 |
| Mark-Sweep | 312 634 | 71 859 | 92 044 | 1 882 000 | 22 195 |
| Batch-Copy | 1 246 000 | 71 467 | 208 216 | 2 978 000 | 38 395 |
| **Other** | | | | | |
| J2SE | 644 484 | 96 518 | 96 732 | 1 943 000 | |
| J2SE -server | 937 903 | 146 367 | 151 440 | 2 914 000 | |
| J2SE -Xint | 47 046 | 7 031 | 8 159 | 102 882 | |
| GCJ | 390 385 | 80 709 | 90 394 | 1 428 000 | |
| GCJ -O3 | 922 230 | 221 592 | 231 575 | 2 008 000 | |
| **C code** | | | | | |
| GCC | 570 498 | | 81 715 | | |
| GCC -O3 | 1 505 000 | | 259 291 | | |

**Table 6.6:** *Performance measurements. Number of iterations in one second, higher is better.*

only really fair comparison between the LJRT and the other Java implementations, regarding execution performance, is when the *Batch-Copy* GC is used. In this case, there is no overhead in the LJRT for synchronizing with an incremental GC, similar to the other Java implementations which all lack real-time support. Here it is not very surprising to see that applications compiled with the LJRT compilers delivers performance well on par with the Sun HotSpot environment, and only about 20% lower performance than highly optimized hand-written C code (not then also that the two benchmark applications which were ported to C are small and of a type very well suited for aggressive optimizations).

The various degrees of performance degradation seen when one of the incremental real-time GCs is used are solely due to GC synchronization. As shown in Table 6.6, performance suffers heavily from using a mark-compact GC due to the read barrier; as reads are typically more common than writes, the expensive synchronization mechanism has a bigger impact on overall performance. The mark-sweep GC performs much better, especially in the more realistic *kfl* and *SDSBlockControl* benchmark programs. The numbers do, however, illustrate that the cost of synchronization may be devastating to performance if care is not taken. The configuration used here corresponds to number 2 in Figure 4.5 (using Posix mutexes which are expensive), so with more efficient synchronization, the penalty of mark-compact would be significantly less.

Incrementality always comes at the cost of increased run-time overhead, and for batch applications it yields no benefit; as the application never sleeps, any GC work will delay the application. A typical real-time control system, on the other hand, consists of a set of periodic tasks. Thus, an incremental GC can be scheduled so that it will not disturb the application, reducing the impact of the GC overhead significantly. In addition, the long GC pauses make a batch GC unsuitable for real-time applications.

One intuitive conclusion to draw from these results would be that using a *Mark-Sweep* GC is always preferable to using a *Mark-Compact* alternative 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 *Sieve* 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 improve general performance without forsaking the achieved real-time performance.

## 6.5 Hard real-time communication

As has been shown in Section 6.4.1, our Java execution environment provides very predictable and stable response times, also in 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 8.

## 6.6 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 6.3—, 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.

Everything that can be invented
has been invented.

Charles Duell, Director of U.S.
Patent Office, 1899

# Chapter 7

# Prospects

THE tools and techniques used for implementing the LJRT compiler and run-time system are believed to be beneficial in other related research areas. This chapter presents a few projects where we have begun implementing prototypes that we think look very promising. A robot control perspective is assumed.

## 7.1 Multi-stage development

One of the greatest differences between embedded systems software development and other software development areas lies in the testing and verification phase. Embedded systems, especially of small to medium size, are typically very limited in tools and resources for verifying software correctness. Timing aspects in multi-threaded applications and hardware device driver issues make things even more complex.

The errors that can be found in a multi-threaded embedded real-time application can roughly be categorized into one of the following types:

**Syntactical errors:** Caught by the compiler and should thus not give problems when running the application.

**Logical errors:** The application has unintentionally, by mistake or by wrongful thinking, been given different semantics than what was intended. Mostly independent of the current run-time environment.

**Concurrency errors:** Failing to protect shared resources and/or taking resource locks in the wrong order may lead to run-time errors (such as deadlocks or corrupted data). Often depend on actual run-time environment and thread model.

**Timing errors:** Real-time threads missing deadlines, or starvation of lower priority threads. Depends on run-time environment and thread model.

Due to the difficulties in verifying software correctness in the real embedded target environment, for example running as an RTAI/Fusion kernel module, almost all logical errors, and as many as possible of the concurrency and timing errors, should preferably be found early in the development process. Then, one can benefit from the high-level tools available for desktop development, and only the (hopefully) very few run-time environment specific errors will be left when running in the real target environment.

To this end, a multi-stage deployment strategy is suggested. That strategy includes stages ranging from portable desktop and simulation suitable programming, to cross compilation into target-specific hard real-time software functions. In principle and theoretically, just like when a Simulink block diagram is cross compiled into embedded software, it should be possible to go directly from the high-level block diagram to the low-level software binary. In practice, we need intermediate stages of verification since we need to verify the generated code and the platform properties for the new setup before the full robot motion control is run[1].

### 7.1.1 Multi-stage deployment

It is desirable to be able to do as much as possible of the development and testing of an application in a standard desktop environment, due to availability of powerful development environments and tools. However, the subsequent porting of the application to the hard real-time embedded system can require a large effort if done in an ad hoc manner. Therefore, a step-wise method for doing the transition from desktop to target is more suitable. Then only one parameter is changed in each

[1]For instance in the Simulink case the execution order of the code blocks is not fully predictable and slight changes of the control schema may result in one additional sample of control delay, which can be illustrated by quite simple examples. For general purpose software functions, assuming that the software tools are not fully safety certified, we also need intermediate stages of deploying and testing our control function

step, to facilitate verification of the different components or identification of problems.

The fundamental principle is that the source code of the application should remain unchanged during all the stages of the deployment. What is changed, as the desktop application gradually is moved towards the embedded target, is, in turn, the class library, the compiler, the computer, the I/O drivers and the thread model. Table 7.1 gives an overview of the properties of the different stages of deployment. We will now review the different stages in detail, based on using Java and our accomplished LJRT platform.

| Stage | VM | native | host | target | real-time | kernel space |
|---|---|---|---|---|---|---|
| J2SE | x | | x | | | |
| JVM+classpath | x | | x | | | |
| LJRT/POSIX | | x | x | | OS | |
| LJRT/POSIX | | x | | x | OS | |
| LJRT/RTOS | | x | | x | x | x |

**Table 7.1:** *Properties of the different stages.* OS *in the real time column means that the real-time properties depend on the OS used. For instance, some desktop operating systems have some real-time capabilities, if, e.g., running as the super-user.*

1a. *Running both the application and the simulated environment in a standard JVM (J2SE) on a workstation.* Most of the work is carried out in a standard Java environment; the application is designed and implemented on the workstation using the J2SDK, and testing is done against a virtual, simulated, robot. Logical bugs in the application are found using standard high-level debug techniques and tools for this environment. In this stage it is possible to determine logical correctness of the application; if it does what it is supposed to do, and in the correct order. It is not possible to determine if the application exhibits correct real-time behavior.

1b. *Running both the application and the simulated environment in a standard JVM on a workstation, but using the free Java library* classpath *instead.* The LJRT platform uses the GNU classpath implementation of the Java standard classes, as it is free software. If the application was developed using the Sun J2SDK in the previous step, the move towards the LJRT platform is started by verifying that the application works correctly when the classpath implementation is used instead of the J2SDK one.

2 *Natively compile application using LJRT, and run against simulated environment on work station.* When we are sufficiently convinced that our application is correct, the next step would be to natively compile and test the code using the LJRT platform, but still run with POSIX threads on the workstation against the simulated environment, and with no real-time threading support. Here we can still use standard debugging techniques and tools, such as DDD and GDB, and verify that native compilation did not induce any new undesired properties of the application.

3 *Running natively compiled LJRT application with POSIX threads on target system.* The next step is to use the LJRT compiler and cross-compile the application for the target system, still using POSIX threads. This allows us to verify that the cross-compiler produces correct output, and that program loading, I/O, etc., on the target system works correctly.

4a. *Running natively compiled LJRT application with native RTOS threads in user space on target system.* Finally, the transition from POSIX to native RTOS threads is done, and the real-time behaviour of the application is tested and verified. At this stage, any errors that occur are known to (or at least very likely) be real-time problems.

4b. *Running natively compiled LJRT application with native RTOS threads, in kernel space, on target system.* If required, for example for interrupt latency reasons or efficiency in communication with hardware drivers, the application may be run entirely in kernel space.

As stated in the introduction, when the tools and the platform have been verified to work, it is possible to directly do the transition from the simulated environment on the desktop, to the target system. The major benefit of the intermediate steps is when things do not work, or when doing verification (or development) of the platform. The possibility of doing the deployment in several steps also makes it much easier to pinpoint at what stage of the deployment an error occurs and, hence, if the source of the error is in the application code, the tools, hardware drivers, or the operating system.

This is, of course very valuable for the developers of the tools and the platform. However, it is equally valuable to the application developer, as it makes it easier to determine if an error is due to a mistake in the application code or a bug in the platform and, hence, if the action to be taken is to start a debugging session or to file a bug report.

### 7.1.2 Experiences

The key feature of the presented approach is that the same source code is used both in the simulated desktop environment and in the target system. In order to verify the feasibility of the presented approach, a motion control application for an ABB IRB-6 industrial robot was developed. With the exception of the drivers for the analog and digital I/O in the target system, the complete application was written in Java. The control software consisted of two threads; the sampling and PI control for each joint was run at $2\,kHz$. The reference generation was performed at $100\,Hz$.

On the desktop, the application was run,in simulated time, on a standard JVM, with a virtual robot consisting of a simple dynamics model and Java3D visualization ([HN99]). On the real robot, the program was compiled to C code using the LJRT Java compiler and to native code with gcc. The target system was a Motorola MVME 2600-1 computer, with a $200\,Mhz$ PowerPC G3 CPU and the operating system was Linux/RTAI fusion, version 0.9.1. Figure 7.1 shows the real robot in the robot lab, and a screen-shot of the virtual robot.

When the application was executed on the target system, both control and real-time performance was measured. The measurements were done by logging to an array in the context of the corresponding thread, and dumped to a file after the completion of the experiment.

Figure 7.2 shows the angles of the 5 joints when a simple pattern is run on the real robot, and the virtual one, respectively. Figure 7.3 shows the step response when all joints are driven between the origin and $2.5rad$. Figure 7.4 shows the real-time performance of the controller thread, on the target system, and you can see that the jitter is quite low, typically below $\pm\,3\,\mu s$.

### 7.1.3 Outlook

As development of embedded real-time software adds complexity compared to software development in general, it is desirable to separate platform concerns from application development. Experiences indicate that the presented method for development and deployment provides such a separation of concerns. Thus, we believe that the use of safe languages and a method that allows application development to be separated from platform issues is an important step in facing the challenges of future control software development.

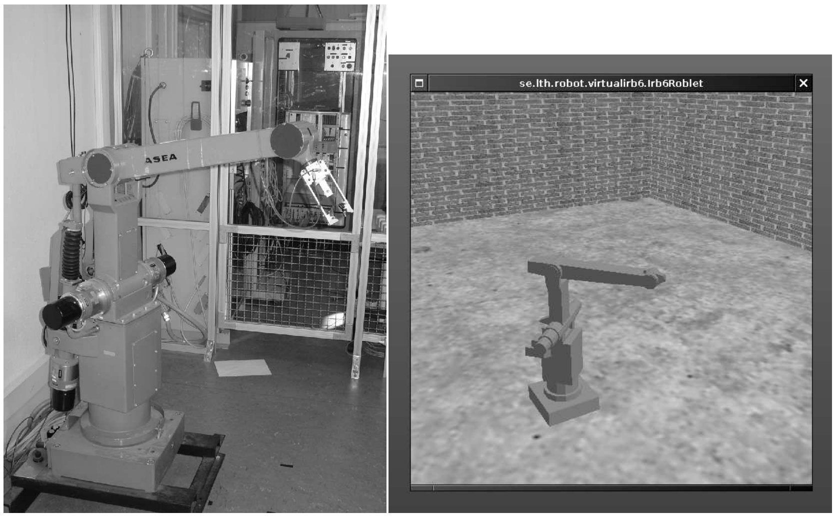


**Figure 7.1:** *The IRB-6 in the robot lab and its virtual counterpart.*

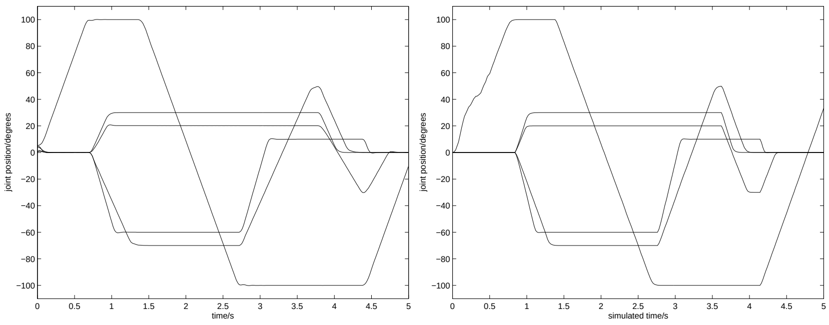


**Figure 7.2:** *Joint angles for the 5 joints when the robot is driven in a pattern of 5 configurations on the real (left) and virtual (right) robots. The differences are that the virtual robot is run in simulated time and that the dynamics model isn't exact.*

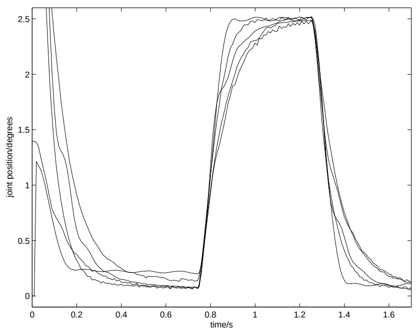


**Figure 7.3:** *Step response for the 5 joints.*

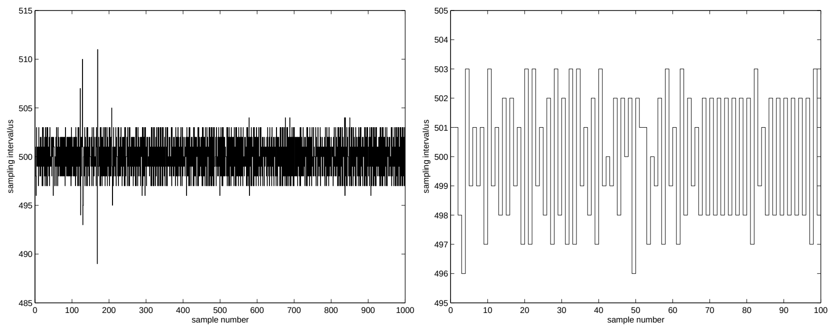


**Figure 7.4:** *Measured sampling intervals for 1000 consecutive sampling instants; the nominal sampling period was* $500\ \mu s$*, and the jitter was typically less than* $\pm 3\ \mu s$*, with a maximum of* $\pm 10\ \mu s$*. The right plot shows a close-up of the first 100 samples.*

## 7.2 Ontology-based compilation

Ongoing research projects in industrial robotics[2] are trying to make it easier to install, program, and (re)configure industrial robot systems in order to make them more accessible to small and medium enterprises. The industrial robot systems available today are only justifiable in enterprises with long or repeated product series and highly trained staff to program and configure the robots. Small to medium enterprises typically manufacture many different products in short series and need to be able to reconfigure the manufacturing line within a day or so, which is virtually impossible with today robots.

One of the problems to be faced when (re)configuring a robot is how to deal with the different peripherals that are used in the robot task. There are many different kinds of peripherals ranging from simple on/off sensors to advanced vision systems and all possible kinds of grippers. Different peripherals have different characteristics (size, weight, speed, payload) and are interfaced (mechanically, electrically, and communication) in different ways. Communication is often not standardized, and done on a quite low level. If the data describing a peripheral was available in a structured format, it would perhaps be possible to automatically generate the peripheral configuration for a specific robot model.

Then there is the problem which specific peripheral mode to choose for a task. This depends on the constraints associated with the specific task; speed, accuracy, payload, and so on. Also here would a database of structured peripheral descriptions be of help. A reasoning system could them try to find the best match from the given constraints.

The idea then is to describe an ontology for robot peripherals, such as grippers, using a standardized ontology language. In the prototype we use OWL [W3C04], but any ontology language could be used. This ontology then serves as specification language for peripherals, which is rather well-suited for automatic processing using either standard XML tools or tailor-made tools. However, a common problem with such specialized description languages is that developed tools are very tightly connected to a specific version of a specific description language. Tools made for gripper descriptions can typically not easily be used also for computer vision system descriptions. Changes in a description language will also imply changes in the tools.

[2]see for example the EU sixth framework programme projects SMErobot (`http://www.smerobot.org`) and SIARAS (`http://www.siaras.org`)

As is not uncommon, such problems become easier to manage by raising the abstraction level one step. By implementing a meta compiler, a compiler for OWL that, as output, generates a compiler for the description language specified in OWL the abstraction level is raised. Instead of having to handle the dependencies between description language and tools manually, there is now one single specification for both description language and tool generation. The fact that these description languages are XML-based helps in that the parsing syntax is given beforehand.

The general steps are depicted in Figure 7.5. A compiler for ontologies expressed in OWL is implemented using JastAdd and a suitable parser generator. Given an OWL ontology as input, this compiler will then automatically generate the description of a compiler for the given description language. The automatically generated compiler can then be used to parse and analyze a description, and generate various forms of output depending on how the compiler was specified. Interface classes to sensors and actuators, or entries in skill server databases are just a few examples of what is possible to achieve.

The automatically generated code will probably need some additional hand-crafted code for a specific description language. But, it is still a far better situation than when the fundamental compiler description needs to be adapted after each change in the description language specification.

## 7.2.1 Initial prototyping

A prototype implementation for to evaluate the above idea is in development. We are using the JastAdd tool, see Chapter 5 together with JavaCC to build a compiler for the OWL language. As a realistic example on a non-trivial description language, a model hierarchy of robotic grippers is constructed, see Figure 7.6. Robotic grippers can be divided into four different categories based on which gripping principle is used; *vacuum grippers*, *magnetic grippers*, *finger grippers*, *pincer grippers*, and *adhesion grippers*. For finger- and pincer grippers, a more fine-grained division is needed because of functional differences with different mechanical solutions.

An OWL representation of the gripper model, see Figure 7.6, serving as input for the OWL compiler is shown in Listing 7.1. Even though it looks rather hard to read for human eyes, as most XML syntax, it is quite easy to write a parser for OWL descriptions since a simple grammar only consists of the following productions (in JastAdd syntax):

```
Element ::= Attribute* Element*;
Attribute ::= <STRING_LITERAL>;
```

Using the JastAdd tools, it is then easy to analyse the AST returned by the parser, and then generate a new abstract grammar from the AST. JastAdd code for generating a parser grammar and useful aspects for the described language has not yet been implemented. It is, however, not more difficult than generating an abstract grammar, it is just different syntax in the generated code.

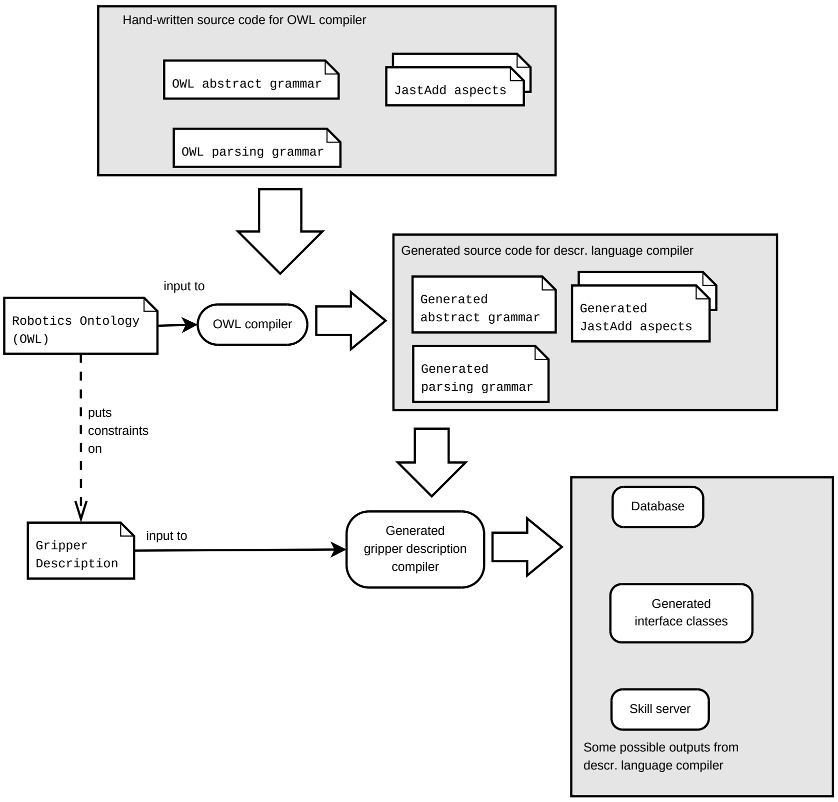

Hand-written source code for OWL compiler
OWL abstract grammar
JastAdd aspects
OWL parsing grammar
Generated source code for descr. language compiler
Generated abstract grammar
Generated JastAdd aspects
Generated parsing grammar
input to
Robotics Ontology (OWL)
OWL compiler
puts constraints on
Gripper Description
input to
Generated gripper description compiler
Database
Generated interface classes
Skill server
Some possible outputs from descr. language compiler

**Figure 7.5:** *Using OWL ontology to generate compiler for robotic grippers.*

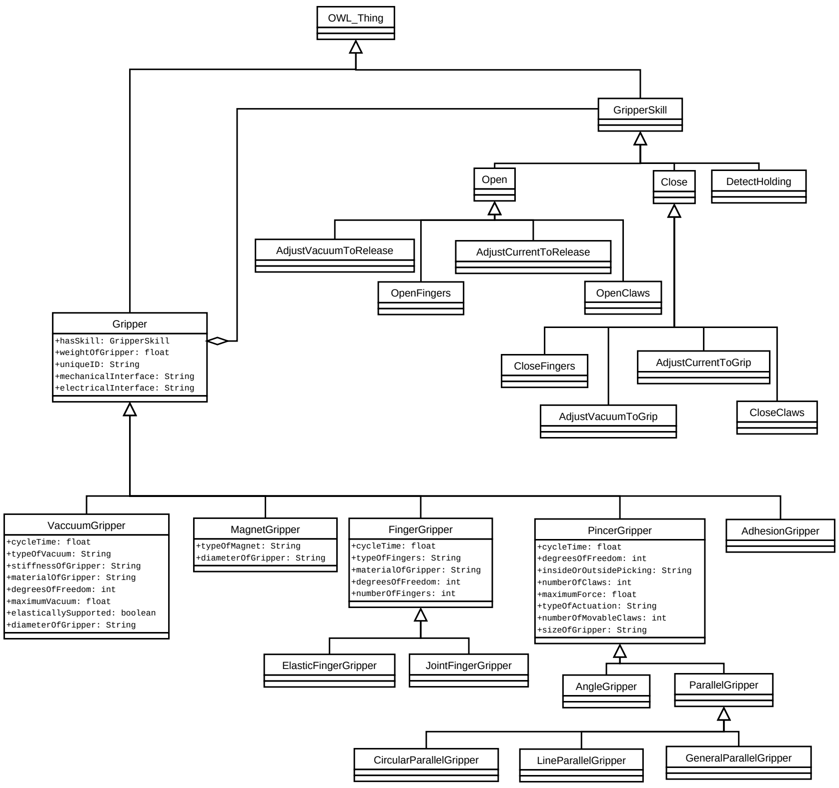


**Figure 7.6:** *Robotic gripper model hierarchy.*

### 7.2.2 Outlook

The current prototype, consisting of less than 400 lines of JastAdd code, can analyze a non-trivial OWL document and then generate a JastAdd abstract grammar for the description language as described by the OWL document. Regardless of which changes are done in the OWL-based specification, both the abstract and concrete grammars for the description language can be automatically generated.

In order to comprise a fully usable compiler for description languages, manually written code, here in the form of JastAdd aspects, is surely needed. If the language specification changes, there is a possibility that one has to make changes to this code also. However, there is a large difference between having to make minor changes in compact aspects, and changing the fundamental grammar descriptions in the compiler.

The described gripper-compiler is just one example of equipment and equipment interfaces. Many more end-effectors (for welding, gluing, grinding, and so on) exist, and there are several other types of external equipment (such as fixtures, feeders, and conveyors). Furthermore, interfacing in a robot work-cell involves more than the equipment, so in total there are at least the following items to cope with in terms of data interpretation:

- External equipment and their interfaces
- Control services
- Human operations
- Interaction devices
- Work-piece data and models
- Task descriptions and robot languages
- Model data for production processes
- Production or robot skill strategies

Up to now, there have been different standardization efforts in each of these areas. Most modern standards are XML based, which is sometimes referred to as being generic and fully portable. However, having XML-based standards in each of the eight areas according to the items above does not really solve the problem; if these standards are based on different taxonomies, the data integration and coherence between different items remains as a problem. That is, the different standards (despite being expressed in XML) are based on different terms, and the information integration in (or around) the robot system still requires extensive engineering efforts.

By relating different standards to a common ontology, and using compilation in several stages as described above, it is believed that the information processing in the robotic work-cell can be much more generic than possible today. Clearly it will be a substantial effort to model all interfaces and devices and so on, but if compiler technology can provide the means for the machine to interpret and relate terminologies from different areas, it will hopefully result in a break-through towards the aim of so called Plug-and-Produce robot systems [SME].

**Listing 7.1:** *Excerpt from OWL representation of grippers.*

```
<owl:Class rdf:ID="Gripper">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasSkill"/>
      </owl:onProperty>
      <owl:someValuesFrom rdf:resource="#DetectHolding"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:someValuesFrom rdf:resource="#Open"/>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasSkill"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:someValuesFrom>
        <owl:Class rdf:about="#Close"/>
      </owl:someValuesFrom>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasSkill"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="MagnetGripper">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:someValuesFrom>
        <owl:Class rdf:about="#AdjustCurrentToGrip"/>
      </owl:someValuesFrom>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasSkill"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasSkill"/>
      </owl:onProperty>
      <owl:someValuesFrom rdf:resource="#AdjustCurrentToRelease"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Gripper"/>
  </rdfs:subClassOf>
</owl:Class>
```

**Listing 7.2:** *Generated abstract grammar for gripper descriptions.*

```
Gripper : Thing ::= ;
MagnetGripper : Gripper ::= <typeOfMagnet:String> ;
VacuumGripper : Gripper ::= <cycleTime:float> <typeOfVacuum:string>
                            <stiffnessOfGripper:String>
                            <materialOfGripper:String>
                            <degreesOfFreedom:int> ;
FingerGripper : Gripper ::= <cycleTime:float> <typeOfFingers:string>
                            <materialOfGripper:String>
                            <degreesOfFreedom:int>
                            <numberOfFingers:int> ;
JointFingerGripper : FingerGripper ::= ;
ElasticFingerGripper : FingerGripper ::= ;
PincerGripper : Gripper ::= <cycleTime:float> <degreesOfFreedom:int>
                            <insideOrOutsidePicking:string>
                            <numberOfClaws:int> ;
AngleGripper : PincerGripper ::= ;
ParallelGripper : PincerGripper ::= ;
AdhesionGripper : PincerGripper ::= ;
CircularParallelGripper : ParallelGripper ::= ;
LineParallelGripper : ParallelGripper ::= ;
GeneralParallelGripper : ParallelGripper ::= ;

GripperSkill : Thing ::= ;
Open : GripperSkill ::= ;
OpenFingers : Open ::= ;
AdjustCurrentToGrip : Open ::= ;
OpenClaws : Open ::= ;
AdjustVacuumToRelease : Open ::= ;
Close : GripperSkill ::= ;
CloseFingers : Close ::= ;
AdjustCurrentToRelease : Close ::= ;
AdjustVacuumToGrip : Close ::= ;
CloseClaws : Close ::= ;
DetectHolding : GripperSkill ::= ;
```

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 8

# 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. In addition to the fact that neither the compiler, nor the runtime system and class library, are complete with regard to the Java language specification and the Java Development Kit (JDK), there are many interesting problems to look into.

## 8.1 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.

### 8.1.1 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 4.4.1, 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.

### 8.1.2 Memory allocation

When using a mark-sweep GC, and not partitioning the heap in constant size blocks, 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.

### 8.1.3 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.

### 8.1.4 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.

## 8.2 Networking

The current trend towards distributed automation systems, and to close control loops over distributed nodes in a network, introduces 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].

## 8.3 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 as 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 Xenomai runtime environment, the concept of loadable Linux kernel modules may present one way of achieving dynamic loading of code.

## 8.4 Ontology-based compilation

The prototype for ontology-based compilation presented in Section 7.2 looks very promising. Implementation should continue so that the prototype can be tested and evaluated in a realistic environment.

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

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

# Chapter 9

# 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 projects most related to our work.

## 9.1 RTSJ

The RTSJ [BBD+00] identifies seven areas where the Java specification must be enhanced in order to facilitate the use of java in real-time systems. This is accomplished by adding new real-time Java packages and by enhancing the JVM so that real-time semantics and timing guarantees may be fulfilled. The seven areas, with a brief description of the needed enhancement, are as follows:

**Thread Scheduling and Dispatching:** *Schedulable objects* are scheduled by an instance of a `Scheduler`. The scheduler is priority-based, but the actual implementation may be replaced. Two new real-time thread classes are supplied, with different characteristics. Instances of *RealtimeThread* have access to the real-time services and may access objects in all types of memory areas including the standard heap. Because of these heap accesses *RealtimeThread* instances may be blocked from execution by the system GC, which makes suitable only for soft real-time tasks. *NoHeapRealtimeThread* is a specialized form of the *RealtimeThread*, of which instances are forbidden to reference heap objects. This makes them insensitive to GC work, and *NoHeapRealtimeThread* instances can interrupt the GC without jeopardizing reference integrity. It is the only type of thread suitable for hard real-time tasks in RTSJ.

**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 objects 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 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.

Although capable of providing hard real-time guarantees, there are several problems associated with using an implementation of RTSJ for developing real-time applications. The new memory area types and the set of rules for how references may cross memory area boundaries adds more complexity to handle for the error-prone programmer.

### 9.1.1 RTSJ implementations

There are now a handful of implementations of the RTSJ available on the market, or under development.

### TimeSys

TimeSys Inc. [Tim06] maintain the official reference implementation of RTSJ which can be downloaded from their website.

### Mackinac

Sun Microsystems have recently released a version of the HotSpot virtual machine supporting RTSJ. Based on Sun's HotSpot technology, but compiling Java classes at initialization time instead of during runtime, applications executing on the real-time HotSpot is predicted to achieve similar performance as equivalent C++ applications. Just like HotSpot, it will take quite some CPU power and memory to run the real-time version (this version needs a 2-way SPARC minimum in order to guarantee hard real-time with decent timing). Future versions are planned to include real-time garbage collection using the ideas developed by Henriksson and Robertz in Lund.

### 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 the RTSJ, see [Sie00, Sie99, SW01, Sie04]. 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.

### PERC

The PERC Java platform from Aonix Inc. [NL98] is another example of a hybrid platform with alleged hard real-time capabilities. NewMonics was one of the founder of the JConsortium with it's competing real-time Java specification, Real-Time Core Extensions (RTCE), but they now seem to have switched side and openly support the RTSJ.

**JRate**

JRate [CS02] is implemented as an extension of GCJ to support the RTSJ. Since it is an ahead-of-time compiled solution, performance should be acceptable also on modest platforms.

## 9.2 RTCE

The RTCE [Con00] 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.

The RTCE initiative now seems to been abandoned, and their website has been shut down.

## 9.3 JOP

JOP [Sch05, Sch06] is a Java processor implementation in hardware designed for time-predictable execution of real-time tasks. It is intended for really small resource-constrained embedded devices. The current implementation only support static memory allocation, but there seem to be plans on implementing a garbage collector.

## 9.4 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.

One 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 virtual dispatch elimination, method in-lining, and dead code elimination.

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.

## 9.5 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.

## 9.6 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 10

# Conclusions

MOTIVATED by the needs to shorten the 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.

## 10.1 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 the LJRT 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 GCI prototype implementation may 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 further 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 trade-off 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.

## 10.2 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 comprehensible, compiler implementation.

- Source code analysis, refactorings, and optimizations, can be conveniently described as cross-cutting 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.

## 10.3 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 ReRAGs 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.
- Two different implementations of the Java exception mechanism which can be used together with an incremental RTGC.
- The *Latency* versus *Predictability* trade-off, concerning the use of non-GC-aware (legacy) code in a real-time Java application, is brought forward.

## 10.4 Concluding remarks

With these conclusions and contributions, we can now look back at the problem statement in Section 1.4 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 in many cases 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 trade-off 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, May 2001.

[Boe81] Barry W. Boehm. *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.

[CS02] Angelo Corsaro and Douglas C. Schmidt. The design and performace of the jRate real-time Java implementation. In *Proceedings of the 4thInternational Symposium on Distributed Objects and Applications, DOA 2002*, October 2002.

[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. In *Proceedings of the 18$^{th}$ European Conference on Object-Oriented Computing (ECOOP) 2004*, Olso, June 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.

[Ekm04a] Torbjörn Ekman. A casy study of separation of concerns in compiler construction using JastAdd II. In *Proceedings of the Third AOSD workshop on Aspects, Components, and Patterns for Infrastructure Software (ACI4IS)*, 2004.

[Ekm04b] Torbjörn Ekman. *Rewritable Reference Attribute Grammars*. Licentiate thesis LU-CS-LIC:2004-3, Lund Institute of Technology, Lund University, Sweden, June 2004. LU-CS-LIC:2004-3.

[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 - the RTLinux Company. *http://www.fsmlabs.com/*, 2004.

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

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1995.

[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.

[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.

[Hed00] Görel Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), 2000.

[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.

[HN99] Mathias Haage and Klas Nilsson. On the scalability of visualization in manufacturing. In *In Proceedings of ETFA '99*, 1999.

[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.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[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.

[Knu68a] Donald E. Knuth. Semantics of context–free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp.95–96 (March 1971).

[Knu68b] 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 realtime 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, 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.

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

[Rey94] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In *Theoretical aspects of object-oriented programming: types, semantics, and language design*, pages 13–23. MIT Press, 1994.

[RH03] Sven Gestegård Robertz and Roger Henriksson. Time-triggered garbage collection — robust and adaptive real-time GC scheduling for embedded systems. In *Proceedings of LCTES'03*. ACM Press, June 2003.

[RH05] Sven Gestegård Robertz and Roger Henriksson. Accurate concurrent GC in an uncooperative environment — performance vs predictability? In preparation, 2005.

[Rob06] Sven Gestegård Robertz. *Automatic memory management for flexible real-time systems*. PhD thesis, Dep. of Computer Science, Lund University, May 2006.

[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.

[Sch05] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[Sch06] Martin Schoeberl. JOP - Java Optimized Processor. World Wide Web, 2006. *http://www.jopdesign.com*.

[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.

[Sie04] Fridtjof Siebert. The impact of realtime garbage collection on realtime Java programming. In *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 33–40. IEEE Computer Society, IEEE, May 2004.

[SME] SMErobot — The European Robot Initiative for Strengthening the Competitiveness of SMEs in Manufacturing. *http://www.smerobot.org*.

[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.

[Tim06] Timesys inc. website, 2006. *http://www.timesys.com/java/*.

[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*.

[W3C04] Web ontology language owl. *http://www.w3.org/2004/OWL/*, 2004.

[Xen06] Xenomai real-time framwork for linux. *http://xenomai.org*, 2006.

# Appendix A

# Acronyms

**AG** Attribute Grammar

**AOP** Aspect-Oriented Programming

**API** Application Programming Interface

**AST** Abstract Syntax Tree

**CPU** Central Processing Unit, microprocessor

**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

**LJRT** Lund Java-based Real Time

**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 Collection

**RTSJ** Real-Time Specification for Java

**RTOS** Real-Time Operating System

**UI** User Interface

**WCET** Worst-Case Execution Time

# Appendix B

# LJRT compiler source

## B.1 Abstract grammar for Java

Below is the complete Java context-free grammar used by the JastAdd tool to produce appropriate AST node types in the LJRT compiler front-end.

```
Program ::= CompilationUnit*;

// 7.3 Compilation Units
CompilationUnit ::= PackageDecl:IdDecl* ImportDecl* TypeDecl*;

// 7.5 Import Declarations
abstract ImportDecl ::= Access;
SingleTypeImportDecl : ImportDecl;
TypeImportOnDemandDecl : ImportDecl;

abstract Access : Expr;

AbstractDot : Access ::= Left:Expr Right:Access;
PackageDot : AbstractDot;
FieldDot : AbstractDot;
MethodDot : AbstractDot;
TypeDot : AbstractDot;
ThisDot : AbstractDot;
SuperDot : AbstractDot;
ArrayTypeDot : AbstractDot;
ArrayNameDot : AbstractDot;
ArrayDot : AbstractDot;
ClassDot : AbstractDot;
SuperConstructorDot : AbstractDot;

Dot : AbstractDot;

// Specialized static/virtual types used in Liveness analysis
StaticMethodDot : MethodDot;
VirtualMethodDot : MethodDot;
StaticMethodAccess : MethodAccess;
```

```
VirtualMethodAccess : MethodAccess;

TypeAccess : Access ::= Package:IdUse* IdUse;
ArrayTypeAccess : TypeAccess ::= /Package:IdUse*/
                                 /IdUse/
                                 Access
                                 <Dimension:int>;
ThisAccess : TypeAccess;
SuperAccess : TypeAccess;

ArrayAccess : Access ::= Expr;

VarAccess : Access ::= IdUse;
MethodAccess : Access ::= Arg:Expr* IdUse;
ConstructorAccess : Access ::= Arg:Expr* IdUse;
SuperConstructorAccess : ConstructorAccess;
PackageAccess : Access ::= Package:IdUse*;
PackageOrTypeAccess : Access ::= IdUse;
AmbiguousAccess : Access ::= IdUse;
ArrayNameAccess : Access ::= <Dimension:int> <dummy:boolean>;

ClassInstanceExpr : Access ::= Access Arg:Expr* [TypeDecl];
ClassInstanceDot : AbstractDot;

ClassAccess : Access ::= ;

ParseName : Access ::= IdUse;
ArrayTypeName : Access ::= EmptyBracket*;
ParseArray : Access ::= Dims*;
ParseMethodName : Access ::= IdUse Arg:Expr*;

abstract TypeDecl ::= Modifiers IdDecl BodyDecl*;

// 4.1 The Kinds of Types and Values
abstract ReferenceType : TypeDecl;
PrimitiveType : TypeDecl ::= Modifiers
                             IdDecl
                             [SuperClassAccess:Access]
                             BodyDecl*;
NullType : TypeDecl;
VoidType : TypeDecl;

UnknownType : ClassDecl;
IllegalType : ClassDecl;

// 4.2 Primitive Types and Values
abstract NumericType : PrimitiveType;
BooleanType : PrimitiveType;
abstract IntegralType : NumericType;
ByteType : IntegralType;
ShortType : IntegralType;
IntType : IntegralType;
LongType : IntegralType;
CharType : IntegralType;
FloatingPointType : NumericType;
FloatType : FloatingPointType;
DoubleType : FloatingPointType;

// 4.3 Reference Types and Values
ClassDecl : ReferenceType ::= Modifiers
                              IdDecl
                              [SuperClassAccess:Access]
                              Implements:Access*
                              BodyDecl*;
InterfaceDecl : ReferenceType ::= Modifiers
                                  IdDecl
                                  SuperInterfaceId:Access*
                                  BodyDecl*;
ArrayDecl : ClassDecl ::= Modifiers
                          IdDecl
                          [SuperClassAccess:Access]
                          Implements:Access*
                          BodyDecl*
                          <ElementType:TypeDecl>
                          <Dimension:int>;

AnonymousDecl : ClassDecl ::= Modifiers
                              IdDecl
                              /[SuperClassAccess:Access]/
                              /Implements:Access*/
                              BodyDecl*;

abstract BodyDecl;
InstanceInitializer : BodyDecl ::= Block;
StaticInitializer : BodyDecl ::= Block;
ConstructorDecl : BodyDecl ::= Modifiers
                               IdDecl
                               Parameter*
                               Exception:Access*
                               [ConstructorInvocation:Stmt]
                               Block;

abstract MemberDecl : BodyDecl;

FieldDecl : MemberDecl ::= Modifiers TypeAccess:Access VariableDecl*;
FieldDeclaration : MemberDecl ::= Modifiers
                                  TypeAccess:Access
                                  IdDecl
                                  [AbstractVarInit]; // Simplified
                                                     // FieldDecl

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

VariableDecl ::= IdDecl EmptyBracket* [AbstractVarInit];

Parameter ::= Modifiers TypeAccess:Access IdDecl EmptyBracket*;
ParameterDeclaration : Parameter ::= Modifiers
                                     TypeAccess:Access
                                     IdDecl
                                     /EmptyBracket*/; // Simplified
                                                      // Parameter

EmptyBracket;

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

```
MethodDecl : MemberDecl ::= Modifiers
                            TypeAccess:Access
                            IdDecl
                            Parameter*
                            EmptyBracket*
                            Exception:Access*
                            [Block]; // Create simplified version

// 8.5 Member Type Declarations
abstract MemberType : MemberDecl ::= TypeDecl;
MemberClass : MemberType ::= ClassDecl /TypeDecl/;
MemberInterface : MemberType ::= InterfaceDecl /TypeDecl/;

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 ;
LongLiteral : Literal ;
FPLiteral : Literal ;
DoubleLiteral : Literal ;
BooleanLiteral : Literal ;
CharLiteral : Literal ;
StringLiteral : Literal ;
NullLiteral : Literal ;

ParExpr : PrimaryExpr ::= Expr;

StringLiteralExpr : PrimaryExpr ::= StringLiteral;

PrimTypeClassExpr : PrimaryExpr ::= <ID>;

ArrayInstanceExpr : PrimaryExpr ::= TypeAccess:Access Dims* [ArrayInit];
Dims ::= [Expr];


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

```
LogNotExpr : Unary ;

CastExpr : Expr ::= TypeAccess:Access 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 : Expr ::= Expr TypeAccess:Access;


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

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

Modifiers ::= Modifier*;
Modifier ::= <ID>;

// Statements

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

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

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

```
WhileStmt : BranchTargetStmt ::= Condition:Expr Stmt;

DoStmt : BranchTargetStmt ::= Stmt Condition:Expr;

ForStmt : BranchTargetStmt ::= 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;

AssertStmt : Stmt ::= first:Expr [Expr];

LocalClassDeclStmt : Stmt ::= ClassDecl;
```

## B.2 Reachability analysis

As an example on what the LJRT compiler source looks like, the complete source code for performing reachability analysis is shown below.

```
aspect Reachable {
    syn boolean BodyDecl.reachable() circular [false] {
        return false;
    }

    syn boolean InstanceInitializer.reachable() circular [false] {
        return getThisClassDecl().reachable();
    }

    syn boolean StaticInitializer.reachable() circular [false] {
        return getThisClassDecl().reachable();
    }


    syn boolean MethodDecl.reachable() circular [false] {
        if (name().equals("main") || name().equals("run")) {
            return true;
        }
        for (Iterator iter = callers().iterator(); iter.hasNext();) {
            BodyDecl m = (BodyDecl) iter.next();
            if (m.reachable()) return true;
        }
        return false;
    }

    syn boolean ConstructorDecl.reachable() circular [false] {
        if (signature().equals("Thread()")) return true;
        for (Iterator iter = callers().iterator(); iter.hasNext();) {
            BodyDecl m = (BodyDecl) iter.next();
            if (m.reachable()) return true;
        }
        return false;
    }

    syn boolean TypeDecl.reachable() circular [false] {
        for (Iterator iter = callers().iterator(); iter.hasNext();) {
            BodyDecl m = (BodyDecl) iter.next();
            if (m.reachable()) return true;
        }
        return false;
    }
}

aspect CallerCrossRefs {
    // Fill in cross-references for callee -> caller


    boolean Program.callerCrossRefsFilledIn = false;
    private Set BodyDecl.privateCallers = new HashSet();
    protected Set TypeDecl.privateCallers = new HashSet();

    public Set BodyDecl.callers() {
        Program root = getProgram();
        if (!root.callerCrossRefsFilledIn) {
            root.fillInCallerCrossRefs();
            root.callerCrossRefsFilledIn = true;
        }
        return privateCallers;
    }

    public Set TypeDecl.callers() {
        Program root = getProgram();
        if (!root.callerCrossRefsFilledIn) {
            root.fillInCallerCrossRefs();
            root.callerCrossRefsFilledIn = true;
        }
        return privateCallers;
    }

    void ASTNode.fillInCallerCrossRefs() {
        for(int i=0; i<getNumChild(); i++) {
            getChild(i).fillInCallerCrossRefs();
        }
    }

    void BodyDecl.fillInCallerCrossRefs() {
        for (Iterator iter = calls().iterator(); iter.hasNext();) {
            BodyDecl m = (BodyDecl) iter.next();
            m.privateCallers.add(this);
        }
        super.fillInCallerCrossRefs();
    }

    void TypeDecl.fillInCaller(BodyDecl caller) {}

    void ClassDecl.fillInCaller(BodyDecl caller) {
        privateCallers.add(caller);
        for (int i=0; i<getNumImplements(); i++) {
            ((TypeAccess) getImplements(i)).type().fillInCaller(caller);
        }
        if (hasSuperClass()) {
            getSuperClass().fillInCaller(caller);
        }
    }

    void InterfaceDecl.fillInCaller(BodyDecl caller) {
        privateCallers.add(caller);
    }
}

aspect GetProgram {

    syn Program ASTNode.getProgram() = getParent().getProgram();

    eq Program.getProgram() = this;

}

aspect TCG {
    // Total Call Graph
    syn Set BodyDecl.calls() {
        HashSet s = new HashSet();
        this.collectCalls(s);
        return s;
    }

    void ASTNode.collectCalls(Set s) {
        for (int i = 0; i < getNumChild(); i++) {
            getChild(i).collectCalls(s);
        }
    }

    void MethodDot.collectCalls(Set s) {
        s.addAll(potentialTargetMethods());
        super.collectCalls(s);
    }

    void FieldDot.collectCalls(Set s) {
        s.addAll(potentialTargetMethods());
    }

    void ClassInstanceExpr.collectCalls(Set s) {
        // Normal Constructor calls
        s.addAll(potentialTargetMethods());
    }

    void ConstructorAccess.collectCalls(Set s) {
        // Inside Constructor declaration
        s.addAll(potentialTargetMethods());
    }

    void InstanceOfExpr.collectCalls(Set s) {
        // Inside Constructor declaration
        s.addAll(potentialTargetMethods());
    }

    void Catch.collectCalls(Set s) {
        // It is not sure that the caught exception is referenced in
        // the catch block.
        s.addAll(potentialTargetMethods());
        super.collectCalls(s);
    }

    syn Set Expr.potentialTargetMethods() = new HashSet();

    eq StaticMethodDot.potentialTargetMethods() {
        MethodDecl m = methodAccess().decl();
        m.getThisClassDecl().fillInCaller(m);
        HashSet s = new HashSet();
        s.add(m);
        return s;
    }

    eq VirtualMethodDot.potentialTargetMethods() {
        TypeDecl f = getLeft().type();
        MethodDecl m = methodAccess().decl();
        if (m.getThisClassDecl() != null ) {
            m.getThisClassDecl().fillInCaller(m);
        }
        HashSet s = new HashSet();
        s.add(m);

        for(Iterator iter=m.overriders().iterator(); iter.hasNext();) {
            MethodDecl mo = (MethodDecl) iter.next();
            if (mo.hostType().instanceOf(f)) {
                if (mo.getThisClassDecl() != null ) {
                    mo.getThisClassDecl().fillInCaller(mo);
                }
                s.add(mo);
            }
        }
        return s;
    }

    eq ClassInstanceExpr.potentialTargetMethods() {
        ConstructorDecl m = decl();
        m.getThisClassDecl().fillInCaller(m);
        HashSet s = new HashSet();
        s.add(m);
        return s;
    }

    eq ConstructorAccess.potentialTargetMethods() {
        ConstructorDecl m = decl();
        m.getThisClassDecl().fillInCaller(m);
        HashSet s = new HashSet();
        s.add(m);
        return s;
    }

    eq FieldDot.potentialTargetMethods() {
        FieldDeclaration f = decl();
        if (f.getThisClassDecl() != null ) {
            f.getThisClassDecl().fillInCaller(getThisBodyDecl());
        }
        HashSet s = new HashSet();
        return s;
    }

    Set Catch.potentialTargetMethods() {
        getParameter().type().fillInCaller(getThisBodyDecl());
        HashSet s = new HashSet();
        return s;
    }

    eq InstanceOfExpr.potentialTargetMethods() {
        getTypeAccess().type().fillInCaller(getThisBodyDecl());
        HashSet s = new HashSet();
        return s;
    }

}

aspect SpecializeStaticVirtualCalls {
        syn boolean MethodDot.isExactMethodDot() = true;
        eq StaticMethodDot.isExactMethodDot() = false;
        eq VirtualMethodDot.isExactMethodDot() = false;

        rewrite MethodDot {
                when (isExactMethodDot() &&
                      methodAccess().decl().isStatic())
                to StaticMethodDot
                   new StaticMethodDot(getLeft(), getRight());
                when (isExactMethodDot() &&
                      !methodAccess().decl().isStatic())
                to VirtualMethodDot
                   new VirtualMethodDot(getLeft(), getRight());
        }

        syn boolean MethodAccess.isExactMethodAccess() = true;
        eq StaticMethodAccess.isExactMethodAccess() = false;
        eq VirtualMethodAccess.isExactMethodAccess() = false;

        rewrite MethodAccess {
                when (isExactMethodAccess() && decl().isStatic())
                to StaticMethodAccess
                   new StaticMethodAccess(getArgList(), getIdUse());
                when (isExactMethodAccess() && !decl().isStatic())
                to VirtualMethodAccess
                   new VirtualMethodAccess(getArgList(), getIdUse());
        }
}


aspect OverridesCrossRefs {

    // Fill in cross-references for
        // - overrides -> overriders
        // - TODO: handle implements relation
        // - TODO: handle sub/super-interfaces

        public Collection MethodDecl.overriders() {
                Program root = getProgram();
                if (!root.overridesCrossRefsFilledIn) {
                        root.overridesCrossRefs();
                        root.overridesCrossRefsFilledIn = true;
                }
                return privateOverriders;
        }

        private LinkedList MethodDecl.privateOverriders =
                                new LinkedList();

        boolean Program.overridesCrossRefsFilledIn = false;

        void ASTNode.overridesCrossRefs() {
                for(int i=0; i<getNumChild(); i++) {
                        getChild(i).overridesCrossRefs();
                }
        }

        void MethodDecl.overridesCrossRefs() {
           for (Iterator iter = overrides().iterator(); iter.hasNext();) {
             MethodDecl m = (MethodDecl) iter.next();
             m.privateOverriders.add(this);
           }
           super.overridesCrossRefs();
        }


}
```

The reachability analysis module consists of about 400 lines of code, of the around 16 000 comprising the complete LJRT compiler.